Integrating with Automic Engine (UC4) by Broadcom

Derek Pascarella
Updated a few seconds ago

Unfollow

Applies To: Ayehu NG

Description

Broadcom's **Automic Engine** is an automation engine for handling various forms of batch job processing, along with other features. While many find **Automic** to be a useful tool, it is very limited in its ability to touch all of the external systems that **Ayehu NG** can. As a result, many users of **Automic** find the need to integrate into another more powerful, robust, and feature-rich tool, like **Ayehu NG**. This tutorial details the steps necessary to communicate with Broadcom's **Automic Engine** REST API to query information on batch jobs, as well as handle tasks such as initiating restarts of those jobs.

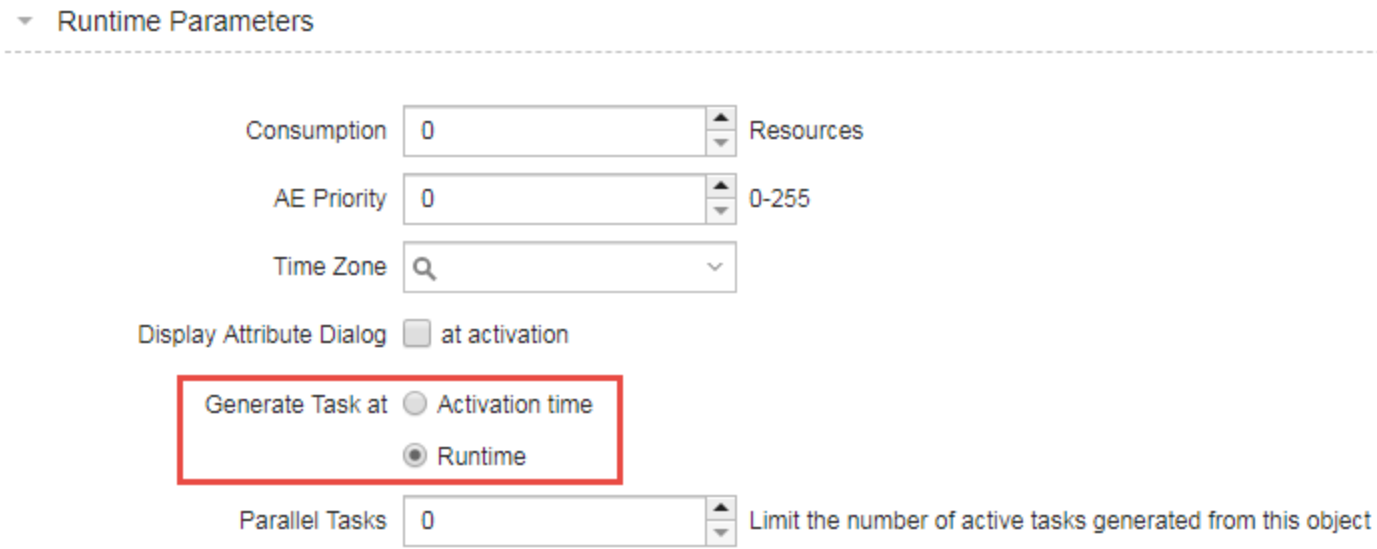
Pre-Requisites

The following are required in order to integrate **Ayehu NG** and **Automic Engine**:

- Ayehu NG 1.3.x +
- Automic Engine
- Knowledge of the **Automic** REST API (http://techdocs.broadcom.com/content/broadcom/techdocs/us/en/ca-enterprise-software/intelligent-automation/SNSC/2-2-1/AE_REST_API_GeneralInfo.html)
- The **Ayehu NG** workflow attached to this article for reference (**Automic - Restart Failed Jobs.xml**)

Automic Preparation

In order for batch jobs to be restarted via the **Automic** API, a setting must be changed in **Automic** to generate tasks at runtime rather than at activation time. See the screenshot below:



Understanding the API

The **Automic** API can be access at its endpoint URL http://automic.mydomain:8088/ae/api/v1/100/, replacing the placeholder "automic.mydomain" with your **Automic** instance's IP address or FQDN.

The API has an **executions** endpoint that returns a list of batch job executions that meet search criteria, which are passed as URL parameters. For example, the URL for finding the five (5) most recent failed executions of a job named "MY_TASK" looks like this:

`http://automic.mydomain:8088/ae/api/v1/100/executions?name=MY_TASK&status=1800,1820&max_results=5`

For the sake of this integration tutorial, we will be using this URL schema in order to find failed batch job executions in order to initiate a restart.

Other parameters and endpoints can be found in Broadcom's **Automic** REST API documentaion (http://techdocs.broadcom.com/content/broadcom/techdocs/us/en/ca-enterprise-software/intelligent-automation/SNSC/2-2-1/AE_REST_API_GeneralInfo.html).

Understanding the Workflow

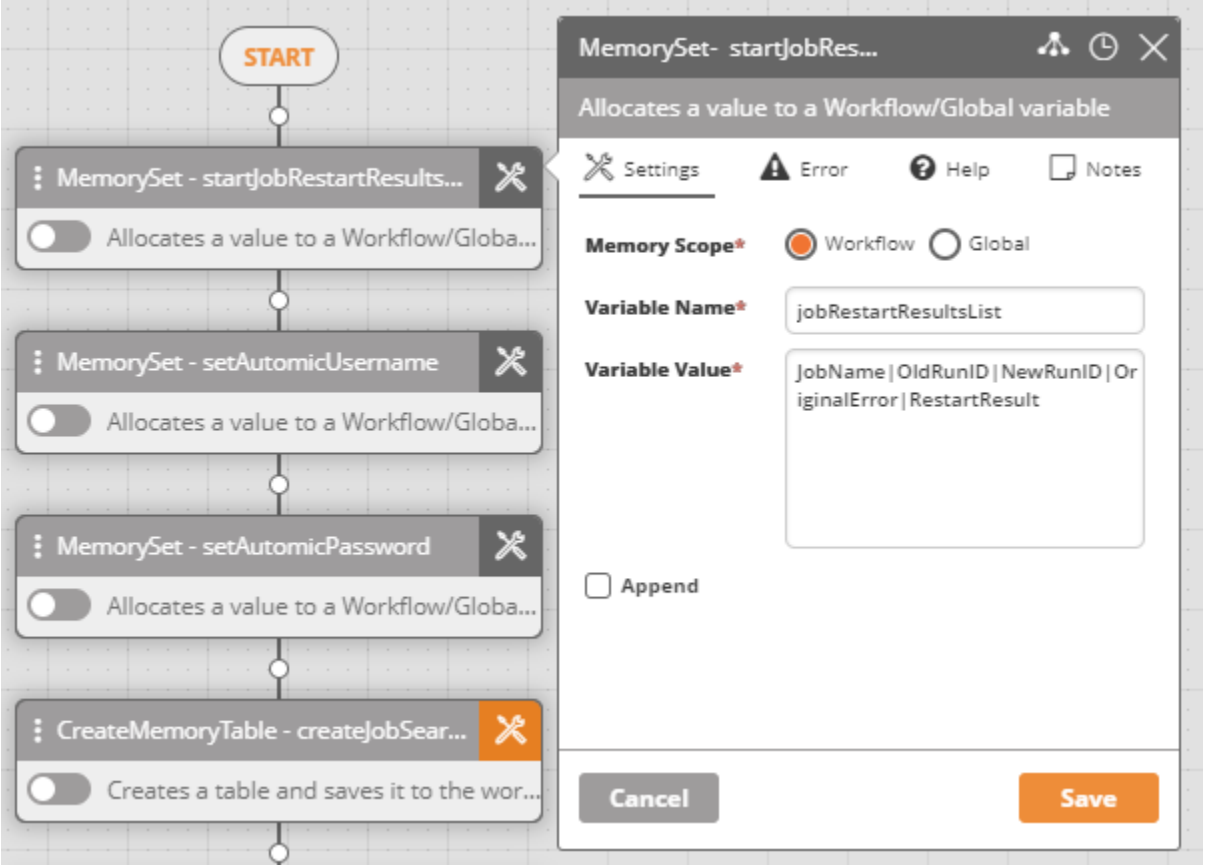
In our example workflow, we will be taking the following steps to find and restart failed batch jobs on **Automic Engine**:

1. Store a list of **Automic** batch job names to be checked.
2. Query the **Automic** API for the most recent failed execution of each batch job.
3. Initiate a restart of each failed job.
4. Write a report in an Excel spreadsheet detailing the failed batch jobs and whether or not they were successfully restarted, including job names, original and new run IDs, and detailed error messages.

Creating the Workflow

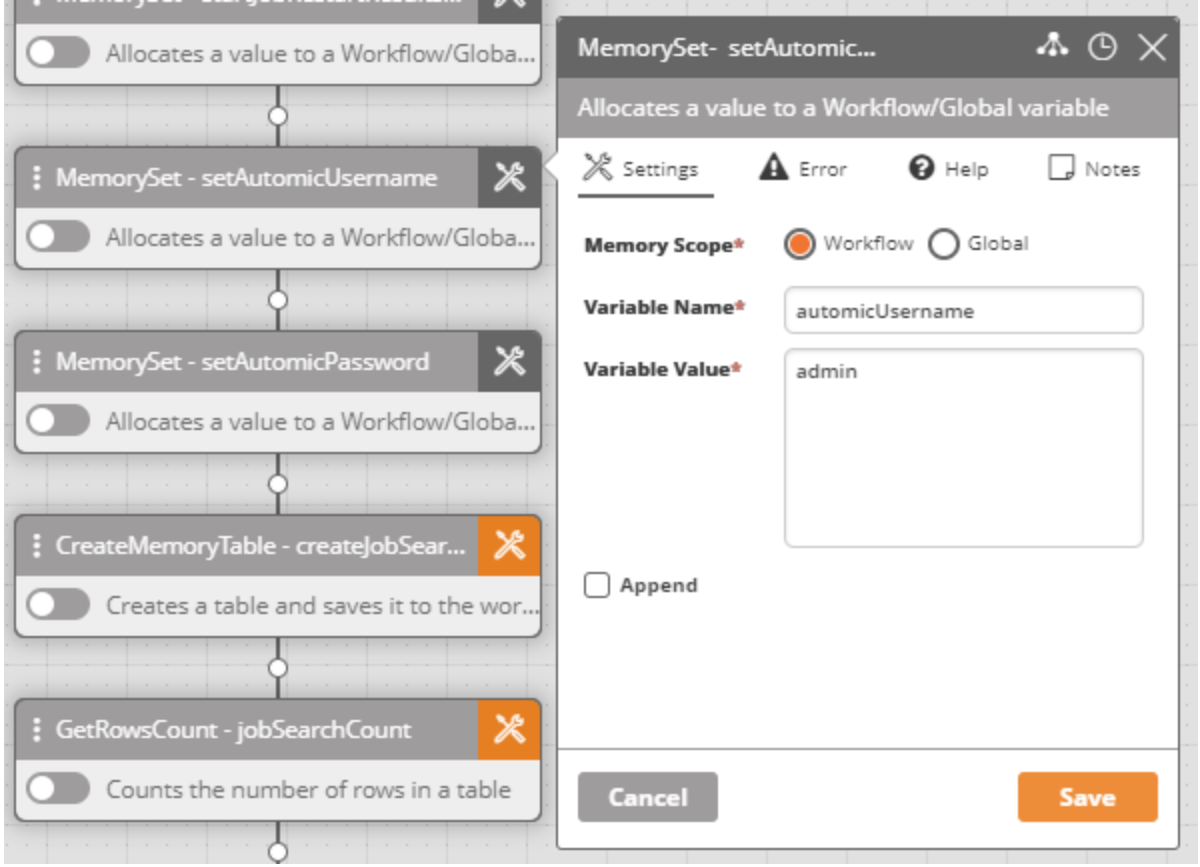
While reading this section of the tutorial, it is advised that you both download and import the workflow attached to this article (**Automic - Restart Failed Jobs.xml**) in order to follow along activity by activity.

In the first activity, we use a **MemorySet** to begin creating the final result table that will be used to generate the Excel spreadsheet containing the detailed report of our failed batch job restart attempts.

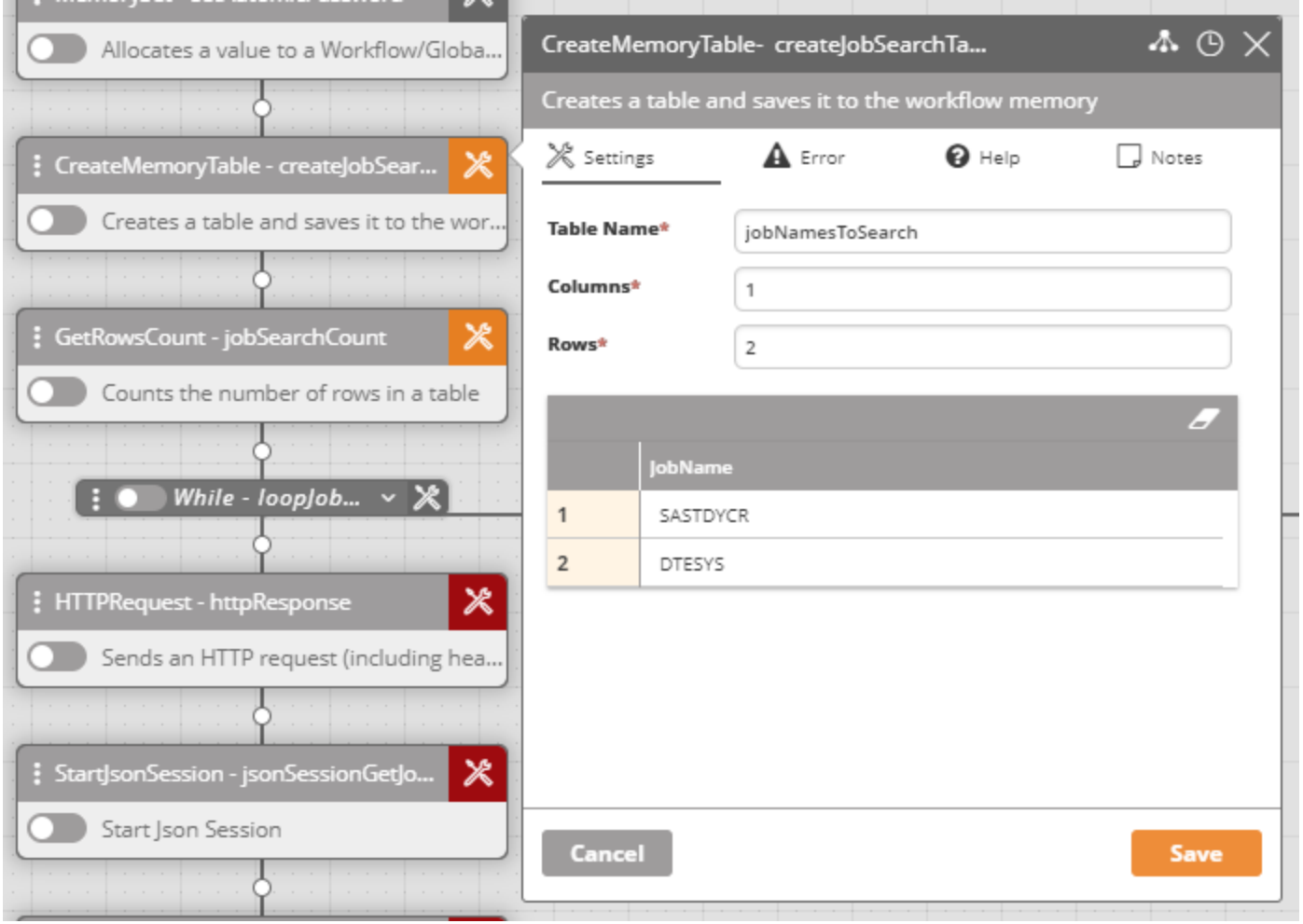


In the screenshot above, you can see each column name from our report separated by the delimiting pipe-character (**|**). Those columns are **JobName**, **OldRunID**, **NewRunID**, **OriginalError**, and **RestartResult**.

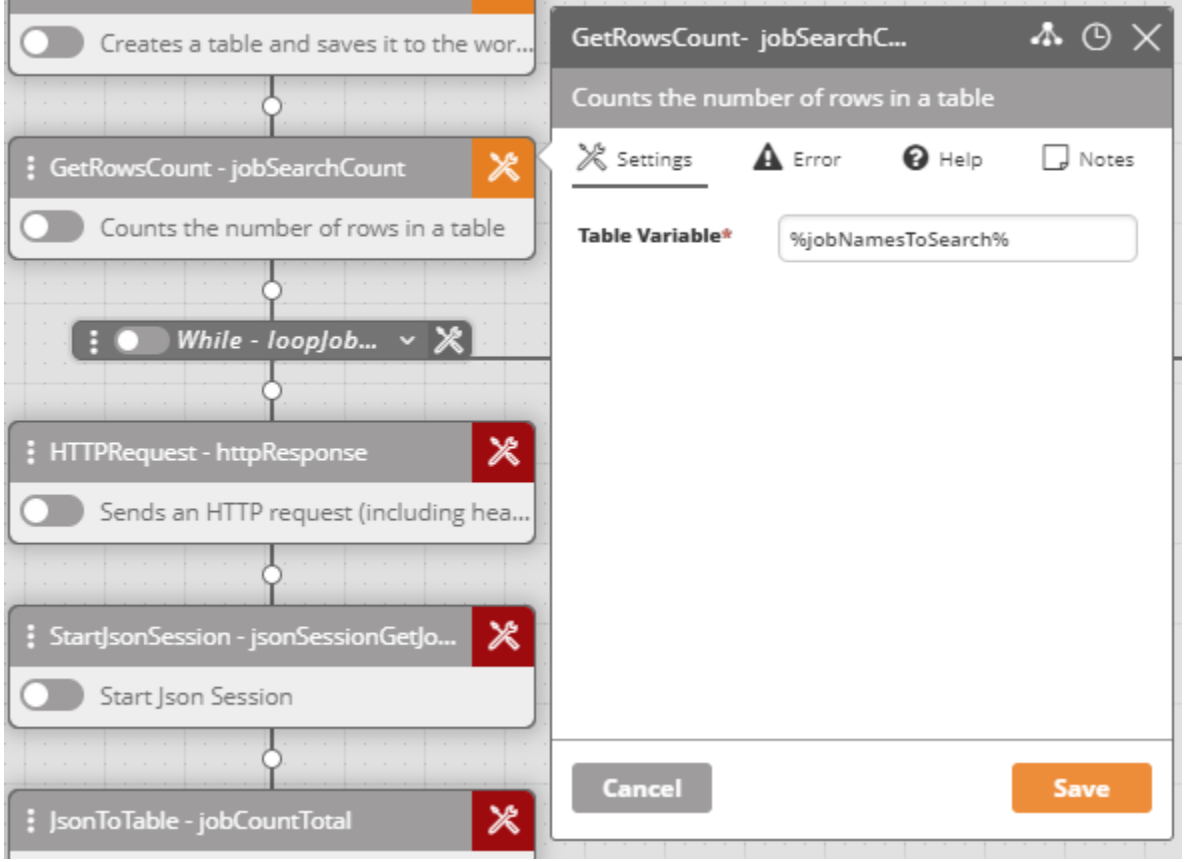
Our next step is to store the username and password being used to authenticate our API session with **Automic**. This is done with the next two **MemorySet** activities, as seen below.



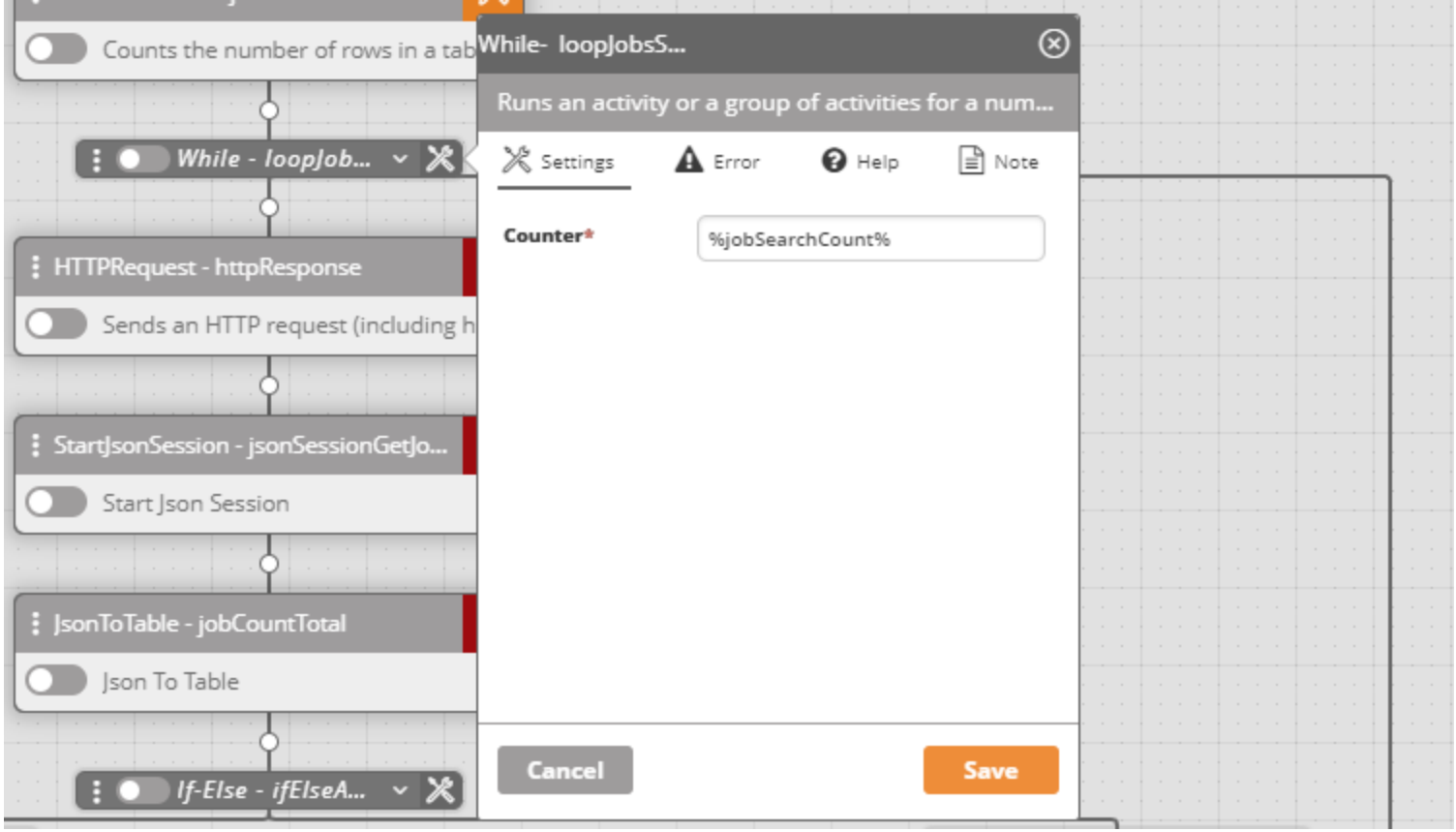
Next, we create a table using the **CreateMemoryTable** activity containing the names of the batch jobs on **Automic** for which we want to find failed executions, as seen in the screenshot below.



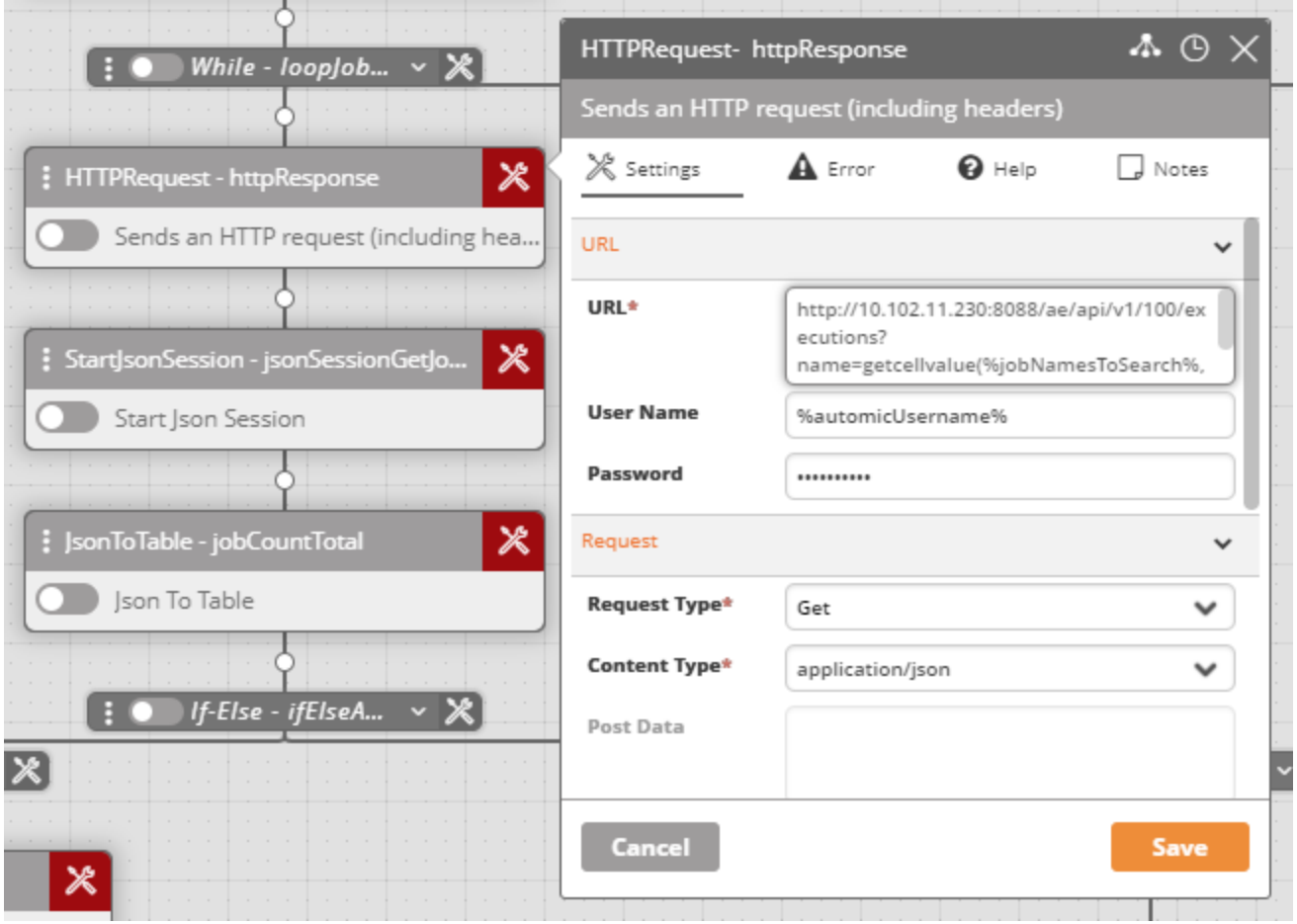
After that, a **GetRowsCount** activity is used to store the number of rows in the **jobNamesToSearch** table that we just created. This is simple and seen in the screenshot below.



Now we begin a while-loop that we will name **loopJobsSearch** that will execute separate iterations for each row of the job name table we created, as seen below.



Next comes a very important step where we use the **HTTPRequest** activity to send an **HTTP GET** request to the **Automic** API's executions endpoint to retrieve the latest failed job matching the name(s) specified in the **jobNamesToSearch** table we created earlier. This activity is configured as seen in the screenshot below.

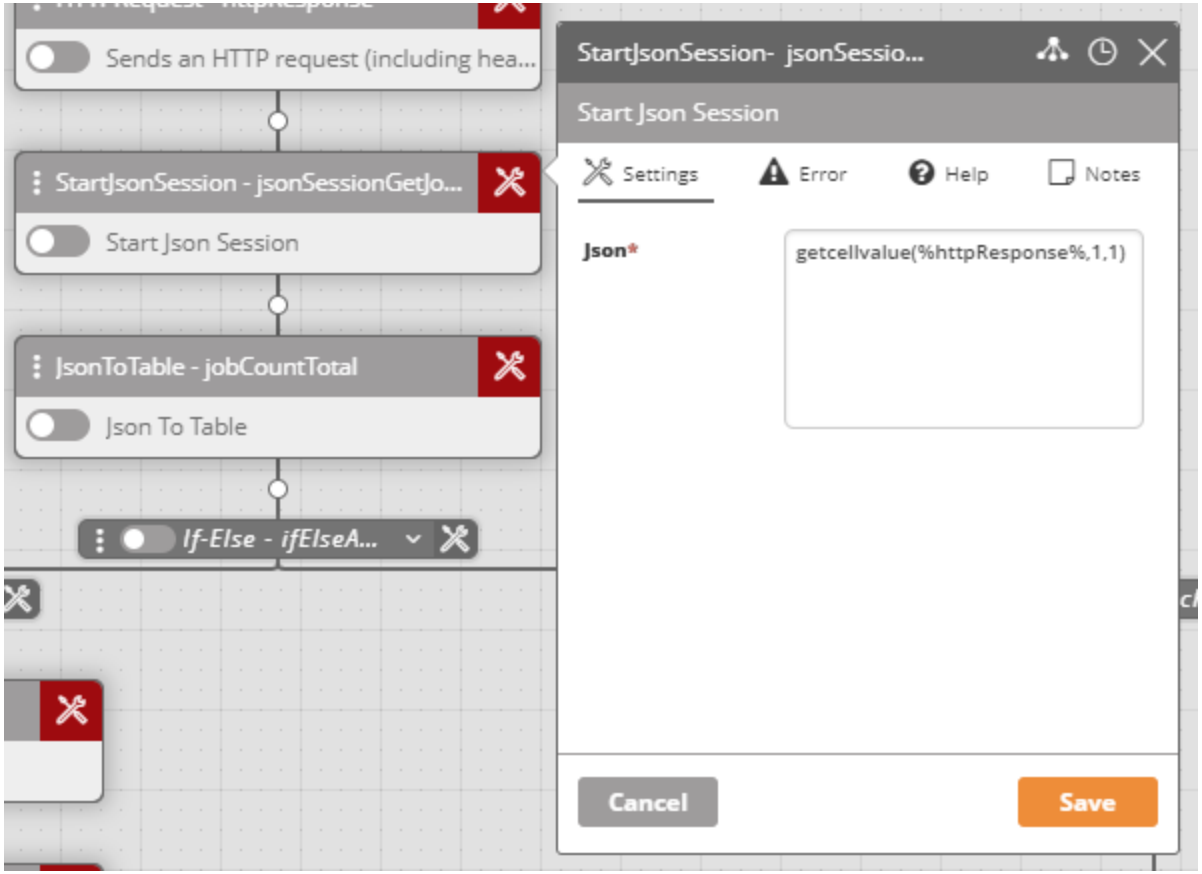


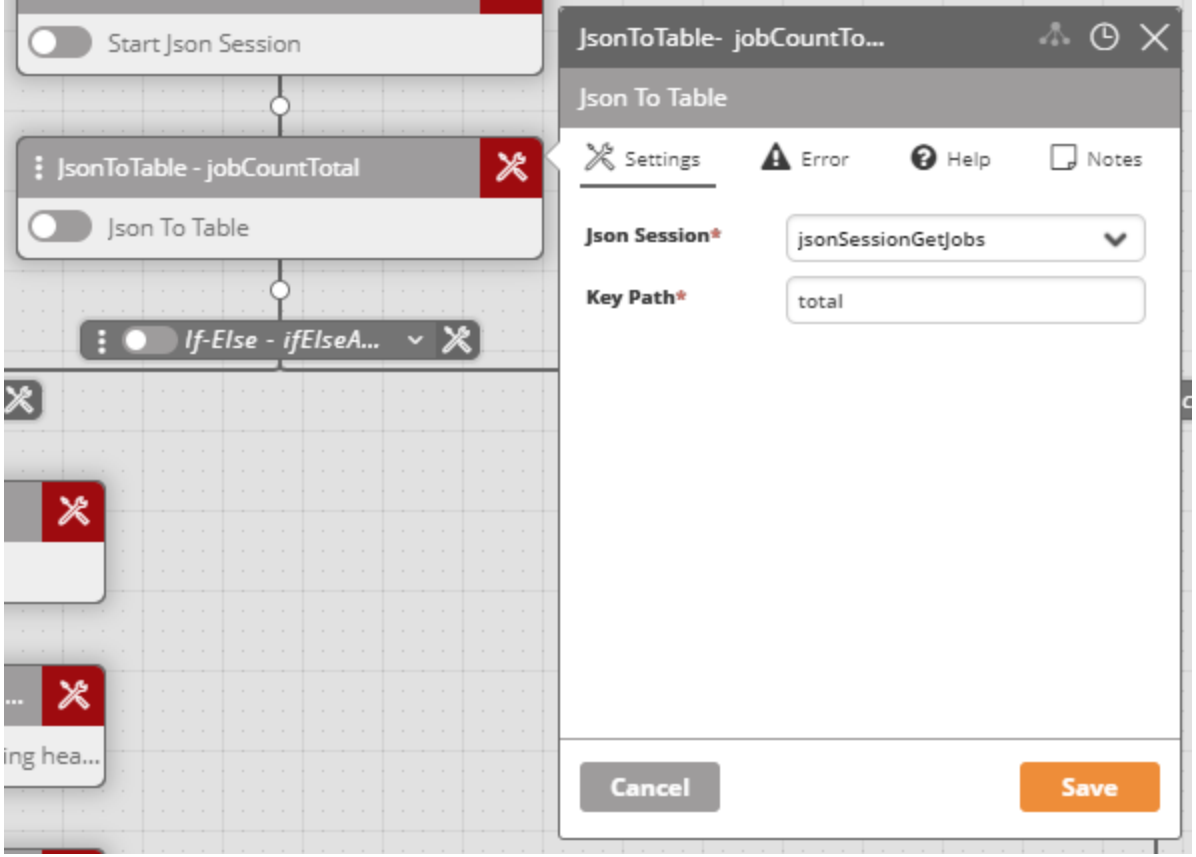
The **HTTPRequest** activity's settings are as follows:

Property	Value
URL	http://10.102.11.230:8088/ae/api/v1/100/executions? name=getcellvalue(%jobNamesToSearch%,%loopJobsSearch%,1)&status=1800,1820&max_results=1
User Name	%automicUsername%
Password	%automicPassword%
Request Type	Get
Content Type	application/json

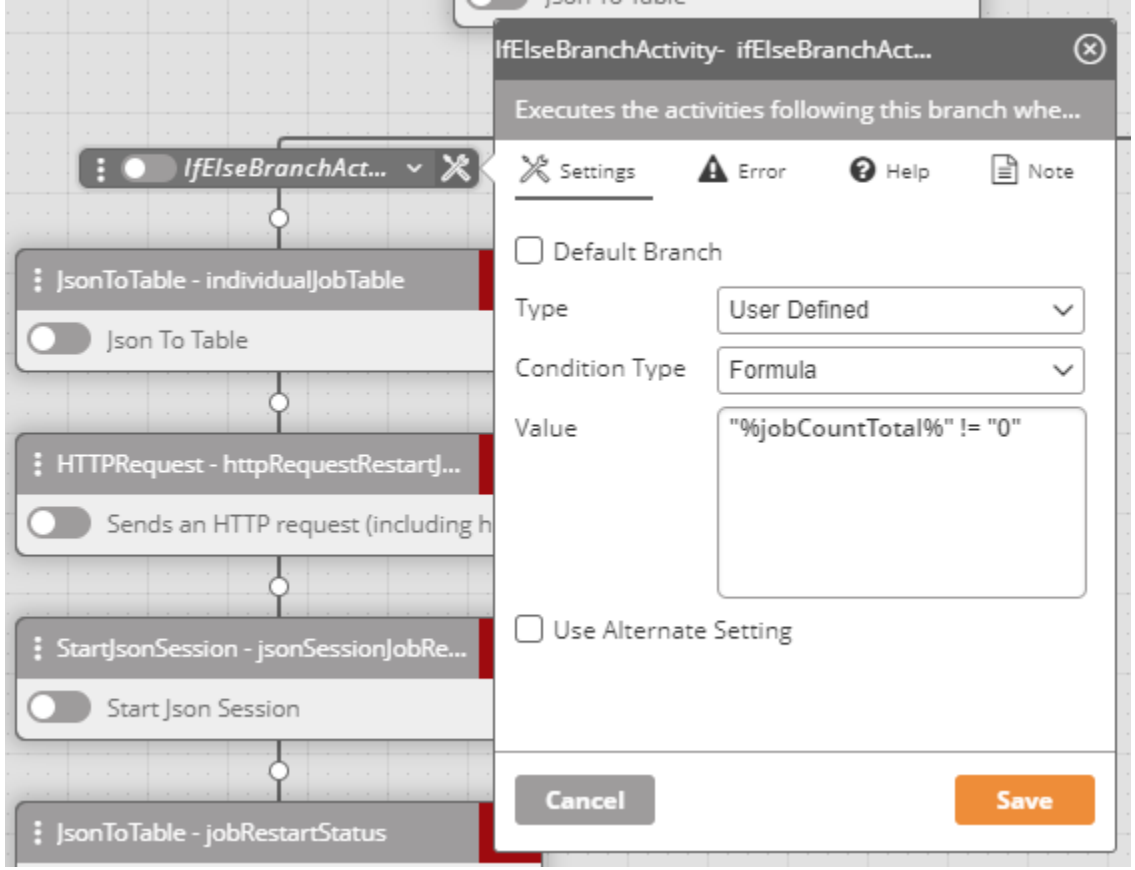
For the **name** parameter, we are using the **getcellvalue** formula native to **Ayehu NG** in order to pull the current cell from the **jobNamesToSearch** table.

Our next steps are to start a JSON session from the response received from the **Automic** API and then convert those results into an **Ayehu NG** native resultset table. The screenshots below show how this is configured.

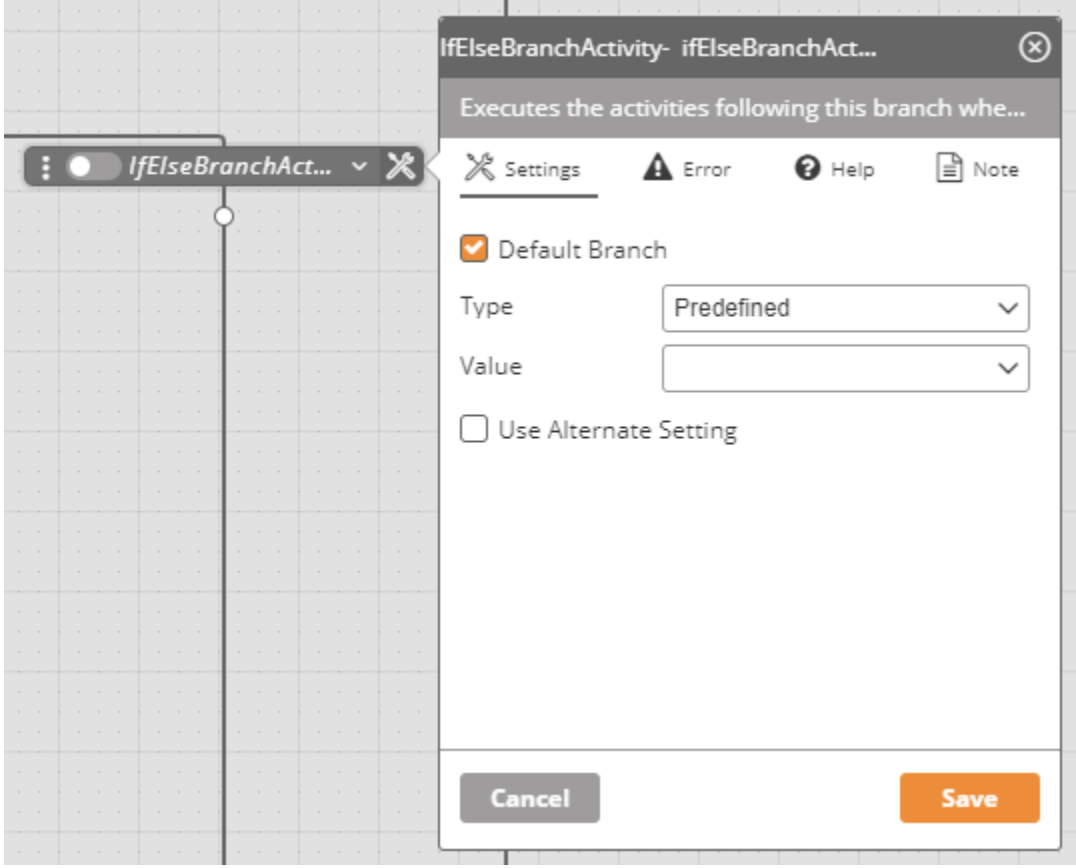




Our next step is to ensure that a failed batch job execution was found before proceeding with the rest of the workflow, since we cannot restart a failed job that doesn't exist. This is achieved by an if-else branch where the left side is configured as seen below, containing the user-defined formula **"%jobCountTotal%" != "0"**.

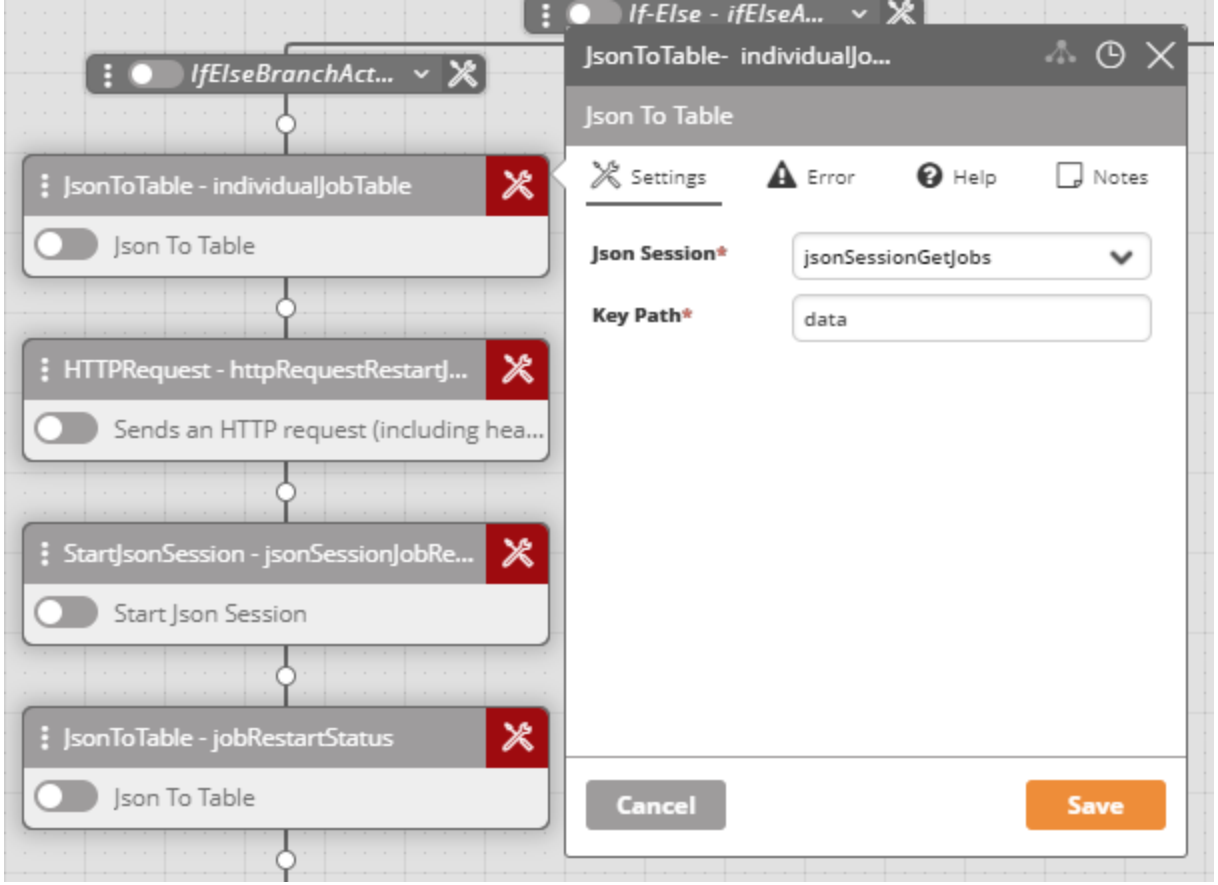


The right side branch is configured as the **Default Branch**, as seen below.

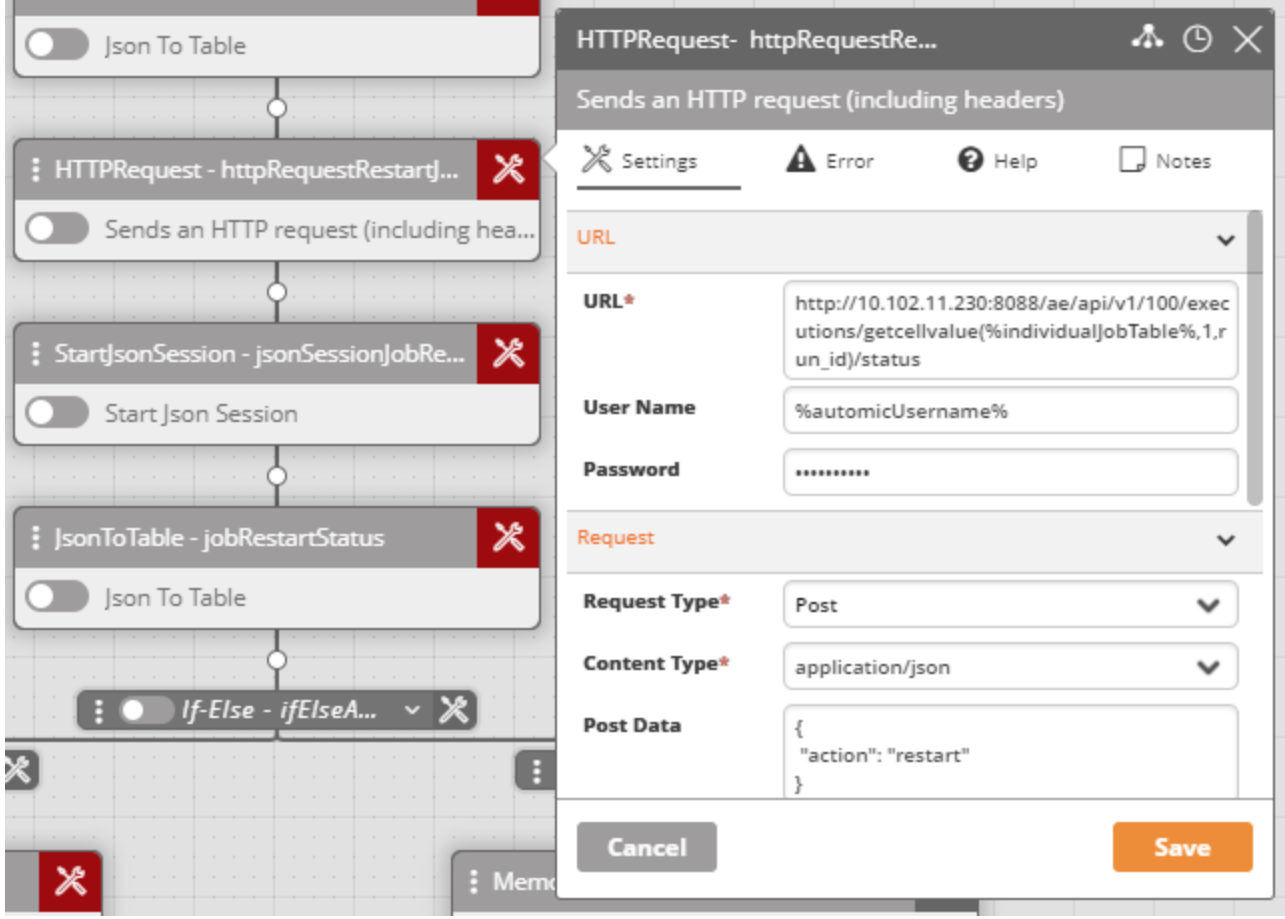


This default branch does nothing, and the while-loop will simply begin its next iteration (if one is available).

The next step is to extract the run ID of the failed batch job, which is stored in the **data** key. This is done with a **JsonToTable** activity configured in the screenshot below.



Now we come to a crucial step where we send a request to the **Automic** API to restart the failed batch job. Our **HTTPRequest** activity is seen below.

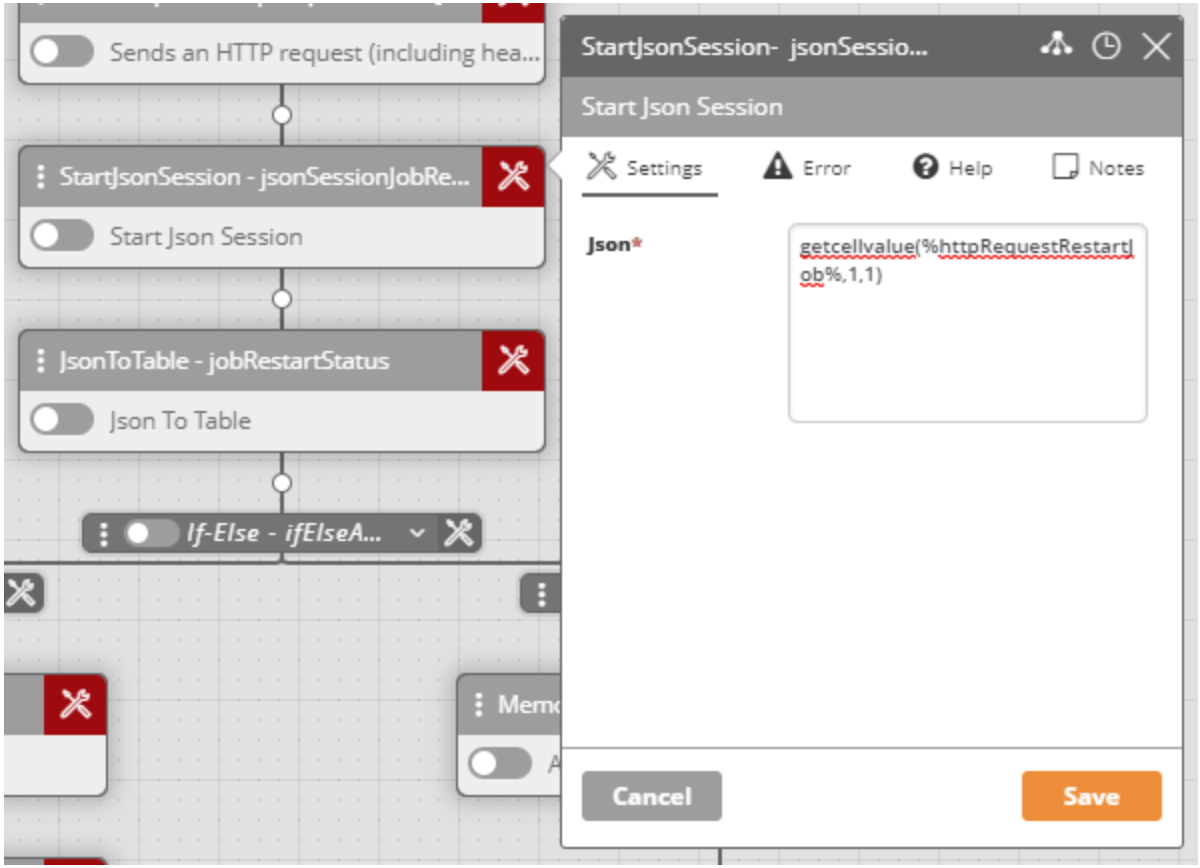


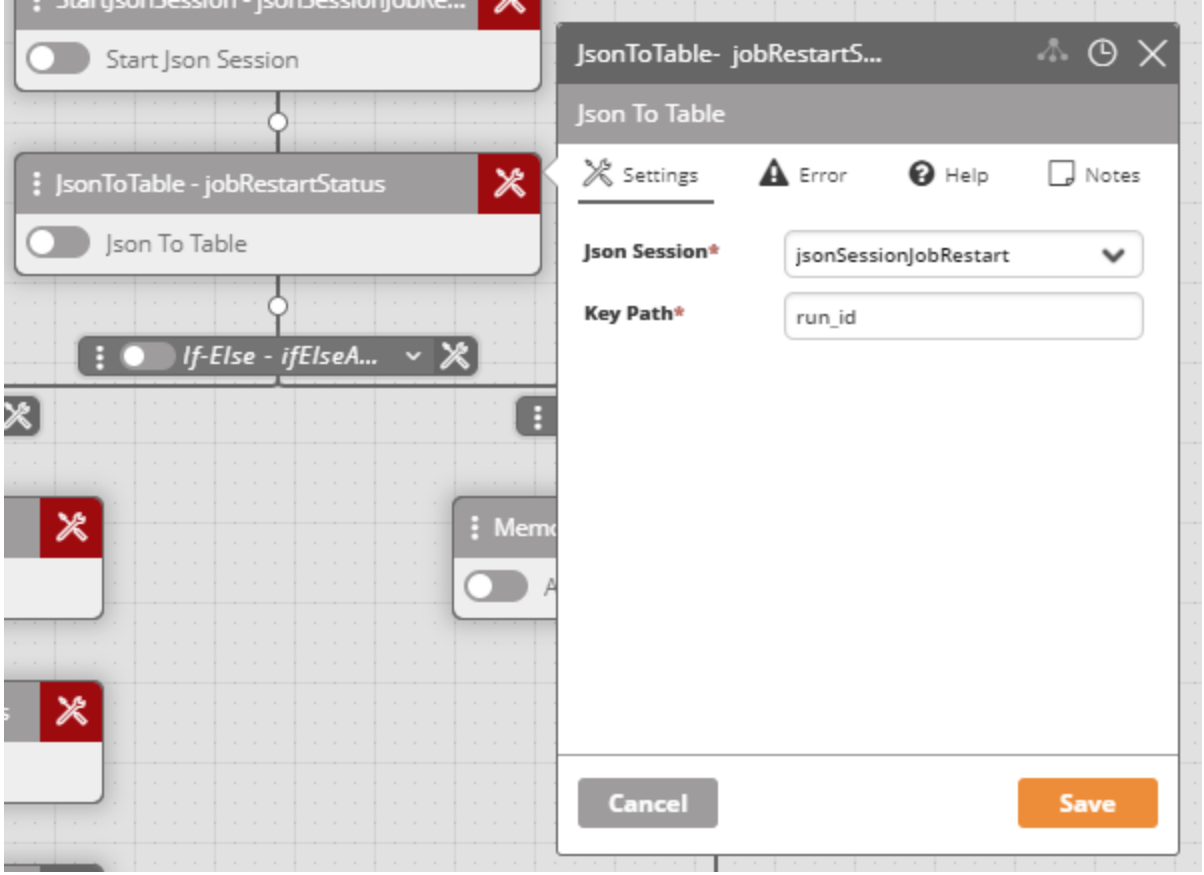
The **HTTPRequest** activity's settings are as follows:

Property	**Value**
URL	http://10.102.11.230:8088/ae/api/v1/100/executions/getcellvalue(%individualJobTable%,1,run_id)/status
User Name	%automicUsername%
Password	%automicPassword%
Request Type	Post
Content Type	application/json

Post Data	{
	"action": "restart"
	}

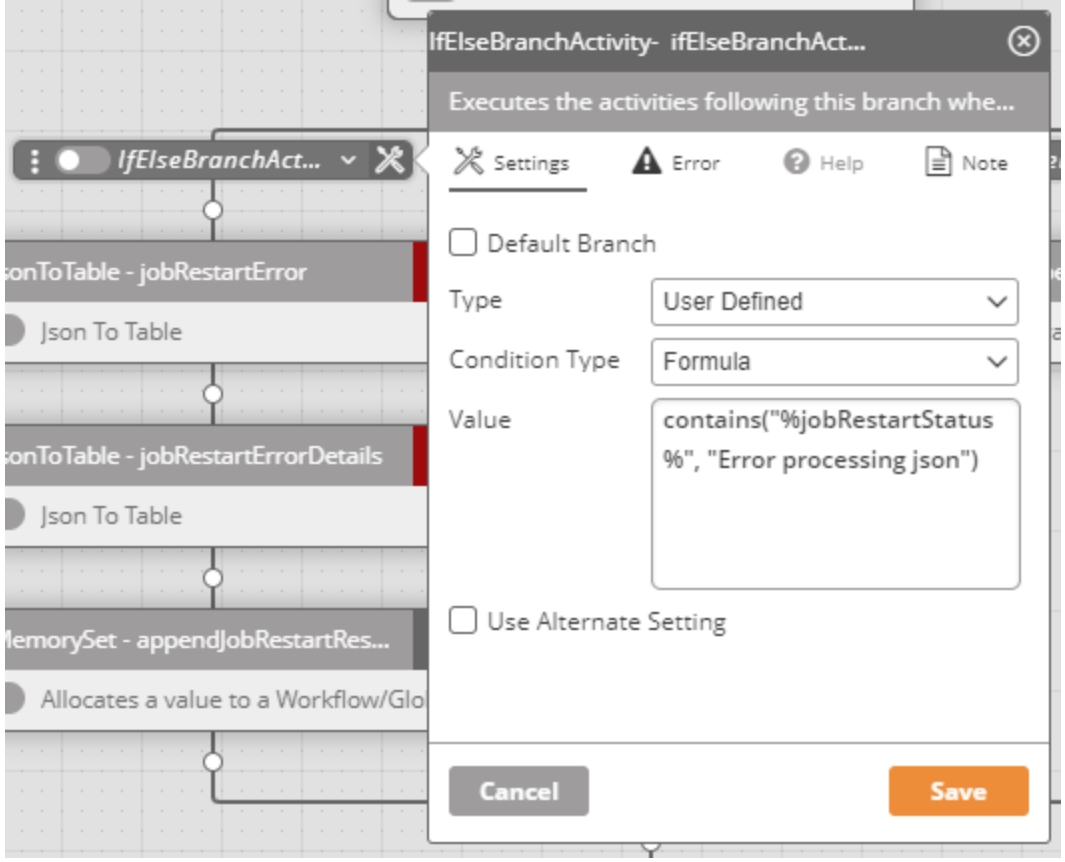
We then start a new JSON session with the results sent back to **Ayehu NG** from the **Automic** API. We then extract the **run_id** key using the **JsonToTable** activity to store the new run ID for the restarted job. The configuration for these two activities is seen in the screenshots below.



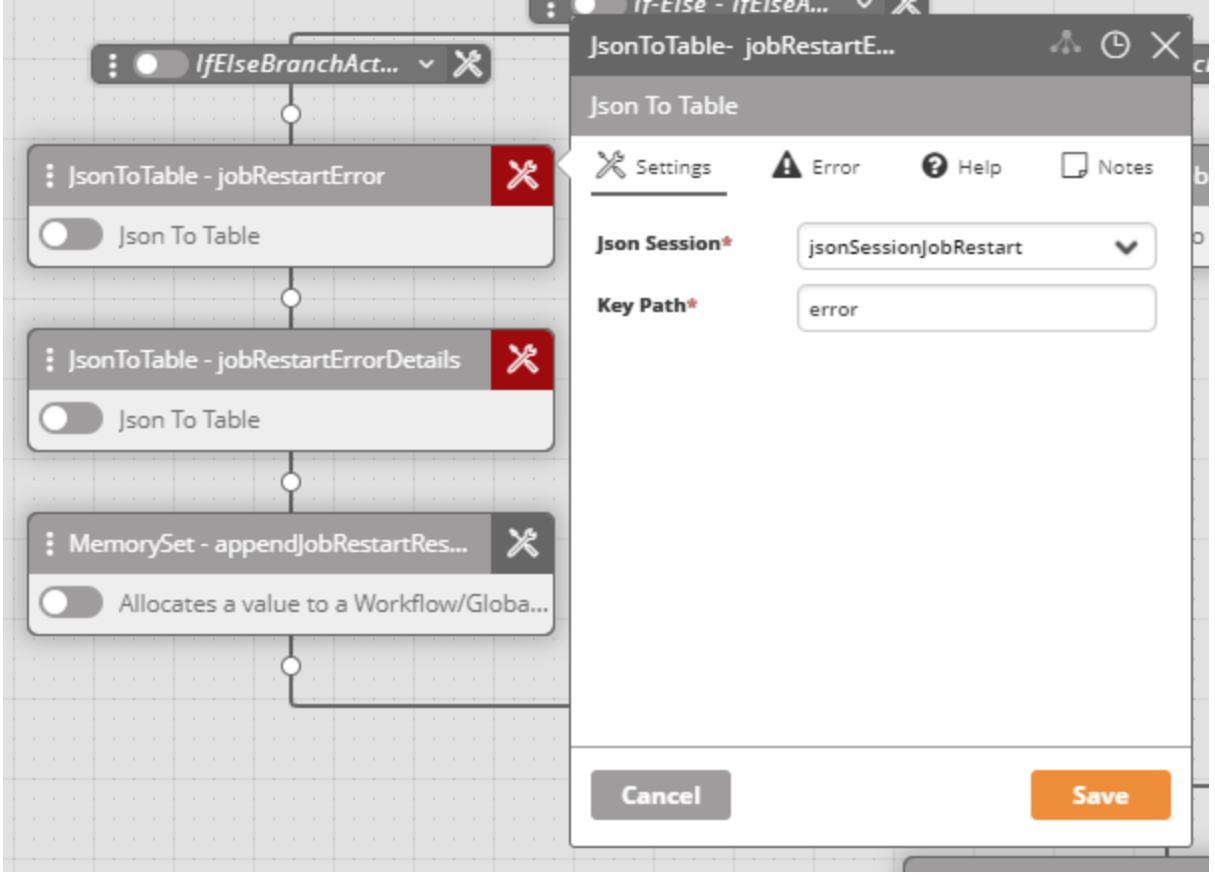


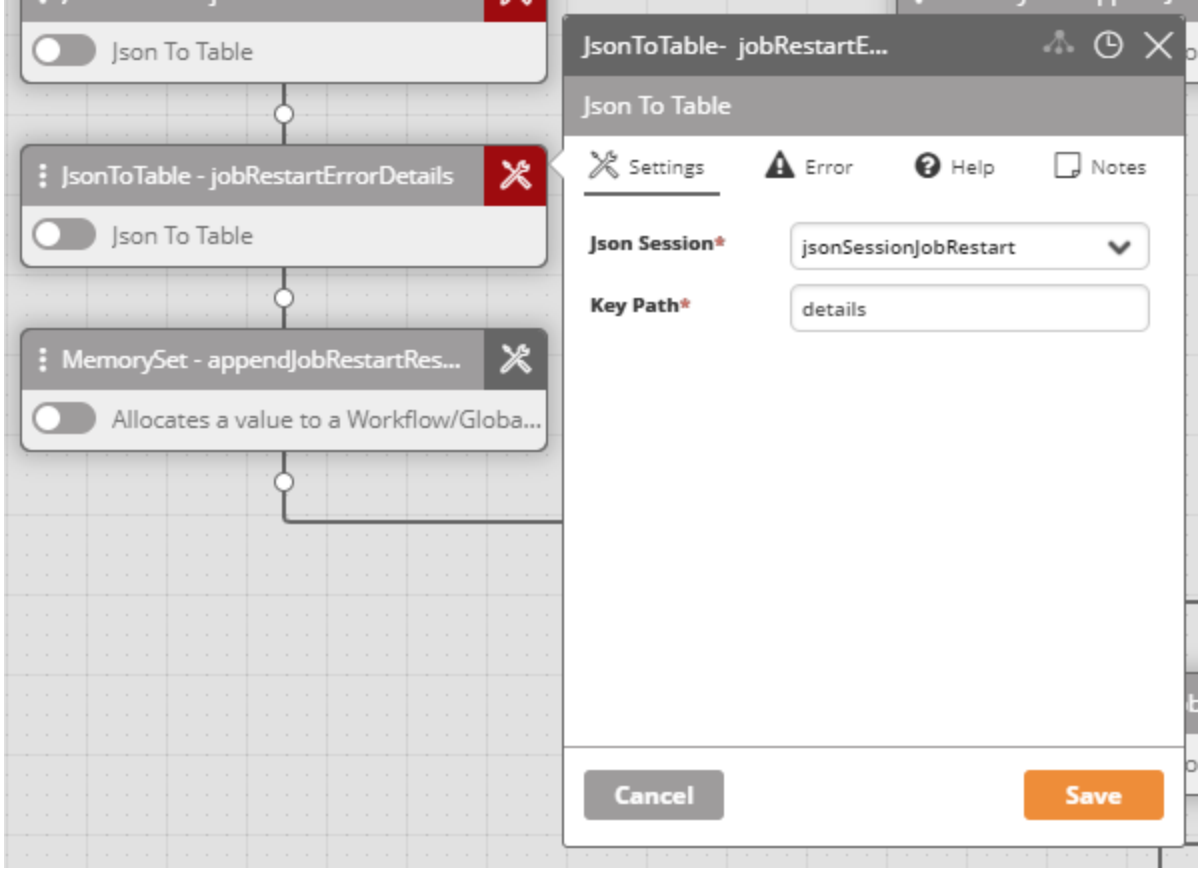
We now come to another crucial step in the workflow. If a batch job restart request fails for any reason, there will be no **run_id** key in the JSON results returned to **Ayehu NG** from **Automic**. Therefore, we now use another if-else branch to determine whether or not the restart was successful.

The left side branch is configured with the user-defined formula **contains("%jobRestartStatus%", "Error processing json")** which checks for the "Error processing json" message coming from the **jobRestartStatus** activity. This formula is configured as seen below.

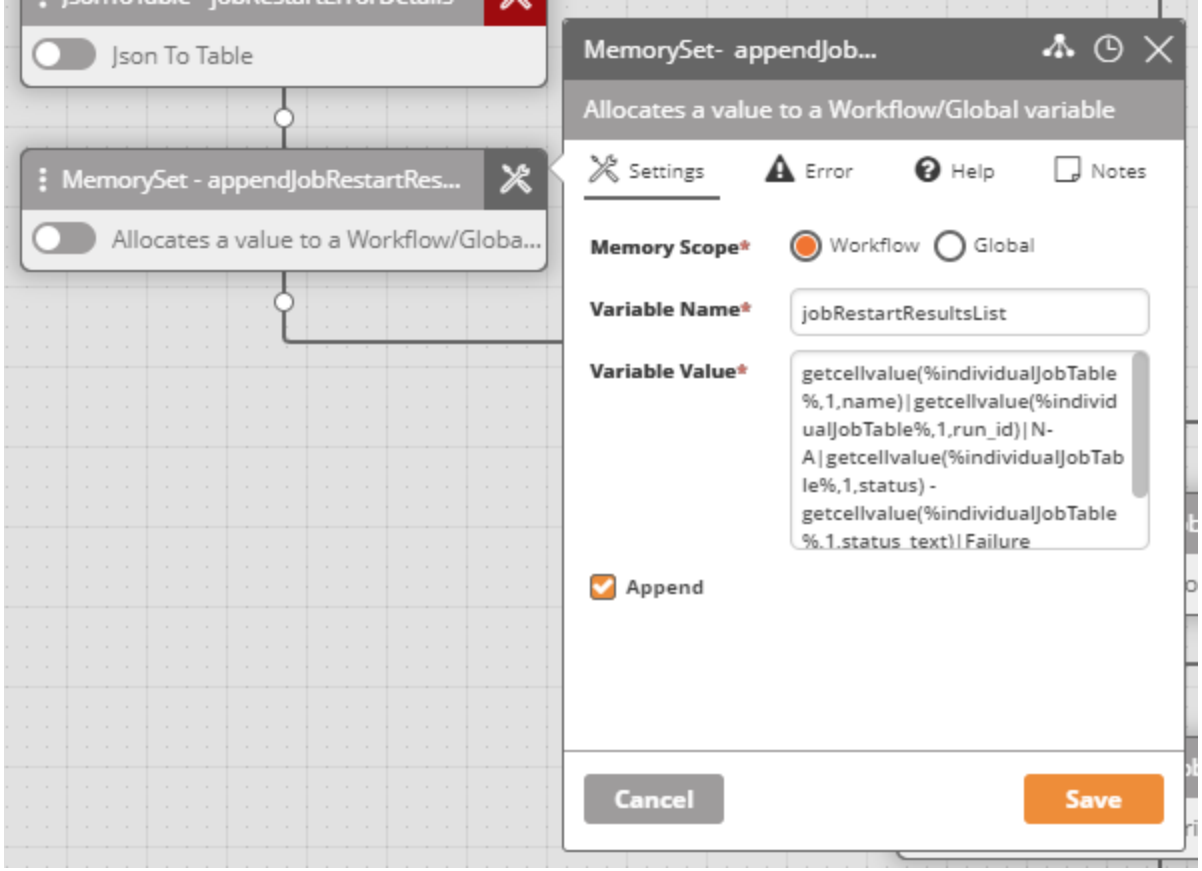


Since the job restart request failed, we then collect additional information about the failure in order to compile a thorough report at the end of the workflow. We do this with two (2) additional **jsonToTable** activities for extracting both the **error** and **details** key from the JSON in the **jsonSessionJobRestart** JSON session. These two activities are configured as shown in the screenshots below.


IfElseBranchAct...
JsonToTable - jobRestartError
Json To Table
JsonToTable - jobRestartErrorDetails
Json To Table
MemorySet - appendJobRestartRes...
Allocates a value to a Workflow/Globa...
JsonToTable- jobRestartE...
Json To Table
Settings
Error
Help
Notes
Json Session*
jsonSessionJobRestart
Key Path*
error
Cancel
Save



The last activity in this branch is a **MemorySet** for storing all of the information on the failed job restart request, as seen below.

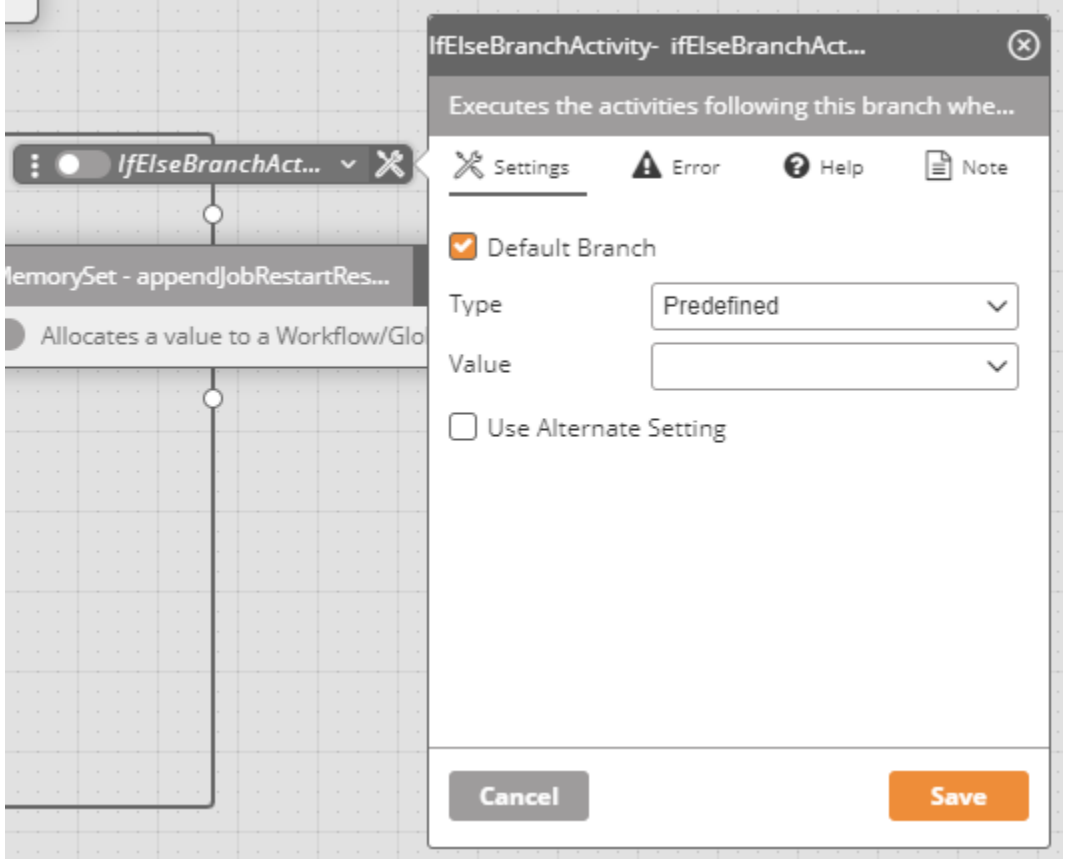


We can see in the screenshot that the **Append** option is checked, since we want to add this text to the **jobRestartResultsList** that was created at the beginning of the workflow. This text follows the same pipe-delimited (**|**) format as the columns and is as follows:

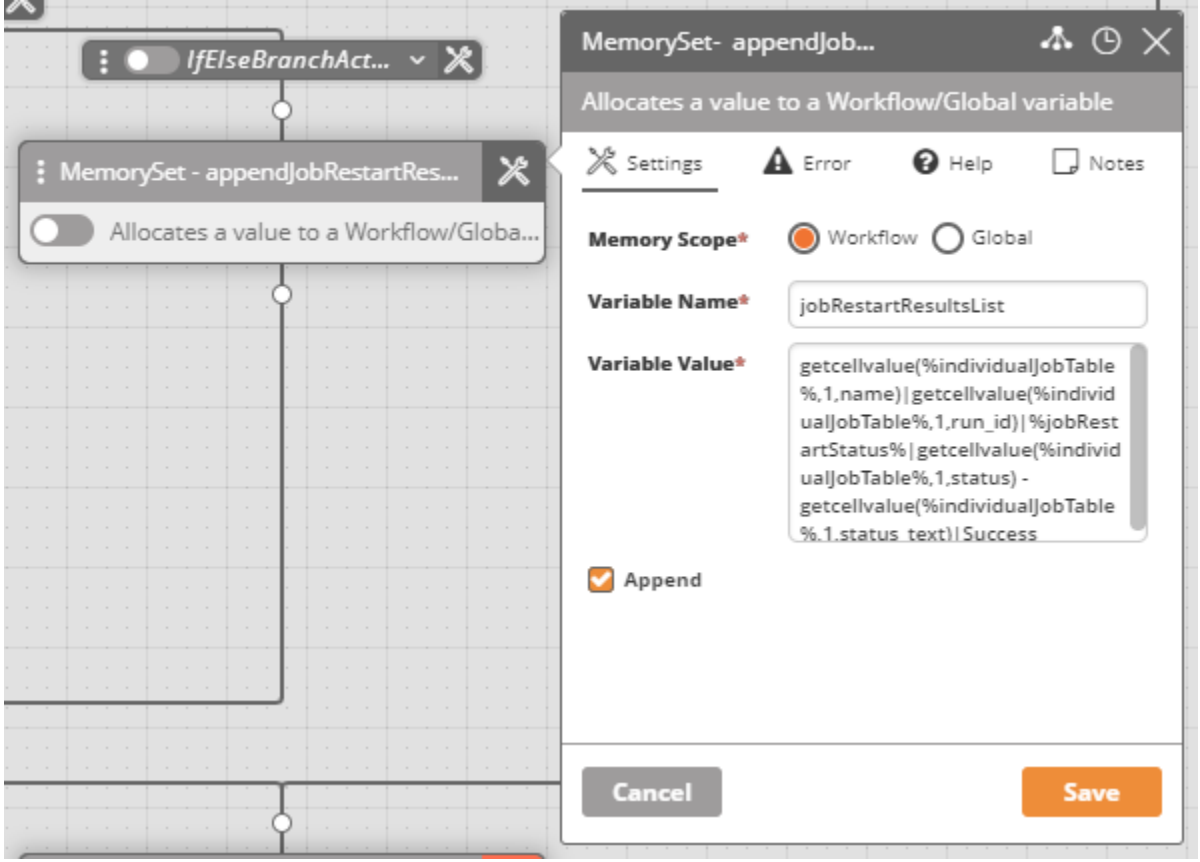
```
getcellvalue(%individualJobTable%,1,name)|getcellvalue(%individualJobTable%,1,run_id)|N-A|getcellvalue(%ir
```

In this text, we see the usage of **getcellvalue** functions to pull the **name**, **run_id**, **status**, and **status_text** values from the JSON response we received and converted to a table, as well as the other error details stored in the **jobRestartError** and **jobRestartErrorDetails** activities, and also the value "N-A" for the "NewRunID" column.

Returning to the start of the if-else branch and looking at the right side branch, we can see that it's configured as the default branch in the screenshot below.



The only action taken in this branch is a **MemorySet** activity for updating the **jobRestartResults** text with information on the successfully restart batch job, as seen below.

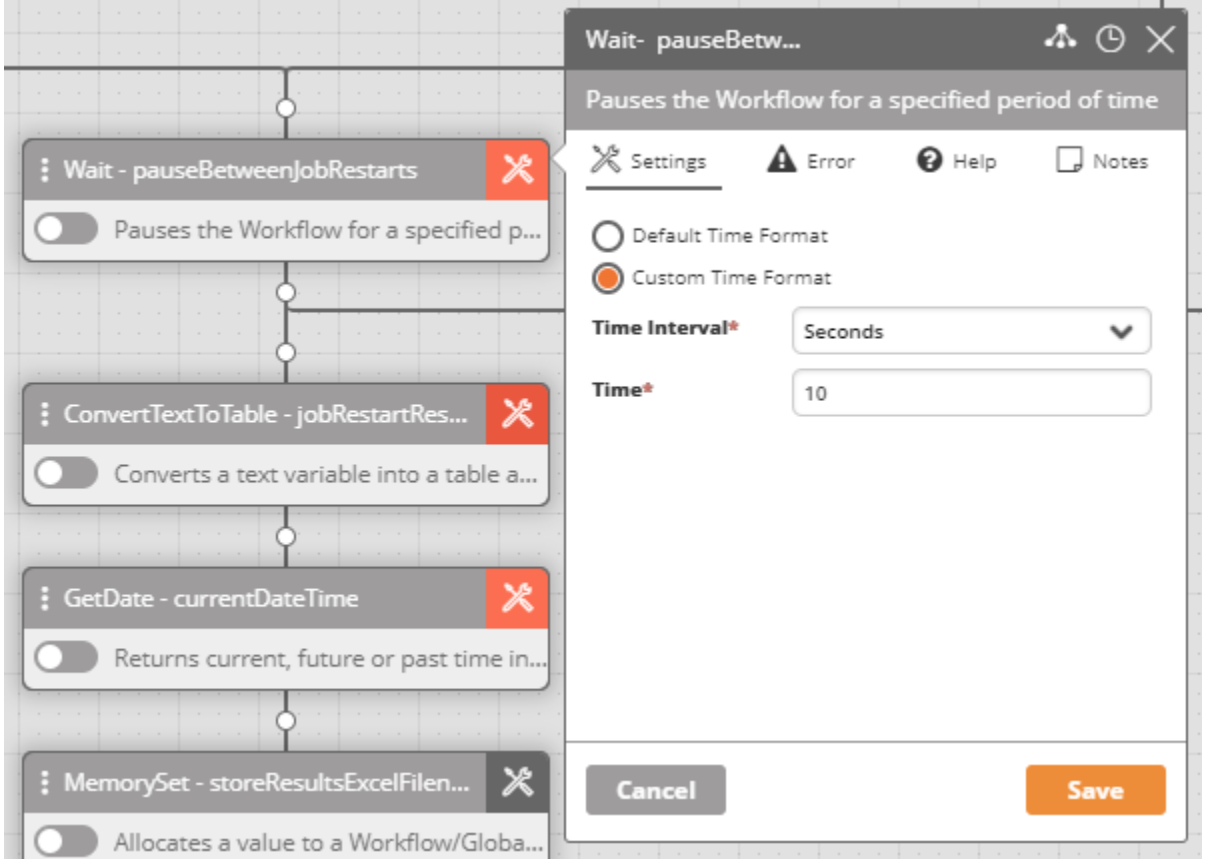


We can see in the screenshot that the **Append** option is checked, since we want to add this text to the **jobRestartResultsList** that was created at the beginning of the workflow. This text follows the same pipe-delimited (**|**) format as the columns and is as follows:

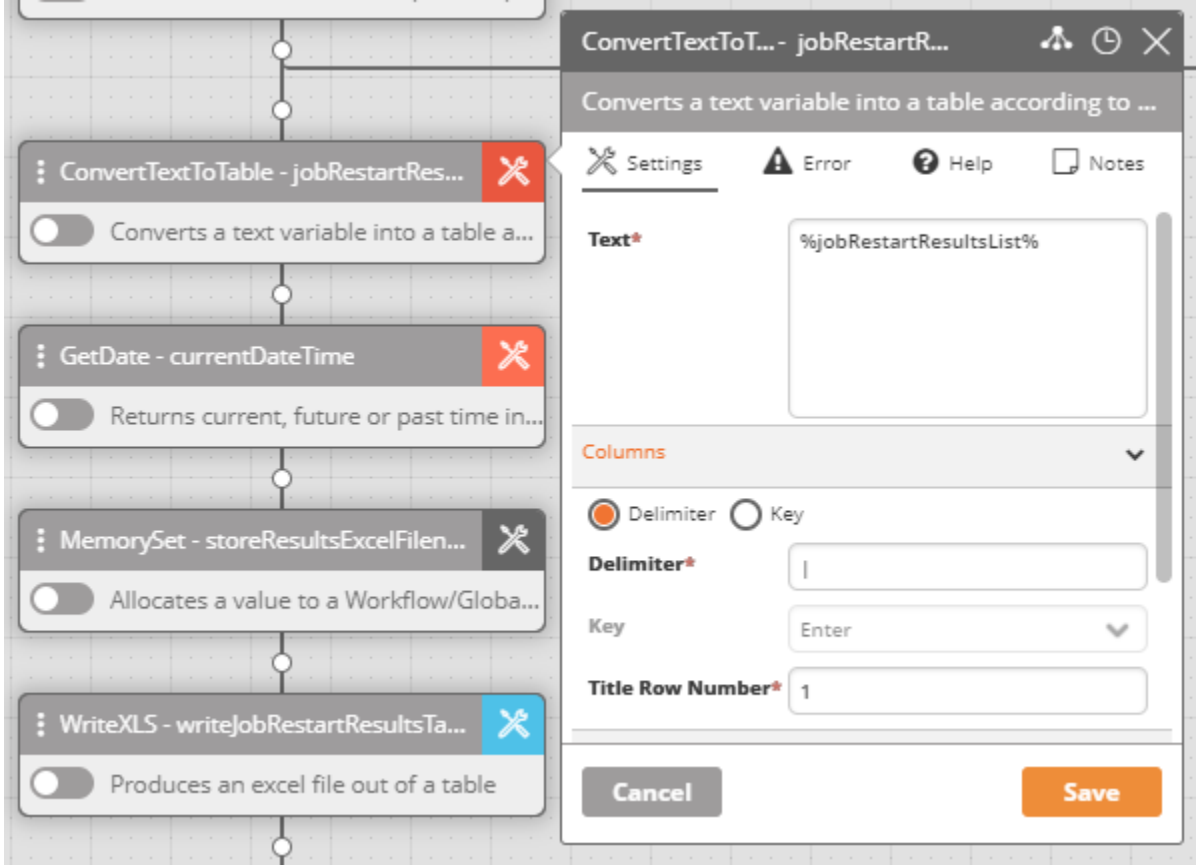
```
getcellvalue(%individualJobTable%,1,name)|getcellvalue(%individualJobTable%,1,run_id)|%jobRestartStatus%|g
```

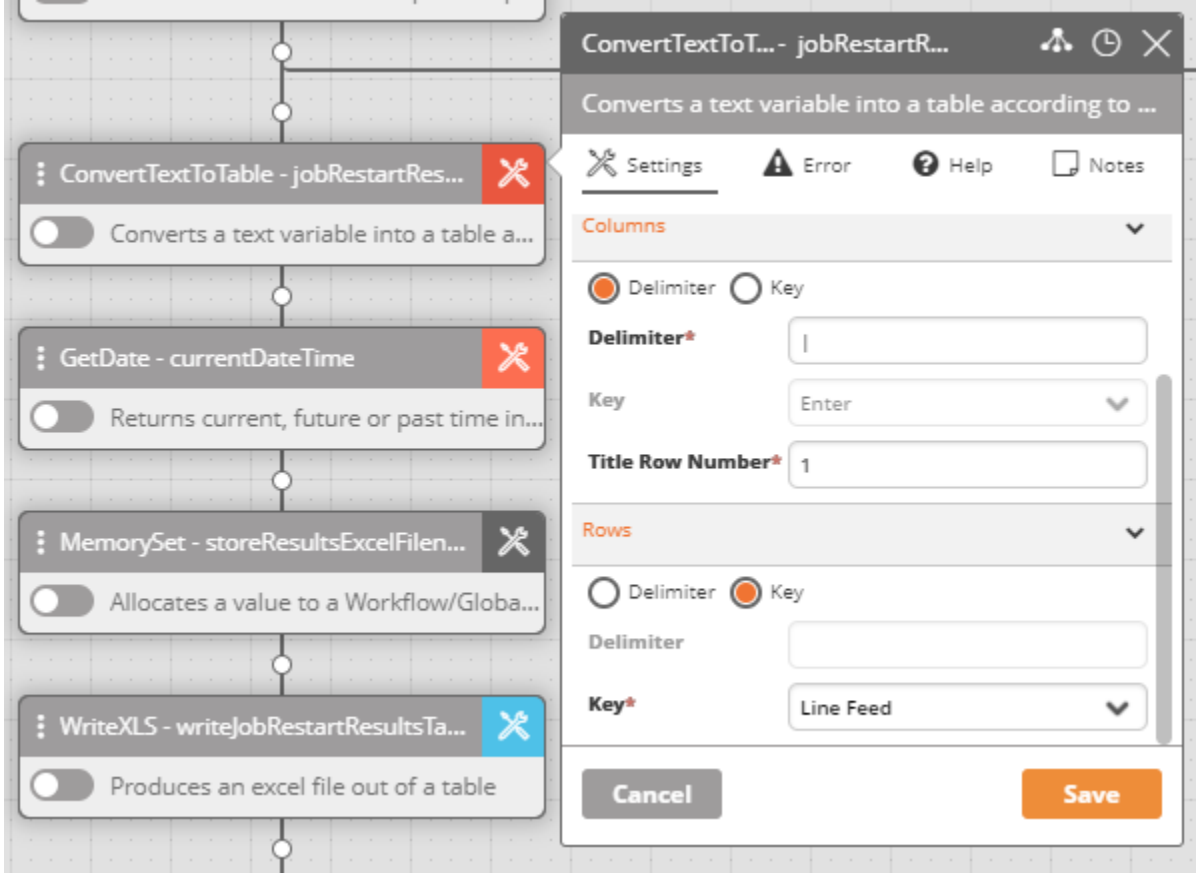
In this text, we see the usage of **getcellvalue** functions to pull the **name**, **run_id**, **status**, and **status_text** values from the JSON response we received and converted to a table, as well as the new run ID stored in the **jobRestartStatus** activity.

After both the parent and nested if-else branches end, we come to an option **Wait** activity. For some users, the **Automic** system works best when a ten (1) second delay exists in between job restart requests. For this example, we have this activity configured as in the screenshot below.



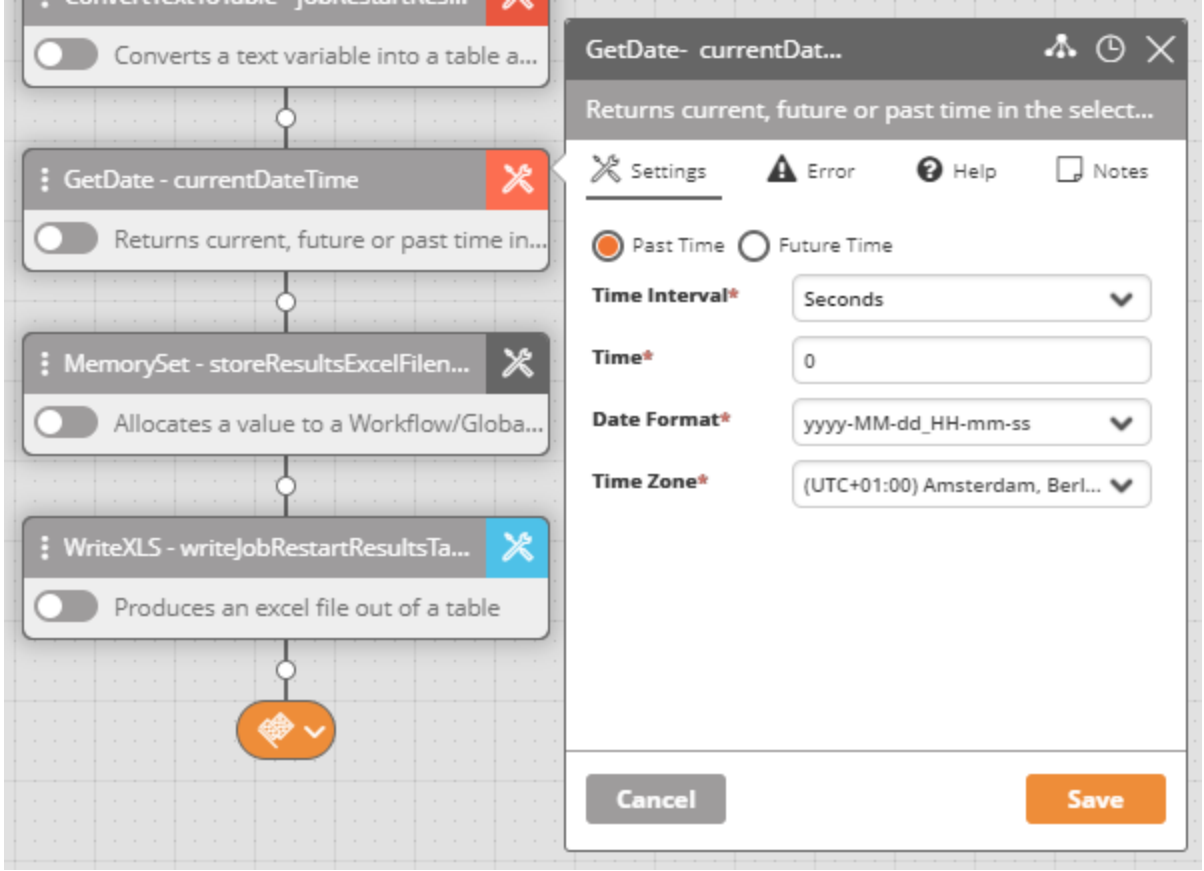
Next, the while-loop branch ends and we come to the last four (4) activities of the workflow. The first step is to convert the plaintext report stored to a native **Ayehu NG** resultset table. The **ConvertTextToTable** activity is configured as seen in the two (2) screenshots below.


ConvertTextToTable - jobRestartRes...
Converts a text variable into a table a...
GetDate - currentDateTime
Returns current, future or past time in...
MemorySet - storeResultsExcelFilen...
Allocates a value to a Workflow/Globa...
WriteXLS - writeJobRestartResultsTa...
Produces an excel file out of a table
ConvertTextToT... - jobRestartR...
Converts a text variable into a table according to ...
Settings
Error
Help
Notes
Text*
%jobRestartResultsList%
Columns
Delimiter
Key
Delimiter*
|
Key
Enter
Title Row Number*
1
Cancel
Save

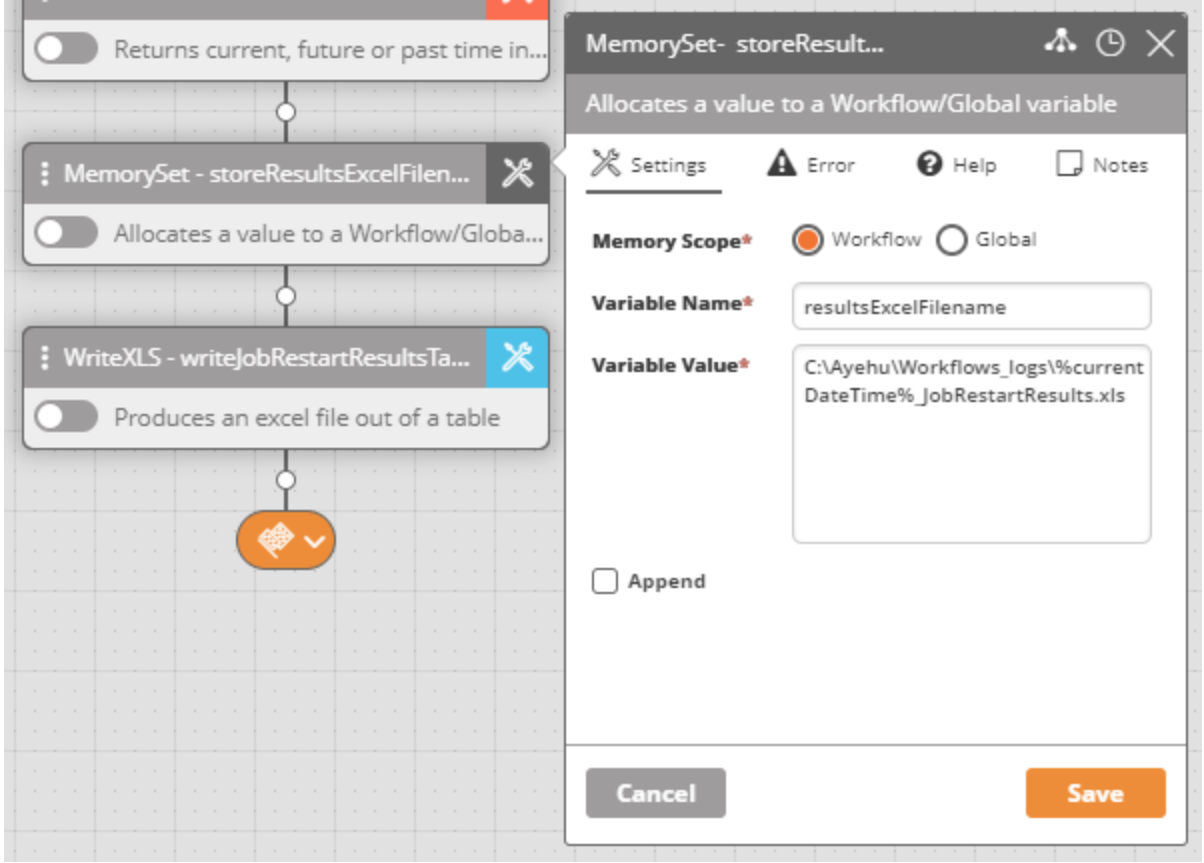


In order to successfully convert the plaintext report to a table, be sure to use the pipe character (**|**) as the delimiter for columns and for rows, the **Key** option should be selected and "Line Feed" as its value.

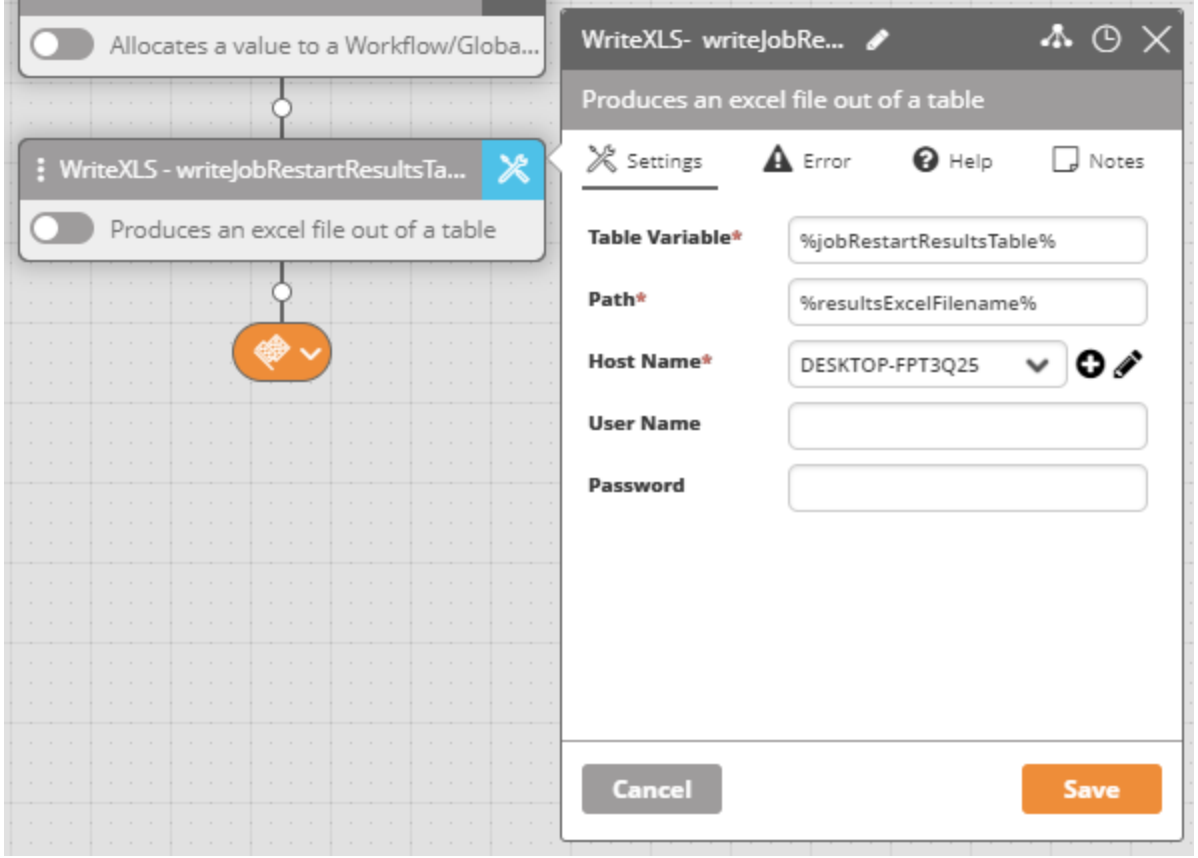
Next, the **GetDate** activity is used to store the current date and time in the format **yyyy-MM-dd_HH-mm-ss** so that it can be used as part of the filename for the Excel spreadsheet that is generated containing our job restart report. This configuration is seen in the screenshot below.



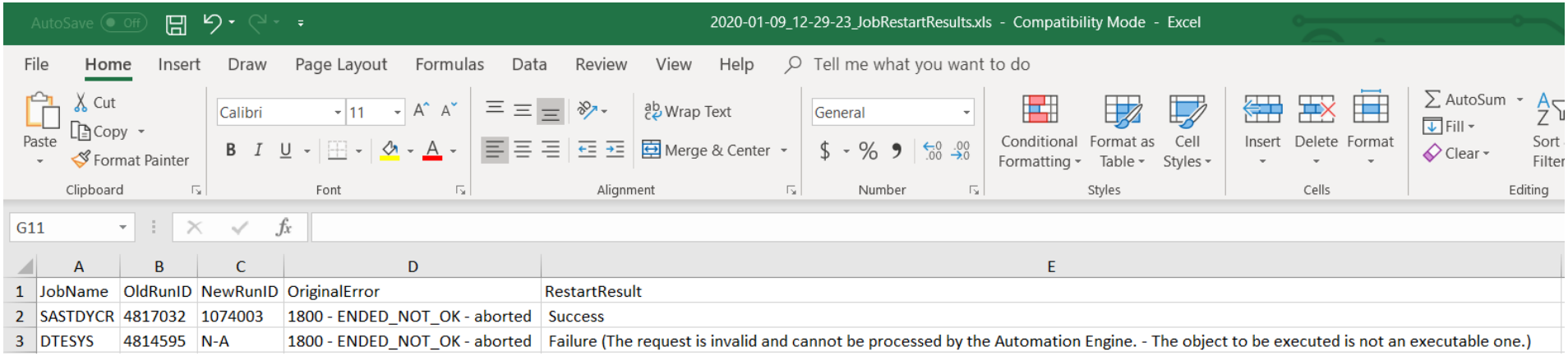
We then use **MemorySet** to store the filename for the Excel spreadsheet as seen below.



The last step is to use the **WriteXLS** activity to write the contents of the **jobRestartResultsTable** table to the filename specified, as seen in the screenshot below.



Below is a screenshot containing an example Excel spreadsheet generated by this workflow.



Click image to view full-sized version.

Automic - Restart Failed Jobs.xml

30 KB · Download

Was this article helpful?

✓ Yes ✕ No

0 out of 0 found this helpful

Return to top

Related articles

How can I read Excel or CSV files using the eyeShare workflow?

Comments

0 comments

Be the first to write a comment.