UiPath REFramework Manual

Author: Mihai Dunareanu
Peer reviewer: Andrei Cioboata
Revision 1.0

Table of contents

About the framework and its purpose

The framework is meant to be a template that helps the user design processes that offer, at a barebones minimum, a way to store, read, and easily modify project configuration data, a robust exception handling scheme and event logging for all exceptions and relevant transaction information.

Because logs generated by each process are a vital component of its report generation, the framework logs messages at each relevant step toward solving a business transaction and sends those logs to the *Orchestrator server*. This is turn can be connected to the *ELK stack (Elasticsearch, logstash, kibana platform)* which enables data storage and countless ways of representing the data.

When we build tools, we try to first define their purpose and, in this scenario, the purpose of our framework is to solve a collection of business transactions. Notice i did not write business process, as all but the most simple business processes are typically composed of multiple, distinct in scope and in purpose, collections of business transactions. Thus, let us henceforth call such a collection of relatable business transactions a *business process component*, a part of a complete business process.

Thus, we could define a business process component as the sum of actions by which the data needed for a set of transactions is obtained, processed, and is input into or out of an *IT resource*.

Such a component needs to be easily deployed to the machines it will run on (Orchestrator server maintains versioning and easy deployment across all runtime machines), needs to be scalable and needs to be able to communicate it's output data with external mediums so that other components of the business process may pick up the work where it left off. Such a medium could be a shared folder, a data server, ftp server, email, Orchestrator server queue e.t.c.

Understanding a business process

Take the following business process: a user has to check fuel prices using web resource 1 (external company website) weekly and update a file with the new values. Another user will then utilize web resource 2 (internal company website) to obtain information about distances traveled by vehicles in company service and correlates this information with the new costs of fuel. He then uses web resource 3 (external company website) to pay for the deliveries.

In this example, we could use three business process components:
- The first, a weekly one, would read data from resource 1 to check and update the fuel price file.
- The second would download information about distances from resource 2 and reference the values obtained by the previous sub-process to filter and further refine that data. Once done, it would save the data.
- The third component would read the information produced by process 2 use it to input data into resource 3.

This business process could, of course, be expressed as the sum of two business process components instead of three, for example by grouping sub-processes 1 and 2 together.

And, of course, it could be also broken up, for example sub-process 2 might be broken up into two further pieces, one that downloads information from resource 2 and another that reads both resource 1 and 2 information and processes it.

This technique of splitting a problem into easily definable, simple components is a great tool in solving any business process, no matter how complex.

It also helps deal with a fundamental aspect of reality, time.

To see exactly how, let's make a change to the business process described above: after the file containing fuel prices was updated a user would need to confirm the validity of the values by opening the file and signing it.

This means that before we utilize web resource 2 to obtain information we first need to wait for the fuel cost file to be signed by a supervisor.

You can easily see how, If we implemented this using 3 components, the change would be small, as we would need to only add one check in subprocess number two: is the file signed. If it is, proceed. If it is not, exit and try again later. This also means that the robot is free to process other tasks.

If, on the other hand, we had grouped components 1 and 2 together, we would have to needlessly refetch data or implement an additional check on the fuel file, and this is obviously not desirable.

We must also acknowledge that business practices do change over time, making small changes like this one likely over long periods of time.

Introduction

About state machines

As you know, UiPath Studio has 3 types of data flow representations: sequence, flowchart and state machine.

While the framework does contain all 3 data flow representations, we chose the state machine for the main body of the program because it provided a cleaner solution to representing our desired dataflow.

This is how wikipedia defines a finite state machine:
"A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition."

Basic rules when using a state machine:
- Since the system can be in only one state at a time, at least one transition condition from a given state to another must become true either by generating a condition in the code running inside the state, an external condition, or a combination of both.

- The transition conditions from each state must be exclusive (two transitions cannot be true at the same time, thus allowing two possible paths of exit from a state).

- Another rule that is agreed upon is that no heavy processing must be done in the Transition actions. All processing should be done inside the state.

Going back to the first chapter, the problems we needed to solve with this template were:
1. Store and read project configuration data
2. Separate *IT resource* start, usage and end
   a. For all retried *transactions*, restart the *IT resource*
3. Implement a robust exception handling and transaction retry scheme
   a. Capture exceptions by type
   b. Use exception type to retry transactions that failed with an application exception
4. Capture and transmit logging for all exceptions and relevant transaction information

Framework component functions

Table 1 shows the calling structure of the framework. That is, which workflows are called, the order in which they are called, and the State of the main state machine where you can find the workflow invoke.

Table 1 - Component call tree structure	
Component file names and locations	State where it is called
Main.xaml	
Framework\InitAllSettings.xaml	Init
Framework\KillAllProcesses.xaml	Init
Framework\InitAllApplications.xaml	Init
Framework\GetTransactionData.xaml	GetTransactionData
Process.xaml	Process
Framework\SetTransactionStatus.xaml	Process
Framework\TakeScreenshot.xaml	Process
Framework\CloseAllApplications.xaml	Process
Framework\KillAllProcesses.xaml	Process
Framework\CloseAllApplications.xaml	End Program
Framework\KillAllProcesses.xaml	End Program

Aside from these, there are additional workflows included but not called by default. You can read about them in the "Additional functions" chapter.

Global Variables

The global variables are those variables whose scope is the main program, or main workflow. They can be found in the main.xaml workflow file, by first clicking anywhere inside the main state machine and then clicking the variables pane. Table 2 is a list of the project's global variables.

These are used to store information that will be available throughout the runtime of the process. It is important to understand where each variable is written and where it is read.

The red cell background represents workflows in which the variable is written and the green cell background workflows in which it is read.

Table 2 - Global variables table			
Name	Data type	Is written in workflows	Is read in workflows
TransactionItem	*QueueItem*	GetTransactionData.xaml	Process.xaml SetTransactionStatus.xaml
TransactionData		GetTransactionData.xaml	GetTransactionData.xaml
SystemError	Exception	Main.xaml	Main.xlsx SetTransactionStatus.xaml
BusinessRuleException	BusinessRuleException	Main.xaml	Main.xlsx SetTransactionStatus.xaml
TransactionNumber	Int32	SetTransactionStatus.xaml	GetTransactionData.xaml
Config	Dictionary(x:String, x:Object)	InitAllSettings.xaml	InitAllApplications.xaml GetTransactionData.xaml Process.xaml SetTransactionStatus.xaml
RetryNumber	Int32	SetTransactionStatus.xaml	SetTransactionStatus.xaml
TransactionID	string	GetTransactionData.xaml	SetTransactionStatus.xaml
TransactionField1	string	GetTransactionData.xaml	SetTransactionStatus.xaml
TransactionField2	string	GetTransactionData.xaml	SetTransactionStatus.xaml

Init State

InitAllSettings.xaml workflow

This workflow outputs a settings Dictionary with key/value pairs to be used in the project. Settings are read from local config file then fetched from Orchestrator assets. Assets will overwrite the config file settings

Table 3 - InitAllSettings.xaml Arguments and values		
dataType and Name	Argument Type	Values
String: in_ConfigFile	Input	"Data\Config.xlsx"
String[]: in_ConfigSheets	Input	{"Settings", "Constants"}
Dictionary(x:String, x:Object): out_Config	Output	Config

InitAllApplications.xaml workflow

Description: Open and initialize application as needed.
Pre Condition: N/A
Post Condition: Applications opened

Table 4 - InitAllApplications.xaml Arguments and values		
dataType and Name	Argument Type	Values
String: in_Config	Input	Config

Init Transitions

At the end of the Init State we should have read our configuration file into the dictionary Config, a global variable, cleaned the working environment by calling the KillAllApplications.xaml workflow only during startup, and initialised all the applications we will work with.

Table 5 - Init Transitions			
Name	Condition	Transition to State	Description
SystemError	SystemError isNot Nothing	End Process	If we have an application exception during the initialisation phase than we lack vital information to begin the process. That is why we end by going to the End Process State
Success	SystemError is Nothing	Get Transaction Data	If during initialisation we have no error than Get Transaction Data.

Get Transaction Data State

GetTransactionData.xaml workflow

Description: Get data from spreadsheets, databases, email, web API or UiPath server queues. If no new data, set out_TransactionItem to Nothing.

For a linear process (not repetitive), set out_TransactionItem only for in_TransactionNumber 1 - first and only transaction.

If the process is repeatable, set io_TransactionData once for in_TransactionNumber 1, then assign a new out_TransactionItem by using the in_TransactionNumber to index io_TransactionData. Be aware that, at the end of the io_TransactionData collection, it is your responsibility to set out_TransactionItem to Nothing, thus ending the process.

Table 6 - GetTransactionData.xaml Arguments and Values		
dataType and Name	Argument Type	Values
Int32: in_TransactionNumber	Input	TransactionNumber
Dictionary(x:String, x:Object): in_Config	Input	Config
QueueItem: out_TransactionItem	Output	TransactionItem
Datatable: io_TransactionData	Input/Output	TransactionData
String: out_TransactionID	Output	TransactionID
String: out_TransactionField1	Output	TransactionField1
String: out_TransactionField2	Output	TransactionField2

Get Transaction Data Transitions

From the GetTransactionData state we have two possible outcomes. The first is that we have obtained new transaction data in TransactionItem variable and so we move on to the Process Transaction state. The other outcome is that either we have exhausted our data collection, and, as a consequence of this, we have set the TransactionItem variable to Nothing or that we get an Application Exception while processing GetTransactionData.xaml, in which case we cannot get Data. This error causes us to go to the End Process State.

Table 7 - Get Transaction Data Transitions			
Name	Condition	Transition to State	Description
No Data	TransactionItem is Nothing	End Process	If TransactionItem is Nothing than we are at the end of our data collection, go to End Process.
New Transaction	TransactionItem isNot Nothing	Process Transaction	If TransactionItem contains data, process it.

Process Transaction State

Process.xaml workflow

In this file all other process specific files will be invoked. If an application exception occurs, the current transaction can be retried. If a BRE is thrown, the transaction will be skipped. Can be a flowchart or sequence. If the process is simple, the developer should split the process into subprocesses and call them, one at a time, in the Process.xaml workflow.

Table 8 - Process.xaml Arguments and values		
dataType and Name	Argument Type	Values
QueueItem: in_TransactionItem	Input	TransactionItem
Dictionary(x:String, x:Object): in_Config	Input	Config

SetTransactionStatus.xaml workflow

This workflow sets the TransactionStatus and Logs that status and details in extra Logging Fields.

The flowchart branches out into the three possible Transaction Statuses: Success, Business Exception and Application Exception.

Each branch analyzes the type of content of TransactionItem. If its not empty and is a QueueItem, then it means we are using a Orchestrator queue, so we must call the "Set Transaction Status" activity to inform Orchestrator about the outcome of our transaction. If TransactionItem is not a QueueItem, we can skip passing it and the "Set Transaction Status" activity will not be triggered.

After that we log the result of the transaction within custom log fields to make it easier to search for within results.

This workflow is also where incrementing of the io_TransactionNumber variable takes place. If we have an application exception and our MaxRetryNumber has not been reached, we increment the io_RetryNumber variable and not the io_TransactionNumber variables. This is done in the Robot Retry flowchart, which manages the retry mechanism of the framework and which is part of the "Handle System Error" sequence.

Table 9 - SetTransactionStatus.xaml Arguments and values		
dataType and Name	Argument Type	Values
Dictionary(x:String, x:Object): in_Config	Input	Config
Exception: in_SystemError	Input	SystemError
BusinessRuleException: in_BusinessRuleException	Input	BusinessRuleException
QueueItem: in_TransactionItem	Input	TransactionItem
Int32: io_RetryNumber	Input/Output	RetryNumber
Int32: io_TransactionNumber	Input/Output	TransactionNumber
String: in_TransactionField1	Input	TransactionField1

String: in_TransactionField2	Input	TransactionField2
String: in_TransactionID	Input	TransactionFieldID

TakeScreenshot.xaml workflow

Usage: Set in_Folder to the folder Name where you want to save the screenshot. Alternatively, supply the full path including file name in io_FilePath. Description: This workflow captures a screenshot and logs it's name and location. It then saves it. If io_FilePath is empty, it will try to save the picture in in_Folder. It uses .png extension.

Table 10 - TakeScreenshot.xaml Arguments and Values		
dataType and Name	Argument Type	Values
String: in_Folder	Input	in_Config("ExScreenshotsFolderPath").ToString
String: io_FilePath	InputOutput	

CloseAllApplications.xaml workflow

Here all working applications will be soft closed.
Pre Condition: N/A
Post Condition: Applications closed

KillAllProcesses.xaml workflow

Here all working processes will be killed
Pre Condition: N/A
Post Condition: N/A

Process Transaction Transitions

The Process Transaction State is where the processing work for all transactions takes place. After the Process.xaml file is executed, we look for an exception having been generated (either Business Rule or Application). In case no exception was caught, it means we were successful.

The SetTransactionStatus.xaml workflow manages both the logging of the Process.xaml output, as well as the management of the next transaction or the retrying of the current one. This workflow is where TransactionNumber and RetryNumber are written, allowing for automatic retry in case of an Application Exception.

Table 11 - Process Transaction Transitions			
Name	**Condition**	**Transition to State**	**Description**
Success	BusinessRuleException is Nothing AND SystemError is Nothing	Get Transaction Data	If we have a Business Rule Exception we log it and go to the next transaction.
Rule Exception	BusinessRuleException isNot Nothing	Get Transaction Data	If we have a business rule exception we log it and move to the next transaction by going to the Get Transaction Data State.
Error	SystemError isNot Nothing	Init	If we have an Application Exception we close all programs, kill them if they fail to close, take a screenshot at the moment the exception happened, and go to Init, where we will reinitialize our working environment and begin anew from the transaction that failed (retrying until we have reached the maximum retry limit)

End Process State

CloseAllApplications.xaml workflow

Here all working applications will be soft closed.
Pre Condition: N/A
Post Condition: Applications closed.

KillAllProcesses.xaml workflow

Here all working processes will be killed.
Pre Condition: N/A
Post Condition: N/A

End Process Transitions

This is the final state, out of which there are no transitions.

Additional Functions

Aside from the functions above, we included a useful workflow that will make credential management easy to implement and secure.

GetAppCredentials.xaml workflow

Usage: Change in_Credential to a previously created Orchestrator asset or a Windows credential and use outputs out_Username and out_Password.

Description: This workflow securely fetches or creates and uses a set of credentials defined at it's input. It first tries to fetch them from Orchestrator. Failing that, it tries to fetch them from the Windows credential manager. If they do not exist, it creates them and then outputs them.

Table 12 - GetAppCredentials.xaml Arguments and values		
dataType and Name	Argument Type	Values
String: in_Credential	Input	"TestRobot-Credential"
String: out_Username	Output	
SecureString: out_Password	Output	

Logging

Log messages are very important to any business process design as they offer a report of what has happened.

As previously stated, log messages are composed of multiple log fields, each with corresponding values. Logs are automatically generated by the robot when important events happen, but also by the developer using a "Log Message" activity, and are pushed to the Orchestrator server, which implements a component that will further push these logs to the Elasticsearch database.

Logged Messages

The following is a list of all the message logs within the framework, the places where the corresponding "Log message" activity is called, the message and the level of the log (info, warn, error, fatal).

Table 13 - Message logs		
Message	Workflow	Log Level
Stop process requested	Main.xaml	Info
Config("LogMessage_GetTransactionDataError").ToString+TransactionNumber.ToString+ ". "+exception.Message+" at Source: "+exception.Source	Main.xaml	Fatal
"SetTransactionStatus.xaml failed: "+exception.Message+" at Source: "+exception.Source	Main.xaml	Fatal
Config("LogMessage_GetTransactionData").ToString+TransactionNumber.ToString	Main.xaml	Info
"Applications failed to close normally. "+exception.Message+" at Source: "+exception.Source	Main.xaml	Warn
Process finished due to no more transaction data	Main.xaml	Info
"System error at initialization: " + SystemError.Message + " at Source: " + SystemError.Source	Main.xaml	Fatal
"Loading asset " + row("Asset").ToString + " failed: " + exception.Message	Framework\InitAllSettings.xaml	Warn
No assets defined for the process	Framework\InitAllSettings.xaml	Trace
Opening applications...	Framework\InitAllApplications.xaml	Info
in_Config("LogMessage_Success").ToString	Framework\SetTransactionStatus.xaml	Info
in_Config("LogMessage_BusinessRuleException").ToString + in_BusinessRuleException.Message	Framework\SetTransactionStatus.xaml	Error

in_Config("LogMessage_ApplicationException").ToString+" Max number of retries reached. "+in_SystemError.Message+" at Source: "+in_SystemError.Source	Framework\SetTransactionStatus.xaml	Error
in_Config("LogMessage_ApplicationException").ToString+" Retry: "+io_RetryNumber.ToString+". "+in_SystemError.Message+" at Source: "+in_SystemError.Source	Framework\SetTransactionStatus.xaml	Warn
in_Config("LogMessage_ApplicationException").ToString+in_SystemError.Message+" at Source: "+in_SystemError.Source	Framework\SetTransactionStatus.xaml	Error
"Take screenshot failed with error: "+exception.Message+" at Source: "+exception.Source	Framework\SetTransactionStatus.xaml	Warn
"CloseAllApplications failed. "+exception.Message+" at Source: "+exception.Source	Framework\SetTransactionStatus.xaml	Warn
"KillAllProcesses failed. "+exception.Message+" at Source: "+exception.Source	Framework\SetTransactionStatus.xaml	Warn
"Screenshot saved at: "+io_FilePath	Framework\TakeScreenshot.xaml	Info
Closing applications...	Framework\CloseAllApplications.xaml	Info
Killing processes...	Framework\KillAllProcesses.xaml	Info

You can see that many of the messages of the logs are made up of a concatenation (the + sign) between strings stored in variables and static strings.

Let's take one such example and break down its meaning. From there, every other log follows the same logic. The message is the following:
in_Config("LogMessage_ApplicationException").ToString+" Retry: "+io_RetryNumber.ToString+". "+in_SystemError.Message+" at Source: "+in_SystemError.Source

The first part of the message, *in_Config("LogMessage_ApplicationException").ToString*, is read from the Config dictionary, which enables easy modification if it is required. It is located in the Constants sheet of the Config excel file, and it's content at the moment of writing this is "System exception."
Next we append the constant string *" Retry: "* to which we append the value of the io_RetryNumber, that is the retry we have reached.
Next we append the *in_SystemError* message and source, as they will shows where the exception occurred and what it's message is.

As you can glean from the explanation of that single log message, it is composed of both static and dynamic parts which are concatenated to form a whole.

Custom Log Fields

Since Elasticsearch is a NO-SQL type database (not relational), we want to have the ability to group logs based on certain criteria. Those criteria will be additional log fields that we have added throughout the framework.

Most of these you need only know about and not modify, while some of them require the developer to modify the values written in those fields. In the table below is a list of the log fields added to the framework, their values, whether or not a developer implementing using the framework needs to change these values and the location, in the program, where they are added.

Table 14 - Custom log fields			
Field Name	Values	Value Change required	Location field is added
logF_BusinessProcessName	"Framework"	Yes	Main.xaml, Init State
logF_TransactionStatus	"Success" "BusinessException" "ApplicationException"	No	SetTransactionStatus.xaml, Success branch SetTransactionStatus.xaml, Business exception branch SetTransactionStatus.xaml, Application exception branch
logF_TransactionNumber	io_TransactionNumber .ToString	No	SetTransactionStatus.xaml, Success branch SetTransactionStatus.xaml, Business exception branch SetTransactionStatus.xaml, Application exception branch
logF_TransactionID	in_TransactionID	Yes	SetTransactionStatus.xaml, Success branch SetTransactionStatus.xaml, Business exception branch SetTransactionStatus.xaml, Application exception branch
logF_TransactionField1	in_TransactionField1	Yes	SetTransactionStatus.xaml, Success branch SetTransactionStatus.xaml, Business exception branch SetTransactionStatus.xaml, Application exception branch
logF_TransactionField2	in_TransactionField2	Yes	SetTransactionStatus.xaml, Success branch SetTransactionStatus.xaml, Business exception branch SetTransactionStatus.xaml, Application exception branch

Field Descriptions and explanations

1. logF_BusinessProcessName

This fields holds the name of the business process.

It's purpose is to group more than one business component into the same dashboard visualisation within Elasticsearch. This is important when a business process and made up of multiple business components, and so this field provides a way to group disjointed pieces of data. As you can see from table 13, it's value needs to be changed. Let's say you have a business process made up of 3 business components. The business process is named "Invoice Management". What you need to do is open each framework built component and put "Invoice Management" as the value of this field.

2. logF_TransactionStatus

Holding the status of the transaction, this log does not need to be changed. You will recall that by passing the global variables named BusinessRuleException and SystemError, holding the exception content, to SetTransactionStatus.xaml, we know exactly what the outcome of the transaction was and we can populate this field.

3. logF_TransactionNumber

The value for this log field is the number of the transaction index, TransactionNumber. As such and because this variable is managed by the system, you do not need to modify its value.

4. logF_TransactionID

The value is that of the variable TransactionID, coming in from the global variables as an input argument to the SetTransactionStatus.xaml workflow. This variable is written in the GetTransactionData.xaml workflow. In other words, once we obtain our new Transaction Item, we should choose an identifier for it. This should be unique, since we will use the value of this field to display transaction outcomes for each different transaction.

5. logF_TransactionField1

The value is that of the variable TransactionField1, coming in from the global variables as an input argument to the SetTransactionStatus.xaml workflow. This variable is written in the GetTransactionData.xaml workflow. In other words, once we obtain our new Transaction Item, we can add additional information regarding it. A single field, logF_TransactionID, might not be enough.

6. logF_TransactionField2

The value is that of the variable TransactionField2, coming in from the global variables as an input argument to the SetTransactionStatus.xaml workflow. This variable is written in the GetTransactionData.xaml workflow. In other words, once we obtain our new Transaction Item, we can add additional information regarding it because a single field, logF_TransactionID, might not be enough.

To wrap up this chapter, we remind you that you need not worry about the fields being logged, with the exception of logF_BusinessProcessName field, where you assign the name of your process, and the transaction specific fields (logF_TransactionID, logF_TransactionField1, logF_TransactionField2), where you should write identifying information about the transaction that you will shortly be processing.

Needles to say, should you require more than three items of information about any particular transactions to be logged, it is just a matter of creating the extra variables in the main program, in GetTransactionData.xaml and in SetTransactionStatus.xaml workflows, and updating the "Add Log Fields" and "Remove Log Fields" activities found within the SetTransactionStatus.xaml with the names of the new fields and the values coming in through the variables you just declared.

Getting started, examples

Deploying the framework

To deploy the framework, follow the steps described below.

- Step 1: Copy its folder to your project location and rename it to represent your project name.
- Step 2: Go into the project folder and, using any text application such as Notepad, open the project.json file. Write the project name you defined in step 1 into the "id" field. Write a project description into the "description" field. Save and close the file.
- Step 3: Open Main.xaml, navigate to the Init State, and change the value of the logF_BusinessProcessName field from the default "Framework" to your business process name.

Defining the scope of the process component and preparing the framework

The first thing to do is to choose data types for the global variables TransactionItem and TransactionData. Remember that TransactionItem stores the data required to complete a single transaction. As such, TransactionData will have to be a collection, list, datatable e.t.c. containing a "collection" of TransactionItems. The framework will then use TransactionNumber as the index that will fetch a new TransactionItem from Transaction Data.

The next step is to check the workflows in which these variables are passed. We will need to modify their data types both in the main.xaml workflow and in any other flow where it is passed as an argument.

- Step 1: Change the data types of TransactionItem and TransactionData in the main program.

- Step 2: Looking at Table 2 - Global variables table, we can see that both variables are passed into GetTransactionData.xaml, Process.xaml and SetTransactionStatus.xaml workflows.

- Step 3: Open GetTransactionData.xaml and Process.xaml and change the type of the arguments to what we decided we need. Save and quit the workflows.

- Step 4: Using Table 1 - Component call tree structure, find where the GetTransactionData.xaml and Process.xaml are called, in Main.xaml. Go to the point of

calling and, for each workflow, click import arguments. The new argument types we have saved in step 3 will show up. In the values section, pass the variables with the changed type from main (Step 1).

- Step 5: You do not need to make the argument change for the SetTransactionItem.xaml workflow, but if you do not select a QueueItem data type for TransactionItem, delete it from the values field and leave that field empty, or pass the null pointer, Nothing.

You should now have a framework that is setup according to your needs.

When developing, follow the following simple rules:
- Always open your applications in InitAllApplications.xaml workflow.
- Always close your applications in CloseAllApplications.xaml workflow.
- Always kill your applications in the KillAllApplications.xaml workflow.
- TransactionNumber is the index that should be used to loop through TransactionData and obtain our new TransactionItem. The looping happens between the Get Transaction Data State and the Process State, and the system manages the incrementing of the index. All the developer needs to do is use it to fetch a new Item.
- The process ends when TransactionItem becomes Nothing, so it's the developer's responsability to assign the null pointer, Nothing, to the TransactionItem at the end of the process.

Usage example 1

Changes to GetTransactionData.xaml

In case your TransactionItem is contained in a bigger data structure, as is the case in Figure 1, where TransactionData is a datatable (result of reading an excel file into memory) you will need to read TransactionData once and then use TransactionNumber, which holds the index of the current transaction, to fetch it's data.

In Figure 1, in the first transaction, we read the whole excel file and pass it to the global variable TransactionData, which is a datatable. In this case, our TransactionItem will be a datarow, a subset of our whole data.
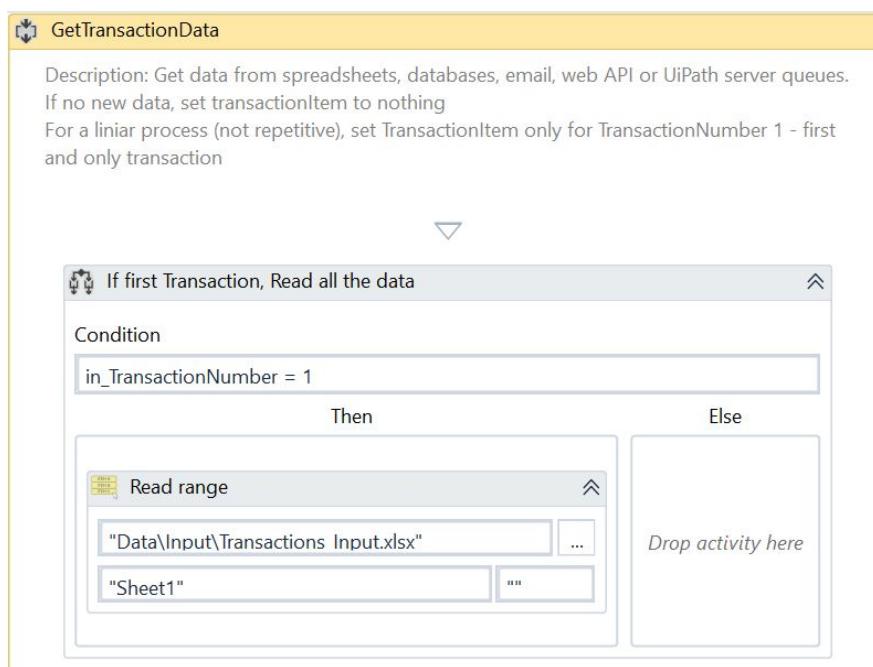

Figure 1 - read TransactionData once and output it to Global variables

We then need to use the index, TransactionNumber, to obtain our TransactionItem.

As a side note, we could have used a for each row activity to read the datarows of our datatable one by one, but we need to use the TransactionNumber index to remember what transaction we processed, and it is precisely this that gives us the ability to retry a transaction by simply not incrementing the index..

So, in figure 2, we use an if to define our loop stop condition. Since TransactionNumber is incremented by the framework we can compare it to the number of rows in the datatable. If it

has become greater than the number of rows, we need to stop our loop. In Table 7 - Get Transaction Data Transitions, we see that the transition we need to go through to end up in the End Process State is "TransactionItem is Nothing", and so, if we have run out of rows, we set TransactionItem to Nothing.

If we have not, we set *out_TransactionItem = io_TransactionData.Rows(in_TransactionNumber - 1)*.

We use *TransactionNumber - 1* because it's initial value is 1, and the index of the rows start at 0.
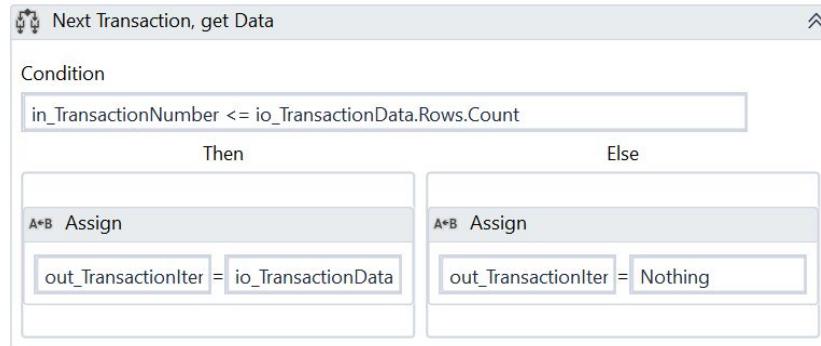

Figure 2 - while we still have rows, read the current one based on

We can see that, as per Table 2 - Global variables table and Figure 3 - Argument list for GetTransactionData.xaml, these variables are passed into the global scope.

Name	Direction	Argument type	Default value
in_TransactionNumber	In	Int32	Enter a VB expression
in_Config	In	Dictionary<String,Object>	Enter a VB expression
out_TransactionItem	Out	DataRow	Default value not supported
out_TransactionField1	Out	String	Default value not supported
out_TransactionField2	Out	String	Default value not supported
io_TransactionData	In/Out	DataTable	Default value not supported
out_TransactionID	Out	String	Default value not supported
Create Argument			

Figure 3 - Argument list for GetTransactionData.xaml

Next, we assign values to the Log Field variables that will carry these values to the SetTransactionStatus.xaml file. Choose one or multiple fields from out_TransactionItem that best represents identifying information for the transaction. For example:
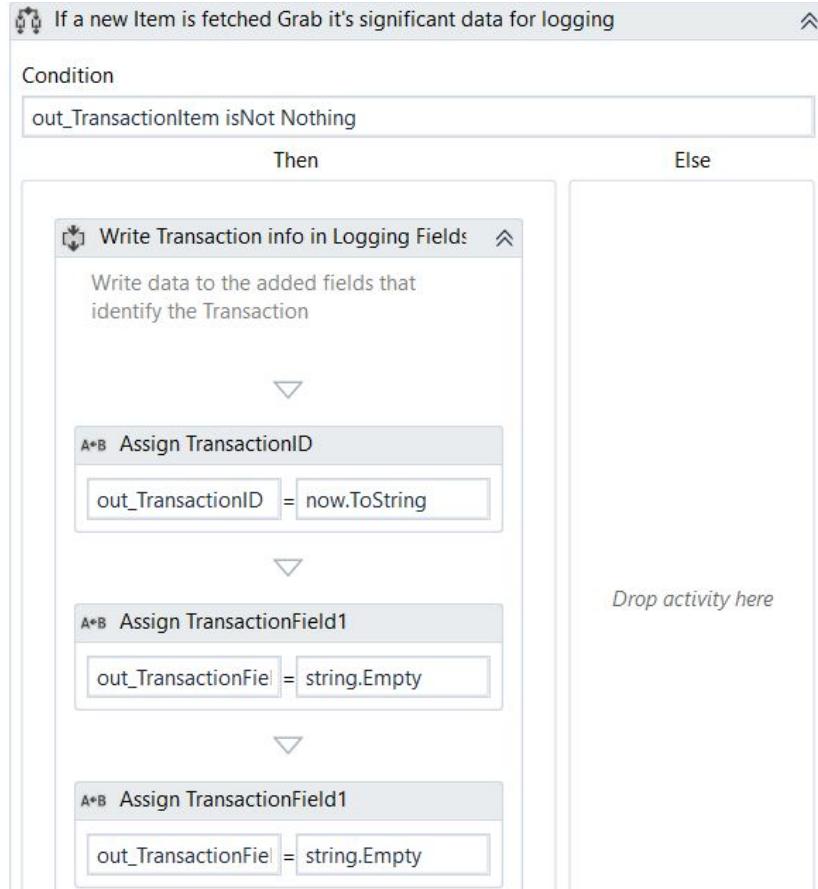*out_TransactionID=out_TransactionItem.Item("Identifier Column name").ToString*

Figure 4 - Transaction Log field value assignments in GetTransactionData.xaml

Changes to Process.xaml

Add the steps that take the data for a single Transaction, stored in the in_TransactionItem variable, and use it to fulfil the process. As your applications are already open and your data is available, you can begin work on the process.xaml file. In our case, in_TransactionItem is of type datarow, so to get the value contained in "column named A", we write *in_TransactionItem.Item("column named A")*, or, if we know "column named A" is the fourth column counting from one, *in_TransactionItem.Item(3).*

Changes to InitAllApplications.xaml

Open all your applications, log them in and set up your environment. Modify the "Log message" activity with information about what applications you are working with.

Changes to CloseAllApplications.xaml

Log out, close all your applications. Modify the "Log message" activity with information about what applications you are working with.

Changes to KillAllApplications.xaml

Kill all applications, in case one of them is not responding and cannot be closed when invoking CloseAllApplications.xaml, they will be killed. Modify the "Log message" activity with information about what applications you are working with.

Usage example 2

If this example the data we need for a Transaction is already obtained and is stored in an Orchestrator Queue.

Changes to GetTransactionData.xaml

Since out data is stored in an Orchestrator server queue, our TransactionItem is of type QueueItem. We simply use the Get Queue Item activity to obtain the next item. Since Orchestrator server is the one serving items from the queue, one by one, we do not need to use TransactionData to store the sum of all Transactions. And, as a consequence of that, we need not worry about using TransactionNumber as an index for TransactionData. When the queue will be empty, we will receive a null pointer, Nothing, from the Orchestrator server. This will in turn cause the program to go to the End Process State.
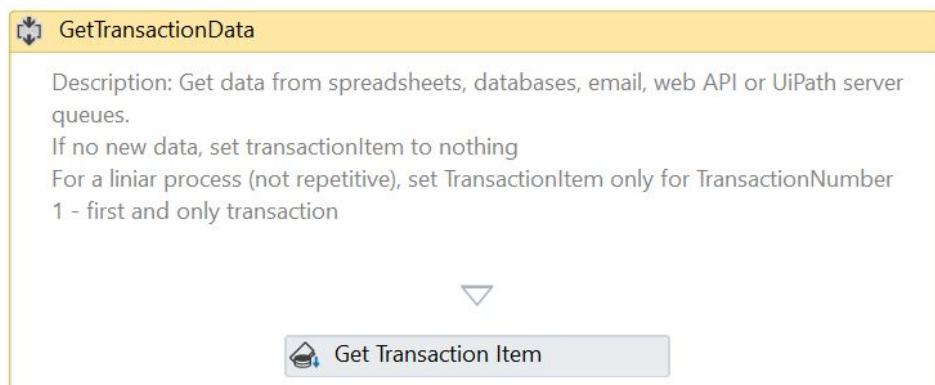

Figure 5 - Get QueueItem activity to get the next TransactionItem

Next, we assign values to the Log Field variables that will carry these values to the SetTransactionStatus.xaml file. Choose one or multiple fields from out_TransactionItem that best represents identifying information for the transaction. For example:
out_TransactionID=out_TransactionItem.Item("Identifier Column name").ToString

Changes to Process.xaml

Add the steps that take the data for a single Transaction, stored in the in_TransactionItem variable, and use it to fulfil the process. As your applications are already open and your data is available, you can begin work on the process.xaml file. In our case, in_TransactionItem is of type QueueItem, so to get the value contained in field "field named A", we write *in_TransactionItem.SpecificContent("field named A")*.

Changes to InitAllApplications.xaml

Open all your applications, log them in and set up your environment. Modify the "Log message" activity with information about what applications you are working with.

Changes to CloseAllApplications.xaml

Log out, close all your applications. Modify the "Log message" activity with information about what applications you are working with.

Changes to KillAllApplications.xaml

Kill all applications, in case one of them is not responding and cannot be closed when invoking CloseAllApplications.xaml, they will be killed. Modify the "Log message" activity with information about what applications you are working with.

Glossary of terms (marked in italics)

Business process component: a subprocess representing a distinct part of a business process.

IT resource: A source of Information technology information. Can be a program of any nature or a data file.

UiPath Orchestrator: A highly scalable server platform, enabling fast deployment, from one robot to dozens, or even hundreds. You can audit and monitor their activities, schedule all types of processes, and manage work queues. Create world-class reporting from Elasticsearch and Kibana tools. Release management, collaboration tools, centralized logging and role-based access are also supported.

ELK Stack: A platform of open source tools that enables the user to reliably and securely take data from any source, in any format, and search, analyze, and visualize it in real time.

Transaction Data: A collection of data items of similar scope, or purpose, that completely represents, from an informational point of view, a set of transactions.

Transaction Item: Data that completely represents, from an informational point of view, a single transaction. Often this is the subset of data contained in the collection, Transaction Data.

Business Rule Exception or BRE: An exception manually triggered by the developer using the "Throw" activity. The basic syntax of the activity's input is: *new UiPath.Core.BusinessRuleException("this is my reason message")*. The developer should throw BREs when he needs particular information to be available in order to continue the process, but, upon testing, discovers it is not available.

Application Exception: An exception that is triggered automatically by activities that fail, or manually by the developer when a condition pertaining to the application environment is not as was expected (for example, inputting data into a program we always expect success if we have the data needed in a correct format, but upon submitting the data we receive an error. We can capture the message and issue an application exception if retrying may solve the problem). The basic syntax of the throw activity is *new System.Exception("this is my reason message")*. There are many types of exceptions.

Workflow: The basic building block of an UiPath application. You can represent data using a sequence, a flowchart or a state machine. Can have arguments and be called from other workflows.