Introduction to Robotic Enterprise Framework: -

Transactions and Types of Processes
The Dispatcher and Performer (Producer and Consumer)
Introducing the ReFramework Template

THE REFRAMEWORK MECHANISMS

1. **Overview of the Automation Project and the Main Workflow(Done)**
2. **The Initialization State and the Configuration File(Done)**
3. **The Get Transaction Data State(Done)**
4. **The Process Transaction and End Process States**
5. **Execution and Logging**
6. **System and Business Exception Handling**
7. **Concurrent Queue Consumption**

1)Overview of the Automation Project and the Main Workflow:-

- The Robotic Enterprise Framework (ReFramework) is a UiPath Studio template built using state machines.
- It acts as the starting point for production-ready RPA projects, especially the ones that require scalable processing.
- It is created to fit all of the best practices regarding logging, exception handling, application initialization, being ready to tackle any business scenario.

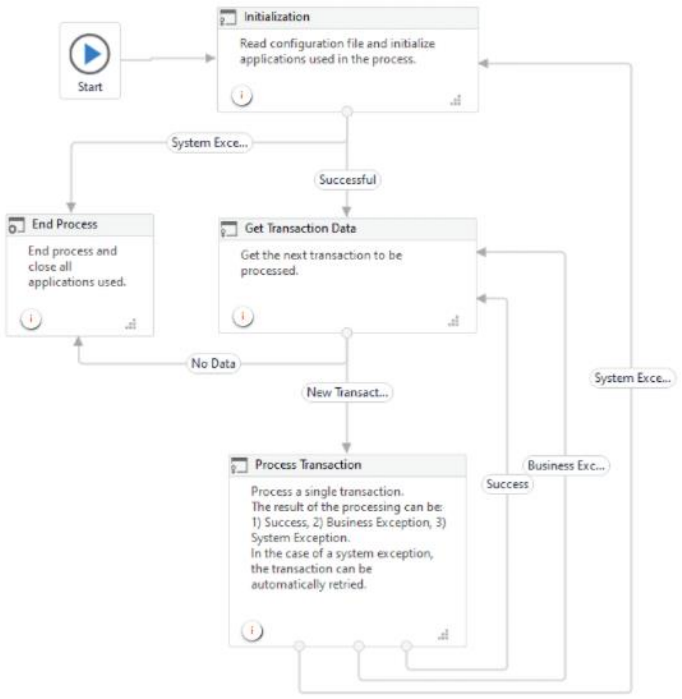
The template contains several pre-built state containers for initializing applications, retrieving input data, processing the data, and ending the transaction.
All of these states are connected through multiple transitions which cover almost every need in a standard automation scenario.

Why use ReFramework?
There are a couple of problems we can face when running automation projects in production.
Fortunately, all of them can be addressed using the ReFramework template.

- Robot stops on the first unexpected error.
- An application crash stops the automation.
- Invalid transaction data stops processing all other transactions.
- Hardcoded and/or sparse values for File Paths, emails, credentials have a negative impact on the security and maintenance.
- Difficult to troubleshoot an error in production.
- Large automation projects are difficult to maintain.

The ReFramework architecture

The REFramework is implemented as a state machine workflow, which is a kind of workflow that defines states that represent a particular circumstance of the execution.
Depending on certain conditions, the execution can transition from one state to another to represent the steps of a process.


Start
Initialization
Read configuration file and initialize applications used in the process.
System Exce...
Successful
End Process
End process and close all applications used.
Get Transaction Data
Get the next transaction to be processed.
No Data
New Transact...
System Exce...
Process Transaction
Process a single transaction. The result of the processing can be: 1) Success, 2) Business Exception, 3) System Exception. In the case of a system exception, the transaction can be automatically retried.
Business Exc...
Success

ReFramework template offers two ways of implementation:
WITH QUEUES
WITHOUT QUEUES
WITH QUEUES
The first way focuses on building a project using the ReFramework with Orchestrator Queues.
For example, this can mean taking the transaction data from the queue and entering it into our UiDemo application.

WITHOUT QUEUES
The second way is instead of using queue items, we use transaction items of a different type such as rows from an Excel file (DataRow), emails (MailMessage), or paths to files (String).
In a future lesson, we will see that it uses the same data structure, only that it will read it from a different source, not an Orchestrator queue, and use each item of the input collection as a transaction.

Overview of the Automation Project and the Main Workflow:-

Nothing just explained about the Over all project
IMPORTANT:-

While going through the overview videos, we recommend focusing on the key functions of each state, the workflows used in each state, the transitions between the states, the argument flow, the exception handling system, the stop mechanism, and logging.

Main takeaways:-
REFramework is the most used template when developing complex business scenarios.
For our process, we are dispatching data from an Excel file, create queue items and process these items by filling details in the UiDemo application.

The Main workflow:-
You are probably familiar already with the Main file, the default entry point for our projects.

As you already know, it uses a state machine diagram with four states: Initialization, Get Transaction Data, Process Transaction, and End Process.

we will learn about the overall organization of the template by checking the Main workflow.

It's important to note that out of the box, the template is set up as a consumer from the producer/consumer model.
As it is meant to be the starting point for any automation project, the template can be configured in many ways, and we will learn about them later.
However, in this example, we are working with a standard consumer implementation with Queue Items.

REFramework template is designed for complex, continuous processes and it contains multiple mechanisms and decision points, the layout used in Main.xaml is State Machine.

Check out the states, their roles, and the transitions between them:

Initialization state:
First, we have the Initialization state. Here, the robot reads the configuration file and initializes the applications used in the process.
(In our example, this is where the start information would be retrieved, UI Demo would be launched and then logged into.)

This state also plays a role in the retry mechanism.

Initialization state → Robot Reads the Config File & Initialize the Application going to use in process.
Initialization state --> Plays role in Retry mechanism.

If this step is completed successfully, the Robot will move to the Get Transaction Data state.
If a System Exception is encountered in the Initialization state, the next state that is executed is the final state, End Process.
This means the execution terminates and the issue should be addressed before running the automation again.

Get Transaction Data state:
The Get Transaction Data state retrieves one by one the transaction items containing the data to be processed.
(In this project, this is where our process retrieves the Queue Items which contain the Cash In, On Us Check and Not On Us Check values.)
In this state, we have two possibilities.

- One is that a new transaction item is retrieved so the robot executes the Process Transaction state next.
- The other is that all transactions are processed, and no more data is available, in this case, the process goes to the End Process state and stops.

GetTransaction Data State--> Retrieve one by one Tranaction item--> with Two Possibility
one -->new transaction item retrieved robot move to the process transaction state
second --> no Transaction item data --> move to end process state and stop

Process Transaction state:-
After successfully retrieving a new transaction, the Process Transaction state performs all needed operations with the data.
(In our case, it enters the values in the UI Demo fields and clicks Accept.)

PTS--> new transaction --> processing--> after processing or during processing
There are three possible outcomes related to this state:

If no exception is encountered,

the outcome is Success, and the automation loops back to the Get Transaction Data state, then the next transaction item is retrieved.
The queue item Status is set to Successful.

If a Business Rule Exception occurs,

we need to perform a few specific actions, and afterward the execution loops back to the Get Transaction Data state.
Note that no automated recovery is performed when Business Rule Exceptions are encountered as they typically need to be addressed by a human user.
The queue item status is set to Failed with the type Business.

If System Exception triggers

A System Exception triggers the execution of a necessary set of steps to recover from the error.
Therefore, all applications are closed, and the execution loops back to the Initialization state to restart them.
→**(Ye baat imp hai)**The queue item status is set to Retried and a new item with the same data is added to the queue with the status New.
→**(Ye baat imp hai**)Supposing only one retry is available after the Robot finishes retrying the new item, its status changes to Failed or Successful, according to the processing result.

Main takeaways

→The overall organization of the template makes it easier to build upon it and is a good starting point for any process.
Depending on certain conditions, the execution can transition from one state to another to represent the steps of a process.
→Several benefits of using the template have been sticking out: a consistent model which enables development standards, better visibility of actions and events, and ease of the development effort.

<u>2) The Initialization State and the Configuration File</u>

The Initialization state, part 1

We will now look into detail on the Initialization state, focusing on its role, exception handling mechanism, the Configuration file, and the workflows that compose this state.

The REFramework template states, and the transitions between the states. In this video, we will explore the Initialization state in depth.
We will cover the exception handling mechanism, the key actions taken in this state, and the argument flow.

In the Initialization state, the robot reads the configuration settings and opens the needed applications.
(In our case, this is where it will open and log into the UI Demo application.)

Init State -->ROBOT--> Read the Config settings and Opens the Needed application.
Init State--> RC & OA

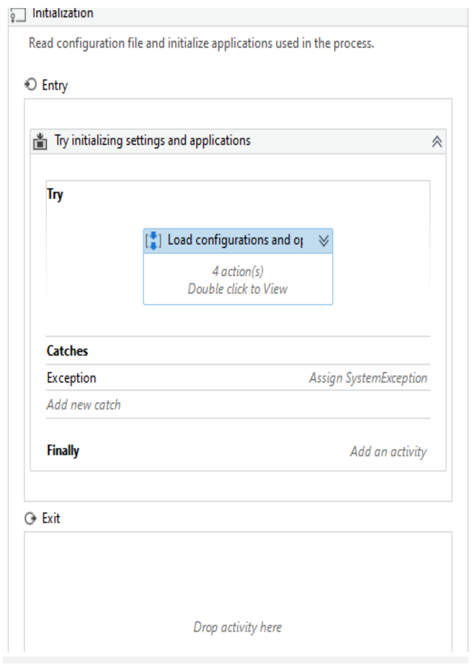
<u>**The first thing we see in this state is the Try Catch activity.**</u>

It's important to note that the overall exception handling is already implemented throughout the framework. Thus, we can find Try Catch activities in every state.

In the Try block, the robot attempts to initialize and if an exception is encountered, it transitions to the End Process state.
So how does this work? When an exception occurs in this state, the exception object is stored in the SystemException variable.
The SystemException variable is initialized with the value Nothing at the beginning of the Try block.

At the starting - SystemVaribale = Nothing

After the try block - If Exception found move to End State bcoz Now--> the systemVariable = Something


Initialization

Read configuration file and initialize applications used in the process.

Entry

Try initializing settings and applications

Try

Load configurations and o

4 action(s)
Double click to View

Catches

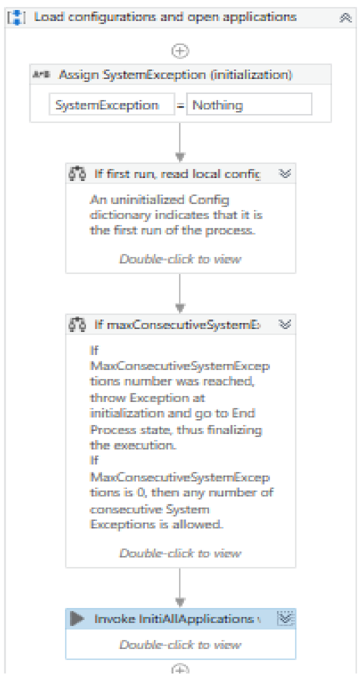
Exception — Assign SystemException

Add new catch

Finally — Add an activity

Exit

Drop activity here


Load configurations and open applications

Assign SystemException (initialization)

SystemException = Nothing

If first run, read local config

An uninitialized Config dictionary indicates that it is the first run of the process.

Double-click to view

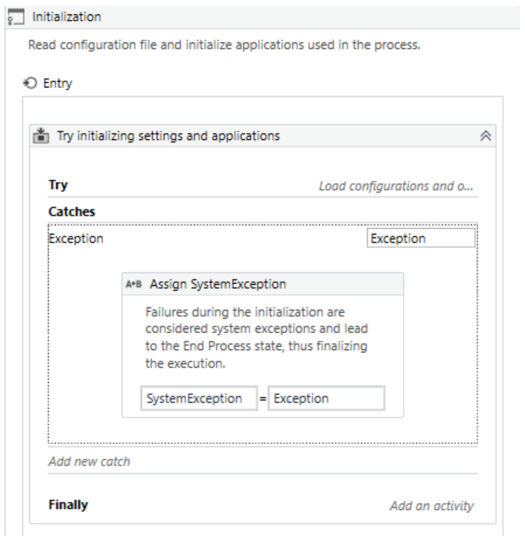
If maxConsecutiveSystemE

If MaxConsecutiveSystemExceptions number was reached, throw Exception at initialization and go to End Process state, thus finalizing the execution.
If MaxConsecutiveSystemExceptions is 0, then any number of consecutive System Exceptions is allowed.

Double-click to view

Invoke InitiAllApplications

Double-click to view


Initialization

Read configuration file and initialize applications used in the process.

Entry

Try initializing settings and applications

Try — Load configurations and o...

Catches

Exception — Exception

Assign SystemException

Failures during the initialization are considered system exceptions and lead to the End Process state, thus finalizing the execution.

SystemException = Exception

Add new catch

Finally — Add an activity

Afterward, if this is the first run of the state, the robot reads the configuration file and attempts to open all applications.
→If an exception is caught, the Catches block is executed where we assign the Exception object to the SystemException variable.

The value of the variable determines the transitions from this state.

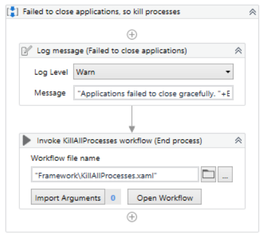
- **If upon exiting the Initialization state the value of SystemException remains Nothing**, the robot transitions to the Get Transaction Data State.

  Tricks:-
  Init State--> RC & OA --> If(SystemVariable is Nothing) then move to next state or Else--> Move to End State | Game of System Variable For catching the Exception.

- **If the value is not Nothing**, meaning it holds an Exception object, the robot logs a Fatal level message and transitions to the End Process state.

  End State:- Log message -->"Applications failed to close gracefully. "+Exception.Message+" at Source: "+Exception.Source
  Exception variable from Catch block

Now that we've covered the Exception handling mechanism used in the Initialization state,


Failed to close applications, so kill processes

Log message (Failed to close applications)

Log Level: Warn

Message: "Applications failed to close gracefully. "+E

Invoke KillAllProcesses workflow (End process)

Workflow file name

"Framework\KillAllProcesses.xaml"

The Initialization State and the Configuration File:- 2

The configuration file

To make it easier to maintain a project and quickly change configuration values, it is a good practice to keep them separated from the workflows themselves.

As we've seen in the previous video, in the InitAllSettings.xaml workflow the ReFramework offers a configuration file. It is named Config.xlsx and can be used to define project configuration values.These values are then read into the Config dictionary.

For easier manipulation, this configuration file is an Excel workbook with three sheets. Let's go

The Config file is used to keep key project settings separate from the workflows, thus allowing us to edit them at any point without needing to modify the project.

Config File-->For--> Project Settings Separate from workflow
Reading the Configuration file is one of the key functions of the Initialization state.
Init State --> Config file--> Purpose is to keep file separate from Project

Let's find out what settings we can define in this file and how the process handles them.

The configuration file is called Config.xlsx and can be found in the Data folder of the project.
Config File --> Found --> DATA folder
By default, it contains three sheets - Settings, Constants, and Assets.
DATA Folder →Config File → Settings-Constants-Assets

Settings sheet.

Here we can store any configuration related to the business process.
This is where we add URLs, file paths, credential names, and any process specific piece of information.

Settings Sheet --> URLS, File PAth, Credential Name,Queue Name Any process Specific piece of Info
(U--------F--------C----Q------I)

For implementations where we use Queue Items, this is where we add the name of the Orchestrator queue which will be used in the process.
(In our case, the queue is called "UiDemo".)
(U--------F--------C----Q------I)
This is also where we add the business process name. (Our process is called "Academy-REF-UiDemo")
(UrFiCnQnInBn)

Additionally we Can Added any Variable Value:-
Path Of Application exe file
name of the Orchestrator credential asset-->even though credentials are added as assets in Orchestrator
threshold based.

For our process, we have three additional values: the path to the UiDemo application …
the name of the Orchestrator credential asset where we have the credential for UI Demo
(note that even though credentials are added as assets in Orchestrator, they are included in the Settings sheet in the Config file) …
and the threshold based on which a transaction is sent for manual processing.

But how is this information read?
There are three columns: Name, Value, and Description.

The Name column always contains a String.
The robot assigns this information as the key in the Config dictionary.
The information in the Value column is assigned as the dictionary value tied to the key from the Name field.
The Description column provides a detailed account of each setting, but it's not used in the process. Its role is to keep developers informed.

Name Column (String) --> Robot Assigns -->info as Key to Config dictionary
Info Value --> With Key

Config(key, value)
Config(Name, Info)
Config(MYurl,www.google.com)

The Constants sheet

The Constants sheet stores "technical" settings that are useful for developers.
It contains information such as the number of retries, default paths, and static log message parts.

- **Number OF Retry**
- **Default Path**
- **Static log message part**

Instead of hard-coding values, we can use the Config object, which enables an easy global value adjustment.
This way, we can fine-tune projects, improve their performance, and smoothly switch their environments, from Dev to Test and then to Production.

An especially important setting is the MaxRetryNumber.
It is used to determine how many times the Robot will retry a transaction which has failed with an Application Exception.

Max Retry--> How many times robot will retry which failed --> by APPLICATION EXCEPTION

The template is designed to work with Orchestrator Queues by default, but it can easily be adapted to suit other types of input data, such as Excel workbooks, emails, files, folders, and so on.

MaxRetryNumber
Orchestrator Queues also have a native and powerful retry mechanism. We are surely going to make good use of it later.
By default, the MaxRetryNumber setting in the Config file is zero as the number of retries is set in Orchestrator.

MaxRetryNumber(Config file(Constant sheet))--> 0 || Value Set in Queue Defalut is -->1

We strongly recommend not to set a different number in the Config file for projects which use queue items as input data.In these situations, the value set in **Orchestrator overrides the one set in the Config file.**

Of course, if the process does not use the Queues functionality, we can set the value of MaxRetryNumber here, allowing the Robot to retry failed transactions.

The Assets sheet

The Assets sheet defines the assets stored in Orchestrator used in the current automation process.

Case: 1

As in the case of the previous sheets, we use the first column, Name, for the Key in our Config dictionary.
We use the second column, Asset … to retrieve the asset with that name from Orchestrator and store its value in the Config dictionary.
In the third column, OrchestratorAssetFolder, we can specify the folder in which the asset is stored. If we leave it empty, the robot uses the default folder.

There are two important notes you should know about assets.

For the first note,
check the name of the UiDemo_ApplicationPath entry in the Assets sheet.
The robot processes the Assets sheet after the Settings sheet and stores the results in the same Config dictionary.

This means that if we define an Asset with the same name as a Setting, the Asset value will overwrite the setting value if the asset is in Orchestrator.

In this example,
the UiDemo_ApplicationPath name is present on both sheets, therefore, if found, the Asset value will be used in the project.
If the asset is not found in Orchestrator, no error is thrown, and the process uses the value defined in Settings sheet.

For the second note,
keep in mind that we can't use this mechanism for Assets of type Credential. Our Config dictionary stores a single value for each Asset, while a Credential type asset holds two values, one for the username and another for the password.

We define Credential Assets in the Settings sheet because they require special handling.

Main takeaways

- Setting up a **config file** for your workflows is an efficient way to configure the initial settings of your workflow environment. Variables that refer to path locations, URLs, and other config information that you need to set up when moving your workflow from one environment to another.
- The above example offers a detailed explanation of the three sheets - Settings, Constants, and Assets.
- Think of it as not having to go inside UiPath Studio to update your variables every now and then. Instead, you can just create a config file, and read it in your workflow.
- As a final note about Config.xlsx, since **the configuration file is not encrypted**, it should not be used to directly store credentials. Instead, it is safer to use Orchestrator assets or Windows Credential Manager to save sensitive data.

The Initialization state, part 2

Discover how the framework can perform actions to make sure that the system is in a clean state before the main process starts.
We've learned about the role of the Initialization state, the Exception handling mechanism, the InitAllSettings workflow, and the Configuration file.
we will focus on the other two workflows invoked in this state: KillAllProcesses and InitAllApplications.

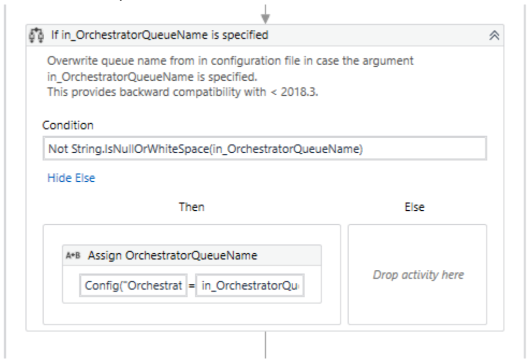
We've successfully read the settings from the Config file in the InitAllSettings workflow and passed the data back to Main.xaml to the Config dictionary variable.

InitAllSetting WFL(Read the Config file) ----> PAss the Data --> Main.xaml --> Config(<t><v>)

overwrite the queue name

In the next step, the Framework offers the option to overwrite the queue name ... by using a Process Configuration or Runtime Argument.

A Process Configuration or Runtime Argument is an argument with the direction In, declared in the project entry point, in our case, Main.xaml.



If in_OrchestratorQueueName is specified

Overwrite queue name from in configuration file in case the argument in_OrchestratorQueueName is specified.
This provides backward compatibility with < 2018.3.

Condition: Not String.IsNullOrWhiteSpace(in_OrchestratorQueueName)

Hide Else

Then: Assign OrchestratorQueueName — Config("Orchestrat = in_OrchestratorQu

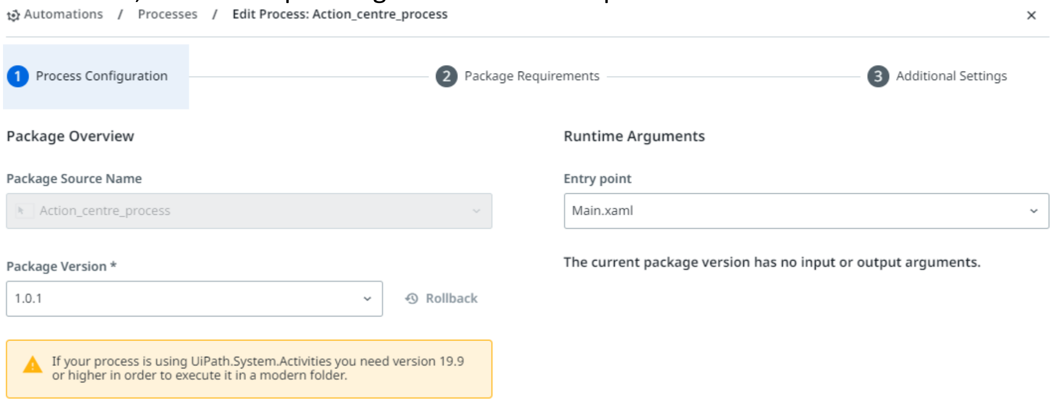
Else: Drop activity here

But where would an Input argument get its value from in Main?

Values can be assigned from Orchestrator. All we need to do is go to the Processes section, select our process, click Edit and provide values for the **Runtime arguments**.

Note that for attended processes, we can also assign values to Runtime arguments from UiPath Assistant.

In this case, we are not planning to overwrite the queue name.



Automations / Processes / Edit Process: Action_centre_process

1 Process Configuration — 2 Package Requirements — 3 Additional Settings

Package Overview

Package Source Name: Action_centre_process

Package Version *: 1.0.1 — Rollback

If your process is using UiPath.System.Activities you need version 19.9 or higher in order to execute it in a modern folder.

Runtime Arguments

Entry point: Main.xaml

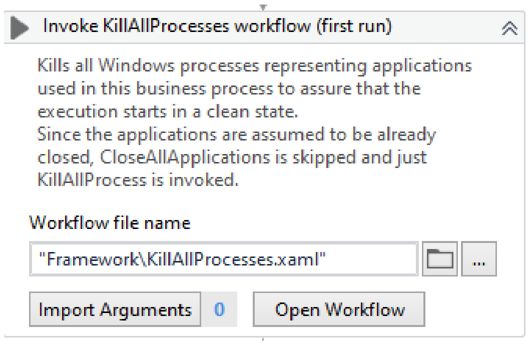
The current package version has no input or output arguments.

<u>The next step is to invoke the KillAllProcesses workflow file</u>. This workflow is meant to force close all applications used in the process.

We do this to make sure that the robot starts in a clean and controlled environment. For example, in our case, when the process starts, an instance of UiDemo may already be open in an incorrect state or even frozen. This would likely cause an exception. The placement of KillAllProcesses here addresses this potential issue.

By default, the workflow is empty, but we added the logic to force close the application we are using ... (in our case, UI Demo.)

Following that, we add a custom log field ... with the name of our process, as defined in the Config file. This way, all logs created further on will have the name of the process. This can help us create reports and visualizations.



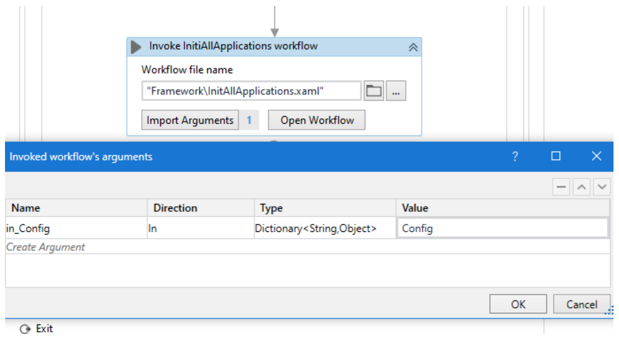
Invoke KillAllProcesses workflow (first run)

Kills all Windows processes representing applications used in this business process to assure that the execution starts in a clean state.
Since the applications are assumed to be already closed, CloseAllApplications is skipped and just KillAllProcess is invoked.

Workflow file name: "Framework\KillAllProcesses.xaml"

<u>Next, the InitAllApplications.xaml workflow</u>, is used to initialize any applications operated during the execution of the process. Note that the config dictionary object is passed to it via the In_Config argument.

By default, the workflow is empty. We recommend placing the steps to open and log into applications in separate workflows, one per application, invoked inside InitAllApplications.
(In our case, we invoked another workflow – UiDemo_Open, ... where we implemented activities to open and authenticate to the UI Demo application.)


Invoke InitiAllApplications workflow
Workflow file name
"Framework\InitAllApplications.xaml"
Import Arguments 1
Open Workflow
Invoked workflow's arguments
Create Argument
OK
Cancel
Exit

Name	Direction	Type	Value
in_Config	In	Dictionary<String,Object>	Config

What are the key functions of this state? What transitions does it contain and what triggers them? What workflows are invoked here and what do they do?

Main takeaways

1. Usually, the activities to open and log into applications are placed in separate workflows, one per application, so we can call these activities wherever is needed.
2. You might ask what is the difference between KillAllProcesses and CloseAll Applications. The first force closes the applications while the second gracefully follows the procedure to close the applications. CloseAllApplications is always used in a Try block with KillAllApplications in the Catches block as a failsafe.
3. The best part of UiPath ReFramework is that it is built in such a way that you can initialize all the applications necessary here in the Initialization state and if the conditions you have built are met, the Robot will then move forward with processing the transaction.

<u>The Get Transaction Data State(Done)</u>

Get Transaction Data.

This state retrieves one by one the transaction items containing the data to be processed.
(In this project, this is where the Robot retrieves the Queue Items containing the Cash In, On Us Check and Not On Us Check values.)

what is a transaction?
We define a transaction as the repetitive part of our process.
It represents the minimum (atomic) amount of data and the necessary steps required to process the data, as to fulfill a section of a business process.

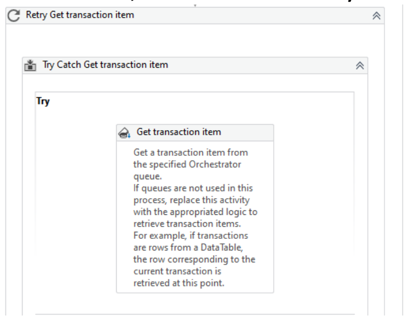
A typical example would be a process that reads a single email from a mailbox and extracts data from it.
We call the data atomic because once it is processed, the assumption is that we no longer need it going forward with the business process.

Note:------> ReFwrk Transtion are Numbered-- Defalut value is One
In the REFramework, the transactions are numbered. The value is stored in the TransactionNumber variable and incremented with each iteration. The TransactionNumber variable is initialized with the value - one.

The next important variable is TransactionItem.
By default, it's of the QueueItem type, but the type can be changed if a different one is needed (for example, DataRow or MailMessage).

(In our case, we need it as a queue item, as we uploaded the transactions from the Excel file to a Queue in Orchestrator.)

When Orchestrator queues are used, the transaction data retrieval is handled by the **Get Transaction Item activity** included by default.
(In this case, it is not necessary to make any modifications to the GetTransactionData.xaml workflow.)



Get a transaction item from the specified Orchestrator queue.
If queues are not used in this process, replace this activity with the appropriated logic to retrieve transaction items.
For example, if transactions are rows from a DataTable, the row corresponding to the current transaction is retrieved at this point.

<u>STOP mechanism</u>

When the Robot gets a **Queue Item from Orchestrator**, the status of the **Item changes to In Process** and it does not get updated until we set it to **Success or Failed by using a Set Transaction Status activity**, after the item is processed.

<u>Why STOP</u>

This is why, if we want to elegantly stop a process, we should do it after finishing processing the current item or just before getting a new queue item.

So, the first thing we do in the Get Transaction Data state is check if a Stop Signal was sent from Orchestrator.
The Should Stop activity gives us the option to soft stop our process execution, before starting a new transaction. If the stop command is received, the automation transitions to the End Process state.
If no stop command is detected, we invoke the GetTransactionData workflow file.

<u>Arguments used in this workflow</u>

IN arguments
We have the following two IN arguments: TransactionNumber and the Config dictionary.

- **TransactionNumber**= 1 from main --> in_TransactionNumber
- Config Dictionary= init all setting workflow = in_Config

OUT arguments
We also have a few OUT arguments, the most important being TransactionItem.
The others: TransactionID, TransactionField1, and TransactionField2 are used for logging purposes.

- **TransactionItem**
- **TransactionID**
- **TransactionField1**
- **TransactionField2**

I/O argumnet
TransactionData can be used as an alternative to store the collection of transaction items when Orchestrator Queues are not used.
As seen here, by default the type is set to DataTable. Note that we are not using this argument in the current process.

TransactionData Collection of transaction item--> i/o TransactionData Datatable

open the workflow file

The first activity is Get Transaction Item, which outputs the queue item from the queue specified in the Config file.

Remember earlier, how we mentioned three OUT arguments that are used for logging purposes?

TransactionID
TransactionField1
TransactionField2

Here is where we assign values to them. Let's find out more about them.

The first one, TransactionID should be unique for each transaction. By default, it is set to the current system timestamp. **The second and third log** fields will be used to log information about the processed transaction: Transaction Field 1 and Transaction Field 2.

(In our case, we set the first one to the concatenation of three values retrieved from the queue item: Cash In, On Us Check and Not On Us Check
TransactionField1= Cash In+On Us Check+Not On Us Check)

if transaction items are invoices

For another example, if transaction items are invoices, then out_TransactionID can be the invoice number, out_TransactionField1 can be the invoice date, and out_TransactionField2 can be the invoice amount.

transaction items = invoices
Then,
out_TransactionID = invoice number
out_TransactionField1 = invoice date
out_TransactionField2 = invoice amount

Now, at some point during process execution, all transactions will have been retrieved and processed, and the process needs to stop as no more transactions are available.

This occurs when TransactionItem becomes Nothing or Null.
When using an Orchestrator queue, that happens by default when there are no more new items to be sent.
If we change the type of the TransactionItem argument, we need to Assign it to Nothing when there are no more transactions to be processed.

we have two possibilities.
One is that all transactions are processed, and no more data is available. In this case, the process goes to the End Process state and stops.
The other is that a new transaction item is retrieved so the robot executes the Process Transaction state next.

Can you explain what a transaction is?
What role does the Get Transaction Data State play in the framework?
Why and how do we use the Stop mechanism?
What workflows do we invoke in this state?
What arguments and variables do we use?

Main takeaways
Get Transaction Data is the data retrieval state, which is used to get the transaction from the queue, data table, folder, database, and other sources. In our example, it looks for any transaction items in the Orchestrator Queue to process.

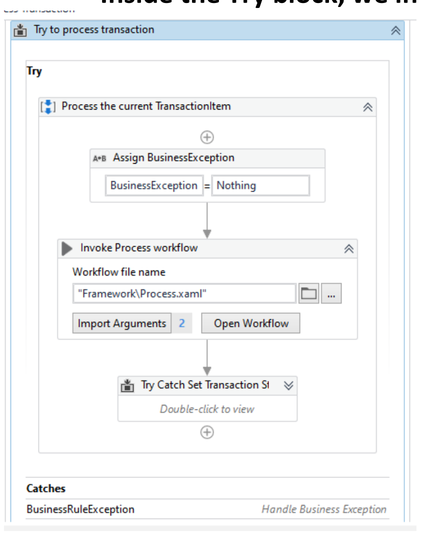
REFramework pre-built functionality looks for a **Stop Request** and will stop the process even though there are items still in the queue to process. This becomes helpful in situations where you want to correctly stop a process and free up the Robot to run a different job.

4). The Process Transaction and End Process States

Process Transaction state

The Process Transaction state includes the key transaction processing steps. There are **three possible outcomes**:
The Success transition is triggered when no exception has been encountered during processing and the automation attempts to get the next transaction item.
The Business Rule Exception transition is triggered when a predefined business rule is broken and the data needs to be checked by a human. The Transaction item is marked as such and the automation attempts to get the next transaction item.
The System Error transition is triggered when a system exception is encountered during processing. The automation sets the correct status for the item, launches the retry mechanism, closes all applications, and loops back to the Initialization state.
The End Process state is the final state of ReFramework, which ends the process and closes all the applications.
it's time to dive into the Process Transaction and End Process states.
we will also spend some time on the Set Transaction Status and Retry mechanisms.

As you can expect, the Process Transaction state starts with a Try Catch activity.
Inside the Try block, we invoke the Process workflow file.


Try to process transaction
Try
Process the current TransactionItem
Assign BusinessException
BusinessException = Nothing
Invoke Process workflow
Workflow file name
"Framework\Process.xaml"
Import Arguments 2 Open Workflow
Try Catch Set Transaction St
Double-click to view
Catches
BusinessRuleException Handle Business Exception

This is where we place the activities which do the actual processing of our data.
(In the case of our process, where we type the data from the Transaction Item into the UiDemo application and click Accept.)
<u>**This workflow takes two input arguments:**</u>
In_TransactionItem which holds **the transaction data**, and **In_Config** which holds the **configuration dictionary**.
In_TransactionItem = transaction data
In_Config = configuration dictionary
Based on the outcome of the execution of the Process workflow,
we can catch a Business Rule Exception and assign its object to the BusinessException variable.
Note that Business Rule Exceptions can only be thrown by a Throw activity … or
a System Exception and assign its **object to the SystemException variable**.
It can consist of any unhandled exceptions thrown in the process.

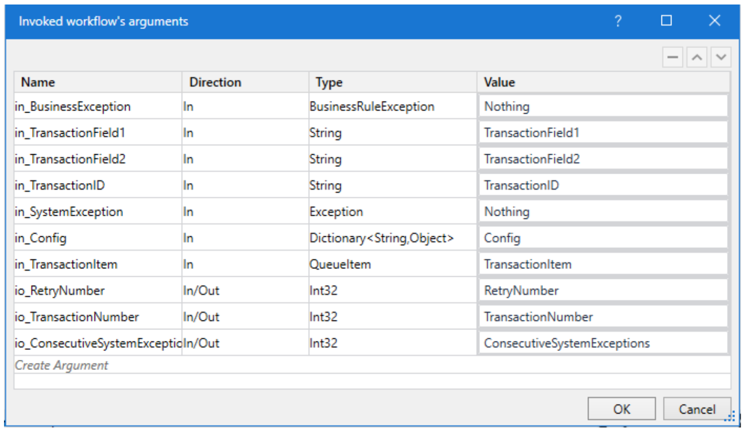
no catch means the Process workflow has been executed successfully.
The Robot selects the state the process transitions to, based on one of these three outcomes.
A System Exception will direct the **process to the Initialization state**
while **both a Business Rule Exception and a No Catch** will direct it to the **Get Transaction Data state.**

The Set Transaction Status workflow

The Set Transaction Status workflow uses quite a few arguments:
This **workflow sets** the **statuses for the queue items**, taking into account the result of the **Process workflow and the number of available retries**.

The Set Transaction Status workflow uses quite a few arguments:
the Config, the SystemError, and the BusinessRuleException arguments, which are used to determine the next state, as well as TransactionItem and TransactionID, as input arguments.

We can find two In/Out arguments here: RetryNumber and TransactionNumber.
They are needed as input, but they are also altered by the Set Transaction Status XAML file. W
hether or not the robot retries a transaction, the Transaction Number or the Retry Number can change.



Invoked workflow's arguments

Name	Direction	Type	Value
in_BusinessException	In	BusinessRuleException	Nothing
in_TransactionField1	In	String	TransactionField1
in_TransactionField2	In	String	TransactionField2
in_TransactionID	In	String	TransactionID
in_SystemException	In	Exception	Nothing
in_Config	In	Dictionary<String,Object>	Config
in_TransactionItem	In	QueueItem	TransactionItem
io_RetryNumber	In/Out	Int32	RetryNumber
io_TransactionNumber	In/Out	Int32	TransactionNumber
io_ConsecutiveSystemExceptio	In/Out	Int32	ConsecutiveSystemExceptions

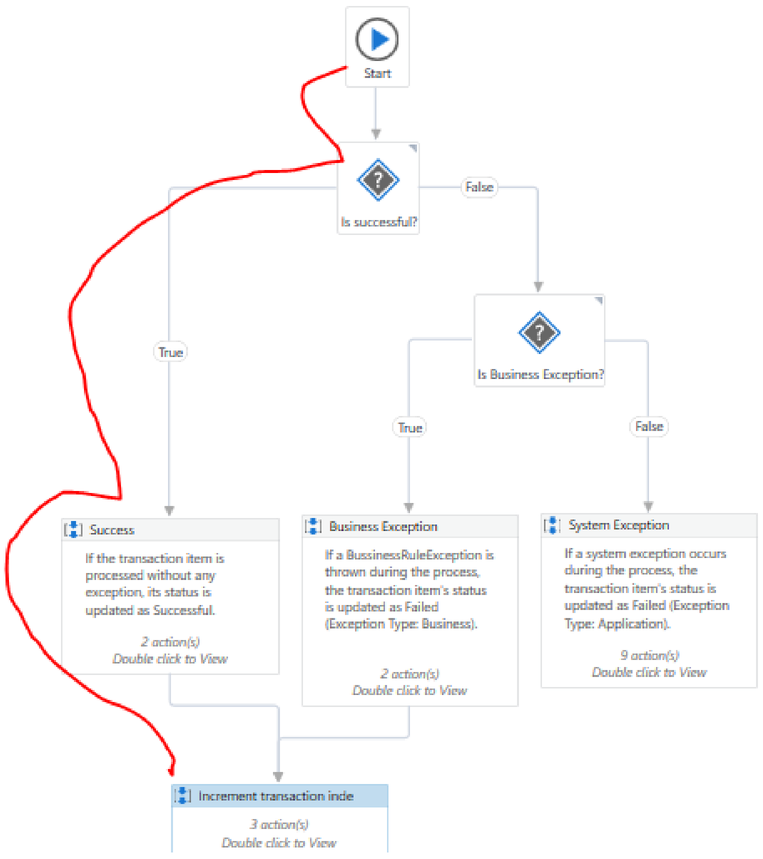
Create Argument

OK Cancel

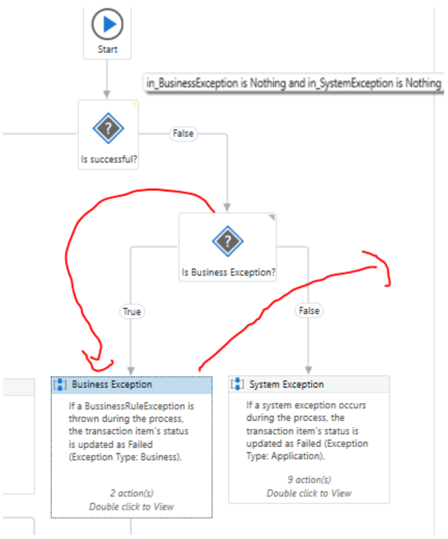
<u>**Let's have a look at the workflow.**</u>

It's a simple flowchart which executes one of the three sequences.
If there is **no exception**, the **Success sequence is executed**.
Here, we use a **Set Transaction Status activity** to change the status of the **Queue Item to "Successful"**.
When we use a different Transaction Item type, we need to make some changes to this workflow, as out of the box, the framework is built to work with queue items.

Condition for Success → in_BusinessException is Nothing and in_SystemException is Nothing → TRUE

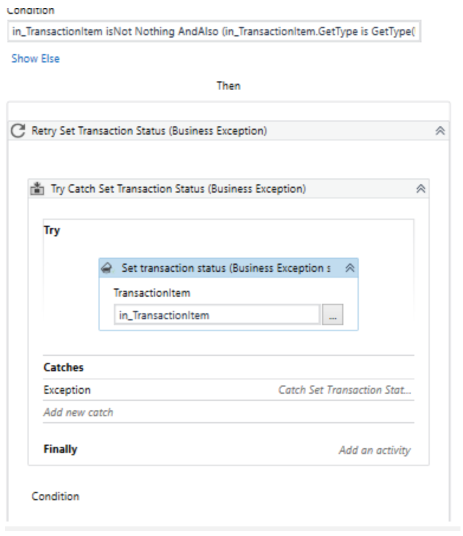

Start
Is successful?
False
True
Is Business Exception?
True
False
Success
If the transaction item is processed without any exception, its status is updated as Successful.
2 action(s)
Double click to View
Business Exception
If a BussinessRuleException is thrown during the process, the transaction item's status is updated as Failed (Exception Type: Business).
2 action(s)
Double click to View
System Exception
If a system exception occurs during the process, the transaction item's status is updated as Failed (Exception Type: Application).
9 action(s)
Double click to View
Increment transaction inde
3 action(s)
Double click to View

if the Robot encounters a Business Exception while processing the transaction,
It sets the status to “Failed”, with the error type “Business”, and the reason set to the message of the caught Business Exception.
The error message is stored in the Message property of the in_BusinessException argument.



If a BusinessRuleException is thrown during the process, the transaction item's status is updated as Failed (Exception Type: Business).
2 action(s)
Double click to View

If a system exception occurs during the process, the transaction item's status is updated as Failed (Exception Type: Application).
9 action(s)
Double click to View


Condition
in_TransactionItem isNot Nothing AndAlso (in_TransactionItem.GetType is GetType(
Show Else
Then
Retry Set Transaction Status (Business Exception)
Try Catch Set Transaction Status (Business Exception)
Try
Set transaction status (Business Exception s
TransactionItem
in_TransactionItem
Catches
Exception | Catch Set Transaction Stat...
Add new catch
Finally | Add an activity
Condition

It’s important to note that for both the items which were successfully processed and the ones that failed with a Business Exception, the following step is to move to the next transaction.

item Failed with Busisness Exception & Successfully Proccesd will got to the NEXT TRANSACTION

Before we do that, we need to increment the transaction number stored in the in/out Transaction Number argument and reset the retry counter stored in the in/out RetryNumber argument to zero.

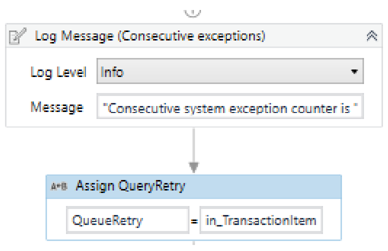
<u>it’s time to see what happens when a transaction fails with a System Error.</u>

This case requires that the Robot takes more actions as it may need to retry the transaction.

The first thing we do in this sequence is to assign the value True to the QueueRetry flag if in_TransactionItem is not nothing and the type of the Transaction Item is QueueItem.

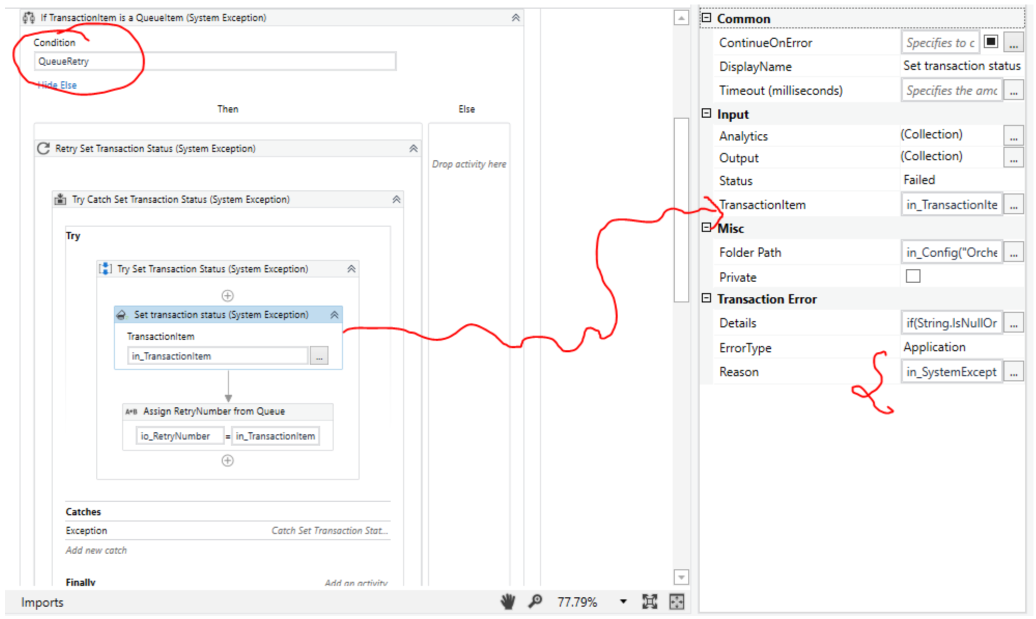
QueueRetry= in_TransactionItem isNot Nothing AndAlso (in_TransactionItem.GetType is GetType(UiPath.Core.QueueItem))

Robot uses the QueueRetry flag to determine whether it should execute the Orchestrator retry mechanism or the one based on the Config file.


Log Message (Consecutive exceptions)
Log Level: Info
Message: "Consecutive system exception counter is "
Assign QueryRetry
QueueRetry = in_TransactionItem

Next, we set the QueueItem status to Failed … with the error type Application, and the message stored in the property of the in_SystemException argument as the reason.

io_RetryNumber = in_TransactionItem.RetryNo


If TransactionItem is a QueueItem (System Exception)

Condition
QueueRetry
Hide Else

Then | Else

Retry Set Transaction Status (System Exception)

Try Catch Set Transaction Status (System Exception)

Try

Try Set Transaction Status (System Exception)

Set transaction status (System Exception)
TransactionItem
in_TransactionItem

Assign RetryNumber from Queue
io_RetryNumber = in_TransactionItem

Catches
Exception — Catch Set Transaction Stat...
Add new catch

Finally — Add an activity

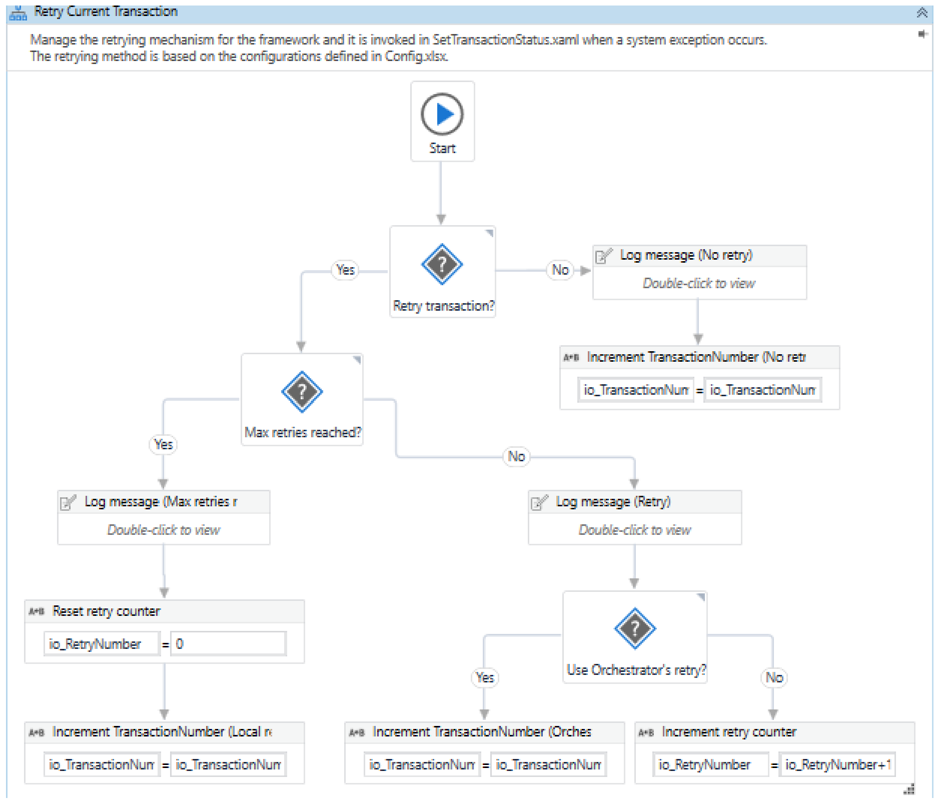
Drop activity here

Imports 77.79%

Common	
ContinueOnError	Specifies to c
DisplayName	Set transaction status
Timeout (milliseconds)	Specifies the amc
Input	
Analytics	(Collection)
Output	(Collection)
Status	Failed
TransactionItem	in_TransactionIte
Misc	
Folder Path	in_Config("Orche
Private	☐
Transaction Error	
Details	if(String.IsNullOr
ErrorType	Application
Reason	in_SystemExcept

invoke the RetryCurrentTransaction workflow file.

STEP1
By default, the framework uses the retry mechanism in the Orchestrator queue, while the MaxRetryNumber in the Config file is zero.
If the retry option is not enabled, after the first flow decision, Retry Transaction, the No branch is executed.
The next activities are a Log message with the level Error and an Assign which increments the Transaction Number.
This means the Robot does NOT retry this transaction, but instead just moves on to the next one.
STEP2
Now, if in our project, the Transaction Item is of a different type than Queue Item (meaning we're not using an Orchestrator queue),
we can enable the retry mechanism simply by setting the MaxRetryNumber in the Config file to a value greater than zero.
In that case, the Yes branch of the Retry decision is executed, and next, the Robot checks if the MaxRetryNumber has been reached.
We don't want to retry a failed transaction forever, if the current retry number reaches the maximum value, the Robot logs an error message once again, … resets the Retry Number to zero, and increments the Transaction Number.
STEP3
But, if the current transaction has been retried fewer times than the maximum retry number indicates, the Robot simply logs a warning.
In the case of a type of transaction item other than a queue item, the Robot increments the current retry number and that's it: it doesn't increment the transaction number value.
If we use Queue Items, the robot increments the Transaction number.


Manage the retrying mechanism for the framework and it is invoked in SetTransactionStatus.xaml when a system exception occurs.
The retrying method is based on the configurations defined in Config.xlsx.

Retry Current Transaction
Start
Retry transaction?
Yes
No
Log message (No retry)
Double-click to view
Increment TransactionNumber (No retr
io_TransactionNum = io_TransactionNum
Max retries reached?
Yes
No
Log message (Max retries r
Double-click to view
Log message (Retry)
Double-click to view
Reset retry counter
io_RetryNumber = 0
Use Orchestrator's retry?
Yes
No
Increment TransactionNumber (Local r
io_TransactionNum = io_TransactionNum
Increment TransactionNumber (Orches
io_TransactionNum = io_TransactionNum
Increment retry counter
io_RetryNumber = io_RetryNumber+1

If an exception occurs while trying to close the applications, we force-close the applications by invoking the Kill all processes workflow in the Catches block.

After restarting all applications, for non-queue item transactions, the same transaction is retrieved in the Get Transaction Data state because the Transaction Number has the same value if retries are still available.
If a Queue Item variable is processed, Orchestrator deals with retries, so it increments the Transaction Number.

Close all applications before returning to the Initialization state and opening them again.
If applications cannot be closed, kill their respective processes.

Do you think you can explain the key functions of the Process Transaction state? What workflows are invoked in this state?
What are the three paths the Set Transaction Status workflow can take?
How does the Retry mechanism work with Queue Items?
How does it work with a different Transaction Item type?
What workflows are invoked in the End Process state?

Main Take aways

1. The result of the processing can be Success, Business Exception or System Exception.
2. In the case of System Exception, the processing of the current transaction can be automatically retried.
3. If the result is Business Exception, the transaction is skipped, and the framework tries to retrieve a new transaction in the Get Transaction Data state.
4. The execution also returns to the Get Transaction Data state to retrieve a new transaction if the processing of the current one is successful.
5. Because the Orchestrator queue is the source of our transactions, the Set Transaction Status activity is used to update their status.

Execution and Logging

Execution and Logging

we will learn about the framework's comprehensive logging structure which uses different

log levels to output statuses of transactions,

exceptions,

transitions between states.

First Debug:- And watch the OUT panel and change the setting from project setting

We want to run the process in Debug mode to check how it performs. To be able to do this, we have already uploaded a few transactions from the Excel file to the queue. Let's debug the process! We can follow its progress in the Output panel.
It starts by killing all the processes and continues with opening the UIDemo application and logging in.
Then it gets one Transaction Item, ... and starts entering the data into the fields.
As you may have already noticed, the UI Automation activities are set to the Hardware Events input method, so the Robot takes control of the mouse and keyboard.
This is something we need to address, so we'll stop the execution.
Let's change this setting at the project level. We can do this by going to Project Settings, UI Automation Classic, Mouse Events & Keyboard Events.
Here we will set Simulate Click & SimulateType to True for both the Run and Debug values.
Since we're here, let's also set the Debug value for the Timeout property to a lower value, so UI interaction errors take less time to occur.
Let's save our changes.
Now if we refresh our Queue, we can see that some of the items were processed successfully while others still display the "New" status.

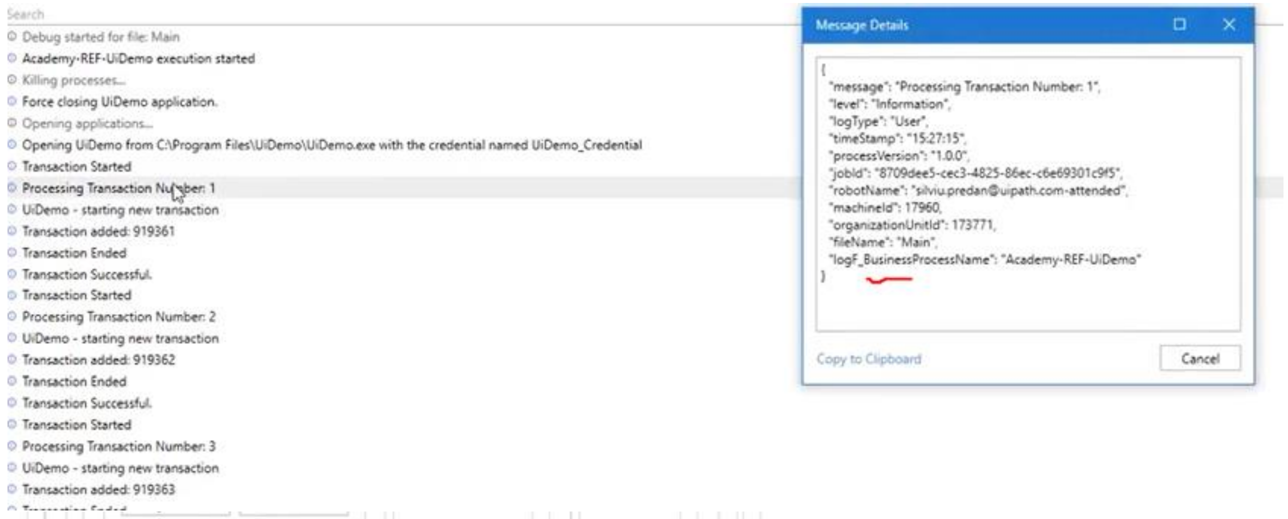
Note that stopping the execution from Studio, like we just did, works like killing the process.
Even though this was not the case now, there is a high chance that one of the queue items will remain suspended with the status "In Progress".

Second Debug:- And watch the OUT panel and change the setting from project setting.

Let's run the process again in debug mode.
It opens the application again and continues the processing with the items which hold the status "New".
OK! All transactions are processed, and because no more data is available, the process goes to the End Process state, closes the application, and stops, as seen in the Output Panel.
Before moving on, let's go to Orchestrator and refresh our Queue.
We can see that all our transactions are now processed successfully.

Let's check the log messages available in the Output panel.

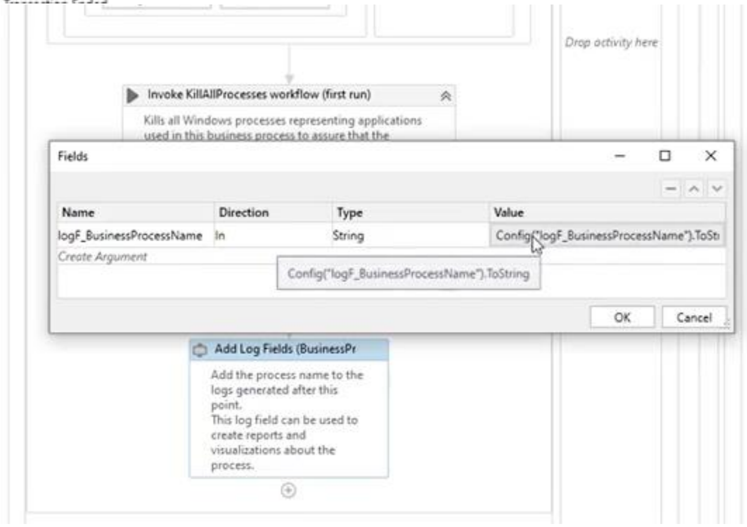
In the panel, we can see only the message for the log entry while the level is indicated by the color. However, if we double click a message, for example, Processing Transaction Number 1 we will see all the log fields.
Here we have the BusinessProcessName log field.
You may remember that its value is set in the Config File in the Settings sheet.
While the **log field is added in the Initialization State using an Add log fields activity**.
Any log message generated after this activity is executed and will contain the Business Process Name field.



```
Search
Debug started for file: Main
Academy-REF-UiDemo execution started
Killing processes...
Force closing UiDemo application.
Opening applications...
Opening UiDemo from C:\Program Files\UiDemo\UiDemo.exe with the credential named UiDemo_Credential
Transaction Started
Processing Transaction Number: 1
UiDemo - starting new transaction
Transaction added: 919361
Transaction Ended
Transaction Successful.
Transaction Started
Processing Transaction Number: 2
UiDemo - starting new transaction
Transaction added: 919362
Transaction Ended
Transaction Successful.
Transaction Started
Processing Transaction Number: 3
UiDemo - starting new transaction
Transaction added: 919363
```

Message Details

```
{
 "message": "Processing Transaction Number: 1",
 "level": "Information",
 "logType": "User",
 "timeStamp": "15:27:15",
 "processVersion": "1.0.0",
 "jobId": "8709dee5-cec3-4825-86ec-c6e69301c9f5",
 "robotName": "silviu.predan@uipath.com-attended",
 "machineId": 17960,
 "organizationUnitId": 173771,
 "fileName": "Main",
 "logF_BusinessProcessName": "Academy-REF-UiDemo"
}
```



Drop activity here
Invoke KillAllProcesses workflow (first run)

Kills all Windows processes representing applications used in this business process to assure that the

Fields

Name	Direction	Type	Value
logF_BusinessProcessName	In	String	Config("logF_BusinessProcessName").ToSti
Create Argument			

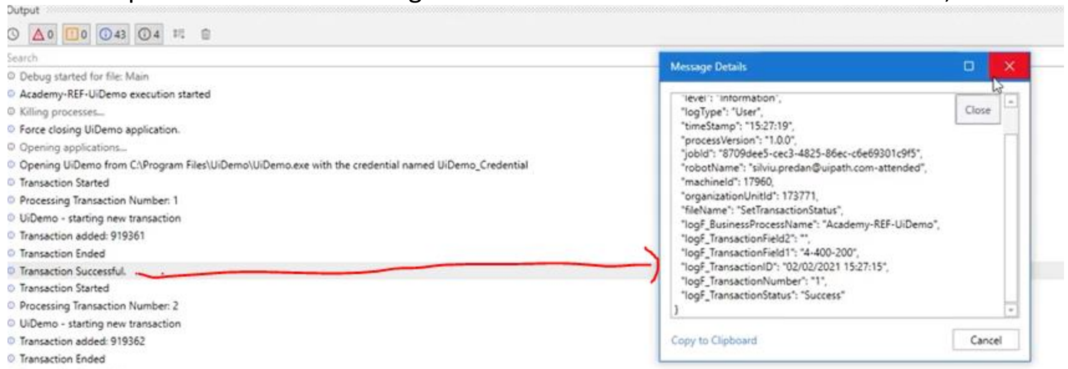
Config("logF_BusinessProcessName").ToString

OK
Cancel
Add Log Fields (BusinessPr

Add the process name to the logs generated after this point.
This log field can be used to create reports and visualizations about the process.

Based on the previous videos, what log fields do you think the Transaction Successful log entry will contain?

We have quite a few additional log fields here such as the Transaction Number, Transaction ID, Field 1, Field 2.

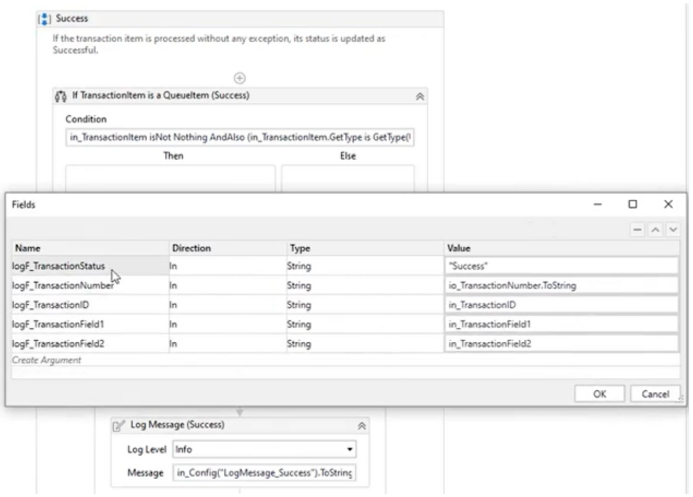


```
Debug started for file: Main
Academy-REF-UiDemo execution started
Killing processes...
Force closing UiDemo application.
Opening applications...
Opening UiDemo from C:\Program Files\UiDemo\UiDemo.exe with the credential named UiDemo_Credential
Transaction Started
Processing Transaction Number: 1
UiDemo - starting new transaction
Transaction added: 919361
Transaction Ended
Transaction Successful.
Transaction Started
Processing Transaction Number: 2
UiDemo - starting new transaction
Transaction added: 919362
Transaction Ended
```

Message Details

```
 "level": "Information",
 "logType": "User",
 "timeStamp": "15:27:19",
 "processVersion": "1.0.0",
 "jobId": "8709dee5-cec3-4825-86ec-c6e69301c9f5",
 "robotName": "silviu.predan@uipath.com-attended",
 "machineId": 17960,
 "organizationUnitId": 173771,
 "fileName": "SetTransactionStatus",
 "logF_BusinessProcessName": "Academy-REF-UiDemo",
 "logF_TransactionField2": "",
 "logF_TransactionField1": "4-400-200",
 "logF_TransactionID": "02/02/2021 15:27:15",
 "logF_TransactionNumber": "1",
 "logF_TransactionStatus": "Success"
}
```

These are set in the Set Transaction Status workflow file.
On the Success branch, we add the ...
Transaction Status,
Number,
ID,
Field1,
Field2.



Success

If the transaction item is processed without any exception, its status is updated as Successful.

If TransactionItem is a QueueItem (Success)

Condition: in_TransactionItem isNot Nothing AndAlso (in_TransactionItem.GetType is GetType(

Fields

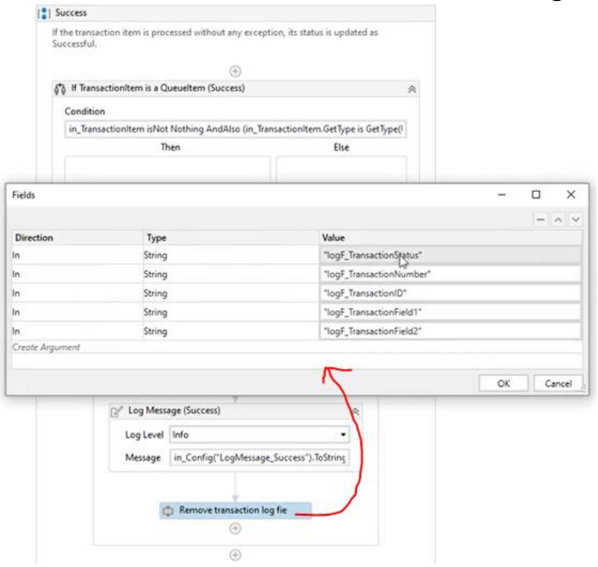
Name	Direction	Type	Value
logF_TransactionStatus	In	String	"Success"
logF_TransactionNumber	In	String	io_TransactionNumber.ToString
logF_TransactionID	In	String	in_TransactionID
logF_TransactionField1	In	String	in_TransactionField1
logF_TransactionField2	In	String	in_TransactionField2

Log Message (Success)

Log Level: Info

Message: in_Config("LogMessage_Success").ToString

We log the message, and then we remove the previously added fields.
We don't want these fields to be added to all log messages, just the ones generated by this activity.



Fields

Direction	Type	Value
In	String	"logF_TransactionStatus"
In	String	"logF_TransactionNumber"
In	String	"logF_TransactionID"
In	String	"logF_TransactionField1"
In	String	"logF_TransactionField2"

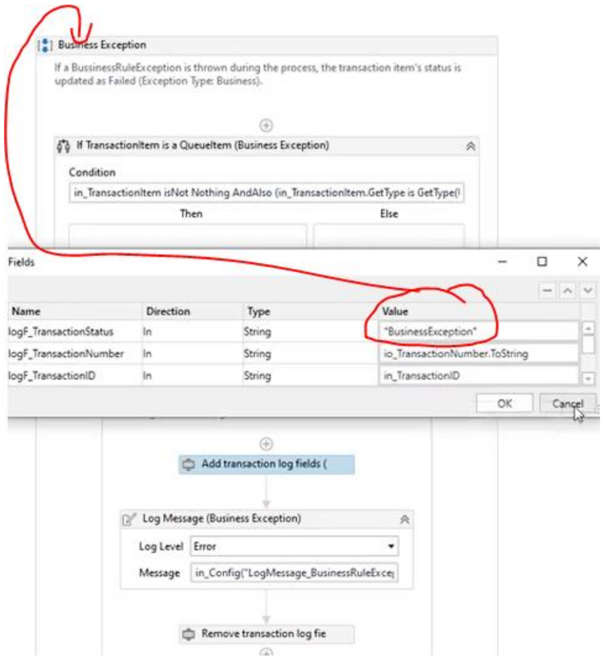
Log Message (Success)

Log Level: Info

Message: in_Config("LogMessage_Success").ToString

Remove transaction log fie

With minor differences, the same happens for logs generated on the Business Exception branch, and the System Exception branch.



Business Exception

If a BussinessRuleException is thrown during the process, the transaction item's status is updated as Failed (Exception Type: Business).

If TransactionItem is a QueueItem (Business Exception)

Condition: in_TransactionItem isNot Nothing AndAlso (in_TransactionItem.GetType is GetType(

Fields

Name	Direction	Type	Value
logF_TransactionStatus	In	String	"BusinessException"
logF_TransactionNumber	In	String	io_TransactionNumber.ToString
logF_TransactionID	In	String	in_TransactionID

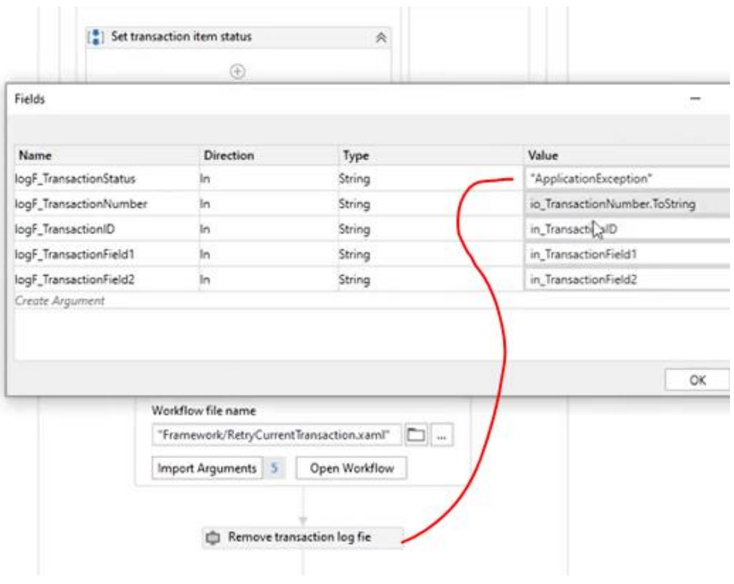
Add transaction log fields (

Log Message (Business Exception)

Log Level: Error

Message: in_Config("LogMessage_BusinessRuleExce

Remove transaction log fie



Set transaction item status

Fields

Name	Direction	Type	Value
logF_TransactionStatus	In	String	"ApplicationException"
logF_TransactionNumber	In	String	io_TransactionNumber.ToString
logF_TransactionID	In	String	in_TransactionID
logF_TransactionField1	In	String	in_TransactionField1
logF_TransactionField2	In	String	in_TransactionField2

Workflow file name: "Framework/RetryCurrentTransaction.xaml"

Import Arguments 5 | Open Workflow

Remove transaction log fie

The values for the **Transaction ID, Field1 and Field2** are set in the **GetTransactionData workflow**.
TransactionID holds the current timestamp.
TransactionField1 holds the information extracted from the queue item.
And we haven't set a value for TransactionField2.

in which we saw how log fields and log messages are organized in the REFramework to help us diagnose and fix published processes.

Main takeaways
The REFramework has a comprehensive logging structure that uses different levels of the Log Message activity to output statuses of transactions, exceptions, and transitions between states.

Although the use of custom log fields is optional, they can be used to include extra information about transactions, which might be helpful during debugging and troubleshooting.

Note that sensitive data should not be included in logs, since they are not encrypted and might lead to privacy issues if leaked.

System and Business Exception Handling

The overall exception handling is already in place throughout the ReFramework, which is why there are Try Catch activities in every state.
Based on the outcome of the execution, we can catch

two different types of exceptions: system and business.

will see system exception handling, business exception handling, and logging in action as our execution will encounter both types of exceptions in the Process Transaction state.

For automations running in production, the execution log is the basis for evaluating the processing results and diagnosing the automation.
(In this demo, we will see how our REFramework based project handles both types of exceptions and how to check the results in Orchestrator.)

how do we generate the exceptions?
For System exceptions we will use the built-in crash mechanism in UIDemo.
We will check the 'Enable crash simulation' option and set the error occurrence to 'Often'.
This will ensure that our demo app will throw a system exception when running.
We can now close the app as the settings are automatically saved and ready for use when it is opened by the Robot.
IMAGE
For Business Exceptions, we have already implemented a rule.
We compute the sum of the values stored in a Transaction Item, and if the sum is greater than the threshold defined in the Config file
then we use a Throw activity to generate a Business Rule Exception.

In the Config.xlsx file, we can see that the 'ManualTransactionThreshold' is set to 10,000.
OK, so if the sum of the Cash In, On Us Check, Not On Us Check values equals more than 10000, then the process will throw a Business Exception.

Watch the Queue
Of course, to test the exception handling mechanism, we've added 10 items to our Orchestrator queue, of which some hold values above the threshold.

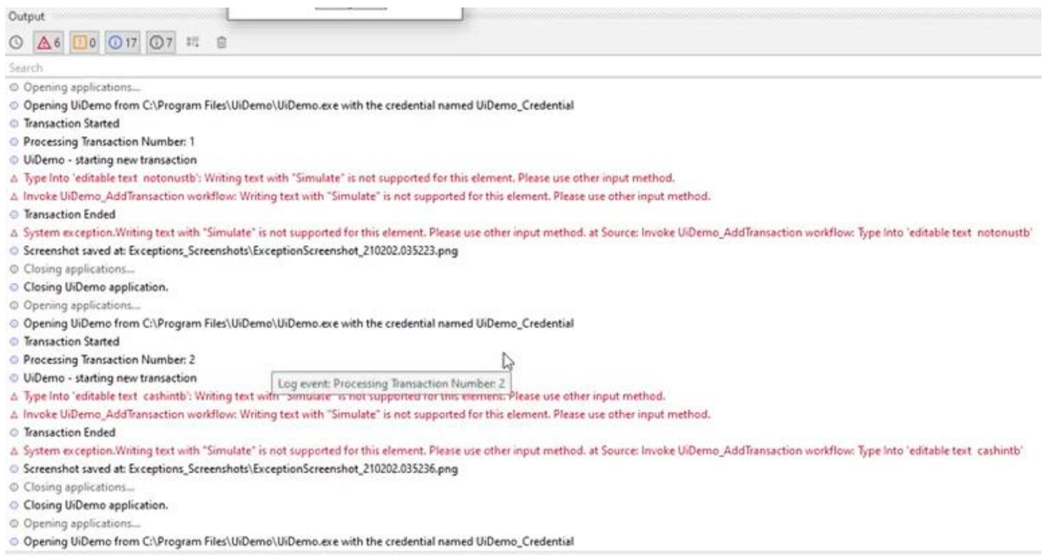
Since we're here, it's good to mention that for testing purposes, we've configured the queue to allow just 1 retry in case of system error.
Image
As seen in previous videos, the retry mechanism can be configured directly in Orchestrator or by using the Config file.
Do you remember when and why we use one option or the other?
Image

As seen in previous videos, the retry mechanism can be configured directly in Orchestrator or by using the Config file.
Do you remember when and why we use one option or the other?

TO excute faster the Project:-
We want to make one small adjustment before testing our process.
The plan is to execute it in Run mode and not in Debug mode.
We want the Timeout value for UI Automation activities to be lower, so we don't wait much for exceptions to occur.
Let's go to Project Settings, UI Automation Classic, and reduce the Timeout Run value to 5000 milliseconds, or 5 seconds.

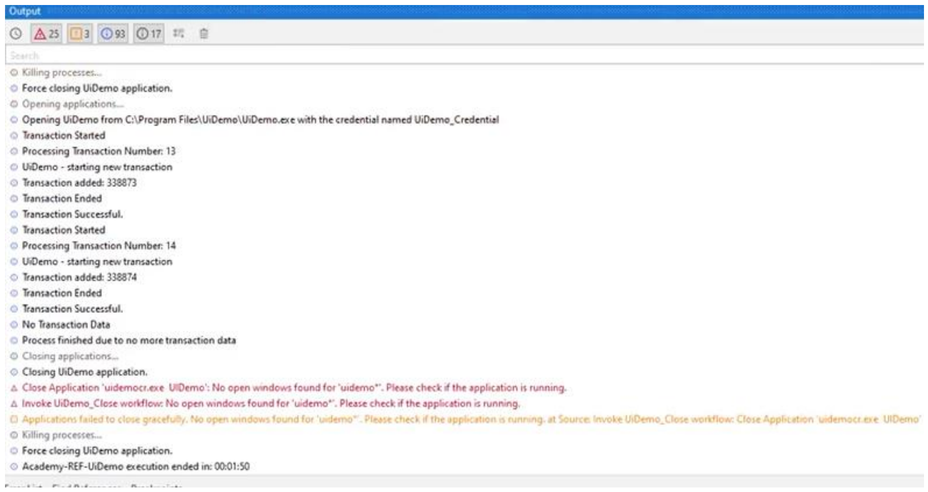
we're all set! Now let's run the project and follow the execution in the Output panel.

We can see that our automation process is opening the UIDemo app, authenticating, and processing each transaction in the Orchestrator Queue.
Now, the target application has crashed while processing Transaction Number 1.
The Robot closes the application and opens it again. 1



```
Opening applications...
Opening UiDemo from C:\Program Files\UiDemo\UiDemo.exe with the credential named UiDemo_Credential
Transaction Started
Processing Transaction Number: 1
UiDemo - starting new transaction
Type Into 'editable text  notonustb': Writing text with "Simulate" is not supported for this element. Please use other input method.
Invoke UiDemo_AddTransaction workflow: Writing text with "Simulate" is not supported for this element. Please use other input method.
Transaction Ended
System exception.Writing text with "Simulate" is not supported for this element. Please use other input method. at Source: Invoke UiDemo_AddTransaction workflow: Type Into 'editable text  notonustb'
Screenshot saved at: Exceptions_Screenshots\ExceptionScreenshot_210202.035223.png
Closing applications...
Closing UiDemo application.
Opening applications...
Opening UiDemo from C:\Program Files\UiDemo\UiDemo.exe with the credential named UiDemo_Credential
Transaction Started
Processing Transaction Number: 2
UiDemo - starting new transaction          Log event: Processing Transaction Number: 2
Type Into 'editable text  cashintb': Writing text with "Simulate" is not supported for this element. Please use other input method.
Invoke UiDemo_AddTransaction workflow: Writing text with "Simulate" is not supported for this element. Please use other input method.
Transaction Ended
System exception.Writing text with "Simulate" is not supported for this element. Please use other input method. at Source: Invoke UiDemo_AddTransaction workflow: Type Into 'editable text  cashintb'
Screenshot saved at: Exceptions_Screenshots\ExceptionScreenshot_210202.035236.png
Closing applications...
Closing UiDemo application.
Opening applications...
Opening UiDemo from C:\Program Files\UiDemo\UiDemo.exe with the credential named UiDemo_Credential
```

But then it crashes again while processing Transaction Number 2.
We've got a few system exceptions logged,
let's adjust the error occurrence to medium so that our process can advance through the remaining transactions faster.
So far, we've mostly checked execution results in the Output panel. If we check Transaction Number 1, for instance, we can see that it failed with an Application Exception.



```
Output
25  3  93  17
Search
Killing processes...
Force closing UiDemo application.
Opening applications...
Opening UiDemo from C:\Program Files\UiDemo\UiDemo.exe with the credential named UiDemo_Credential
Transaction Started
Processing Transaction Number: 13
UiDemo - starting new transaction
Transaction added: 338873
Transaction Ended
Transaction Successful.
Transaction Started
Processing Transaction Number: 14
UiDemo - starting new transaction
Transaction added: 338874
Transaction Ended
Transaction Successful.
No Transaction Data
Process finished due to no more transaction data
Closing applications...
Closing UiDemo application.
Close Application 'uidemocr.exe  UIDemo': No open windows found for 'uidemo*'. Please check if the application is running.
Invoke UiDemo_Close workflow: No open windows found for 'uidemo*'. Please check if the application is running.
Applications failed to close gracefully. No open windows found for 'uidemo*'. Please check if the application is running. at Source: Invoke UiDemo_Close workflow: Close Application 'uidemocr.exe  UIDemo'
Killing processes...
Force closing UiDemo application.
Academy-REF-UiDemo execution ended in: 00:01:50
```

<u>Let's check the same transaction in Orchestrator.</u>
To do this, we go to the Queues section in the correct folder in Orchestrator and select View Transactions for our queue. This is where we can check the processing result of an automation built using the REFramework and Queue Items and placed into production.
Here too we can see that the first transaction failed with Application exception.

Furthermore, if we click More Actions, View Details ... we can find out more about each transaction. We can see the Specific Data, the Exception Type, the Reason, and that it was retried successfully.

Now, let's look at a failed transaction because of a 'Business' exception.
We can see that the reason is that the sum of the transaction items is 30,153 and therefore exceeds the configured cap of 10,000 for automated processing.
A key aspect is that since it is a 'Business' exception, it must be analyzed by a person; hence this transaction was not retried.

1. What are the next steps which follow a caught Business exception in the Process Transaction state?
2. What about for a Business Rule Exception?
3. Where can you find the list of Queue Items and their statuses in Orchestrator?
4. How can you find out why a queue item failed?

Main takeaways:-

1. The REFramework offers a robust exception handling scheme and can automatically recover from failures, update statuses of transactions and gracefully end the execution in case of unrecoverable exceptions.
2. This feature is closely related to the logging capabilities so that all information about exceptions is properly logged and available for analysis and investigation.
3. It's probably a good idea to emphasize that Business Rule Exceptions will never be thrown automatically. The developer needs to explicitly implement the logic to throw the exceptions (at design time) by using the Throw activity.

Concurrent Queue Consumption

- What is the concurrent queue consumption and what are its benefits?
- How can we execute a job using multiple robots?
- What options do we have for setting up the dynamic allocation?
- How do we send a stop command from Orchestrator to terminate a job smoothly?

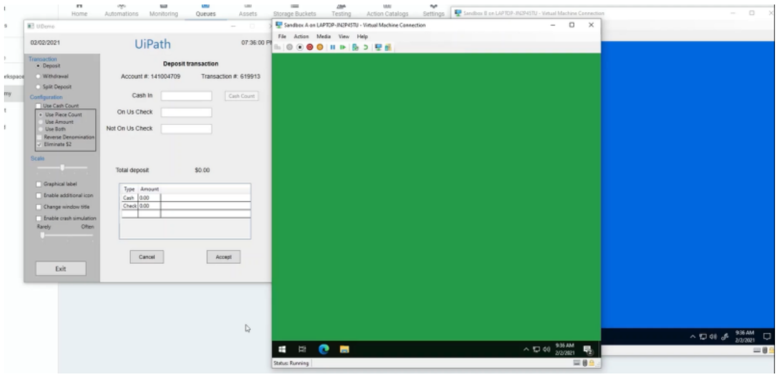
We want to focus on a powerful feature facilitated by the collaboration between the template and Orchestrator, namely, the distribution of workload among multiple available robots.

In the first part of this video, we will see how we can assign multiple robots to run the same process and consume the same queue.

In the second part, we will find out how we can send the stop command from Orchestrator to terminate a job smoothly.

We are already running the UI Demo process on the local machine. We've started the execution from Studio.

We also have two virtual machines prepared. They hold Robots connected to Orchestrator and have the UI Demo application installed



let's see how we can run the UI Demo consumer on the two virtual machines.

With the correct folder selected, we navigate to Automations, Jobs, and select Start a Job.

We have three important settings here.

1. The Allocate dynamically option lets us set the number of times we want to run the process.
2. Dynamic allocation with no explicit user and machine selection allows us to execute a foreground process multiple times under the user and machine that becomes available first.
3. Background processes get executed on any user, regardless of whether it's busy or not, as long as we have sufficient runtimes.

The process is executed under a specific User. We can control this by editing the User setting. Specifying only the user results in Orchestrator allocating the machine dynamically.

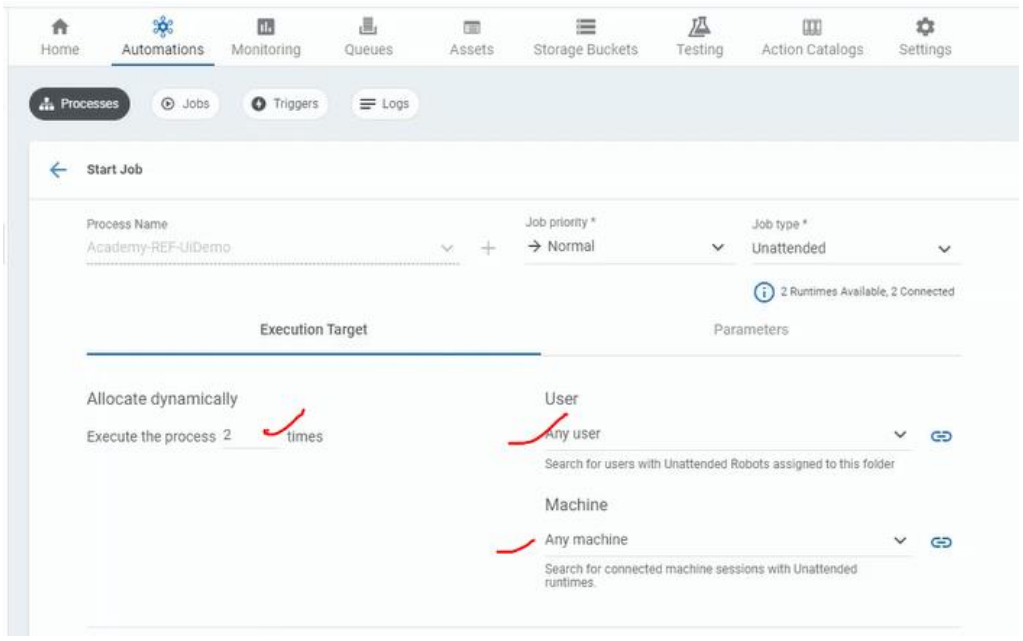
The process is executed on one of the host machines attached to the selected machine template. Specifying the template displays an additional Connected Machines option, allowing us to select a specific host machine from the pool of connected host machines.

Specifying only the machine results in Orchestrator allocating the user dynamically.
Specifying both the user and the machine means the job launches on that very user-machine pair.

Let's set the number of times we want to execute the process to two, leave the User and the Machine options set to Any, ... and click Start.

As we can see, we have two processes running on the sandboxes, started from Orchestrator, and one process running locally started from Studio.

The three robots are processing different items in parallel, thus consuming the queue much faster.



Start Job

Process Name

Academy-REF-UiDemo

Job priority *

Normal

Job type *

Unattended

2 Runtimes Available, 2 Connected

Execution Target | Parameters

Allocate dynamically

Execute the process 2 times

User

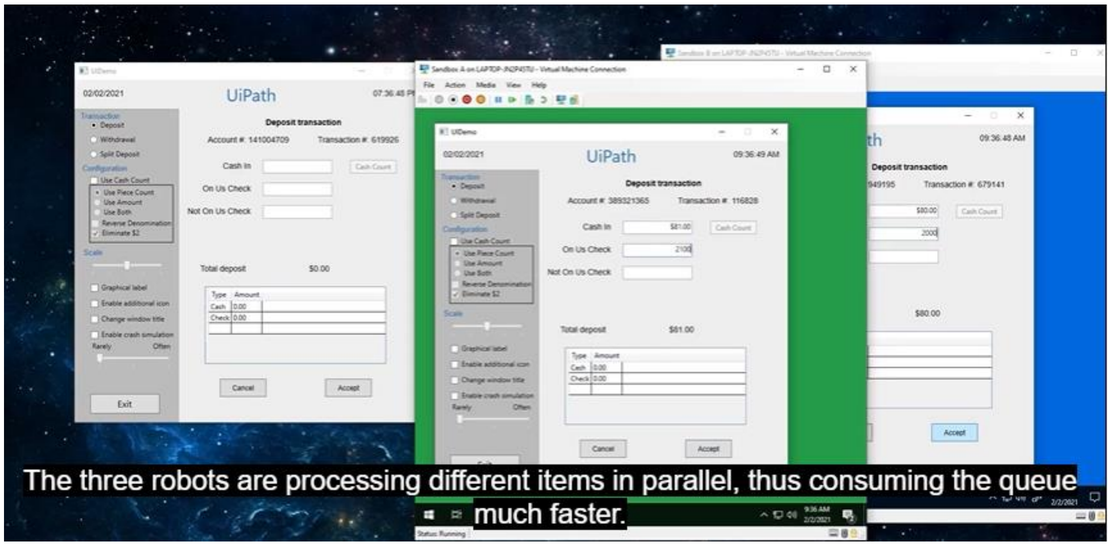
Any user

Search for users with Unattended Robots assigned to this folder

Machine

Any machine

Search for connected machine sessions with Unattended runtimes.

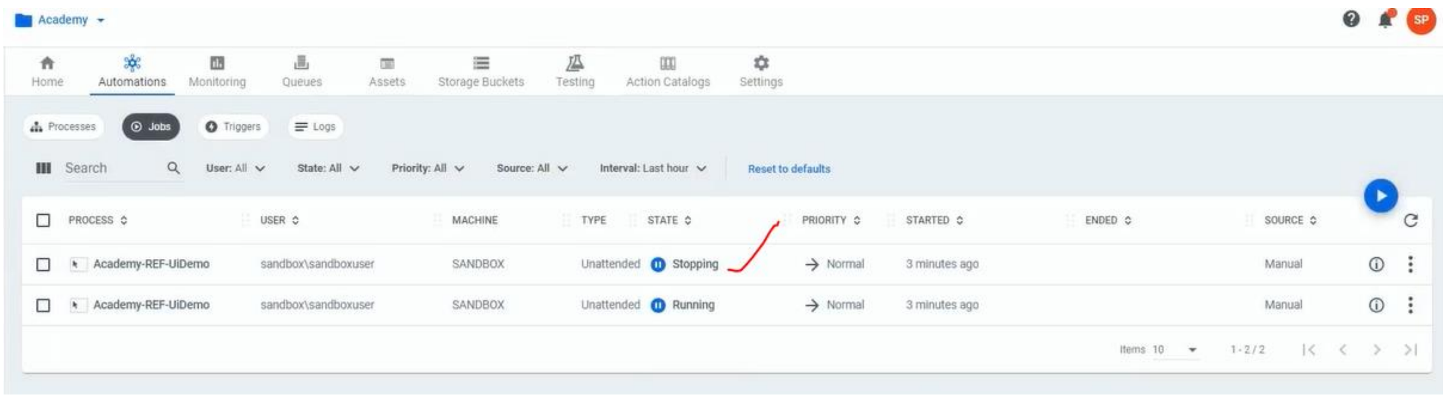


The three robots are processing different items in parallel, thus consuming the queue much faster.

Let’s find out how dynamic allocation influences the processing speed.

Let’s go back to Orchestrator, access the Queues section, and select View Chart for our Queue.
While we were running on a single robot, we were processing around 18 transactions per minute.
Now that we are running on two additional robots, we’re processing 43 transactions per minute.
OK, this covers how we can start a process to run on multiple robots.

We’ve talked about the stop mechanism built into the framework in a previous video.
As a best practice, it’s better to stop a job than to kill it.
This will send a signal to the robot, telling it that the process should stop. Do you remember what activity checks for this signal and what state it is in?



PROCESS	USER	MACHINE	TYPE	STATE	PRIORITY	STARTED	ENDED	SOURCE
Academy-REF-UiDemo	sandbox\sandboxuser	SANDBOX	Unattended	Stopping	Normal	3 minutes ago		Manual
Academy-REF-UiDemo	sandbox\sandboxuser	SANDBOX	Unattended	Running	Normal	3 minutes ago		Manual

If we now look at the green Virtual Machine, we see that the process has stopped.

For the Stop command from Orchestrator to work, the Should Stop activity is required in the process workflow.
This activity returns a Boolean that indicates if the Stop button was clicked in Orchestrator.

In the REFramework template, the **Should Stop** activity is the first activity executed in the ‘Get Transaction Data’ state.
This is because we want to perform the check ... before starting a new transaction.

If the Should Stop variable holds the value True, then the Then block will be executed, ... the information will be logged, and the Nothing value will be assigned to the TransactionItem variable, thus triggering the transition to the End Process state.

Now, if we go back to ‘Jobs’ in Orchestrator, and select the ‘View Logs’ action for the stopped job, we can see that stop process was requested.
This will now reflect in a slower processing speed, as displayed in the UIDemo queue chart.

you could answer a few key questions?

What is the main benefit of concurrent queue consumption?
How can we execute a job using multiple robots?
What options do we have for setting up the dynamic allocation?
How do we send the stop signal?
What activities are executed after the stop signal is sent for a specific job?

Main takeaways
How easy it is to distribute the workload with the REFramework and Orchestrator.
We saw the stop mechanism in action and why it is better to stop a job than to kill it.