Embedding a Console in a C# Application

Dave Kerr

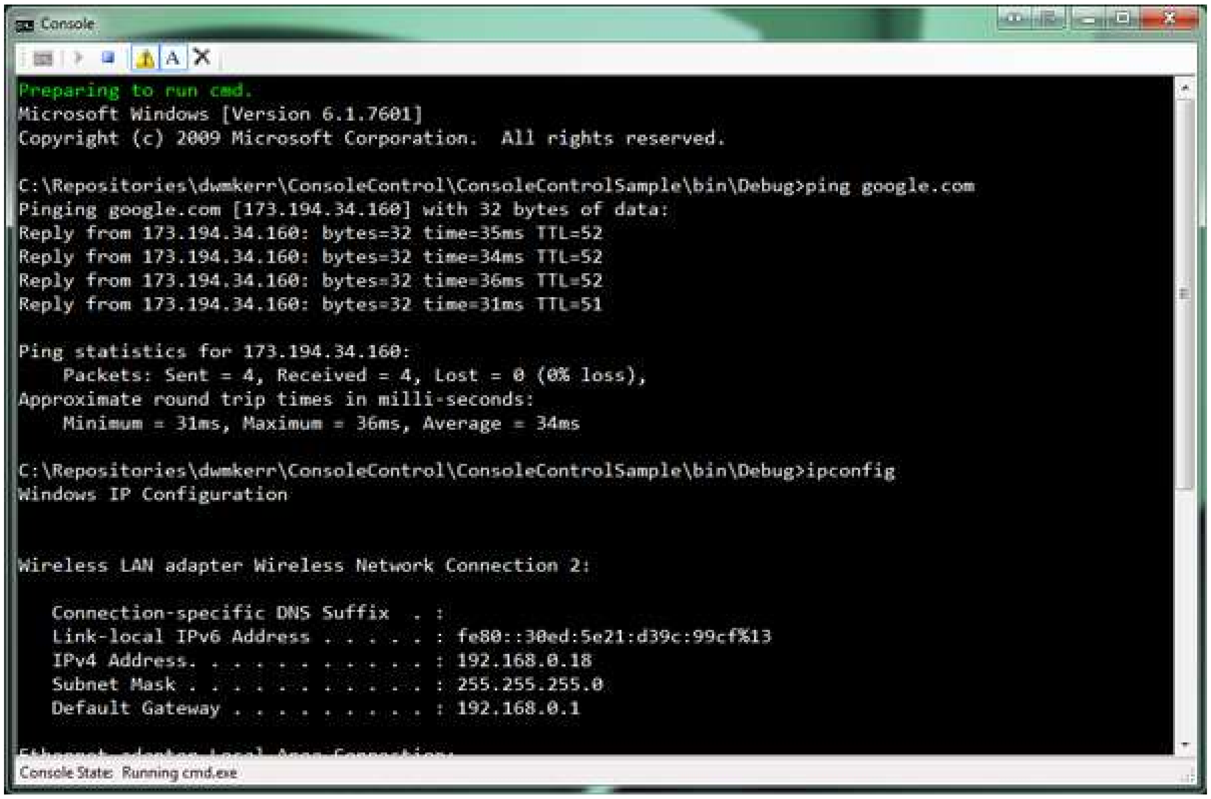
22 Mar 2015 MIT

Embed a functional console window in a C# application

Download source code - 142.6 KB

Download binaries and sample applications - 90.5 KB

Get the latest code on GitHub



```
Preparing to run cmd.
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Repositories\dwmkerr\ConsoleControl\ConsoleControlSample\bin\Debug>ping google.com
Pinging google.com [173.194.34.160] with 32 bytes of data:
Reply from 173.194.34.160: bytes=32 time=35ms TTL=52
Reply from 173.194.34.160: bytes=32 time=34ms TTL=52
Reply from 173.194.34.160: bytes=32 time=36ms TTL=52
Reply from 173.194.34.160: bytes=32 time=31ms TTL=51

Ping statistics for 173.194.34.160:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 31ms, Maximum = 36ms, Average = 34ms

C:\Repositories\dwmkerr\ConsoleControl\ConsoleControlSample\bin\Debug>ipconfig
Windows IP Configuration


Wireless LAN adapter Wireless Network Connection 2:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::30ed:5e21:d39c:99cf%13
   IPv4 Address. . . . . . . . . . . : 192.168.0.18
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.0.1
```

Console State: Running cmd.exe

Introduction

As developers, ever so often, we have to kick off some command line programs. Compiling and building tools, utilities, and so on. Recently, I have been writing a small application that calls Oracle's '`impdp`' and '`expdp`' tools for you. While working on it, I thought - it'd be nice to see the console program running, but *inside* my application. This article shows you how to create a Console Control that lets you do just that.

Tip: The `ConsoleControl` is available for both WPF and WinForms.

You can follow the project on GitHub at https://github.com/dwmkerr/consolecontrol to track issues, suggest features and find documentation.

Using the Control

> **Tip:** Use Nuget! Install the package `ConsoleControl` for WinForms or `ConsoleControl.WPF` for WPF.

If you just need the control, download the binary and sample application and add a reference to *ConsoleControl.dll*. You'll get the `ConsoleControl` in the toolbox, drop it into a `Form`, and you're done. The key function is:

StartProcess(string fileName, string arguments)

This function starts the process specified by `fileName`. It will use the arguments supplied. The process will run, but its output will be sent to the console control. The console control can also take input from the keyboard.

Some other useful functions are:

StopProcess

Kills the running process.

WriteOutput(string output, Color color)

Writes the string '`output`' to the console window, using the specified color.

WriteInput(string input, bool echo, Color color)

Writes the `string` '`input`' to the running process' input stream. If '`echo`' is `true`, then the input will also be written to the Console Control (as if the user had keyed it in).

ShowDiagnostics

If this property is set to `true`, diagnostics will be shown. These include:

- A message saying '**Running X**' when you start a process
- A message saying `'`**X exited.**' when a process ends
- Exceptions written in red if the process cannot be run

IsInputEnabled

If this property is set to `true`, then the user can type input into the console window.

IsProcessRunning

This property evaluates to `true` if the process is currently running.

CurrentProcess

The current running process (if any).

InternalRichTextBox

Returns the rich text box which displays the console output.

How Does It Work?

Essentially, all we need to do to get the required functionality, is create a process and have direct access to its input and output streams. Here's how we can do that. Create a new User Control named '`ConsoleControl`' and add a rich text box to the `UserControl`. Now, we'll create the most key function - `StartProcess`:

```
/// <summary>
/// Runs a process
/// </summary>
/// <param name="fileName">Name of the file.</param>
/// <param name="arguments">The arguments.</param>
public void StartProcess(string fileName, string arguments)
{
    // Create the process start info
    var processStartInfo = new ProcessStartInfo(fileName, arguments);

    // Set the options
    processStartInfo.UseShellExecute = false;
    processStartInfo.ErrorDialog = false;
    processStartInfo.CreateNoWindow = true;

    // Specify redirection
    processStartInfo.RedirectStandardError = true;
    processStartInfo.RedirectStandardInput = true;
    processStartInfo.RedirectStandardOutput = true;
```

The first thing we do is create the process start info. To make sure that it doesn't show a window, we make sure `CreateNoWindow` is set to `true`. We will also specify that we're redirecting standard output, standard error, and standard input.

```
// Create the process
currentProcess = new Process();
currentProcess.EnableRaisingEvents = true;
currentProcess.StartInfo = processStartInfo;
currentProcess.Exited += new EventHandler(currentProcess_Exited);

// Try to start the process
try
{
    bool processStarted = currentProcess.Start();
}
catch (Exception e)
{
    // We failed to start the process. Write the output (if we have diagnostics on).
    if(ShowDiagnostics)
        WriteOutput("Failed: " + e.ToString() + Environment.NewLine, Color.Red);
    return;
}

// Store name and arguments
currentProcessFileName = fileName;
currentProcessArguments = arguments;
```

Now that we have the process start info, we can actually start the process. We also store the file name and arguments for later. The class must also contain two background workers, to read the output stream and error stream. Now we start them - and enable editing if the `IsInputEnabled` flag is on. We also store the standard input, output and error streams.

```
    // Create the readers and writers
    inputWriter = currentProcess.StandardInput;
    outputReader = TextReader.Synchronized(currentProcess.StandardOutput);
    errorReader = TextReader.Synchronized(currentProcess.StandardError);

    // Run the output and error workers
    outputWorker.RunWorkerAsync();
    errorWorker.RunWorkerAsync();

    // If we enable input, make the control not read only
    if (IsInputEnabled)
        richTextBoxConsole.ReadOnly = false;
}
```

The background workers that read the standard input and error streams work in the same way, so we only need to see one of them:

```
/// <summary>
/// Handles the DoWork event of the outputWorker control
/// </summary>
/// <param name="sender">The source of the event.</param>
/// <param name="e">The <see cref="System.ComponentModel.DoWorkEventArgs"/>
///    instance containing the event data.</param>
void outputWorker_DoWork(object sender, DoWorkEventArgs e)
{
    // Keep working until we're told to cancel
    while (outputWorker.CancellationPending == false)
    {
        // Just keep on reading the buffer
        int count = 0;
        char[] buffer = new char[1024];
        do
        {
            // Create a builder
            StringBuilder builder = new StringBuilder();

            // Read and append data
            count = outputReader.Read(buffer, 0, 1024);
            builder.Append(buffer, 0, count);

            // Report the progress
            outputWorker.ReportProgress(
                  0, new OutputEvent() { Output = builder.ToString() });
        } while (count > 0);
    }
}
```

All we are doing here is reading from the specified buffer. We then use the '`ReportProgress`' function of the worker to tell the control to update the screen. The Report Progress event handler works as below:

```
/// <summary>
/// Handles the ProgressChanged event of the outputWorker control
/// </summary>
/// <param name="sender">The source of the event.</param>
/// <param name="e">The <see cref="System.ComponentModel.ProgressChangedEventArgs"/>
///    instance containing the event data.</param>
void outputWorker_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    if (e.UserState is OutputEvent)
    {
        // Get the event data
        var outputEvent = e.UserState as OutputEvent;

        // Write the output
```

```
            WriteOutput(outputEvent.Output, Color.White);
        }
    }

    /// <summary>
    /// Writes the output to the console control
    /// </summary>
    /// <param name="output">The output.</param>
    /// <param name="color">The color.</param>
    public void WriteOutput(string output, Color color)
    {
        if (string.IsNullOrEmpty(lastInput) == false && output.Contains(lastInput))
            return;

        //  Invoke on the UI thread...
        Invoke((Action)(() =>
        {
            // Write the output
            richTextBoxConsole.SelectionColor = color;
            richTextBoxConsole.SelectedText += output;
            inputStart = richTextBoxConsole.SelectionStart;
        }));
    }
```

The event handler is trivial - we just cast the data and write the output. Writing the output is just a case of adding the text to the rich text box. `inputStart` is also updated - this lets us keep track of where the user can type. This is used in the `KeyDown` event of the internal rich text box, as we see below:

```
/// <summary>
/// Handles the KeyDown event of the richTextBoxConsole control
/// </summary>
/// <param name="sender">The source of the event.</param>
/// <param name="e">The <see cref="System.Windows.Forms.KeyEventArgs"/>
///    instance containing the event data.</param>
void richTextBoxConsole_KeyDown(object sender, KeyEventArgs e)
{
    // If we're at the input point and it's backspace, bail
    if ((richTextBoxConsole.SelectionStart <= inputStart) &&
                e.KeyCode == Keys.Back) e.SuppressKeyPress = true;

    // If we're before the input point and it's anything but arrows, bail.
    if ((richTextBoxConsole.SelectionStart < inputStart) &&
      !(e.KeyCode == Keys.Left || e.KeyCode == Keys.Right ||
                   e.KeyCode == Keys.Up || e.KeyCode == Keys.Down))
        e.SuppressKeyPress = true;

    if (e.KeyCode == Keys.Return)
    {
        // Get the input
        string input = richTextBoxConsole.Text.Substring(inputStart,
               (richTextBoxConsole.SelectionStart) - inputStart);

        // Write the input (without echoing)
        WriteInput(input, Color.White, false);

        // We're done
        e.SuppressKeyPress = true;
    }
}
```

If the selection is before the input start, we don't let the user enter text. If the return key is pressed, we write it to the output stream.

This is the bulk of the code - as you can see, it is not complex, but it does the job.

Final Thoughts

This control is basic - it may support what you need, if not, do feel free to use it as a baseline for your own projects. Some things that the control *doesn't* do are listed below:

1. Restrict output to 80 characters horizontally
2. Respect changes to the console foreground and background
3. Support programmatic clear-screen, etc. Remember - we're streaming output, if you need to run a program that manipulates the console, this is not the control for you!

Ctrl-C

There has been quite some discussion in the messages about the complexity of sending a command like '**Control C**' to a cmd process in the console window. Just so that others can find the solution until something formal is actually published, VisualG seems to have a good solution which you can find here:

- http://www.codeproject.com/Articles/335909/Embedding-a-Console-in-a-C-Application?msg=4320232#xx4320232xx

Scroll to Bottom

Again, thanks to VisualG for the solution that let's you automatically scroll to the bottom - this'll come in a release soon! Until then, see the message below:

- http://www.codeproject.com/Articles/335909/Embedding-a-Console-in-a-C-Application?msg=4642047#xx4642047xx

License

About the Author

Dave Kerr

Software Developer
United Kingdom

Follow my blog at www.dwmkerr.com and find out about my charity at www.childrenshomesnepal.org.

Comments and Discussions

163 messages have been posted for this article Visit **https://www.codeproject.com/Articles/335909/Embedding-a-Console-in-a-C-Application** to post and view comments on this article, or click **here** to get a print view with messages.

Permalink
Advertise