WMP Attended REFramework

Contents

[I. About UiPath REFramework 2](#_Toc1461426)

[II. About WMP Attended REFramework (WMP REF) 2](#_Toc1461427)

[A. Purpose 2](#_Toc1461428)

[B. Overview 2](#_Toc1461429)

[III. UiPath REFramework vs WMP Attended REFramework: 2](#_Toc1461430)

[A. Dispatcher & Performer 2](#_Toc1461431)

[UIPath REFramework 2](#_Toc1461432)

[WMP Attended REF 2](#_Toc1461433)

[B. Queues and TransactionItems 3](#_Toc1461434)

[UiPath REFramework 3](#_Toc1461435)

[WMP Attended REFramework 3](#_Toc1461436)

[C. Processing Transaction Items 3](#_Toc1461437)

[D. End Process 4](#_Toc1461438)

[IV. WMP Attended Framework 4](#_Toc1461439)

[A. WMP Attended Framework (folder) 4](#_Toc1461440)

[B. Initializing the WMP Attended Framework 4](#_Toc1461441)

[Updated Config File: 4](#_Toc1461442)

[Main.Xaml 5](#_Toc1461443)

Log:

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Note** | **Date Modified** | **Modified By** |
| 1.0 | Document Creation | 2/19/19 | wchance |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# About UiPath REFramework

Link to Framework documentation

# About WMP Attended REFramework (WMP REF)

## Purpose

WMP Attended REFramework or WMP REF is a Templated Framework for building and deploying Attended or Assistive bots. It is an iteration of the UiPath REFramework, UiPath's best practices and enterprise bot framework, and includes additional framework and workflow functionality that is specific to attended bots. Finally, the WMP REF enables the logging of transactions and the overall status of the bot's execution or "Run."

## Overview

WMP Attended REFramework comes prebuilt with an updated Config file, variable types, and framework for logging transaction status. Loading of a "Transaction Queue" within the framework should be done with a new workflow file named "dispatcher" which is only run prior to capturing the first transaction, populating the TransactionData (DataTable) variable. Once the "dispatcher" has been updated and the TransactionData variable initialized, the bot uses the framework to get a single transaction's data and then execute the transaction process. Transaction data is stored in the TransactionItem (DataRow) variable and will be passed into Process.Xaml. Completion of each Transaction (success or failures) requires the bot to "Set Transaction Status" where the bot will set the status, iterate the transaction number, or retry the transaction. The WMP REF enables the bot to write the status of the transaction to a new "Transaction Log" (dt\_transactionLog). Once all transactions are complete, the bot transitions to End Process State, where it writes the Transaction Log to excel. Finally, the bot references the full transaction log (from excel) and creates updated excel data tables with views for analysis on an excel dashboard within the transactionLog file.

# UiPath REFramework vs WMP Attended REFramework:

## Dispatcher & Performer

### UIPath REFramework

UiPath REF enables bots and users to work with Orchestrator Queues to drive transactional processes - this is best suited for Unattended, Autonomous bots. UiPath provides developers with training on UiPath REFramework that stresses the use of a two-part bot process called "Dispatcher" and "Performer." This best practice allows for the "Dispatcher" Bot to load an Orchestrator Queue with transactional data and the "Performer" Bot to then utilize the Orchestrator Queue to create transactions and execute the transactional process.

### WMP Attended REF

WMP Attended REF combines the "Dispatcher" and "Performer" Bots into a single bot process. A transaction queue is created and loaded with data at the start of the Bot's process - "TransactionData". The bot then uses the GetTransactionData workflow to grab the first transaction from "TransactionData" and execute the bot's transactional process.

Note - All orchestrator related activities in the framework have been commented out and labeled for future use.

## Queues and TransactionItems

### UiPath REFramework

UiPath REF uses orchestrator queue items to drive transactional processes - the traditional UiPath REF uses the "TransactionItem" variable as type "QueueItem." When data is grabbed from a particular Orchestrator Queue using the Get Transaction Item activity, it is stored and identified as "TransactionItem" from an indexed TransactionData (DataTable Variable).

This data is passed out to main along with a TransactionNumber and additional Transaction Variables (TransactionID, TransactionField1, and TransactionField2). TransactionNumber allows for the TransactionItem to be indexed at the end of each process and iterates in order to grab the next QueueItem. All additional Transaction Variables are used for Logging Purposes in Orchestrator.

### WMP Attended REFramework

WMP Attended REF uses the TransactionData (DataTable Variable) to create a Transactional Queue within the bot's framework. This "queue" should be loaded once, during the first transactional run of the bot's process - e.g. "Init" where "Config is Nothing" or within "Get Transaction Data" where TransactionNumber = 1. Decision Needed on where to make this live

It's best practice to use a separate workflow (.Xaml) file that is specific to capturing data and creating the DataTable queue. Within this "Dispatcher" workflow, an out argument of type DataTable (out\_DataTable) should be initialized at the start of execution and then loaded with data via Scrape DataTable, Read Range, Manual Entry, etc. Within Main, The out argument should be set to TransactionData. (out\_DataTable = TransactionData). If the "Dispatcher" workflow executes correctly and tests show that it has loaded Main's TransactionData variable then the bot should be ready to execute on transactiondata

Like the traditional UiPath REF, the WMP Attended REF uses the variables named, "TransactionItem" and "TransactionNumber" to execute on the transactional process and iterate through the TransactionData, DataTable, Queue. Within the WMP REF, the TransactionItem variable is set to type DataRow, as the Transaction Queue is of type DataTable. \*Note all arguments and variables associated with TransactionItem have been updated to type DataRow throughout this framework.

In order to enable the transactional queue for execution of a transactional process, the GetTransactionData workflow file has been updated to grab TransactionItems (DataRow) from the Queue (DataTable) based on the TransactionNumber which is always equal to "1" during the first process execution (TransactionNumber is iterated on within the SetTransactionStatus workflow). Within the Get Transaction Data state, TransactionData (DataTable) must be passed in/out of GetTransactionData workflow as io\_TransactionData - this allows for iteration and for the arguments of out\_TransactionItem (and transaction variables) to be initialized and passed to main.

## Processing Transaction Items

Both UiPath REF and WMP Attended REF use the Process Transaction State to execute on a transactional process. The two workflows used within Process Transaction State include Process.Xaml and SetTransactionStatus.Xaml.

Process.Xaml - The key difference for the WMP Attended REF within Process is the use of the in/out argument, io\_dict\_statusLog. (See WMP Attended Framework).

SetTransactionStatus.Xaml - This workflow will be executed for each and every transaction. Within SetTransactionStatus, the bot will set the "status" of the transaction based on results from executing the Process Workflow - results will be "Successful", "Business Rule Exception", or "System Error". When the Bot sets the status of each transaction, the TransactionNumber will be iterated on (+1). The only exception to iterating the TransactionNumber is enabling the bot's retry mechanism (default is set to 0 retries) which results in a retry of the last transaction due to a System Error. If this mechanism is set to 0, then the TransactionNumber will iterate and the bot will set the transaction's status to "Failed"

A successful transaction means that all exceptions are equal to "nothing" and the Process resulted in completion without a "Throw." The bot will follow the true arm of "successful" for the SetTransactionStatus workflow.

A "failed" transaction is resultant from a Throw of type BusinessRuleException or SystemException during execution of the Process. If the Bot experiences a BusinesRuleException, it is considered the False argument for "success" and will then follow the true arm of the BusinessException argument in the SetTransactionStatus workflow while a SystemException will follow the false arm of the argument.

## End Process

Both UiPath REF and WMP Attended REF will result in closing of all applications per the original REF design. WMP REF differs in that the Bot will navigate directly to "End Process" following the execution of the last transaction item in the TransactionData queue. This Transition has been labeled as "Run Complete". Within the End Process State, WMP REF will also write a Transaction Status Log to Excel (see WMP Attended Framework)

# WMP Attended Framework

## WMP Attended Framework (folder)

1. The WMP REF adds an additional folder to the UiPath REFramework named, "WMP Attended Framework." Within the framework and overall state machine, the multiple workflow files associated with this folder are invoked, allowing for the creation of a Transaction Status Log. When a bot reaches the End Process State within Main, the Transaction Status Log is written to an Excel File, "transactionLog," which can be found in "Data\TransactionLog" folder
2. DepWorkflows - this folder contains workflows that are used by the Attended\_ workflows and are action based. For example, dt\_columnIndexToExcel, enables the bot to convert a column index number (e.g. 10) to a letter (e.g. J), which is used to write the range to Excel

## Initializing the WMP Attended Framework

### Updated Config File:

1. StatusLog Tab: The updated Config file includes a new tab, "statusLog." The data in this tab is similar to the settings tab, in that it will be read into a new dictionary variable called **dict\_statusLog**, which is used throughout the framework. This dictionary passes in a set of string variables that enable transaction logging and the creation of an excel based Transaction Log which is written at the end of each execution during End Process
   1. Do Not Change: In the statusLog tab do not modify A1:A15, as these variables are already "called" via assigns throughout the framework. They are used throughout main and within SetTransactionStatus. All Values have been assigned to a specific value or NA - these values will be updated within the framework via an Assign Activity
   2. Update/Modify: All other variables under Name that are associated with a process step, should be updated with the associative process step name - e.g. 01-runSharePointProcess. By adding additional variables and being descriptive with the process step names, the developer can track the bot's status throughout each step as well as the time it takes to execute the process. This also enables the developer to build in additional functionality for business rules and/or failures - instead of throwing the transaction to Main, a flowchart can be used to drive additional logic if a process step failed. The below example process highlights this use

***Process***: Search to see if a customer exists, if they do, first grab the customer's accounts, then gather any customer specific data, and finally write to excel

1. 01-customerExists --> 02-scrapeCustomerAccounts --> 03-scrapeCustomerData --> 04-writeToExcel

***Exception***: If the customer exists but contains no accounts, still write the customer's known information to excel.

1. 01-customerExist --> 02-scrapeCustomerAccounts (failed) --> 04-writeToExcel

***Using the Log***: The dict\_statusLog can drive the decision in the above example, as the process status for 02-scrapeCustomerAccounts can be set to "Failed" instead of being thrown(dict\_statusLog("02-scrapeCsutomerAccounts") = "Failed". A decision in the flowchart can then tell the bot to jump strait to 04-writeToExcel

1. Settings Tab - environment
   1. Added a new config variable, "environment." Set this variable to "production", "test", or "dev" as the config("environment") will write the dt\_transactionLog to the named tab in the excel transactionLog. For all other config variables, feel free to make updates/changes as you see fit.

### Main.Xaml

Init State

1. InitAllSettings.Xaml
   1. Arguments:
      1. Out Argument = Config
      2. Out Argument = dict\_statustLog
   2. Initializes New Argument - out\_dict\_statusLog as New Dictionary (of string, string)
      1. Dict\_statusLog is passed in via the Config file's, "statusLog" tab
      2. When importing arguments from main, out\_dict\_statusLog should be set to dict\_statusLog
   3. Update initial statusLog values for overall bot run - out\_dict\_statusLog("variable")
      1. runStatus = "In Progress"
      2. runStatus-Logged - datetime.now
      3. runStart = "Complete"
      4. runStart-Logged - datetime.now
2. Attended\_initTransactionStatusLog
   1. Arguments:
      1. In Argument = dict\_statusLog
      2. Out Argument = dt\_transactionLog
   2. Initializes out\_dt\_transactionLog by iterating through in\_dict\_statusLog's, dictionary key values, creating new columns for each value in the dict\_statusLog

Get Transaction Data State

1. Dispatcher
   1. Arguments:
      1. Out Argument out\_dt = TransactionData
   2. Conditions:
      1. Precondition: If the bot is executing in the Get Transaction State for the first time, then transactionNumber should be set to 1.
      2. Postcondition: TransactionData (DataTable) is initialized and contains data
   3. Update statusLog values - dict\_statusLog("variable")
      1. Pre
         1. dispatcher = "In Progress"
      2. Post
         1. dispatcher = "Complete"
         2. dispatcher-logged = datetime.now
         3. transactionData-Total = TransactionData.rows.count.tostring
2. Try - GetTransactionData
   1. Arguments:
      1. in\_transactionNumber = transactionNumber
      2. in\_config = config
      3. in\_rowCount = TransactionData.rows.count
      4. io\_transactionData = TransactionData
      5. out\_transactionItem = TransactionItem
      6. \*out\_transactionID, Field1, Field2 (can be removed)
   2. Conditions:
      1. Precondition: TransactionData includes one or more row of data
      2. Postcondition: TransactionItem contains a DataRow from TransactionData
   3. Update statusLog values - dict\_statusLog("variable")
      1. Pre
         1. getTransactionData = "In Progress"
      2. Post
         1. getTransactionData = "Complete"
         2. getTransactionData-Logged = datetime.now
         3. transactionStatus = "In Progress"
         4. transactionNumber = transactionNumber.toString

Process Transaction State

1. Try - Process the current Transaction Item
   1. Arguments:
      1. in\_TransactionItem = TransactionItem
      2. in\_Config = Config
      3. io\_dict\_statusLog = dict\_statusLog
   2. Conditions
      1. Precondition: TransactionItem contains transactional data and is ready for process execution
      2. Postcondition: Transaction is completed resulting in Success, BusinessRulesException, or SystemException
   3. Update statusLog values - dict\_statusLog("variable")
      1. Pre
         1. processTransaction = "In Progress"
      2. Post - Success (no throws)
         1. processTransaction = "Complete"
         2. processTransaction-Logged = datetime.now
      3. Post - BusinessRulesException
         1. processTransaction = "Failed"
         2. processTransaction-Logged = datetime.now
      4. Post - SystemException
         1. processTransaction = "Failed"
         2. processTransaction-Logged = datetime.now
2. Finally - SetTransactionStatus
   1. Arguments:
      1. in\_Config = Config
      2. in\_SystemError = SystemError
      3. in\_BusinessRulesException = BusinessRulesException
      4. in\_TransactionItem = TransactionItem
      5. io\_RetryNumber = RetryNumber
      6. io\_TransactionNumber = TransactionNumber
      7. io\_dt\_transactionLog = dt\_transactionLog
      8. io\_dict\_statusLog = dict\_statusLog
   2. Conditions:
      1. Precondition: process transaction complete or failed
      2. Postcondition: The Transaction's status is captured, the dict\_statusLog is written to dt\_transactionLog, TransactionNumber updated (+1), and the dict\_statusLog is made ready for the next transaction. OR the run is complete
   3. Within the SetTransactionStatus Workflow:
      1. Update statusLog values - dict\_statusLog("variable")
         1. transactionStatus = "Success", "Failed", "System Error"
         2. transactionLogTime = datetime.now

*If Final Transaction*

* + - 1. runStatus = "Complete", "Complete - SystemException"
      2. runStatus-Logged = datetime.now
      3. Attended\_get\_dict\_statusLog\_forTLog
         1. in\_dict\_statusLog = io\_dict\_statusLog
         2. out\_arr\_statusLog = arr\_statusLog
    1. Add Data Row
       1. Data Row Array = arr\_statusLog
       2. DataTable = io\_dt\_transactionLog
    2. Attended\_update\_dict\_statusLogValues
       1. io\_dict\_statusLog
       2. Saves run values as temporary variables, then updates all values within the dict\_statusLog to NA, finally saves the temp variables back to the associated value in the dict\_statusLog (e.g. runStart-Logged)

End Process

1. Attended\_writeTransactionStatusLog
   1. Arguments
      1. in\_dt\_transactionLog = dt\_transactionLog
      2. environment = Config("environment")
      3. out\_str\_logName\_excelTbl = str\_excelTable Name
   2. Conditions:
      1. Precondition: All transactions are complete or the run has resulted in a system error failure without retry
      2. Postcondition: the dt\_transactionLog is written to the excel file, transactionLog.xlsx in the correct tab (environment)
   3. Notes -
      1. Environment (config variable) - designates what tab you would like to write the dt\_transctionLog to in Excel. If you set environment to "test" in config, then the dt\_transactionLog will write to the "test" tab
      2. First Bot Run (relates to environment in config)
         1. Adds RunID to dt\_transactionLog as RunID = 1
         2. Writes the dt\_transactionLog to the excel tab as new, captures the range of the written log, and converts the range to a table, str\_excelTableName
      3. All Other Bot Runs
         1. Gets the table range and identifies the last row
         2. Gets the value from the last row's RunID and sets RunID for current run to RunID+1
         3. Adds RunId to dt\_transactionLog as RunID
         4. Appends dt\_transactionLog to transactionLog.xlsx
2. Attended\_build\_ExcelDashboard
   1. Arguments
   2. Conditions
      1. Precondition: Bot has written dt\_transaction log to the transactionLog.xlsx file
      2. Postcondition: Excel summary views of the dt\_transactionLog have been created tabs – SummaryByDate, SummaryToday, SummaryByRunID
   3. Notes
      1. Process Steps:
         1. Load Data + Initialization and Build DataTable for Outputs
         2. Update DataTable Values
         3. Create Summary Data Table – DateHistory
         4. Create Summary Data Table – DailyHistory
         5. Create Summary Data Table – RunID
         6. Assign variables for updating excel tables
         7. Update Excel – Excel Application Scope
         8. Write Summary tables to excel

Appendix

Write Excel Log Steps:

1. Add new columns to dt\_transactionLog for added logging capabilities - transactionTime, runTime, runDate, runID

2. Calculate Run Time for process start to transaction end as timeSpan var

3. Assign the table name in excel based on the Config("environment") - this selects the correct tab and creates a table with the tab's name in it

4. Read Cell - this checks if the excel worksheet is blank or contains a table

5. Excel application scope

\*\*1st RUN\*\*

1. If Blank - Assign runID as "1" for the first run,

2. If Blank - For Each Row - calculate the transaction time and assign transactionTime, runTime, runID, runDate to dt\_transactionLog

\*note - the bot will have to complete a for each row ever time. This step is within the If statement due to the need of assigning first run and then dynamically assigning all runs after first run

3. If Blank - get the dynamic range... passes the column and row index into a workflow and retrieves the endRange (e.g. R10)

4. If Blank - assign tableRange, Write Range to "A1" (add headers)

5. If Blank - Create Table based on tableRange

\*\*ALL FUTURE RUNS\*\* (unless environment changes)

1. Get table range from str\_excelTableName = str\_existingRange

2. Read range from tableRange

3. Get last rowID from the last row in the historical log and assign runID +1

For each row, assign transactionTime, runTime, runDate, and runID

Append