UiPath Framework Training

Mini-Bot Walkthrough

Jordan Mohler, 2/25/2019

Bot Details

Description

This bot is an adaptation of the process created in UiPath Academy Training- Level 1: Lesson 4, Practice 2. Using the UiPath REFramework, the process will read an input excel file containing the 25 largest cities in the United States. It will the search for the weather in each of those cities in Internet Explorer and write the results to a text file.

Input / Output

The input file is an excel file called MiniBot_Input.xlsx. The file consists of two columns, City and State, and 25 rows (not including the headers). The rows represent a list of the 25 largest cities in the United States.

The output file will be a text file called LargestCitiesWeatherReport.txt. It will contain 25 lines of text that match the following pattern: “The temperature in [City], [State] is [Temperature] degrees”.

Both files should be placed in the Data subfolder of the project.

Walkthrough

Setup

First, we should clean up the Framework to match what we will need for the process. We will not be using Orchestator Queues or Credentials, so the GetAppCredentials and SetTransactionStatus files from the Framework may be deleted. Because we are deleting the GetAppCredentials workflow, we can also remove the UiPath.Credentials.Activities dependency.

Additionally, we will not be performing unit tests, so the Test_Framework folder and _Tests workflow may be deleted. Do not worry if this causes validation errors in your code, we will clean those up next. Also feel free to delete TakeScreenshot, but you may use this workflow if you want.

Next, let’s clean up the arguments and variable panels in Main. The following variables can be removed: TransactionID, TransactionField1, TransactionField2, RetryNumber. Additionally, the OrchestratorQueueName argument may be deleted. Because we are not using Orchestrator Queues for this process, change the type TransactionItem from QueueItem to DataRow. Don’t forget to also change the type of TransactionItem in GetTransactionData and Process.

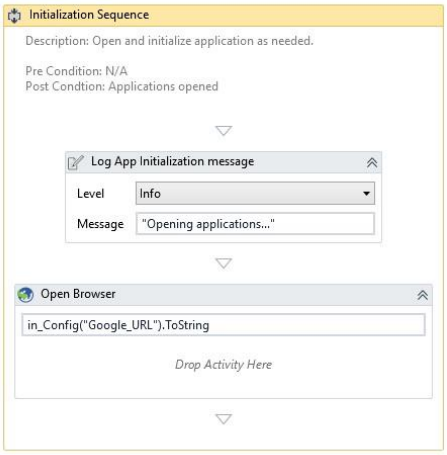
Finally, let’s clean up the validation errors. Using the Debugging and Framework Tips and Tricks Guides, try to resolve all validation errors in the program.

Initialization

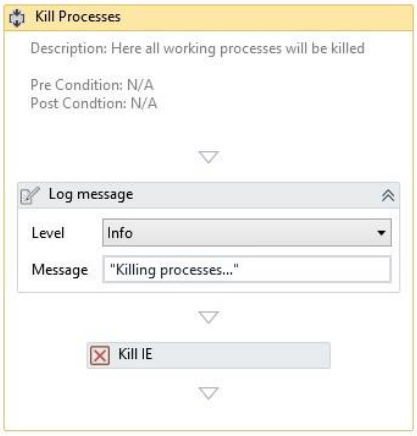
There are three workflows that are invoked by default in the Init block of the Main workflow: InitAllSettings, KillAllProcesses, and InitAllApplications.

For this application, the only application we need to open is Internet Explorer (Notepad and Excel will be opened by their initial activities or run in the background, so they don’t need to be explicitly opened). In InitAllApplications, add an *Open Browser* activity. We don’t want to use a hard-

coded value, so add a row to the Settings sheet in the Config.xslx file with the name `Google_URL` and value `https://google.com`. Then in the *Open Browser* activity, set the URL to `in_Config("Google_URL").ToString`. The InitAllApplications workflow should appear as follows:



Similarly, for the KillAllProcesses workflow, we need only to kill Internet Explorer. Add a *Kill Process* activity and set the ProcessName to `"iexplore"`. The KillAllProcesses workflow should appear as follows:

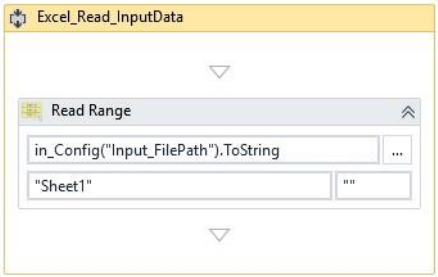


Nothing needs to be changed in the InitAllSettings workflow, but we do need to setup the Config file to be read in. Add the following names to the Config file: `Input_FilePath`, `Output_FilePath`.

The final step for initializing the process is to read in the input file. Create a new Sequence workflow called Excel_Read_InputData.xaml and put it in a folder called Excel. We will need to export the DataTable we read in, so create an out argument called `out_TransactionData` of type DataTable. Additionally, we will need the Config file, so make an in argument called `in_Config` of type Dictionary<String, Object>. Next, add a *ReadRange* activity to the file. Set the WorkbookPath to `in_Config("Input_FilePath").ToString` and the Range to an empty string (""). Finally, set the Output DataTable to `out_TransactionData`.

Note: If you use the *ReadRange* activity from App Integration > Excel, you will need to surround it with an *ExcelApplicationScope* activity. Alternatively, you can use the *ReadRange* activity from System > File > Workbook

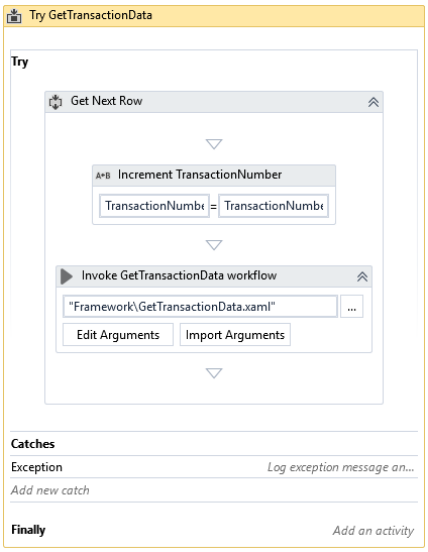
The Excel_Read_InputData workflow should appear as follows:



Then, in the Main workflow, within the Init *State* activity, navigate to the 'If first run' Sequence. Below the 'Invoke KillAllProcesses workflow' activity, invoke the workflow you just created. Make sure to import the necessary arguments.

Getting Transaction Data

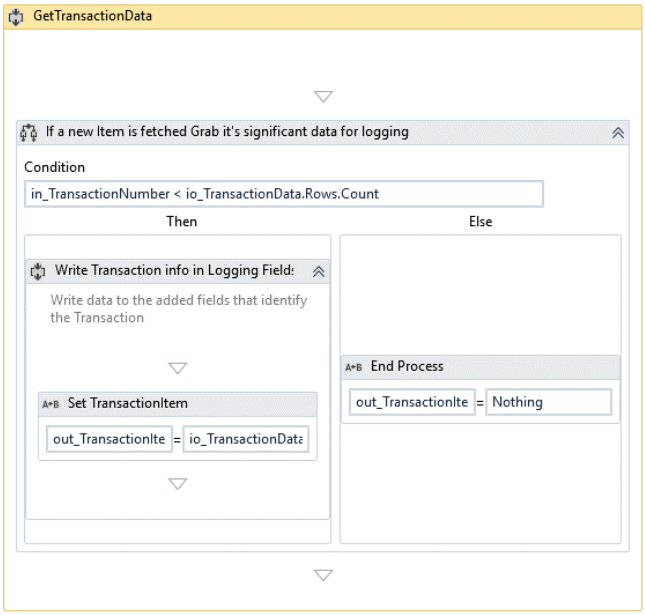
If we were using Orchestrator Queues, by default there would be an attached Transaction Number. Because we are not, we have to increment the TransactionNumber variable ourselves. In the GetTransactionData State in the Main workflow, increment TransactionNumber before invoking the GetTransactionData workflow. To accomplish this, assign TransactionNumber to TransactionNumber+1. Then, because we are incrementing before getting the first transaction item, we must set the default value of TransactionNumber to 0. The 'Try GetTransactionData' sequence should appear as follows:



Next, navigate to the GetTransactionData workflow. Our input data was pulled in previously from Excel, so the purpose of this workflow is now to grab a row from that input data. If it is still present, delete the *Get Transaction Item* activity. Change the condition in the *If Statement* activity to `in_TransactionNumber < io_TransactionData.Rows.Count`. Next, we need to set Transaction Item to the row corresponding to the TransactionNumber. In the 'Write Transaction info..' sequence, add an *Assign* activity that sets `out_TransactionItem` to `io_TransactionData.Rows(in_TransactionNumber-1)`.

Note: Transaction numbers are 1-indexed, while DataTables are 0-indexed. The above code works if you did not Add the Headers to your DataTable. If you did Add Headers, then use `io_TransactionData.Rows(in_TransactionNumber)`

Finally, if TransactionNumber is greater than the number of rows in our TransactionData, then we have reached the end of data and should set `out_TransactionItem` to `Nothing`. The GetTransactionData workflow should appear as follows:



Processing the Data

Navigate to the Process workflow. The first thing we will have to do is attach the Internet Explorer browser opened previously. Manually open Internet Explorer and go to https://google .com. Then use the *Attach Browser* activity and indicate the browser window just opened.

Note: Make sure to indicate only the area inside the browser, do not include the url bar.

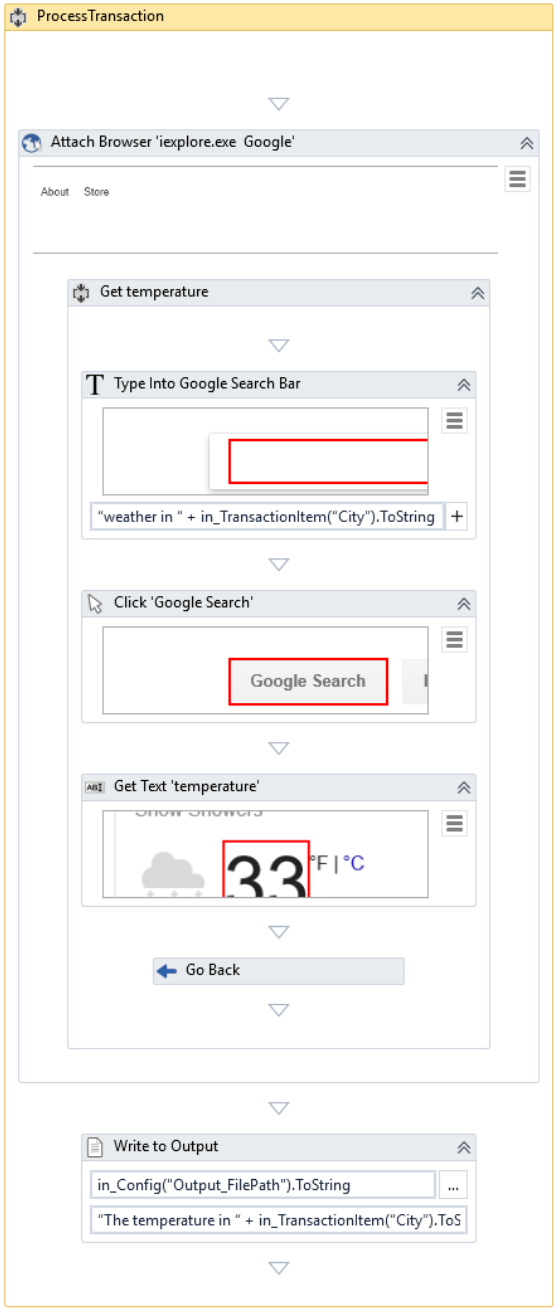
Within the *Attach Browser*:

1. Add a *Type Into activity* and indicate the Google search bar. Make sure to set it to Simulate Type and Empty Field. Then set the input text to:
   `"weather in " + in_TransactionItem("City").ToString + ", " + in_TransactionItem("State").ToString`
2. Add a *Click* activity and indicate the "Google Search" button. Make sure to set Simulate Click.
3. Once you see the search results page, use a *Get Text* activity and indicate the temperature. Store the retrieved text in a variable of type String called Temperature. Make sure this variable is in the outermost scope of the workflow (ProcessTransaction)
4. Add a *Go Back* activity to return the browser to the main Google Search page.

Following the *Attach Browser*:

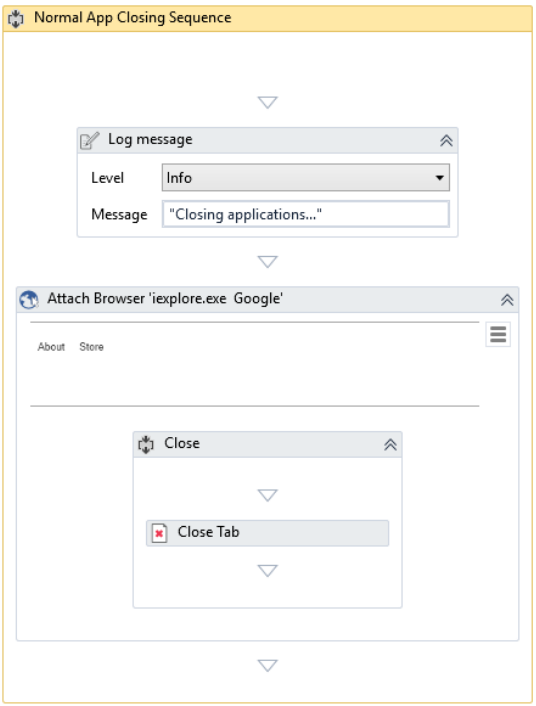
1. Add an *Append Line* activity with `in_Config("Output_FilePath").ToString` as the filename with the following text value:
   `"The temperature in " + in_TransactionItem("City").ToString + ", " + in_TransactionItem("State").ToString + " is " + Temperature + " degrees."`

The Process workflow should appear as follows:



End Process

The last step is to fill out the CloseAllApplications workflow. In the workflow, reattach the Internet Explorer browser on the main Google Search page using an *Attach Browser* activity. Finally, within the *Attach Brower*, add a *Close Tab* activity. CloseAllApplications should appear as follows:



Preparing for Production

One major responsibility of a developer is to design with production in mind. Thus, for this process, the following questions should be considered:

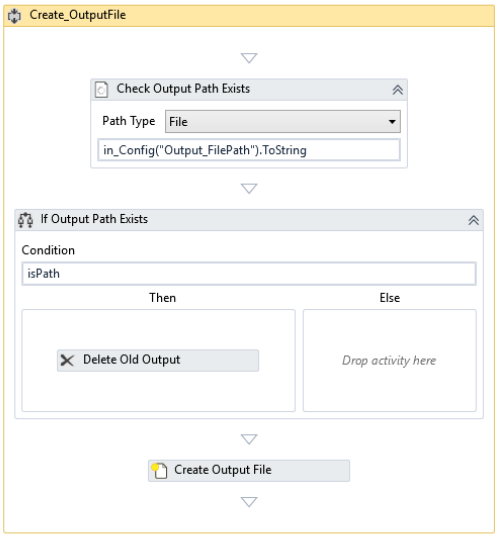
1. Would this still work if I increased in the input data?
2. Would this still work if I run this more than once?

The process as-is should still work if we increased the amount of input data; however, because we used an *Append Line* activity to write our data to output, future runs of this process will keep appending to the same file. This is not the desired functionality. So, to prevent this from happening we must make a clean output file. To do this, create a sequence workflow called Create_OutputFile.

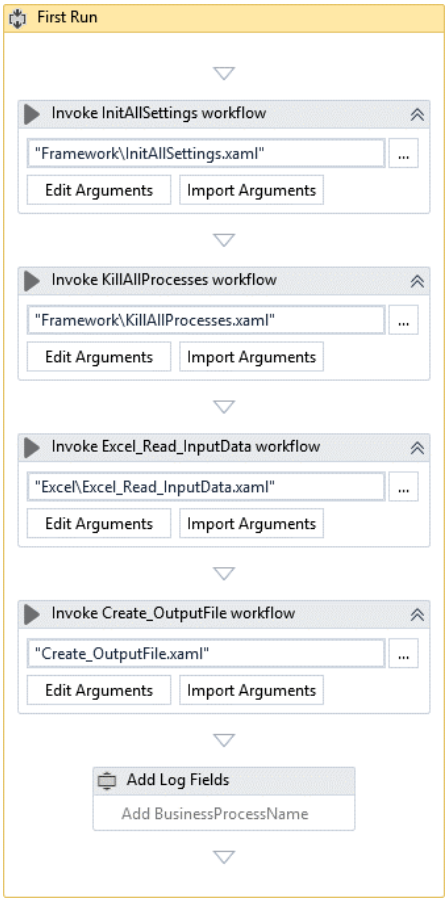
Note: .txt files are not inherently attached to an application/system. This is why we did not follow the traditional naming convention of ApplicationName_Action_Object and why we did not place the workflow in a folder.

In the Create_OutputFile workflow first we need to check if an output file already exists at the file path listed in Config. First, make an in argument called `in_Config` of type Dictionary<String, Object>. Next, add a *Path Exists* activity and set the path to `in_Config("Output_FilePath").ToString`. Store the resulting output in a variable of type Boolean called IsPath.

Then, add an *If Statement* activity and set the Condition to IsPath. In the 'Then' block, add a *Delete* activity and set the path to the same value as the *Path Exists* activity. Next, add a *Create File* activity and use the same path as before. The Create_OutputFile workflow should appear as follows:



The last step is then to invoke the Create_OutputFile workflow in Init. Navigate to Main > Init > If first run… > First Run. Under the 'Invoke Exel_Read_InputData workflow' activity, invoke the Create_OutputFile workflow. Don't forget to pass in the necessary arguments. Main > Init > If first run… > First Run should now appear as follows:



Code Cleanup

Congratulations! You have completed the initial programming of the Mini-Bot. Don't forget to make your code compliant with best practices. This includes:

- Naming all activities appropriately
- Adding annotations to the top of each workflow with: description, pre-conditions, post-actions
- Adding annotations on any activities that require additional explanation
- Removing any unneeded variables and arguments

Additionally, don't forget to debug your code. While debugging, make sure you try to mess up the code. Try:

- Closing Internet Explorer mid-process
- Changing all the values in the Config file
- Inserting bad data into the input file

Does the process still act the way you want? If not, keep working!

Workflows

Filename	Variables	In Arguments	Out Arguments
Main	TransactionItem: *DataRow* SystemError: *Exception* BusinessRuleException: *Exception* TransactionNumber: *Int32* Config: *Dictionary<String, Object>* TransactionData: *DataTable*		
InitAllSettings		in_ConfigFile: *String* in_ConfigSheets: *String[]*	out_Config: *Dictionary<String, Object>*
KillAllProcesses			
InitAllApplications		in_Config: *Dictionary<String, Object>*	
Exel_Read_InputData		in_Config: *Dictionary<String, Object>*	out_TransactionData: *DataTable*
GetTransactionData		in_Config: *Dictionary<String, Object>* in_TransactionNumber: *Int32* io_TransactionData: *DataTable*	out_TransactionItem: *DataRow* io_TransactionData: *DataTable*
Process	Temperature: *String*	in_Config: *Dictionary<String, Object>* in_TransactionItem: *DataRow*	
Create_OutputFile	IsPath: *Boolean*	in_Config: *Dictionary<String, Object>*	
CloseAllApplications			