UiPath REFramework Manual

Revision 1.0

Table of contents

Table of contents **2**

About the framework and its purpose **3**

Understanding a business transaction **4**

Introduction **5**

About state machines 5

Framework component functions **6**

Init State 8

InitAllSettings.xaml workflow 8

InitAllApplications.xaml workflow 8

Init Transitions 9

Get Transaction Data State 10

GetTransactionData.xaml workflow 10

Get Transaction Data Transitions 11

Process Transaction State 12

Process.xaml workflow 12

SetTransactionStatus.xaml workflow 13

TakeScreenshot.xaml workflow 14

CloseAllApplications.xaml workflow 14

KillAllProcesses.xaml workflow 14

Process Transaction Transitions 15

End Process State 16

CloseAllApplications.xaml workflow 16

KillAllProcesses.xaml workflow 16

End Process Transitions 16

Getting started, examples **17**

Getting Started 17

Usage example 1 19

Changes to GetTransactionData.xaml 19

Changes to Process.xaml 20

Changes to InitAllApplications.xaml 20

Changes to CloseAllApplications.xaml 20

Changes to KillAllApplications.xaml 21

Usage example 2 21

Changes to GetTransactionData.xaml 21

Changes to Process.xaml 21

Changes to InitAllApplications.xaml 21

Changes to CloseAllApplications.xaml 21

Changes to KillAllApplications.xaml 22

Glossary of terms **23**

About the framework and its purpose

The framework is meant to be a template that helps the user design processes that offer, at it's barebones minimum, a way to store and read 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)* 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 business transactions.

Thus, we could define a business transaction as the process by which an *IT resource* is input data into or out of.

Understanding a business transaction

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 processes:

- 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 process to filter and further refine that data. Once done, it would save the processed data.
- The third process would read the information produced by process 2 use it to input data into resource 3.

This business process could be expressed as two business transactions instead of 3, for example by grouping process 1 and 2 together. And, of course, it could be broken up even more, for example process 2 might be broken up into two further subprocesses, 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.

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.”

An important conclusion is that, since the system can be in only one state at a time, at least one transition condition from the state to another must become true either by generating a condition in the application running inside the state, an external condition, or a combination of both.

Another is that 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 in a 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

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/APost Condtion: 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 transactionItem to Nothing. For a liniar process (not repetitive), set TransactionItem only for TransactionNumber 1 - first and only transaction.

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
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 call the setTransactionStatus activity. After that we log the result of the Transaction within custom log fields to make it easier to search for within results. If TransactionItem is not a QueueItem, we can skip passing it and the SetTransactionStatus actvity will not try to communicate with Orchestrator!

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 filename 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 Condtion: Applications closed

KillAllProcesses.xaml workflow

Here all working processes will be killed
Pre Condition: N/A
Post Condtion: 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 of the retrying of the current now. 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 Condtion: Applications closed.

KillAllProcesses.xaml workflow

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

End Process Transitions

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

Getting started, examples

Getting Started

The first thing to do is to choose a 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.

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.

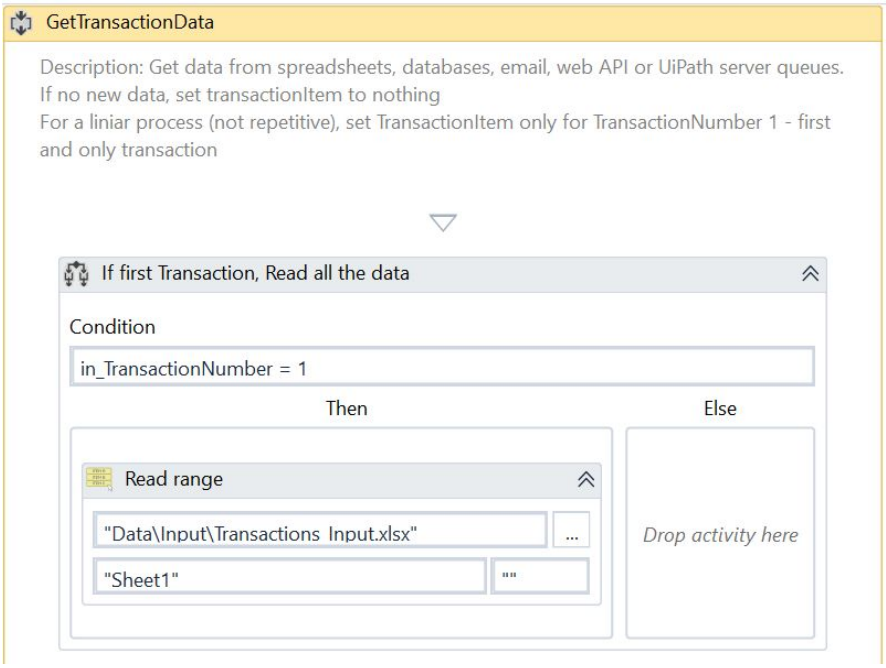

Figure 1 - read TransactionData once and output it to Global variables

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

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.

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.

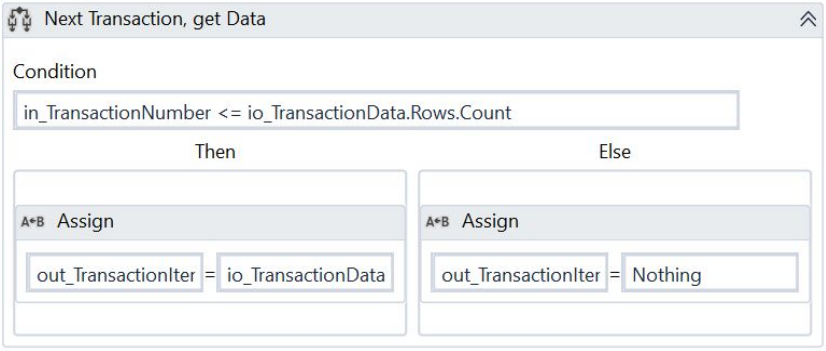

Figure 2 - while we still have rows, read the current one based on

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

Changes to Process.xaml

Add the steps that take the data for a single Transaction, stored in the TransactionItem variable, and use it to fulfil the process.

Changes to InitAllApplications.xaml

Open all your applications, log them in and set up your environment.

Changes to CloseAllApplications.xaml

Log out, close all your applications.

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.

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.

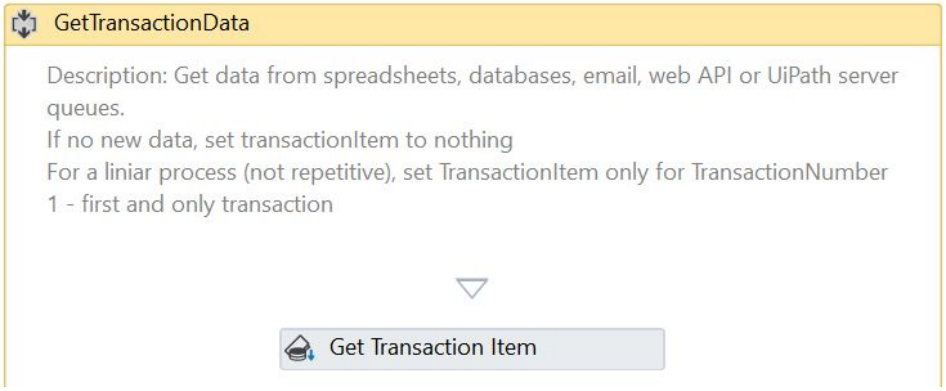

Figure 3 - Get QueueItem activity to get the next TransactionItem

Changes to Process.xaml

Add the steps that take the data for a single Transaction, stored in the TransactionItem variable, and use it to fulfil the process.

Changes to InitAllApplications.xaml

Open all your applications, log them in and set up your environment.

Changes to CloseAllApplications.xaml

Log out, close all your applications.

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.

Glossary of terms

IT resource: A source of Information technology information. Can be a program, data file
Orchestrator server:
ELK Stack:
Transaction Data:
Transaction Item:
Business Rule Exception or BRE:
Application Exception:
Workflow: