Adobe RPA Template Framework Manual

1. About the RE framework

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. 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.

2. Introduction

2.1. 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.

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.

The problems we needed to solve with this template were (example usecase)

1. Read project configuration data, and also read input file from the folder and store data to Queues.
2. Calculate the numbers given in the input sheet using calculator application and updating the result back to queues
3. Implement a robust exception handling and transaction retry schema
    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.

3. RE Framework Component functions

Table 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.

Main.xaml	**Component file name and location**	**State where it is used**
	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, End
	Framework\KillAllProcesses.xaml	process, End

4. 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. Below Table consists of list of the projects 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.

Global Variables tables			
Name	**Data Type**	**is written in workflows**	**is read in workflows**
TransactionItem	Queue item	GetTransactionData.xaml	✓ Process.xaml ✓ SetTransactionStatus.xaml
Transaction Data		GetTransactionData.xaml	✓ Get TransactionData.xaml
System Error	Exception	Main.xaml	✓ Main.xlsx ✓ SetTransactionStatus.xaml
BusinessRuleException	BusinessRuleException	Main.xaml	✓ Main.xlsx ✓ SetTransactionStatus.xaml
TransactionNumber	Int32	SetTransactionStatus.xaml	✓ Get TransactionData.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	SetTransactionStatus.xaml	✓ SetTransactionStatus.xaml

TransactionField1	string	SetTransactionStatus.xaml	✓ SetTransactionStatus.xaml
TransactionField2	string	SetTransactionStatus.xaml	✓ SetTransactionStatus.xaml

5. Init State

5.1. 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

InputAllSettings.xaml Arguments and Values		
Data Type and Name	**Argument type**	**Values**
string: in_ConfigFile	Input	Data\Config\Config-Dev.xlsx"(Based on bot deployed environment)
String[]: in_ConfigSheets	Input	{"Settings", "Constants","kafka"}
Dictionary(x:String, x:Object): out_Config	Output	config

5.2. InitAllApplications.xaml workflow

Description: Open and initialize application as needed. (Depends on the usecase, we have used only Calculator application)
Pre Condition: N/A
Post Condition: Applications opened

InitAllApplications.xamArguments and Values		
dataType and Name	**Argument Type**	**Values**
String: in_Config	Input	Config

5.3. 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 initialized all the applications we will work with.

Init Transactions			
Name	**Condition**	**Transition to state**	**Description**
System Error	System Error is not 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	System Error is nothing	Get Transaction Data	a If during initialisation we have no error than Get Transaction Data

6. Get Transaction Data State

6.1. GetTransactionData.xaml workflow

Description: Get data from spreadsheets, databases, email, web API or UiPath queues (Depends on the usecase, we have used Queues for the process)

GetTransactionData.xaml Arguments and Values		
Data Type and Name	**Argument Value**	**Values**
Int32: in_TransactionNumber	Input	Transaction Number
Dictionary(x:String, x:Object): in_Config	Input	Config
QueueItem: out_TransactionItem	Output	TransactionItem

Datatable: io_TransactionData	in/out	TransactionData
String: out_TransactionID	out	TransactionID
String: out_TransactionField1	out	TransactionField1
String: out_TransactionField2	out	TransactionField2

6.2. 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.

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 Transition	TransactionItem is Not Nothing	Process Transaction	If TransactionItem contains data, process it.

7. Process Transaction State

7.1. 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.

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

7.2. 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.

SetTransactionStatus.xaml Arguments and values		
dataType and Name	**Argument types**	**Values**
Dictionary(x:String, x:Object): in_Config	in	Config
Exception: in_SystemError	in	SystemError
BusinessRuleException: in_BusinessRuleException	in	BusinessRuleException
QueueItem: in_TransactionItem	in	TransactionItem
Int32: io_RetryNumber	in/out	RetryNumber

Int32: io_TransactionNumber	in/out	TransactionNumber
String: in_TransactionField1	in	TransactionField1
String: in_TransactionField2	in	TransactionField2
String: in_TransactionID	in	TransactionFieldID

7.3. <u>TakeScreenshot.xaml workflow</u>

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.

7.4. <u>CloseAllApplications.xaml workflow:</u>

Description: Here all working applications will be soft closed. (Depends on usecase, close safe all the applications in this xaml)
Pre Condition: N/A
Post Condition: Applications closed

7.5. <u>KillAllProcesses.xaml workflow</u>

Description: Here all working processes will be killed (Add all the dependent applications list in the config file where name should be same as application name in Taskmanager instance)
Pre Condition: N/A
Post Condition: Applications will be killed

7.6. <u>Process Transaction Transitions</u>

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.

Process Transaction Transitions			
Name	**Condition**	**Transition to state**	**Description**
Success	BusinessRuleException is Nothing AND SystemError is Nothing		If we have a Business Rule Exception we log it and go to the next transaction.
Rule Exception	BusinessRuleException isNot Nothing		f 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		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)

8. <u>End Process State</u>

8.1. <u>CloseAllApplications.xaml workflow:</u>

Description: Here all working applications will be soft closed. (Depends on usecase, close safe all the applications in this xaml)
Pre Condition: N/A
Post Condition: Applications closed

8.2. <u>KillAllProcesses.xaml workflow</u>

Description: Here all working processes will be killed (Add all the dependent applications list in the config file where name should be same as application name in Taskmanager instance)
Pre Condition: N/A
Post Condition: Applications will be killed

8.3. <u>End Process Transitions</u>

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

9. Additional Reusable Components

Aside from the functions above, we included a useful workflows that can be reused

9.1. 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.

GetAppCredentials.xaml Arguments and values		
dataType and Name	**Argument Type**	**Values**
String: in_Credential	in	Asset Name to fetch credentials from orchestrator
String: out_Username	out	
SecureString: out_Password	out	
string: In_ProcessName	in	Process name to be captured from config file
string: In_BotEnvironment	in	DEV/UAT/Prod To be captured from config file
dictionary: in_Config	in	

9.2. KafkaInsightsComponent

Usage: Kafka insights components is to insert data into DB using Kafka API's, Input all the mandatory details.

Description: Kafka componenet will try to insert data to DB using API's upon failure it will throw an exception

Kafka_Insights_Component.xaml Arguments and Values		
DataType and Name	Argument type	Values
Dictionary:In_Config	in	Config variable
in_stStatus	in	Transaction status<Success/Failure>
in_stProcessID	in	Process ID
in_stExecutionId	in	Unique reference ID of the transaction
in_stStartTime	in	Start time of the process
in_stRemarks	in	Remarks for the transactions
in_Query	in	Query should be in the form of Key Value pair (Refer Config File Kafka sheet fot more details)

10. 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 database

10.1. 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).

Message Logs		

Message	Workflow	LogLevel
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
in_Config("LogMessage_Success").ToString	Framework\InitAllApplications.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

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.

Lets 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 its 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 its 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.

10.2. Custom Log Fields

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.

Custom log fields

Field Name	Values	Value Change required	Location field is added	Description
logF_BusinessProcessName	"Framework"	yes	Main.xaml, Init State	This fields holds the name of the business process
logF_TransactionStatus	Success BusinessException ApplicationException	no	SetTransactionStatus.xaml, Success branch SetTransactionStatus.xaml, Business exception branch SetTransactionStatus.xaml, Application exception branch	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.
logF_TransactionNumber	io_TransactionNumber .ToString	no	SetTransactionStatus.xaml, Success branch SetTransactionStatus.xaml, Business exception branch SetTransactionStatus.xaml, Application exception branch	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.
logF_TransactionID	in_TransactionID	yes	SetTransactionStatus.xaml, Success branch SetTransactionStatus.xaml, Business exception branch SetTransactionStatus.xaml, Application exception branch	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.
logF_TransactionField1	in_TransactionField1	yes	SetTransactionStatus.xaml, Success branch SetTransactionStatus.xaml, Business exception branch SetTransactionStatus.xaml, Application exception branch	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.
logF_TransactionField2	in_TransactionField2	yes	SetTransactionStatus.xaml, Success branch SetTransactionStatus.xaml, Business exception branch SetTransactionStatus.xaml, Application exception branch	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.

11. Getting started, examples

11.1. Deploying the framework

To deploy the framework, follow the steps described below.

- Copy its folder to your project location and rename it to represent your projectname.
- 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.
- Open Config file from Data\Config folder and update logF_BusinessProcess Name to your business process name.

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 its the developers responsibility to assign the null pointer, Nothing, to the **TransactionItem** at the end of the process.

Sample Code that we have developed is transactional based

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

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.

<u>Changes to CloseAllApplications.xaml:</u>Log out, close all your applications. Modify the Log message activity with information about what applications you are working with.

<u>Changes to KillAllApplications.xaml:</u>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. (We have already added Reusable component to kill applications to utilize that just pass application names based on task manager to config value)

<u>Changes to Process.xaml:</u>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).**