C# CLIWrap Chaining Multiple Statements

Normally, each call to CLIWrap is performed inside an independent terminal session.

So, there multiple calls will not affect eachother, like one to set an environment variable, another to change folders, and a third to start a process, with that environment variable and working directory.

A couple of ways exist to still make this happen.

Compound Statements

This is the quick and dirty method of chaining statements together with '&&', such as:

```
echo 'value' && cd /etc/nging/sites-available && cat ./newconfig.conf
```

The above is gibberish statements, but illustrates how multiple commands can be chained into one statement.
And, we can pass the above as a single statement to a CLIWrap call.

The downside of this is that, we wouldn't know which failed, if a failure occurred.

With Directives

CLIWrap provides some directives for the more common commands that would be chained together.

WithEnvironmentVariables - allows us to set environment variables for the specific command session.

WithWorkingDirectory - allows us to change the working folder before the command executes.

If your commands that require execution in the same session go beyond the above, you could think about a persistent shell stream. See below.

Persistent Shell Streams

This technique uses a single open CLIWrap session, and we pass command statements through standard input, one at a time.

In this technique, we are ensured that all statements will execute in the same session, and that we get responses and exitcode from each one.

And when we are done, we close the session, similar to a user closing an interactive terminal session with 'exit'.
We close with exactly that as well.

Below is mockup of this reactive (conversational) method of sending subsequent commands, and process the response of each, before exiting the session.

```
using CliWrap;
using System;
using System.Text;
using System.Threading.Tasks;
using System.IO.Pipelines;

class Program
{
    // Marker to help us identify which line contains the exit code
    private const string ExitCodeMarker = "__EXITCODE__";

    static async Task Main(string[] args)
    {
        // Set up a string builder to accumulate lines of output
        // for parsing in real time
        var outputBuffer = new StringBuilder();

        // Create a persistent bash session
        var bash = Cli.Wrap("/bin/bash")
            .WithStandardInputPipe(PipeSource.Pipe())
            .WithStandardOutputPipe(PipeTarget.ToDelegate(line =>
            {
                // Write everything for debugging/logging
                Console.WriteLine($"[OUTPUT] {line}");

                // Also accumulate in buffer for parsing
                outputBuffer.AppendLine(line);
            }))
            .WithStandardErrorPipe(PipeTarget.ToDelegate(line =>
            {
                // Log errors
```

```
            Console.WriteLine($"[ERROR] {line}");
        }));

    var stdin = bash.StandardInput.PipeWriter;
    var bashTask = bash.ExecuteAsync();

    try
    {
        // Example 1: Change directory
        var cdExitCode = await RunCommandAndGetExitCodeAsync(stdin, outputBuffer, "cd /my/dir");
        if (cdExitCode != 0)
        {
            Console.WriteLine($"[INFO] cd failed with exit code {cdExitCode}. Aborting...");
            goto Cleanup;
        }

        // Example 2: Export an environment variable
        var exportExitCode = await RunCommandAndGetExitCodeAsync(stdin, outputBuffer, "export
MY_VAR=some_value");
        if (exportExitCode != 0)
        {
            Console.WriteLine($"[INFO] export failed with exit code {exportExitCode}. Aborting...");
            goto Cleanup;
        }

        // Example 3: Start a process
        var processExitCode = await RunCommandAndGetExitCodeAsync(stdin, outputBuffer, "./start_process");
        Console.WriteLine($"[INFO] Process completed with exit code {processExitCode}.");
    }
    finally
    {
    Cleanup:
        // Ensure we exit the shell
        await stdin.WriteAsync("exit\n");
        await stdin.CompleteAsync();
    }

    // Wait for the bash session to end
    await bashTask;
}
```

```
/// <summary>
/// Sends a command to the bash session, then echos the exit code, parses it,
/// and returns the integer exit code to the caller.
/// </summary>
private static async Task<int> RunCommandAndGetExitCodeAsync(
    PipeWriter stdin,
    StringBuilder outputBuffer,
    string command)
{
    // Clear out old data from the buffer
    outputBuffer.Clear();

    // 1) Send the command
    await stdin.WriteAsync($"{command}\n");
    Console.WriteLine($"[COMMAND SENT] {command}");

    // 2) Immediately echo $? along with a unique marker
    await stdin.WriteAsync($"echo \"{ExitCodeMarker}=$?\"\n");

    // 3) Wait for the exit code marker to appear in the output
    int exitCode = await WaitForExitCodeAsync(outputBuffer);

    return exitCode;
}

/// <summary>
/// Periodically checks the outputBuffer for our exit code marker.
/// Once found, parses and returns the integer exit code.
/// </summary>
private static async Task<int> WaitForExitCodeAsync(StringBuilder outputBuffer)
{
    // We do a simple polling approach here, but you could make it
    // more sophisticated if needed (e.g., reading line by line in real time).
    while (true)
    {
        // Parse the buffer for our exit code marker
        var text = outputBuffer.ToString();
        var markerIndex = text.IndexOf(ExitCodeMarker);
        if (markerIndex >= 0)
```

```
        {
            // e.g., line might look like:  __EXITCODE__=0
            var lineStart = text.IndexOf(ExitCodeMarker);
            var lineEnd = text.IndexOf('\n', lineStart);

            // Extract the portion containing the marker and exit code
            string line;
            if (lineEnd > 0)
                line = text.Substring(lineStart, lineEnd - lineStart);
            else
                line = text.Substring(lineStart);

            // line should look like "__EXITCODE__=0" or something similar
            var parts = line.Split('=');
            if (parts.Length == 2 && int.TryParse(parts[1], out int code))
            {
                return code;
            }
        }

        // Not found yet, or can't parse. Wait a bit and try again.
        await Task.Delay(100);
    }
  }
}
```

Explanation

1. **Persistent Shell**:
   - We start a bash shell using `Cli.Wrap("/bin/bash")`, and we capture its stdin, stdout, and stderr.
2. **RunCommandAndGetExitCodeAsync**:
   - Sends the actual command (e.g., `cd /my/dir`) followed by an `echo` statement to print `"$?"` (the exit code) with a unique marker (like `__EXITCODE__`).
   - We then wait for that marker to appear in the aggregated output buffer.
3. **Parsing the Exit Code**:
   - When we see a line like `__EXITCODE__=0`, we know the exit code is `0`.
   - We parse that integer and return it to the caller.
4. **Reactive Flow**:
   - Each time we run a command, we do so synchronously in terms of logic:
     - Send the command,
     - Wait for its exit code,

    - Decide what to do next (continue, abort, etc.).
  - This simulates a more "*conversational*" or "*interactive*" approach rather than blindly sending all commands at once.
5. **Polling vs. Event-Driven**:
  - In the above example, we do a small `while (true)` loop with `Task.Delay(100)` to poll the buffer.
  - For small commands, this works fine. For more robust scenarios, you could implement a more event-driven approach where each line from stdout is checked in real time as soon as it arrives. But the principle—pushing a unique marker into the output and then scanning for it—remains the same.

Key Features of the above approach (taken from a previous attempt that didn't retrieve the exitcode):

1. **Dynamic Decision-Making**:
  - The `SendCommandAndWaitAsync` method sends a command and then waits for output before deciding the next action.
  - You can parse `outputBuffer` after each command to make decisions (e.g., check exit codes or specific responses).
2. **Real-Time Feedback**:
  - Output is captured immediately and can be logged, processed, or used to trigger further actions.
3. **Seamless Flow**:
  - Commands are executed only after verifying the outcome of the previous one, making the interaction feel more "conversational."
4. **Flexibility**:
  - You can customize the waiting mechanism (`Task.Delay`) or introduce a more sophisticated strategy, like waiting for specific keywords or end markers in the output.

Example Interaction

Suppose you’re running the following commands:

```
cd /my/dir
export MY_VAR=some_value
./start_process
```

The program might produce:

```
[COMMAND SENT] cd /my/dir
[OUTPUT] 0
[COMMAND OUTPUT] 0
[COMMAND SENT] export MY_VAR=some_value
```

```
[OUTPUT]
[COMMAND OUTPUT]
[COMMAND SENT] ./start_process
[OUTPUT] Process started successfully
[COMMAND OUTPUT] Process started successfully
```

Considerations:

- **Command Completion Signals**:
  - Some commands might take longer to execute (e.g., `./start_process`). You might need to enhance the waiting mechanism to look for specific output or end markers (e.g., "done").
- **Error Handling**:
  - Ensure you handle errors gracefully if commands fail or produce unexpected output.
- **Performance**:
  - Avoid overly aggressive polling with `Task.Delay`; look for meaningful signals in the output to avoid unnecessary delays.

Summary

- **Yes**, you can run multiple commands in a persistent shell using `CLIWrap` and still capture **each individual command's exit code**.
- You just need to manually insert `echo $?` (or an equivalent marker-based approach) after each command to retrieve that exit code.
- By parsing the marker from stdout, your C# code can determine which exit code belongs to which command in a more "conversational" workflow.

Revision #2
Created 27 January 2025 00:49:08 by glwhite
Updated 19 March 2025 15:41:14 by glwhite