ES CONSOLE PROCESS ENGINE USER MANUAL

Overview

ES Console Process Engine is a pure Java component that allows controlled execution of a console process (non GUI program). By precisely automating the control of legacy (or new for that matter) console applications, user intervention is no longer required. All output from the program is logged to a file for monitoring purposes.

Console processes have 3 standard streams. They are called ‘input’, ‘output’, and ‘error’. The engine monitors the ‘output’ stream by parsing the data and looking for certain values. Once these values are found, the engine has the option to write values to the ‘input’ stream, simulating user intervention. Once a value is fed to the ‘input’ stream, it is possible to look for totally new ‘output’ values to continue the ‘input’ feeding.

Of course, the supplying of values to standard ‘input’ is only optional. The engine can simply be used to run a console process that writes to standard ‘output’, but provide logging to a file (for those processes not requiring user intervention).

Console Process Engine Settings

All configuration settings for the engine come from an XML file. The default file is called ‘Console Process Engine.xml’ and can be found in the installation directory. The settings can either be configured through the XML file or the graphical user interface (preferred way). Figure 1 is a sample screenshot of the graphical user interface settings dialog.

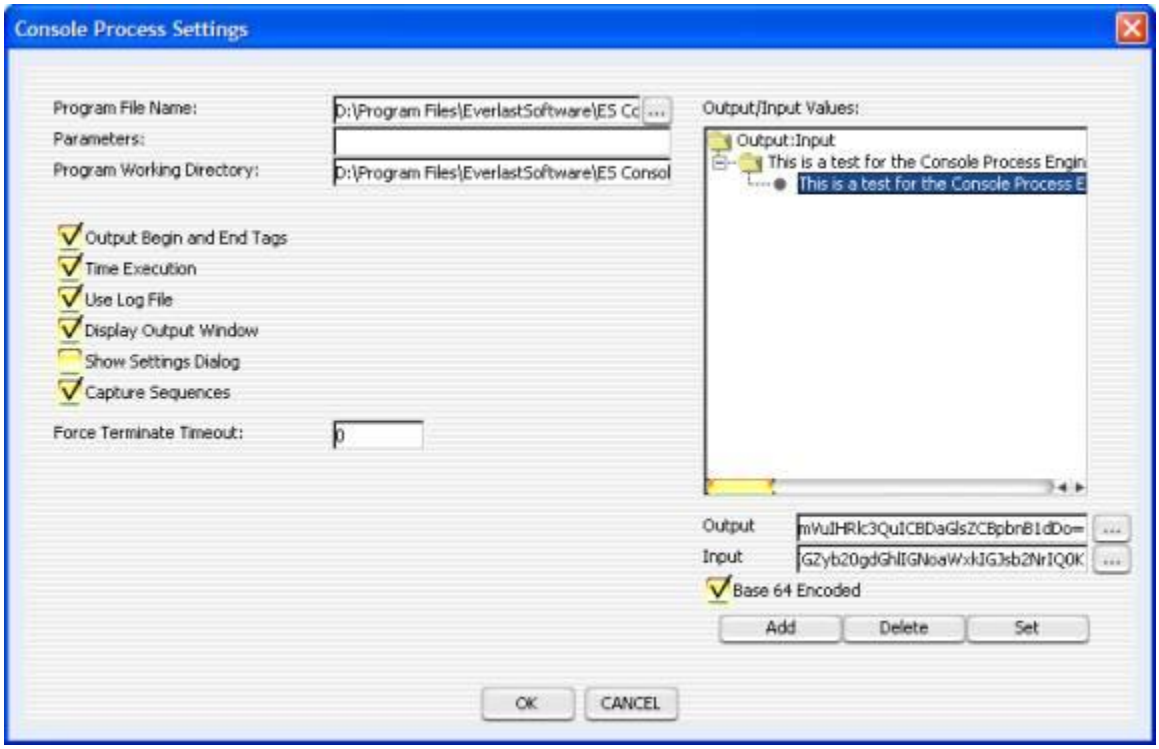

FIGURE 1

The following is a list of the various settings and what they mean (GUI name with XML tag name in parenthesis):

1) Program File Name (programFileName) – This is the name of the console program to execute.

2) Parameters (parameters) – This is an array of the parameters to pass to the program when it begins execution.

3) Program Working Directory (programWorkingDirectory) – This is the working directory (current directory) the program will execute in.

4) Output Begin and End (outputBeginAndEnd) – If set to true, the fact that the process started execution is logged, as well as when it completes. Upon completion, the process's exit code is also written to the log file.

5) Time Execution (timeExecution) – This flag indicates whether or not the total execution time of the process is timed. If so, the total time is logged upon completion.

6) Use Log File (useLogFile) – This flag indicates whether all output should be logged to a file or not. It is highly recommended to leave this set to true, unless of course there is a good reason not to have a log file.

7) Display Output Window (displayOutputWindow) – By default, an output window is displayed (on non-headless environments) as the console process runs when set to true. Any standard output messages (and fed input messages) appear in this window. For performance reasons, if the console program can execute unmonitored (so it only logs), this flag should be set to false.

8) Show Settings Dialog (showGUI) - If true, the console process engine settings dialog will be displayed at execution time.

9) Capture Sequences – If set to true (default), the engine will automatically capture any input typed into the display console window (if visible and enabled) and save the input/output values for the next execution. This makes it easy to simply execute the program one time, type the appropriate characters, and save the settings for any future execution. The capture will only save once the console program terminates.

10) Force Terminate Timeout (forceTerminateTimeout) – This is the number of milliseconds to wait before trying to force terminate the executed program process. If set to 0, force termination is disabled.

11) Output/Input Values (values) – This is the entire block for all the standard 'input' and 'output' values for reading/writing. Any time an item in the output/input tree is selected, the 'Output', 'Input', and 'Base 64 Encoded' fields will dynamically update. The 'Output:Input' node at the top of the tree should be ignored. It's required to use the 'Add' button to create new root output/input values. It will not be used in any way by the Console Process Engine.

12) Output (output) – A value to look for in order to feed input. The button next to this field with the '…' brings up a file dialog in order to browse for a file that contains the exact output data desired. When this button is used, the data will automatically be base 64 encoded for convenience.

13) Input (input) – The input to feed in based on finding the processed output value. This field also has a '…' button to browse for a file. This applies to the input that will be fed however.

14) Base 64 Encoded (base64Encoded) – This should only be used if non-displayable characters must be passed to standard 'input'. The 'newline' or 'endline' is a good example. So is a 'null', which is ASCII character 0. The 'output' value would then be a string that was base 64 encoded. In the sample demonstrated in Figure 1, base 64

encoding was used because the 'endline' character was expected by the program, and this character cannot be represented in a GUI component accurately. If selected, the tree will attempt to automatically base 64 decode the output and input values to give it's best visual display it can (don't worry, the values will still be utilized correctly even if they don't display properly). The 'Input' and 'Output' fields will still show the encoded values however.

15) Add – Creates a child stub output/input value underneath the item currently selected in the tree.

16) Delete – Deletes the currently selected output/input value in the tree (including any children underneath that item).

17) Set – Updates the currently selected item in the tree with the values in 'Output', 'Input', and 'Base 64 Encoded'. This does NOT immediately save the settings to the XML file, clicking the 'Ok' button does this. Set is required in order to make changes to an existing output/input value or to replace the stub values created for the new item when the 'Add' button is used.

The following are only included in the XML file for advanced usage:

18) com.everlast.io.OutputInputArray – This is simply another block required for each set of 'input' and 'output' values. Each time the engine must process one level deeper in the hierarchy of values, another one of these blocks must be set.

19) children – This is required for each level of the value hierarchy past the 1st.

20) onCompletionProgramFileName – This is the same as 'programFileName', except this is the optional program to run when the real program completes execution.

21) onCompletionProgramWorkingDirectory – The working directory for the completion program.

22) onCompletionParameters – The parameters for the completion program.

23) onErrorProgramFileName – Similar to 'programFileName', except this is the optional program to run when an error occurs while the real program is executing.

24) onErrorProgramWorkingDirectory – The working directory for the error program.

25) onErrorParameters – The parameters for the error program.

26) captureSequences – If set to true (default), the engine will automatically capture any input typed into the display console window (if visible and enabled) and save the input/output values for the next execution. This makes it easy to simply execute the program one time, type the appropriate characters, and save the settings for any future execution. The capture will only save once the console program terminates.

27) capturePartial – If set to true (and captureSequences is set to true), the engine will attempt to capture any input fed in and save this to the XML dynamically. This is only useful for only feeding partial data to input of a program. Sometimes this simply speeds up the process of a certain console program that still requires user intervention at a later point in the process. This is false by default.

An XML example values clause utilizing children:

```
<values>
  <com.everlast.io.OutputInputArray>
    <output>quit help.</output>
    <input>2</input>
    <base64Encoded>false</base64Encoded>
    <children>
      <com.everlast.io.OutputInputArray>
        <output>finished</output>
        <input>quit</input>
        <base64Encoded>false</base64Encoded>
      </com.everlast.io.OutputInputArray>
    </children>
  </com.everlast.io.OutputInputArray>
  <com.everlast.io.OutputInputArray>
    <output>error</output>
    <input>abort</input>
    <base64Encoded>false</base64Encoded>
  </com.everlast.io.OutputInputArray>
</values>
```

When the engine finds ‘quit help.’ written to standard ‘output’, it will feed a ‘2’ into standard ‘input’. From there, it will look for a different phrase ‘finished’. If found, the engine will supply ‘quit’ to standard ‘input.

If the engine finds ‘error’ instead of ‘quit help.’, it will write ‘abort’ to standard input.

An example values clause utilizing base 64 encoding is as follows:

```
<values>
  <com.everlast.io.OutputInputArray>
    <output>dGltZQ==</output>
    <input>bg0K</input>
    <base64Encoded>true</base64Encoded>
  </com.everlast.io.OutputInputArray>
</values>
```

When the 'output' is decoded, it actually contains the value 'time'. The decoded 'input' value contains 'n', followed by carriage return and line feed characters (non visible in a text file). This is the reason encoding was required. It's important to note that BOTH the output AND input must be encoded if the 'base64Encoded' flag is enabled.

Executing the Console Process Engine

There are two basic ways to execute the Console Process Engine. One way uses a launcher that allows selection and/or the creation of a new configuration for a console program to execute (friendly and easy way to execute on demand). The other way is to execute a specific console program configuration directly (used more for scheduled executions or batching executions).

Executing the Console Process Engine with Launcher

The launcher for the Console Process Engine is a friendly way to select and execute a specified console program. Often times, a user will want to have more than one console program configured for execution through the engine. The launcher allows the user to create, setup, and execute a configuration using a friendly dialog. Figure 2 (below) demonstrates the dialog.

FIGURE 2

On Windows systems, a convenience EXE file called 'es_console_process_engine_launcher.exe' exists in order to execute the engine launcher without having to execute Java directly. The launcher can also be invoked by simply executing the 'es_console_process_engine.jar' file. The manifest inside the jar causes the launcher to execute by default.

In order to choose an already existing configuration, click the combo box and select the desired engine name. A new configuration can be created by selecting '<NEW>' from the combo box.

The 'Execute' button executes the console program with its saved configuration.

The 'Settings' button allows the viewing and modifying of the configuration for the console program.

The 'Cancel' button simply causes the launcher to terminate.

Note: All configurations shown in the dialog are from the working directory of the execution of the launcher.

Executing the Console Process Engine Directly

To execute the engine, simply execute Java (typically 'java.exe' or 'javaw.exe'), and the classname of 'com.everlast.io.ConsoleProcessEngine' supplying 'Console Process Engine', with the classpath set to all jar files in the installation directory.

Examples:

(Windows, command prompt)

'java.exe –cp .;es_console_process_engine.jar;activation.jar; es_lookandfeel.jar com.everlast.io.ConsoleProcessEngine "Console Process Engine"'

(Windows, no command prompt, full path to jar files)

'javaw.exe –cp ".;d:\program files\everlastsoftware\ES Console Process Engine\es_console_process_engine.jar;d:\program files\everlastsoftware\ES Console Process Engine\activation.jar;d:\program files\everlastsoftware\ES Console Process Engine\es_lookandfeel.jar" com.everlast.io.ConsoleProcessEngine "Console Process Engine"'

(Windows, no command prompt, 'Nightly Backup Run.xml' file)

'javaw.exe –cp .;es_console_process_engine.jar;activation.jar; es_lookandfeel.jar com.everlast.io.ConsoleProcessEngine "Nightly Backup Run"'

(UNIX)

'java –cp es_console_process_engine.jar:activation.jar:es_lookandfeel.jar com.everlast.io.ConsoleProcessEngine "./Console Process Engine"'

If the console program is executed successfully, it will run as a separate process, independent of the Console Process Engine itself. Therefore, situations may arise where the program crashes, or becomes unstable, and the Console Process Engine is an inconsistent state (or vice-versa). If this occurs, it may be required to use a process destroyer of some kind (Windows Task Manager for example) to kill the hanging process.

On Windows systems, a convenience EXE file called 'es_console_process_engine.exe' exists in order to execute the engine without having to execute Java directly. The EXE file reads the INI file supplied as the first parameter ('ES_CONSOLE_PROCESS_ENGINE.INI' if no parameter is supplied). Not only is it easier to execute, the process shows up as the EXE name instead of 'java' in the task manager.

As mentioned earlier, the data read/written to the streams are logged in a file. This is so someone can validate and view the results of an execution. The files are located in a subdirectory called 'logs' under the location where the XML file is. This typically is 'c:\Program Files\EverlastSoftware\ES Console Process Engine\logs'. Each log file will contain a unique number to ensure each execution's results are segregated.

Capturing Input/Output from a Program Automatically

The Console Process Engine has the ability to automatically capture the input being fed into a console program. This makes it easy to record/capture the process to be repeated with any future execution. Of course this can also be accomplished manually, by creating the parent/child relationship for all the output/input values (Figure 1). However, the automatic capture makes it much easier.

The automatic capturing is enabled by default. Any information typed into the console display output window will be fed into the console program and saved upon terminate of the process. This can be disabled via the XML or hiding of the display output window.

Some non-displayable characters may appear strange in the output window (such as backspace, delete, etc.). However, be assured that the real characters are being input to the console program appropriately. If at any point during the capture process a mistake is made, simply modify the settings for the engine configuration (Figure 1) and delete the entries that were captured in the 'Output/Input Values' tree (or remove from the XML if so desired).

Configuring Multiple Engines

Often times, more than one console program will be setup to run. Any number of engines can be configured by simply copying the XML file and renaming it if the launcher is not used.

For example, take the standard 'Console Process Engine.xml' file and copy it to 'Nightly Backup Run.xml'. Open the new file and change the configuration information, and the new task can be run whenever desired.

If utilizing the Windows 'es_console_process_engine.exe' program, an associated INI file must be copied and renamed as well. This new INI filename must be supplied as the first parameter when executing 'es_console_process_engine.exe'.

Examples:

(A nightly backup legacy application)

'es_console_process_engine.exe' "c:\program files\ everlastsoftware\ES Console Process Engine\Nightly Backup Run.xml"

(A Zip compression program)

'es_console_process_engine.exe' "c:\program files\ everlastsoftware\ES Console Process Engine\Zip Compression.xml"

INI File Settings

The INI files are currently used by the Windows exe files and should only be modified by advanced users. The following is an example of the contents of the 'ES_CONSOLE_PROCESS_ENGINE.INI' file:

[Options]

; This should be the full path and filename of the jar file.

JarFilePath=c:\Program Files\EverlastSoftware\ES Console Process Engine\es_console_process_engine.jar;c:\Program Files\EverlastSoftware\ES Console Process Engine\activation.jar;c:\Program Files\EverlastSoftware\ES Console Process Engine\es_lookandfeel.jar

; This should be the full path and file prefix of configuration information

ConfigFilePath=c:\Program Files\EverlastSoftware\ES Console Process Engine\Console Process Engine

; This should be the name of the class to run

JavaClass=com.everlast.io.ConsoleProcessEngine

JVMParameter=-Xmx512M

The 'JarFilePath' should be altered if the jar files are being moved somewhere else (or copies exist somewhere else) besides the installation directory.

'ConfigFilePath' is the XML file to use (minus '.xml' for the real filename). For example, if the real file was 'Nightly Backup Run.xml', the 'ConfigFilePath' would read 'Nightly Backup Run'.

'JavaClass' is the ConsoleProcessEngine and should never change.

'JVMParameter' can be any number of parameters to pass to the Java Virtual Machine on startup. If a particular parameter is wanted, simply insert/modify it here.

Special String Tags

The ES Console Process Engine has the ability to specify special tags in the file prefixes, file extensions, output directories, and output/input values (see Figure 1). This allows one to have great control over where files should go and what they should be named, as well as utilizing environment variables and other special tags for feeding input or parsing output. These special tags are indicated by a value inside two '%' characters. For example, in order to have the file be named the current date, the following would be set in the file prefix field:

%month%-%day%-%year%

Those tags would expand to something like '02-15-2005', '10-03-1995', etc.

Mixing of hard coded Strings and tags is possible. One could do the following in order to look for an output String followed by an endline/newline character (\n):

Backup finished!%line.separator%

If the user name of the logged in user is desired, the following can be done:

%user.name%

The following is a list of all the practical tags that may be used:

guid – Generates a random GUID (Globally Unique Identifier)
year – Four digit year
month – Two digit month
day – Two digit day
hour – Two digit hour
minute – Two digit minute
second – Two digit second
millisecond – Four digit millisecond
longtime – Single timestamp value to the millisecond
user.name – User name of the current user
host.name – The host name of the machine
host.address – The IP address of the host machine
user.home – The home directory for the current user
user.dir – The current user's directory
java.io.tmpdir – The temp directory
launch.text – The text value selected by the user in the launcher
launch.engine – The engine name selected by the user in the launcher
xml.dir – The directory where the engine XML files are being utilized
log.dir – The directory where the main log XML files are being written
line.separator – Endline/newline character.

NOTE: The tags only work for input/output values if the values are not Base 64 encoded.

Base 64 Utility

The Console Process Engine comes with an optional Base 64 encoding utility in order to generate the necessary strings for the XML files (where applicable). Once executed, the utility simply pops up a dialog, asking for a file to read for encoding. Once the entire file contents are read and encoded, another dialog pops up with the encoded string. This string can be copied to the clipboard via 'Ctrl-C', and pasted into the appropriate XML file. On headless systems, the utility can be pointed directly to files using parameters as follows (2 file example):

'java –cp .;es_console_process_engine.jar;activation.jar;es_lookandfeel.jar com.everlast.email.Base64Utility myfile.dat anotherfile.sql'

The output, when executed in this manner, will be displayed through the standard output stream.

Advanced Tips and Troubleshooting

Console applications can behave strangely at times. The Console Process Engine does it's best to control these wild processes, but sometimes they can loose control. Often times it is because of how the program is being run. Some can be run directly (such as 'scp.exe', a secure copy program), while others must be run through a command shell like 'cmd.exe' on Windows (for a program such as 'pkzip', a compression program). If a program appears to not behaving properly when executing it directly on Windows, try utilizing Windows 'command.com' or 'cmd.exe' program. For example:

'cmd.exe /C pkzip.exe' would execute from a command prompt on Windows 2000 without any problems. In order to replicate this functionality through the engine, the following settings would be required in the XML file:

```
<programFileName>cmd.exe</programFileName>
<programWorkingDirectory>c:\compress</programWorkingDirectory>
<parameters>
  <string>/C</string>
  <string>c:\compress\pkzip.exe</string>
</parameters>
```

A program that executes without problems would simply appear as follows (as example):

```
<programFileName>backupprog.exe</programFileName>
<parameters>
  <string>c:\myfile.txt</string>
</parameters>
```

If at any time, error 267 is generated while 'cmd.exe' or 'command.com' is the 'programFileName', make sure the 'programWorkingDirectory' is set to the same directory where the REAL program to execute exists. The example above utilizing 'pkzip.exe' demonstrates the working directory being set to the location where 'pkzip.exe' resides.

Error 2 is an indication that the program file could not be found at the given location for execution.

Error 3 is an indication that a specified directory in the program file path could not be found.

Deadlocks can appear to occur with certain console programs. This is typically a problem because the console program is not flushing the standard streams properly. The C/C++ libraries, for example, have a function called '_flushall()'. After any 'printf' call, or any others that write to standard output, a flush is required in order to read the stream. Otherwise, the Console Process Engine may not be able to function properly. If that is the case, the Console Process Engine will not be able to provide a solution.