

TaskAgent & Windows PowerShell

During the Argos Software users' conference in October 2008, I presented a session on how to extend the standard TaskAgent functionality using scripting with T-SQL and Windows PowerShell. In several blog posts I will try to make the examples and materials from that session available for all Argos customers.

Prerequisites

You will need to install the latest version of the TaskAgent (available for download on our FTP server here: <ftp://ftp.abecas.com>) and Windows PowerShell version 1.0 (available for download here: <http://www.microsoft.com/downloads/details.aspx?familyid=6CCB7E0D-8F1D-4B97-A397-47BCC8BA3806>) in order to be able to use the examples below.

Scenario 1: Automate file and directory maintenance

One common task which I have to do quite often involves maintenance of different files and directories. I wanted to configure a simple TaskAgent job which will monitor a specific file system directory every 15 minutes and will delete files which match certain criteria – in this specific case, delete all files which have been created more than 8 hours ago. Setting up in the TaskAgent a job schedule which triggers the job execution every 15 minutes is very easy. However, configuring a task to delete the necessary files is not straight forward. Initially I tried to do this with a simple batch file, but getting the current system date and time and using it to filter files proved to be quite difficult. After some research I found an easy answer which was using the newest scripting offering from Microsoft – Windows PowerShell, which is a standard part of Windows Server 2008 and is also available for Windows XP and Windows Server 2003. With PowerShell, all I needed to write is a single line of script code to achieve my task goal:

```
dir "D:\Temp" | ? {$_.CreationTime -lt (get-date).AddHours(-8)} | del -recur
```

This single line of code performs the following:

1. Retrieves the list of all files and sub-directories in **D:\Temp**
2. Iterates through the list and checks if the creation date & time of the file/sub-directory is more than 8 hours ago
3. Each file/sub-directory, which matches the criteria above, is deleted and in the case of directories, all sub-directories and their content is deleted as well.

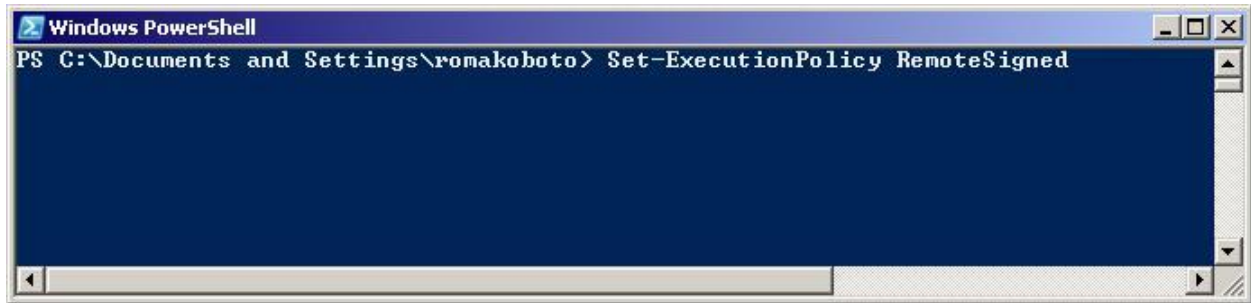
It is beyond the scope of this blog post to provide a full explanation and reference for using the Windows PowerShell – you can refer to the multitude of available online resources for that (one such reference can be found here: <http://channel9.msdn.com/Wiki/WindowsPowerShellWiki/>). However, I do want to show you how to integrate the PowerShell scripts in the TaskAgent to extend its functionality and automate a wide variety of tasks.

Here is what we need to do for our single-line script above:

1. If you have not installed the Windows PowerShell on the TaskAgent server, do so.
2. Open the PowerShell command prompt, enter the following command:

```
Set-ExecutionPolicy RemoteSigned
```

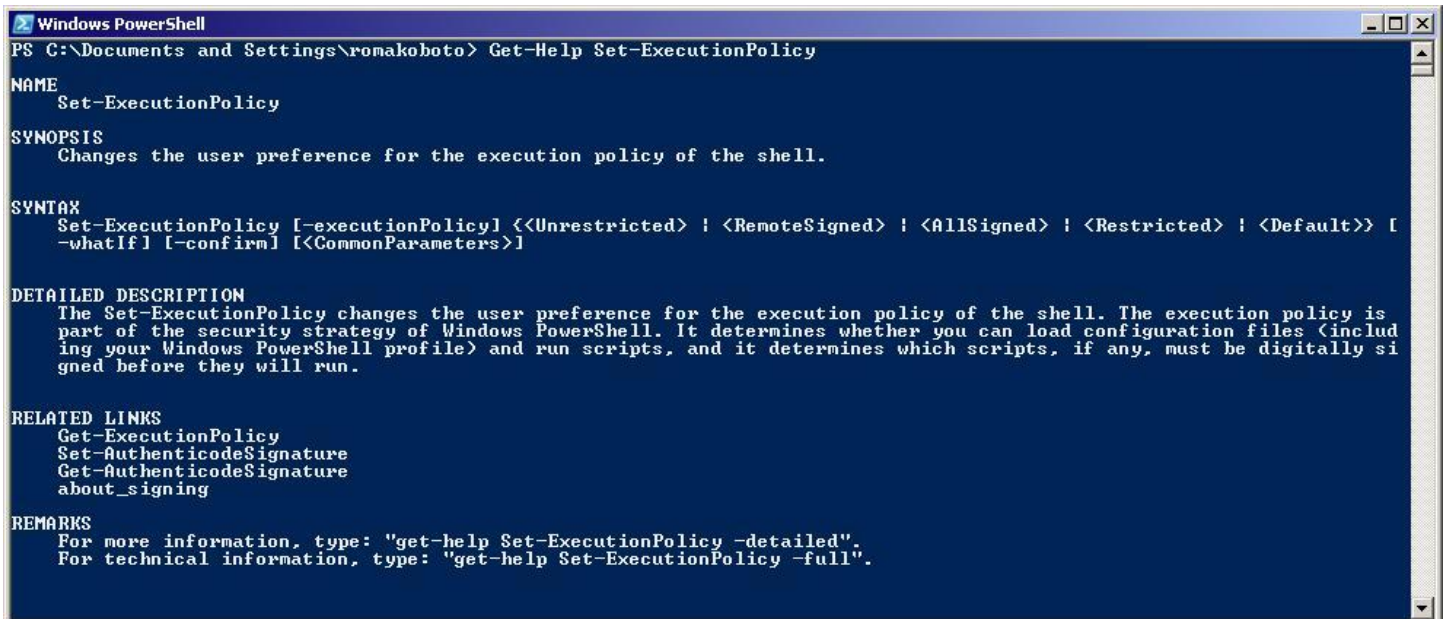
and press "Enter".



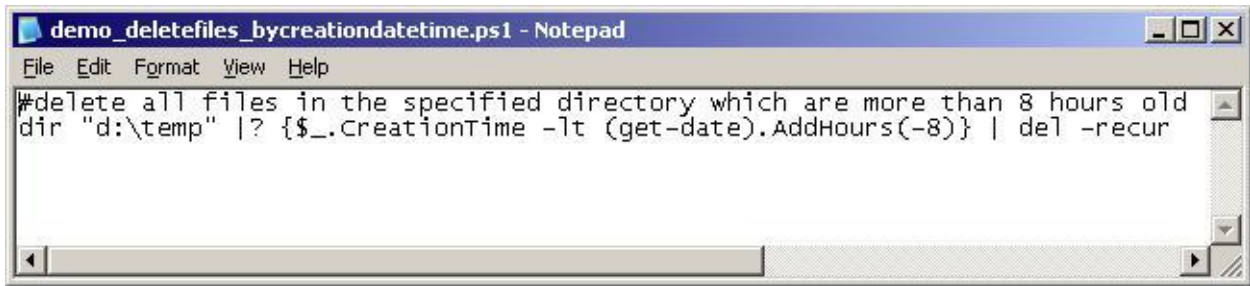
This command will enable the execution of PowerShell script files which are created on the TaskAgent server, because by default this option is disabled to prevent the execution of malicious scripts. (note: Windows PowerShell provides also options to enable only the execution of digitally signed scripts – you will need to have a certificate which can be used to sign your scripts). If you enter the command

```
Get-Help Set-ExecutionPolicy
```

You will get detailed description for the available options.



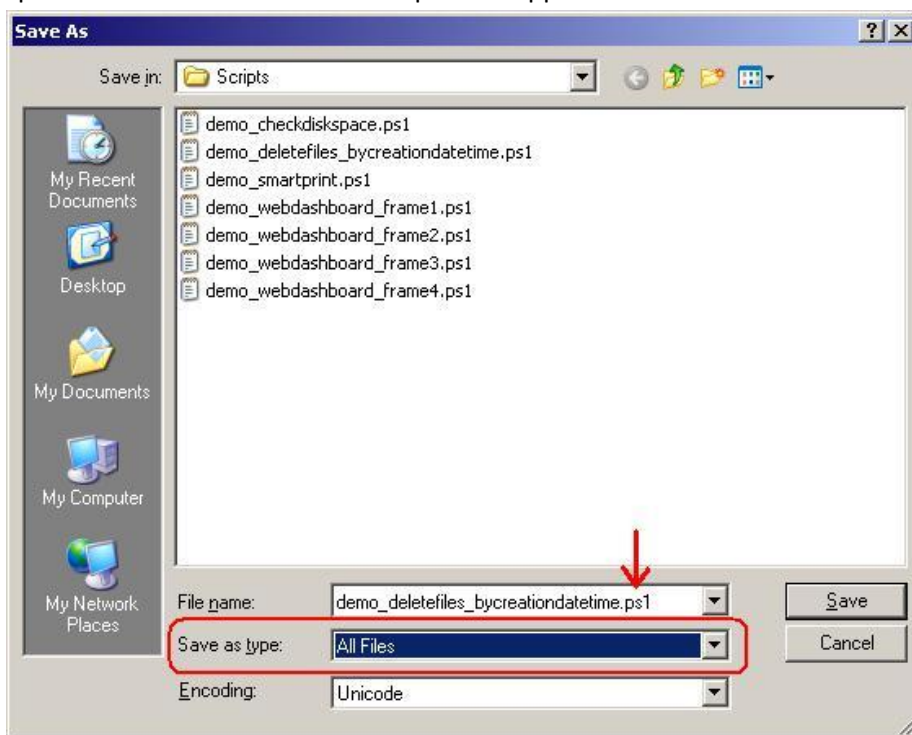
3. Create a simple text file using Notepad, copy and paste the script line above into your text file and save the file in a directory accessible by the TaskAgent using as filename **demo_deletefiles_bycreationdatetime.ps1**.



A screenshot of a Notepad window titled "demo_deletefiles_bycreationdatetime.ps1 - Notepad". The window contains a PowerShell script with the following text:

```
#delete all files in the specified directory which are more than 8 hours old  
dir "d:\temp" |? {$_.CreationTime -lt (get-date).AddHours(-8)} | del -recur
```

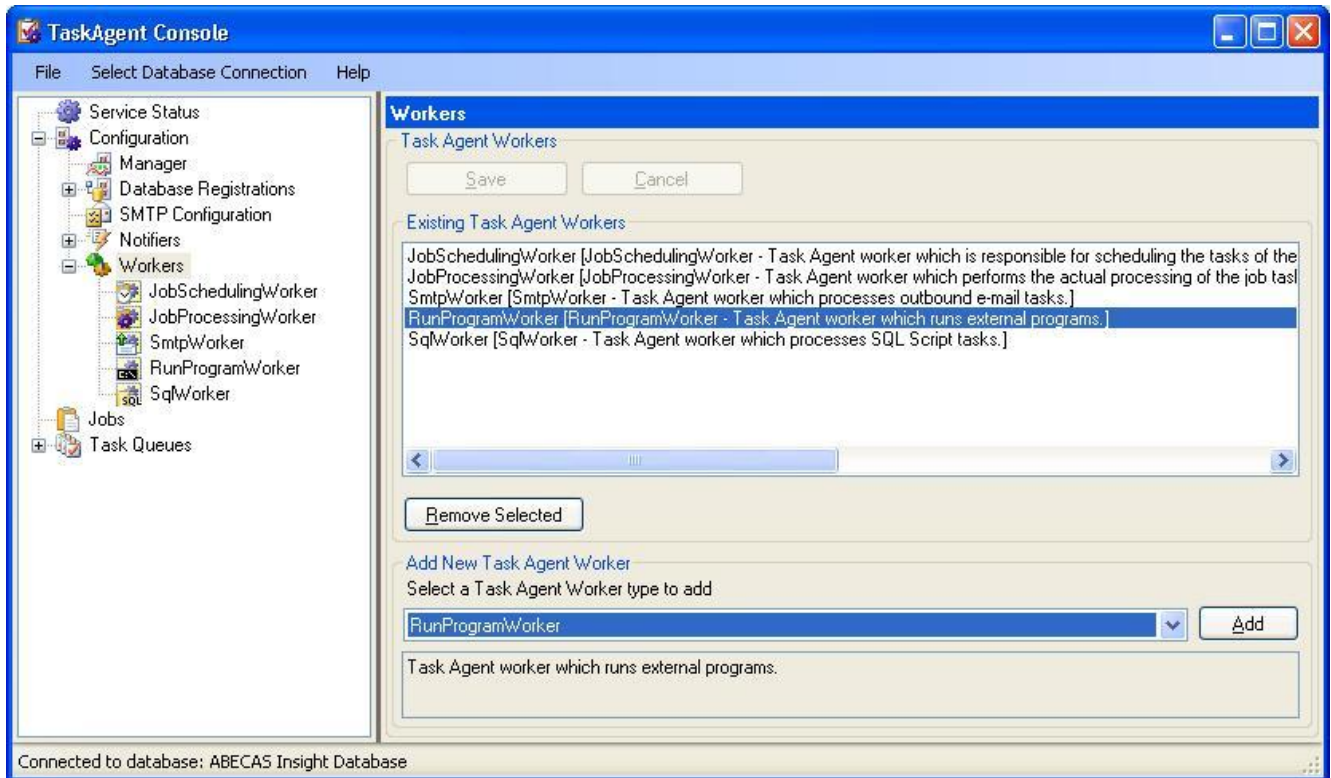
Note the extension **.ps1** – by default, this is the extension used for PowerShell script files. If you are using Notepad to create the script files, when saving the files make sure that you specify in the “Save As Type” the option “All Files” – otherwise Notepad will append the default .TXT extension to your filename.



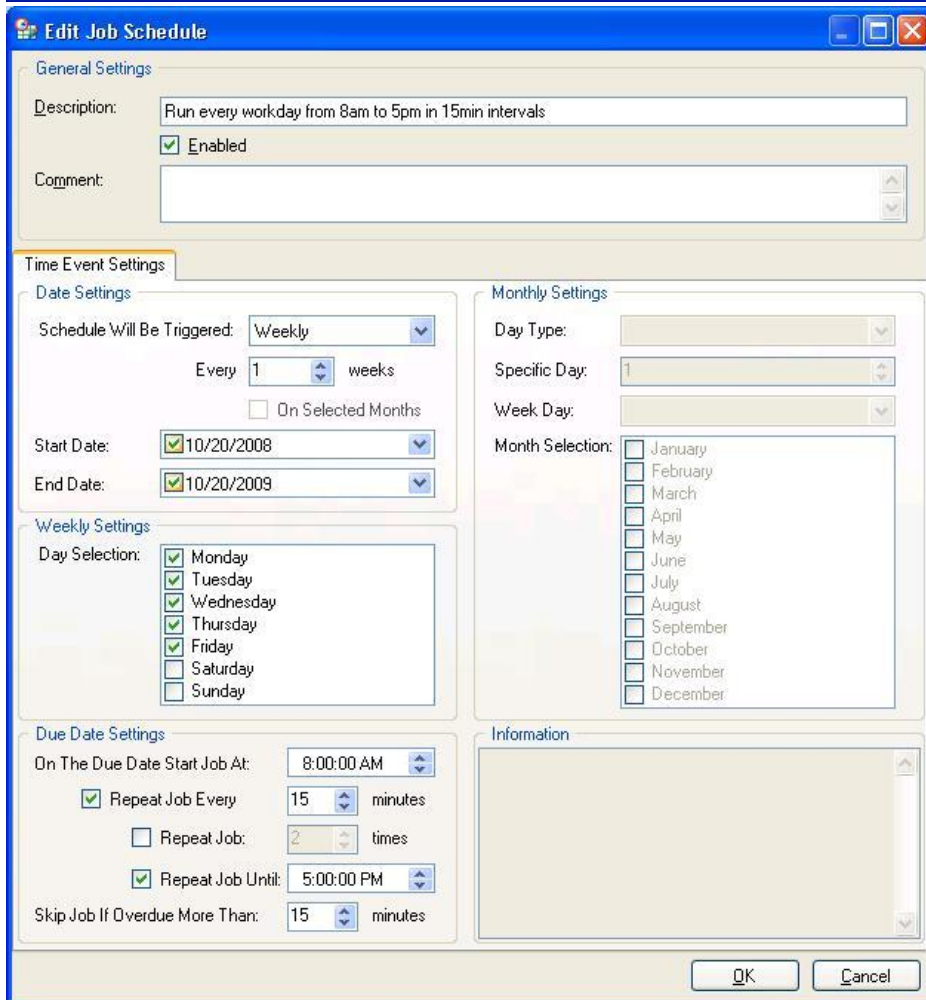
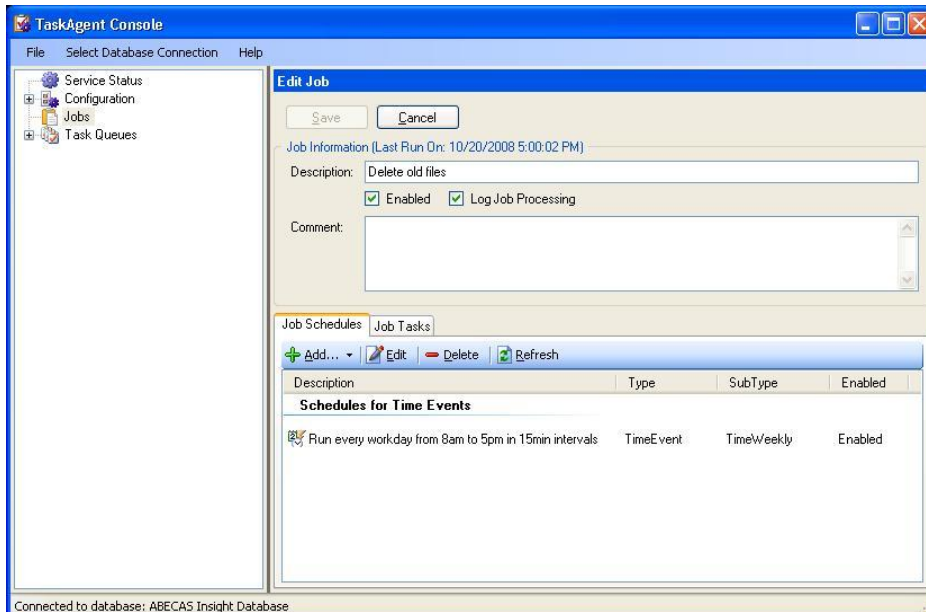
You can use any text editor to create or modify the script files. The only requirement is to save the files as plain text ASCII or Unicode files. My favorite text editor is UltraEdit – it provides among other things color syntax highlighting for the PowerShell scripts which makes it easier to edit them. In the screenshot below, there is an extra line which starts with # - this designates a comment in your PowerShell script file. Such lines will be ignored during the script execution, they are used only to document your script code – very useful for the future script maintenance.

```
#delete all files in the specified directory which are more than 8 hours old  
dir "d:\temp" |? {$_.CreationTime -lt (get-date).AddHours(-8)} | del -recur
```

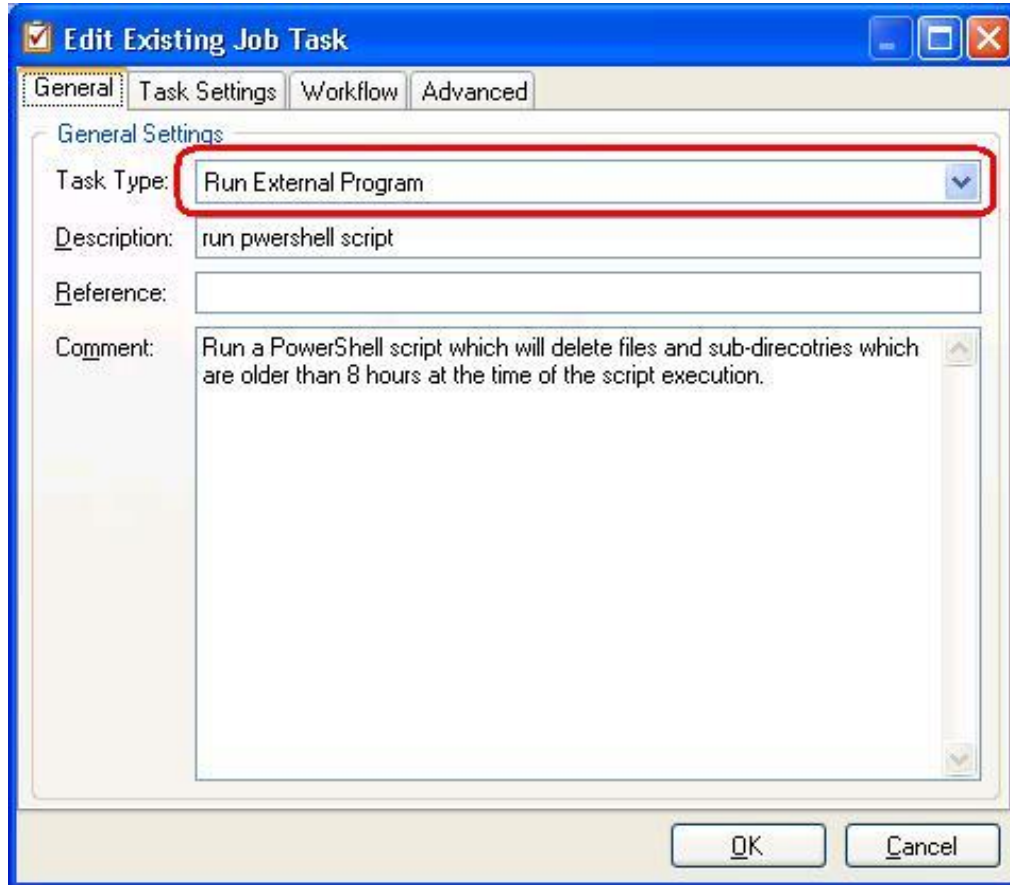
4. Open the TaskAgent console and make sure the RunProgramWorker is registered and enabled – this is the worker which will be used to process the script tasks



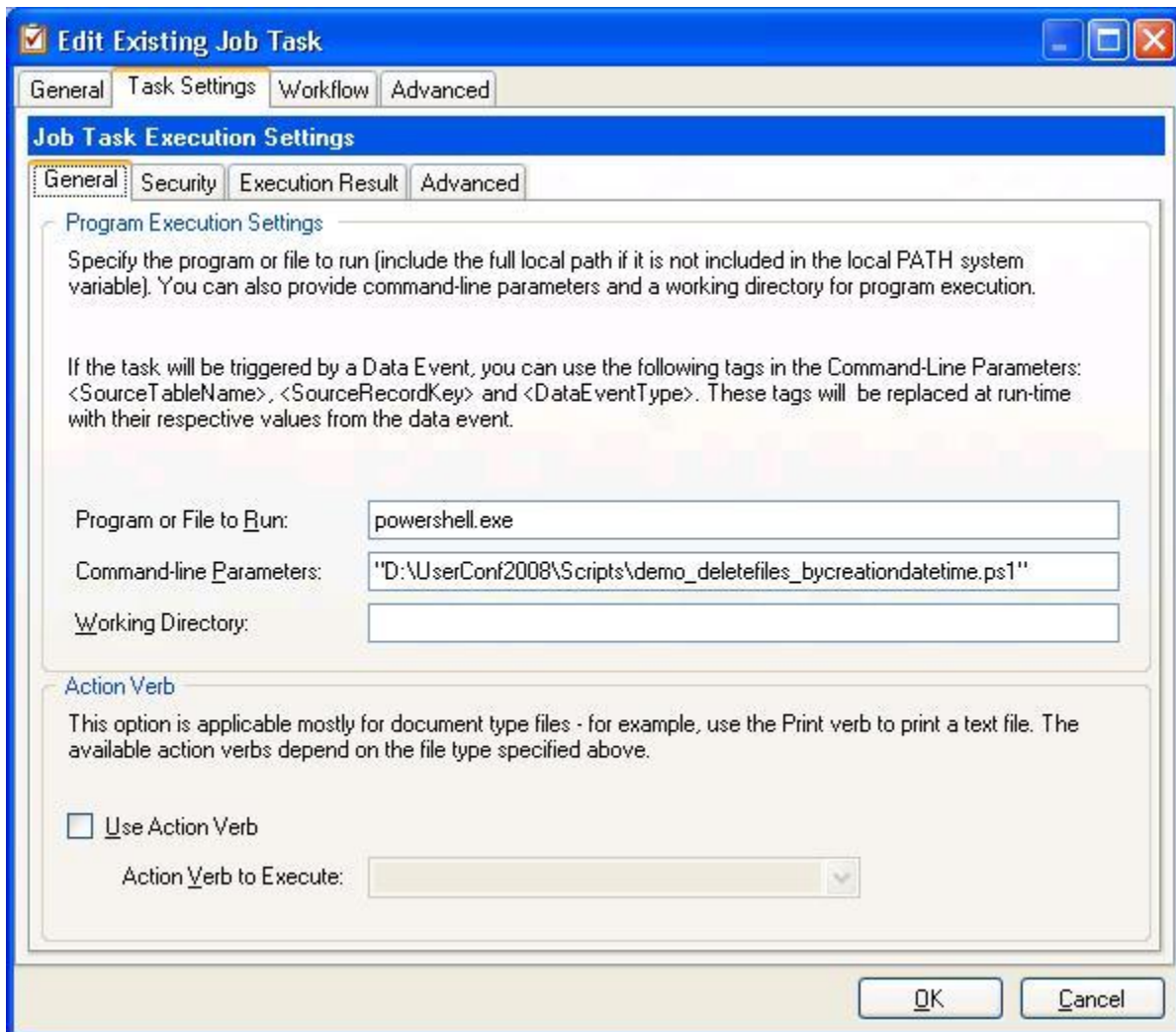
5. In the TaskAgent console, create a new job and add a date & time schedule for it – for this example I used a time schedule which is triggered every 15 minutes, every work day from 8am to 5pm – feel free to set the time schedule as you see fit. (note: to test the script task you do not even need a schedule – just a job with the necessary task, the job can be triggered manually using the option “Run Job Now”)



6. Add a task for the job created above – for the task type, select “Run External Program”

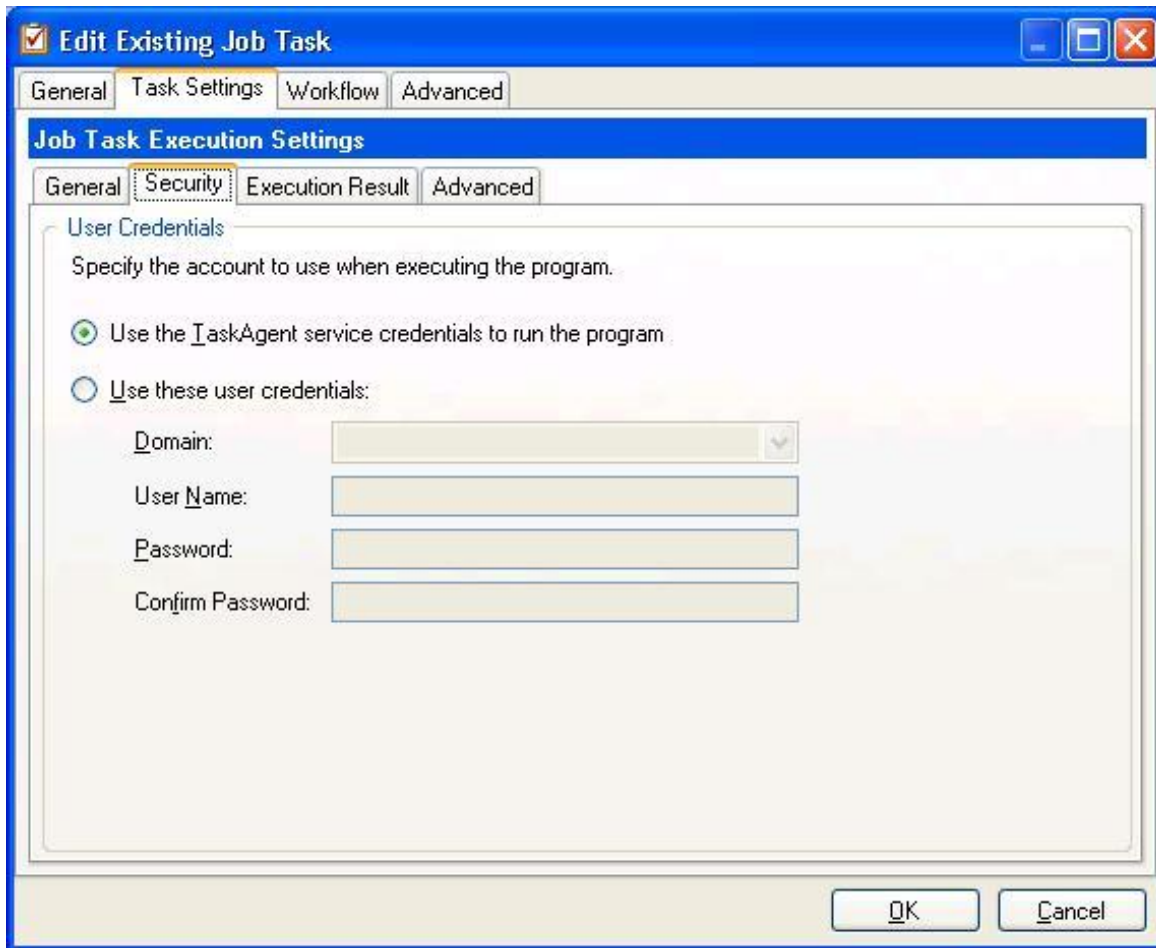


7. On the “Task Settings” – “General” tab, in the “Program or File to Run” field, enter the PowerShell executable “powershell.exe” and in the “Command-line Parameters” field, enter the full path to the PowerShell script file.

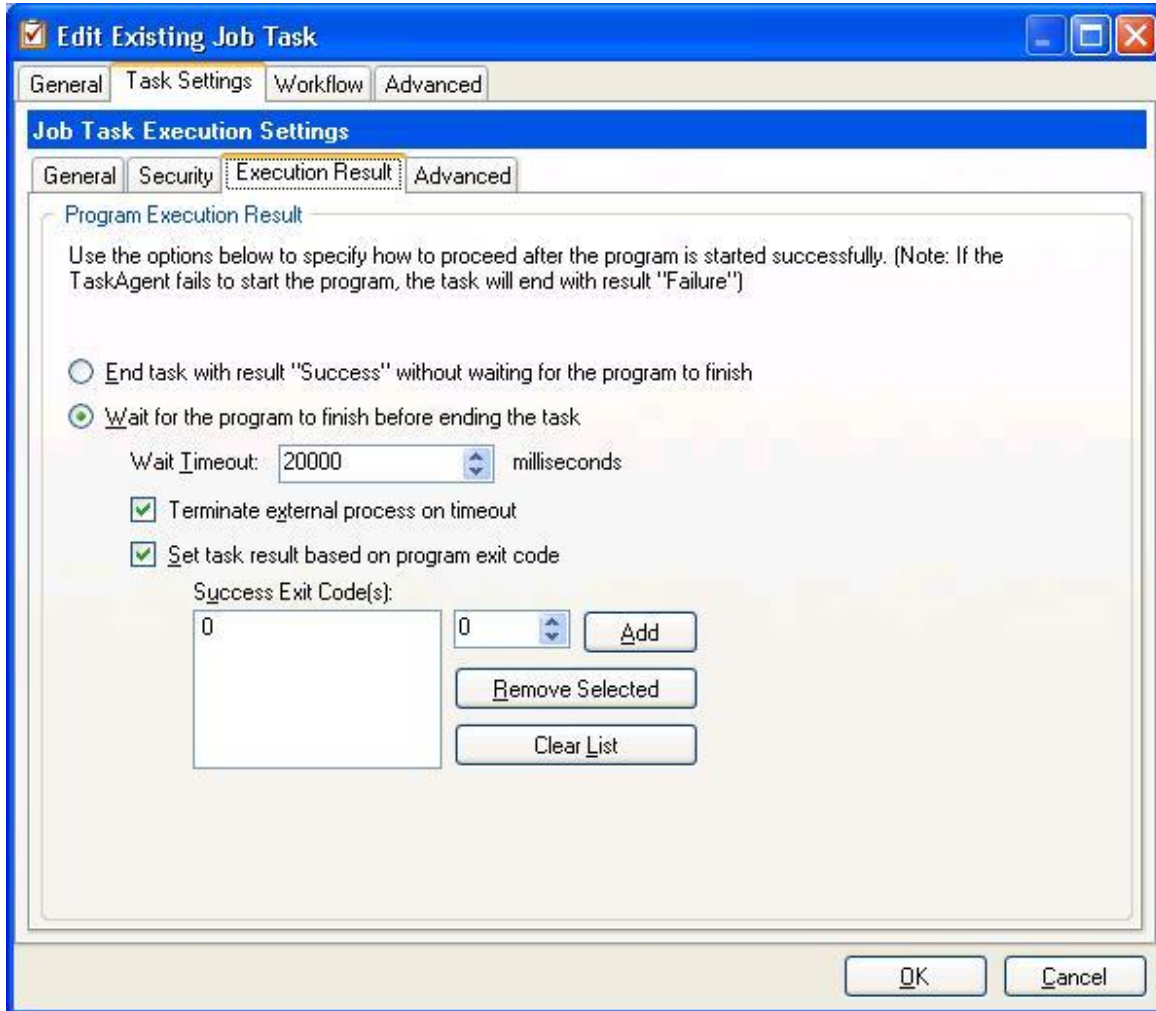


As a rule, I always surround the file paths with double-quotes to avoid errors if the file path contains spaces.

- On the “Task Settings” - “Security” tab, you can provide the Windows credentials which the TaskAgent should use when executing the external program (in our case this is the powershell.exe). By default, the TaskAgent will use the same credentials as the ones which the TaskAgent service is using.



- On the “Task Settings” – “Execution Result” tab, you can specify what the TaskAgent should do after the external program is successfully started. In this specific case, I have configured the task to wait for 20000 milliseconds (20 seconds) for powershell.exe to finish its execution. If the powershell.exe does not finish within the allocated period, the TaskAgent will terminate the external process. In addition, the TaskAgent can examine the exit code of the external process to determine whether the execution was successful or not. In this specific case, I have configured the task to accept only 0 as a successful exit code.



This is all – now save the job and we are ready to test it! You can manually trigger the job by right-clicking on it in the TaskAgent console – Jobs list and selecting the option “Run Job Now”.

Scenario 2 – Check the available disk space and send an e-mail notification

This scenario attempts to automate a standard system administrative task by periodically checking the available disk space on multiple servers and sending e-mail notifications if the free disk space is below a certain limit. To accomplish this I will use again a Windows PowerShell script which will collect the necessary information using Windows Management Instrumentation (WMI) objects and send the notifications using .NET objects. Lets start:

1. We will start by creating the PowerShell script. For your convenience, you can download the source for this demo and use the file **demo_checkdiskspace.ps1**

I will briefly explain what each part of the script is doing:

```
#array with the server names to check
$serversToCheck = "VMSEVER1", "VMSEVER2", "ADSERVER1", "RADSERVER"

#array to store the results
$results = @{}
```

In the first section we just declare the list of computer names which should be checked and an empty array variable to store the results. Remember to change the list of server names to something which applies to your network.

```
#loop through the list of servers...
foreach ($server in $serversToCheck)
{
    #get information about the server logical disk drives (include only the drive name, size and free space
    $diskList = Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName $server |
        Select-Object -Property Name, Size, FreeSpace

    #loop through the server drives and add a few custom properties
    foreach ($disk in $diskList)
    {
        add-member -inputobject $disk -type noteproperty -name Machine `
            -value $server;
        add-member -inputobject $disk -type noteproperty -name PercentFree `
            -value ([long]$disk.FreeSpace/[long]$disk.Size);
        add-member -inputobject $disk -type scriptproperty -name Status `
            -value {if($this.PercentFree -lt 0.20){"CRITICAL"}elseif($this.PercentFree -lt 0.35){"WARNING"}else{"OK"}};

        #add the drive info to the results array
        $results += $disk;
    }
}
}
```

In the second section the script iterates through the list of server names and uses the WMI object Win32_LogicalDisk to collect the disk drives information (marked with red). Since the servers may have multiple drives, the script iterates through the disk drives information of each server and adds to the result information the server name, calculates the percent free disk space and uses the calculated percentage to place the server disk drive one of the following 3 groups: less than 20% - CRITICAL, less than 35% - WARNING and everything else – OK (marked with blue). You can change these percentages to different levels if necessary.

```
#return from the results only data for drives with status different than OK
#format the output in a table, grouped by server name, with sizes in gigabytes
$badDisks = $results | where-object -filterScript {$_.Status -ne "OK"}
```

The third part of the script further filters the results collected so far and leaves only the ones with CRITICAL or WARNING status. You can omit this step if you would like to send in the e-mail a report which includes information for all disk drives, not only the ones which are running low in free disk space.

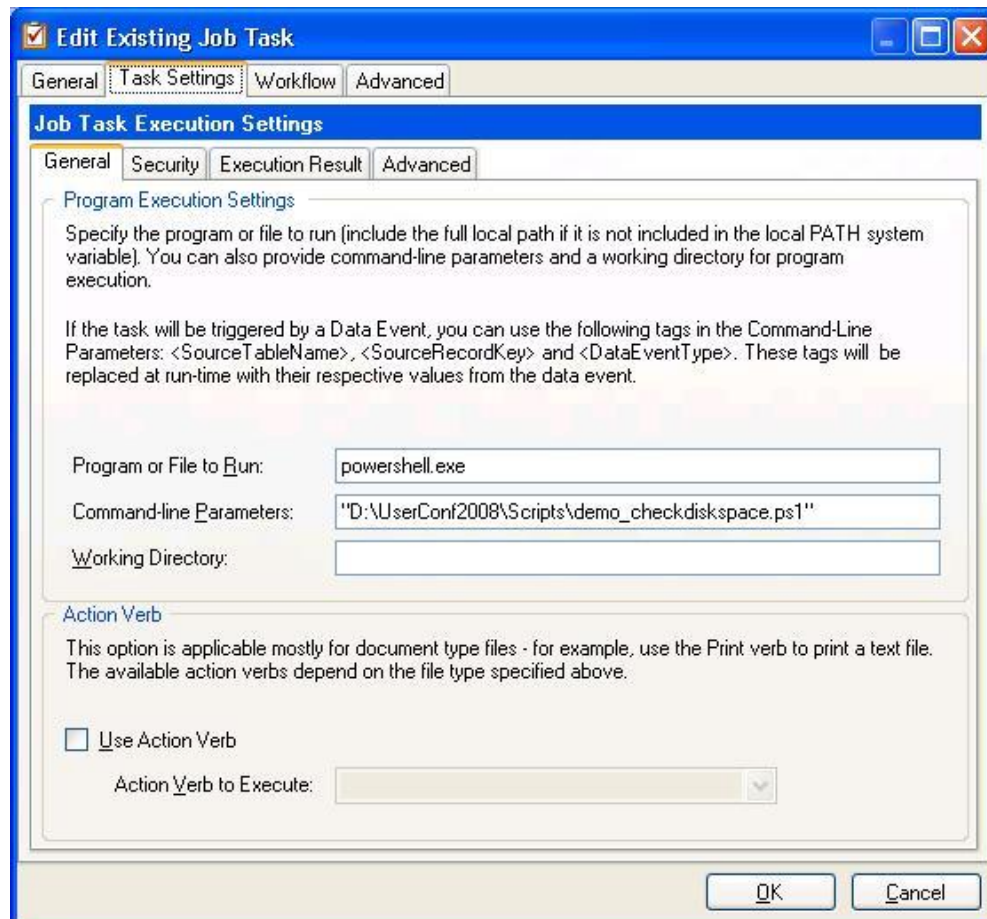
```
#check if there are any disks which are running on low disk space
if ($badDisks)
{
    #format the output (store it in a file)
    $badDisks | format-table -groupBy Machine -property `
    @{label="Drive"; expression={$_Name}}, `
    @{label="Disk Size "; expression={"(0,7:N2) GB" -f ($_.Size /1gb)}}, `
    @{label="Free Space"; expression={"(0,7:N2) GB" -f ($_.FreeSpace /1gb)}}, `
    @{label="Percent Free"; expression={"(0,12:P)" -f $_.PercentFree}}, `
    @{label="Free Status"; expression={$_.Status}} | out-file "d:\results\baddisks.txt" -encoding Unicode

    [string]$msg = get-content -path "d:\results\baddisks.txt" -encoding Unicode

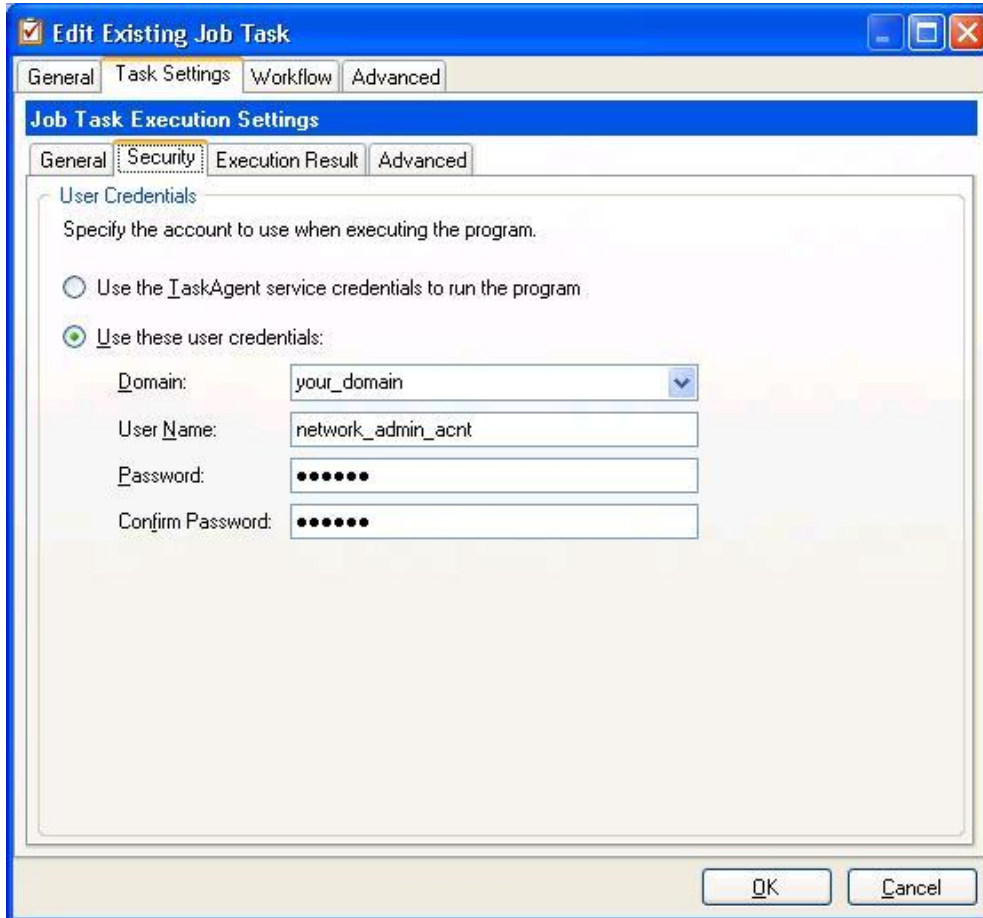
    #send out an alert e-mail
    $smtpClient = new-object system.net.mail.smtpClient
    $smtpClient.host = "smtp.dsl.net"
    $from = "PowerShell Alert <rossent@argossoftware.com>"
    $to = "rossent@gmail.com"
    $title = "Low Disk Drive Space Alert"
    $body = "The following drives are running low on diskpace `n" + $msg
    $smtpClient.Send($from,$to,$title,$body)
}
```

The last section of the script checks if there are any results and formats them in a nice report which is written to a text file (marked with red). The script then uses a .NET object to prepare an e-mail message and send the results in the body of the message (marked with blue). Remember to change the From and To e-mail addresses. If your SMTP server requires authentication, you can also specify the necessary username and password.

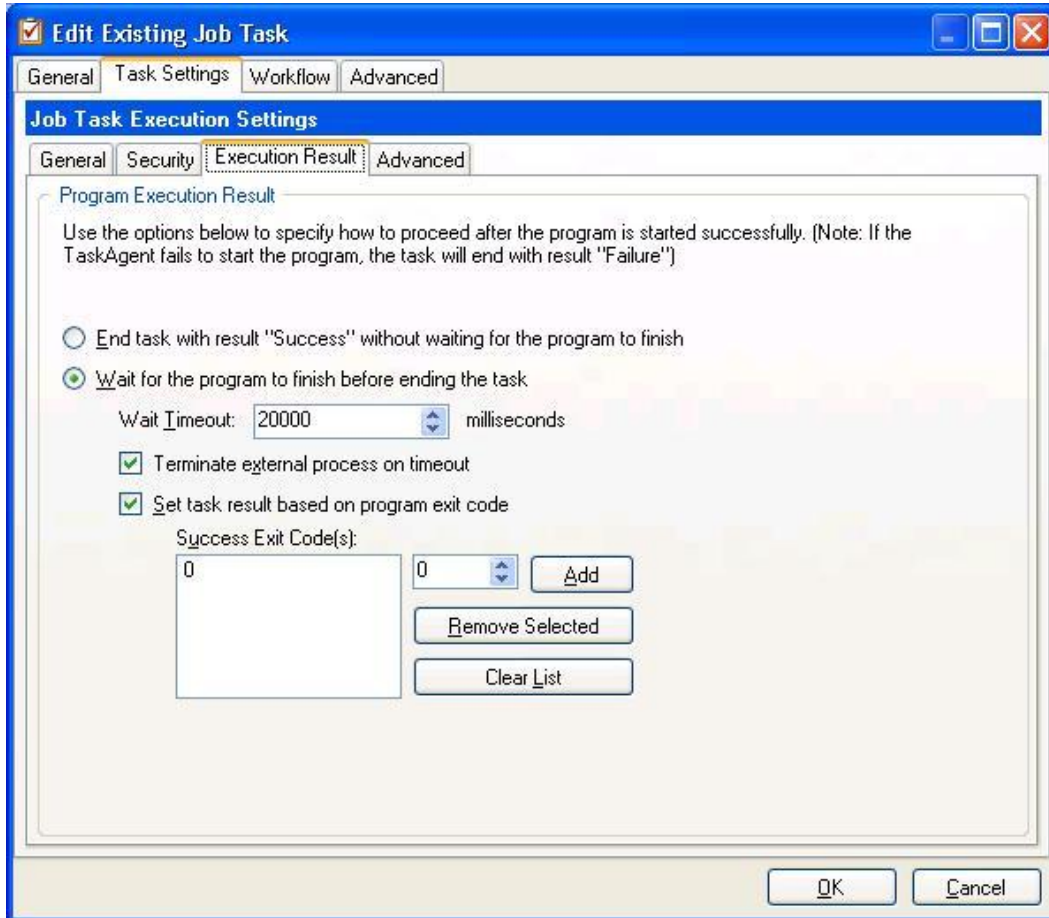
2. Create a TaskAgent job and add a time event for it. (see Scenario 1 for more details how to do this)
3. Add a task for the job which we created above. The task type should be "Run External Program".
4. On the "Task Settings" – "General" tab, in the "Program or File to Run" field, enter the PowerShell executable "powershell.exe" and in the "Command-line Parameters" field, enter the full path to the PowerShell script file.



- On the “Task Settings” - “Security” tab, you can provide the Windows credentials which the TaskAgent should use when executing the external program (in our case this is the powershell.exe). By default, the TaskAgent will use the same credentials as the ones which the TaskAgent service is using. Our PowerShell script is using WMI to interrogate multiple servers for their disk drives – this requires administrative permissions which the TaskAgent service account may not have. If this is the case, on the “Security” tab enter the credential for a Windows account which has the necessary permissions.



- On the “Task Settings” – “Execution Result” tab, you can specify what the TaskAgent should do after the external program is successfully started. In this specific case, I have configured the task to wait for 20000 milliseconds (20 seconds) for powershell.exe to finish its execution. If the powershell.exe does not finish within the allocated period, the TaskAgent will terminate the external process. In addition, the TaskAgent can examine the exit code of the external process to determine whether the execution was successful or not. In this specific case, I have configured the task to accept only 0 as a successful exit code.



Save the job and test it. If everything is correct and some of the servers in your list are running low on free disk space, you should get a report containing information similar to this:

Machine: VMSEVER1				
Drive	Disk Size	Free Space	Percent Free	Free Status
C:	408.36 GB	138.69 GB	33.96 %	WARNING
E:	698.12 GB	45.52 GB	6.52 %	CRITICAL
F:	698.64 GB	199.39 GB	28.54 %	WARNING
Machine: VMSEVER2				
Drive	Disk Size	Free Space	Percent Free	Free Status
C:	272.23 GB	71.61 GB	26.30 %	WARNING
Machine: RADSERVER				
Drive	Disk Size	Free Space	Percent Free	Free Status
E:	14.99 GB	5.04 GB	33.61 %	WARNING

Not bad for a short script – by integrating the TaskAgent with PowerShell, WMI and .NET we can automate a multitude of administrative and business tasks.

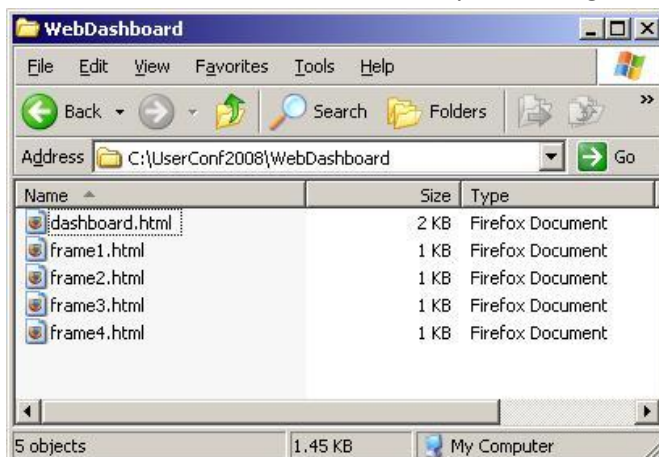
Scenario 3 – Web Dashboard powered by TaskAgent and Windows PowerShell

The third task which I wanted to automate involves the periodic update of an online dashboard which displays a summary of different types of business data – . This time I will use some T-SQL scripting from within the PowerShell script to pull the necessary data from the database. In addition, to make the script usable across different databases, I will be passing the database connection information into it via command-line parameters. Here’s how this is done:

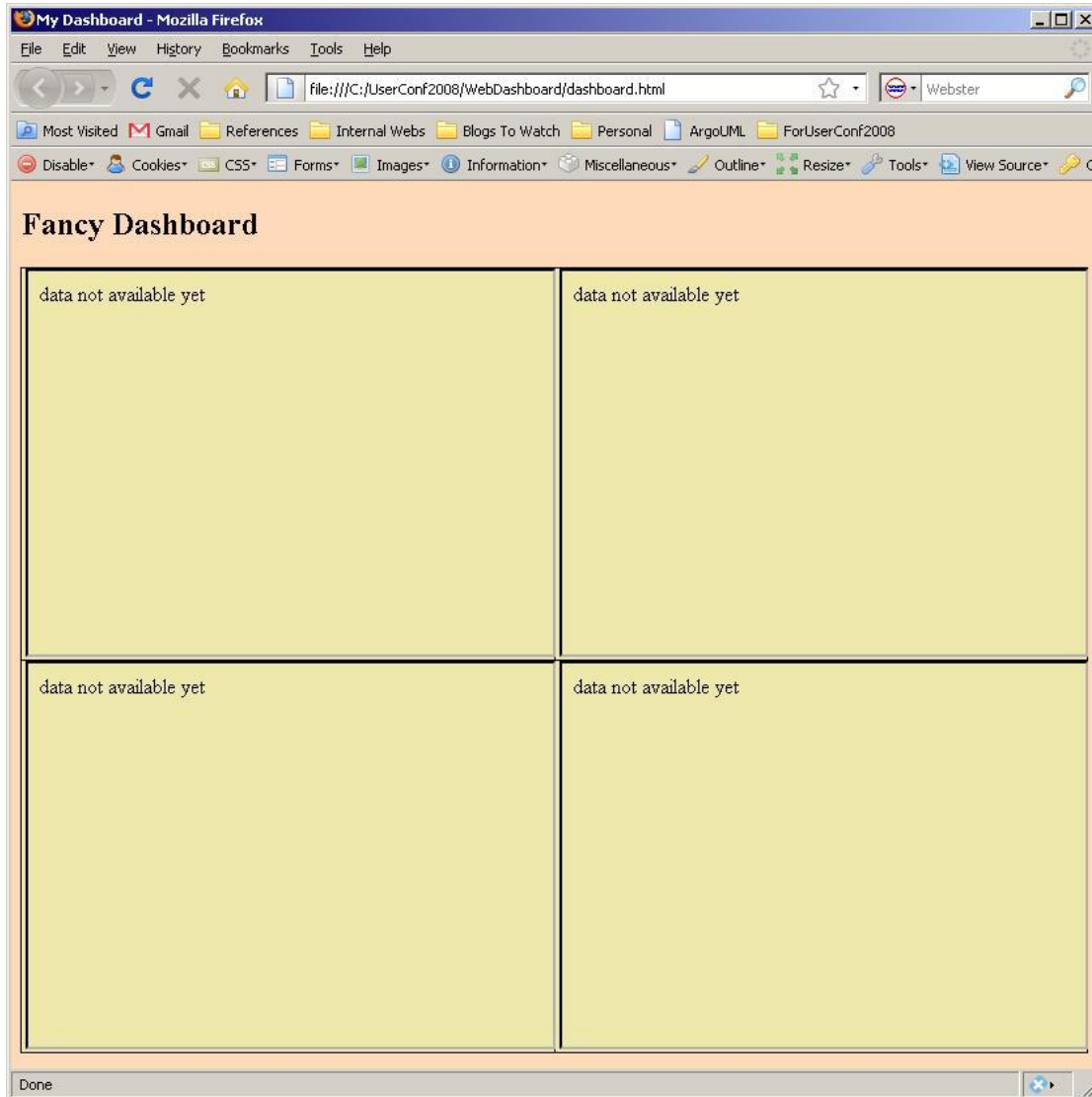
1. We will start by creating the “frame” for our online dashboard. I will use a simple HTML file - **dashboard.html** - which has just a heading text and a table with two rows and two columns – this will give us the four cells where the dashboard data will be presented. Each dashboard cell will be a separate web page represented by an IFRAME element. Here is the content of the **dashboard.html** file:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>My Dashboard</title>
    <style>
      BODY{background-color:PeachPuff;}
      TABLE{border-width: 1px;border-style: solid;border-color: black;border-collapse: collapse;}
      TH{border-width: 1px;padding: 3px;border-style: solid;border-color: black;background-color:Thistle}
      TD{border-width: 1px;padding: 3px;border-style: solid;border-color: black;background-color:PaleGoldenrod}
    </style>
  </head>
  <body>
    <H2>Fancy Dashboard</H2>
    <table width=100% height=100%>
      <tr height=300>
        <td width=50% height=100%>
          <iframe src="frame1.html" width=100% height=100%></iframe>
        </td>
        <td width=50% height=100%>
          <iframe src="frame2.html" width=100% height=100%></iframe>
        </td>
      </tr>
      <tr height=300>
        <td width=50% height=100%>
          <iframe src="frame3.html" width=100% height=100%></iframe>
        </td>
        <td width=50% height=100%>
          <iframe src="frame4.html" width=100% height=100%></iframe>
        </td>
      </tr>
    </table>
  </body>
</html>
```

In addition to the dashboard HTML file, create four files for the different frames – **frame1.html**, **frame2.html**, **frame3.html** and **frame4.html** – the files can be simply blank or contain some default text to be displayed while there is no summary data. These files will later be automatically replaced by the TaskAgent with ones containing our live data. Here is how the directory containing the dashboard HTML files should look:



Here is how the empty dashboard should look while there is no data to display:



2. Create the PowerShell scripts which will do all the processing. We will use a separate script to update each dashboard cell. This will provide the flexibility of updating the different parts of the dashboard at different intervals (using different TaskAgent jobs) or using data from different data sources. The demo will use a single job to process all four tasks, but it is very easy to separate the tasks into different jobs with different schedules. You can use the ready files available from the download section – **demo_webdashboard_frame1.ps1**, **demo_webdashboard_frame2.ps1**, **demo_webdashboard_frame3.ps1**, **demo_webdashboard_frame4.ps1**

I will briefly explain what the purpose is of the different sections of the first script file. The scripts for the other frames are similar – only the SQL query and the frame HTML file which they update are different.

```

$server = "<not specified>"
$database = "<not specified>"

#parse the command-line params
foreach($arg in $args)
{
    if ($arg.Length -lt 3)
    {
        continue;
    }

    $param = $arg.SubString(0, 3)
    switch -regex ($param)
    {
        "/S:" {$server = $arg.SubString(3); continue;}
        "/D:" {$database = $arg.SubString(3); continue;}
    }
}

#check if the required command-line params were provided
if ($server -eq "<not specified>")
{
    throw "SQL Server name is not specified. Use the parameter /S: to provide the SQL Server to use";
}

if ($database -eq "<not specified>")
{
    throw "Database name is not specified. Use the parameter /D: to provide the Database to use";
}

```

The first section of the script handles the input parameters. The script expects two parameters – the SQL server name (specified by **/S:<server_name>**) and the database name (specified by **/D:<database_name>**). It checks if the required parameters are provided and extracts their values. If any of the required parameters is not provided, the script throws an exception with the appropriate message.

```
#set the SQL command
$commandTextHeader = "SELECT TOP 5 " +
"      IsNull(o.TransactionNumber, '<not available>') as TransactionNumber, " +
"      o.TransactionDate, " +
"      IsNull(c.Name, '<not available>') as Customer " +
"FROM    OMTRANSACTIONHEADER as o " +
"      join idmaster as c on c.ROWID = o.R_Customer ORDER BY o.CreationDateTime"

#create the database connection
$conn = New-Object System.Data.SqlClient.SqlConnection
$conn.ConnectionString = "Server = $server; Database = $database; Integrated Security = True"
$conn.Open()

#create the header command object
$cmdHeader = New-Object System.Data.SqlClient.SqlCommand
$cmdHeader.CommandText = $commandTextHeader
$cmdHeader.Connection = $conn

#get the header data
$headerTable = New-Object System.Data.DataTable
$reader = $cmdHeader.ExecuteReader()
$headerTable.Load($reader)

#close the database connection
$conn.Close();

#exit the script with exit code 1 if no header data was found
if ($headerTable.Rows.Count -eq 0)
{
    exit 1;
}
```

The second part of the script prepares a SQL command object and a SQL connection object and executes a SQL query to get the summary data. The section marked with blue is the SQL query which is different in each of the frame scripts. The database connection is created using the information specified by the command-line parameters. It uses integrated security (Windows authentication) instead of requiring username and password.

```
#specify the frame output filename
$outputFileName = ("d:\results\frame1.html")

$headText = "<title>Last 5 Sales Orders</title>"
$headText = $headText + "<style>"
$headText = $headText + "BODY(background-color:yellow;)"
$headText = $headText + "TABLE(border-width: 1px;border-style: solid;border-color: black;border-collapse: collapse;)"
$headText = $headText + "TH(border-width: 1px;padding: 3px;border-style: solid;border-color: black;background-color:thistle)"
$headText = $headText + "TD(border-width: 1px;padding: 3px;border-style: solid;border-color: black;background-color:PaleGoldenrod)"
$headText = $headText + "</style>"

$titleText = "Last 5 Sales Orders"

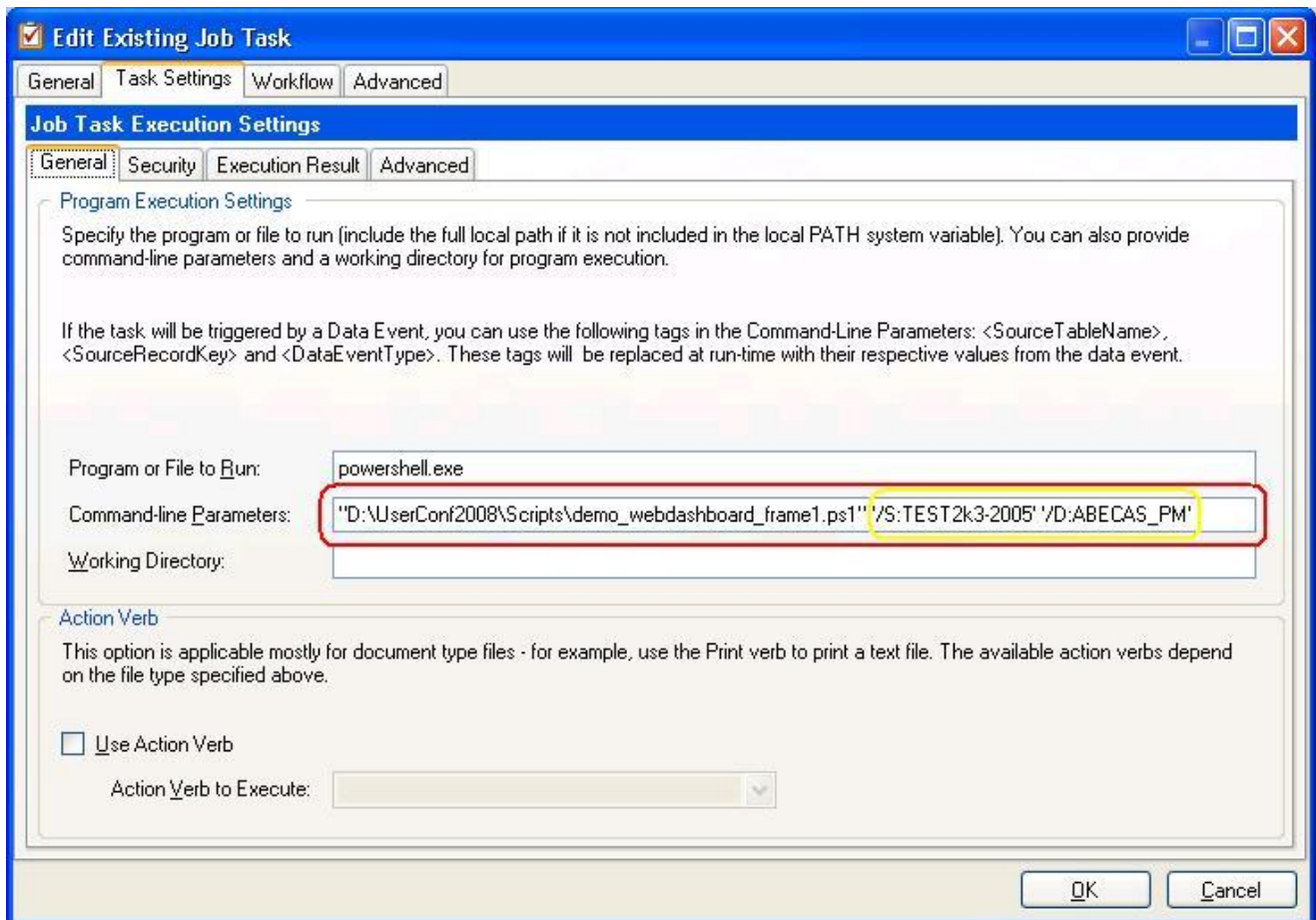
$bodyText = ("<H4>Last 5 Sales Orders</H4><H5>(as of {0:D} {0:t})</H5>" -f (get-date))

$headerTable |
    ConvertTo-HTML -head $a -title $titleText -body $bodyText |
    Out-File $outputFileName
```

The last part of the script formats the retrieved SQL data in an HTML table and writes the results to an HTML file. The section marked with blue is the HTML file name which is different for each of the frame scripts.

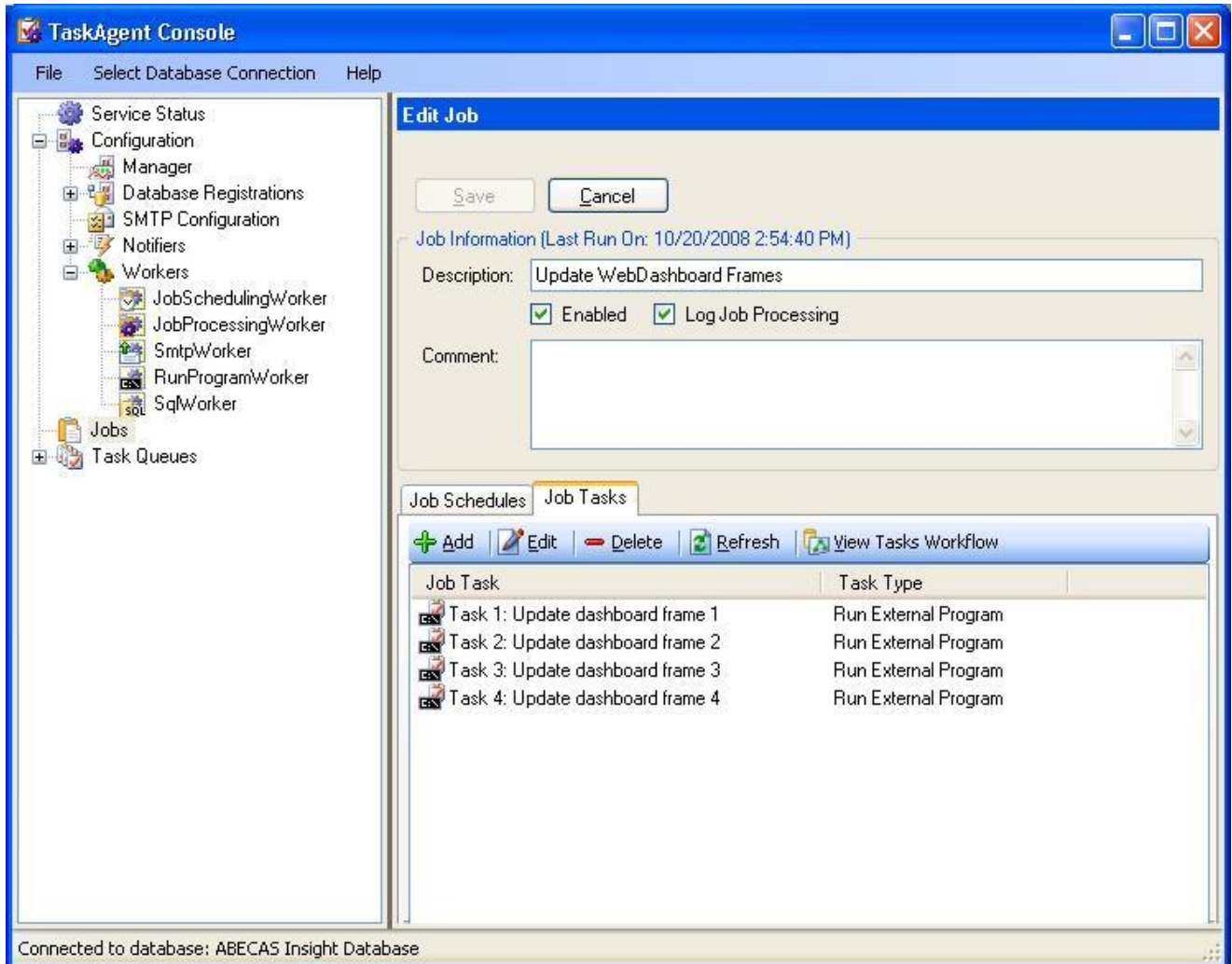
3. In the TaskAgent console, create a job with a time schedule (similar to the way it is done for the prior two scenarios)
4. Create a task for the job created above and select "Run External Program" as the task type.

- On the “Task Settings” – “General” tab, in the “Program or File to Run” field, enter the PowerShell executable “powershell.exe” and in the “Command-line Parameters” field, enter the full path to the PowerShell script file. The difference here is that in addition to the script file, we provide as part of the command-line parameters the database connection information which should be used by the PowerShell script. In this particular case, we are providing only the SQL server name and the database – the PowerShell will use Windows authentication to connect to the database. This requires that the Windows credentials which will be used to execute the task have a corresponding SQL login on the specified SQL server. I chose to go this route to avoid entering in plain text the SQL server username and password. I recommend that you use this approach as well because it does not expose any secure information.

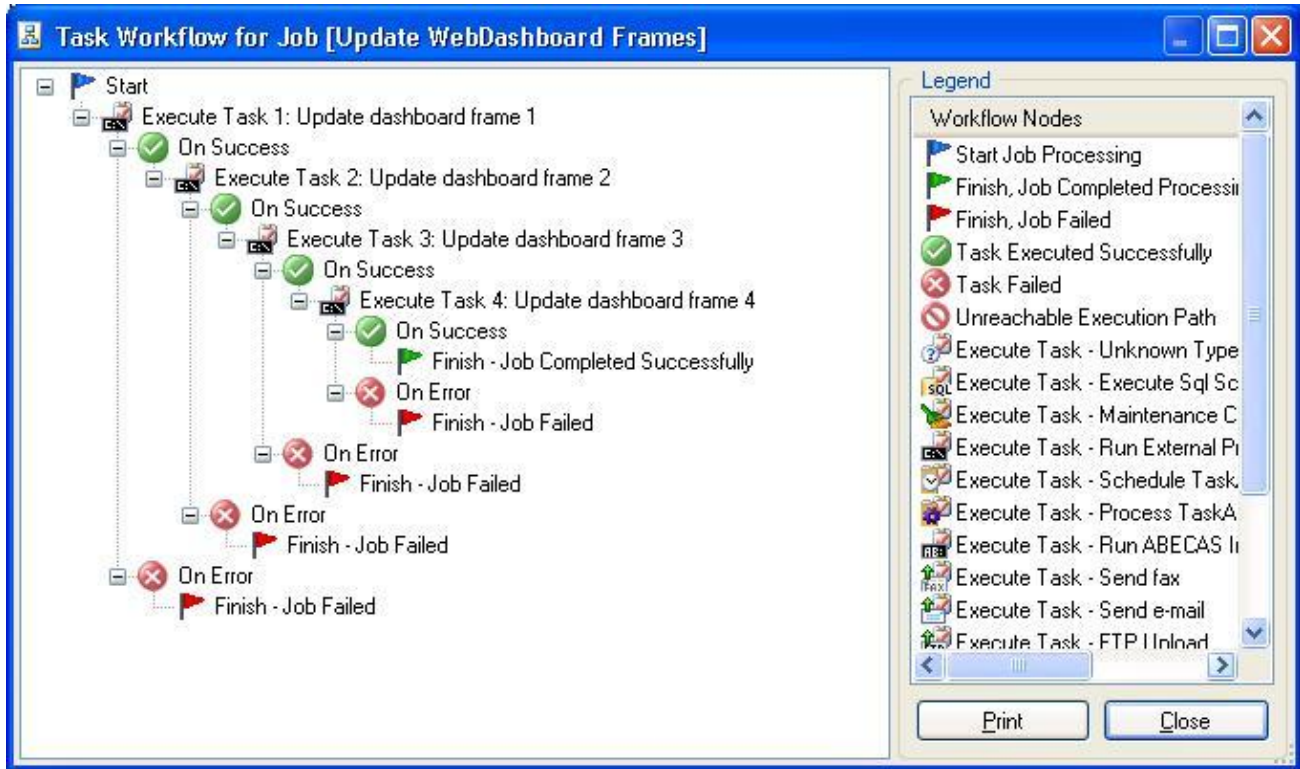


Note that the script input parameters are surrounded by single-quotes.

6. On the “Security” tab provide the necessary Windows account credentials (which also have a corresponding SQL login account) if the TaskAgent service account does not provide them.
7. On the “Execution Result” tab configure what the TaskAgent should do after the external program has been started.
8. Repeat the steps 4 through 7 three more times – the only difference each time will be the PowerShell script file which you specify in the Command-line parameters field. The end result should be a job with four tasks – each task executing one of the four PowerShell scripts which we created in Step 2.



Do not forget to configure the job task workflow – use the “Workflow” tab to specify how the tasks should be executed. The end result should be similar to this:



Save the job and give it a test run. Depending on your TaskAgent configuration it may take a minute or two to complete the execution of all four tasks. At the end, the updated web dashboard will look similar to this.

The screenshot shows a Mozilla Firefox browser window titled 'My Dashboard - Mozilla Firefox'. The address bar shows the file path: file:///D:/UserConf2008/WebDashboard/dashboard.html. The dashboard is titled 'Fancy Dashboard' and contains four data tables arranged in a 2x2 grid.

Last 5 Sales Orders

(as of Monday, October 20, 2008 2:55 PM)

TransactionNumber	TransactionDate	Customer
123456	8/8/2000 12:00:00 AM	SEARS BRAND CENTRAL
23456	8/8/2000 12:00:00 AM	KMART DISC. STORE FRESNO
<not available>	8/8/2000 12:00:00 AM	SEARS BRAND CENTRAL
<not available>	8/8/2000 12:00:00 AM	KMART DISC. STORE FRESNO
25689	8/9/2000 12:00:00 AM	WALMART INC.

Top 5 Customers

(as of Monday, October 20, 2008 2:55 PM)

Customer	SalesOrders
ABC Company	89
Mauna Loa Nut Corp.	72
Driscoll Strawberry	45
Flor Expo	14
KMART DISC. STORE FRESNO	13

Top 5 Sales Reps

(as of Monday, October 20, 2008 2:56 PM)

SalesPerson	SalesOrders
Harvey Clement	70
Driver One	31
Texaco	30
Master Payee	26
A-1 Auto Transport	19

Sales Orders By Quarter

(as of Monday, October 20, 2008 2:56 PM)

Quarter	SalesOrders
4	112
2	106
3	95
1	80

Depending on the job schedule, the TaskAgent will periodically update the dashboard with the latest information from your database. You can even configure the different tasks to pull information from different databases – this way the dashboard will be presenting on a single page summary information from multiple data sources. By modifying the SQL queries you can present different summaries. You can even add new “cells” to the dashboard to expand the presented information. The current style of the dashboard is not perfect and requires the attention of a web designer but I hope that at least its business value is clear.

Online Resources

Below is the list of some of the available online resources which I have used.

- Windows PowerShell
 - <http://www.microsoft.com/windowsserver2003/technologies/management/powershell/default.mspx>
 - <http://www.microsoft.com/technet/scriptcenter/topics/winps/manual/start.mspx>
 - <http://channel9.msdn.com/wiki/windowspowershellquickstart/>
- Batch Files
 - <http://commandwindows.com> (Great One!)
- SQL Scripts
 - <http://msdn.microsoft.com/en-us/library/bb510741.aspx>