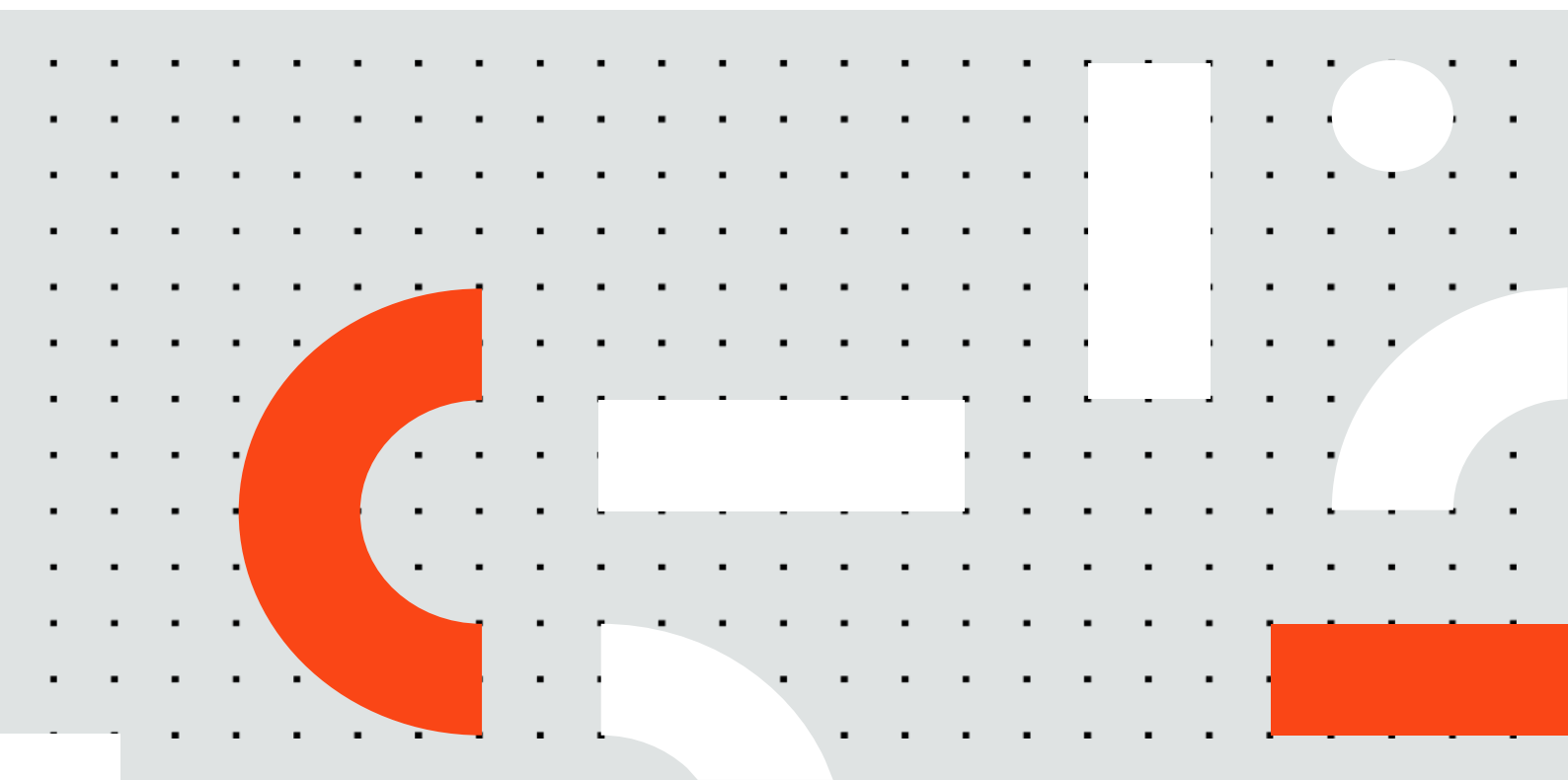




Robotic Enterprise Framework



Overview.....	4
Transaction Processing.....	4
Main Features.....	5
Settings	5
Logging.....	9
Exception Handling and Recovery	11
Architecture.....	12
States	12
Shared Variables	15
Workflows.....	17
Framework\InitAllSettings.xaml	17
Framework\KillAllProcesses.xaml	18
Framework\InitAllApplications.xaml	19
Framework\GetTransactionData.xaml	19
Framework\Process.xam	21
Framework\SetTransactionStatus.xaml	22
Framework\RetryCurrentTransaction.xam	24
Framework\TakeScreenshot.xaml	25
Framework\CloseAllApplications.xaml	25
Using the Framework	26
Changes to Framework Files.....	26
Data\Config.xlsx	26
Main.xaml	27
Framework\GetTransactionData.xaml	28
Framework\Process.xam	29
Framework\SetTransactionStatus.xaml	30



Practical Example 1: Using Queues	31
Practical Example 2: Using Tabular Data	33
Test Framework.....	36
Distribution and Support to Extensions	38

A well-organized project can directly impact the success of an RPA initiative. Besides choosing processes favorable to automation (i.e., mature processes with well-defined steps and low exception rate) and creating clear documentation (i.e., Solution Design Document and Process Definition Document), the quality of the implementation itself plays a major role for a positive outcome.

Although different RPA implementations can have their own unique traits, a common set of practices can usually be seen in successful projects. Among those, flexible configuration, robust exception handling and meaningful logging make projects easier to implement, understand and maintain. In addition, in the case of large implementations, scalability also becomes an important factor due to the volume of data processed.

The Robotic Enterprise Framework, REFramework, is an UiPath Studio template with features that cover these essential practices and can be used as the starting point for most RPA projects, especially the ones that require scalable processing. Although the REFramework can be adapted to any process, its advantages are especially evident when the framework is used to implement transactional processes. Since transaction items are independent from each other, it is possible to handle exceptions and manage logging at transaction level, offering more detailed information about each processed item and making it easier to retry or eventually skip failed transactions.

This guide describes the framework in detail with realistic use cases and practical examples. Firstly, section *Transaction Processing* introduces different types of processes and explains how they are related to the REFramework. After that, an overview of the main aspects of the framework is offered in section *Main Features*. Next, the workflows that compose the framework are detailed in *Architecture section*. Section *Using the Framework* clarifies how to use the framework in practice and includes two step-by-step examples. Moreover, the *Test Framework* section outlines how to use the unit testing capabilities of the framework. The last section, *Distribution and Support to Extensions*, presents the framework's license and policies related to distribution and support.

Transaction Processing

Although business processes can have different characteristics, it is usually possible to classify them based on how they repeat certain steps when processing data.

For example, consider a business process that extracts certain data from a PDF file specified by a user and inputs that data into a web system. In this scenario, to extract



data from a different PDF file, the user must execute the process again and pass the new file as input.

However, if a user specifies a batch of PDF files instead of just one, the same processing steps are repeated for each of the files in the batch. In this case, if each of the PDF files can be processed independent of each other, then it is possible to say that each file is a transaction within the whole process. In other words, a transaction represents a single unit of work that can be independently processed.

Although the kind of transaction depends on the process, it is important to clearly identify transactions within the process to be automated. The REFramework natively considers the processing of transactions and performs the same steps defined in the **Process Transaction** state on each transaction. The States section gives more details about specifying a source of transactions and how each one is processed.

Main Features

Other than naturally enabling transactional processing, the REFramework also has other features that are helpful in the implementation of stable and scalable automation projects: settings, logging, and exception handling.

Settings

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. In such cases, a configuration file can be used to define parameters that are used throughout the project and to avoid hardcoded values in workflows.

The REFramework offers a configuration file, named **Config.xlsx** and located in the **Data** folder, which can be used to define project configuration parameters.

Table 1 - Examples of constants.

Name	Value	Description
Department	Accounting	Default name for department.
Bank Code	ABC123	Code of the bank to be used for payments.

These parameters are then read into the **Config** dictionary variable of the **Main.xaml** file. This dictionary is passed as argument to different files of the framework.

For easier manipulation, this configuration file is an Excel workbook with three sheets:

- **Settings:** Configuration values to be used throughout the project and that usually depend on the environment being used. For example, names of queues, folder paths or URLs for web systems.
- **Constants:** Values that are supposed to be the same across all deployments of the workflow. For example, the department name or the bank name to be used as input in a certain screen.
- **Assets:** Values defined as assets in Orchestrator.

The rows from the **Settings** and the **Constants** sheets indicate keys and values that are read into the **Config** dictionary during the initialization phase of the framework. The **Name** column represents a key in **Config** and the **Value** column defines the value associated with that key. The **Description** column offers an explanation about the row, but it is not included in the dictionary. Table 1 provides an example of how to define constants in the **Constants** sheet.

For instance, if a process needs to define a constant for a department name, then that can be added to the **Constants** sheet: the name is *Department*, the value is *Accounting*, and the explanation is *Default name for department*. Then, during the implementation of workflows, developers can use *Config("Department")* to retrieve the value *Accounting*. Figure 1 illustrates this relationship between the configuration file **Config.xlsx** and the **Config** dictionary.

Config.xlsx Configuration File

Name	Value	Description
Department	Accounting	Default name for department.
BankName	Bank ABC	Default name for bank.

Config Dictionary

Key	Value	Usage
Department	Accounting	Config("Department").ToString
BankName	Bank ABC	Config("BankName").ToString

Figure 1 - Correspondence Between Config.xlsx and Config Dictionary.



There are many constants defined by default and the **Description** column details their purpose. Among those, one particularly important is **MaxRetryNumber**, which specifies how many times a robot attempts to retry processing a transaction that failed with a system exception (section *Exception Handling and Recovery* offers details about exceptions).

If an Orchestrator queue is being used as a source of transactions, then the value of **MaxRetryNumber** should be zero, indicating that the retrying management is done by Orchestrator. If queues are not used, the value of **MaxRetryNumber** should be changed to an integer that represents the desired number of retries.

Another important constant is **MaxConsecutiveSystemExceptions**, which specifies the maximum number of consecutive **System Exceptions** allowed before stopping the job. To disable this feature, the value of the constant should be set to 0.

The **Assets** sheet behaves differently than the other two, since the **Name** column establishes the key to be included in the **Config** dictionary, and the **Value** column determines the name of the asset as defined in Orchestrator.

Figure 2 shows the relationship between the assets defined in Orchestrator, their definition in the **Assets** sheet of the **Config.xlsx** file and their usage in workflows by means of the **Config** dictionary.

Orchestrator Asset

Asset Name	Type	Text	Description
CountryName	Text	Romania	Default name for country.

Assets Sheet in Config.xlsx Configuration File

Name	Asset	OrchestratorAssetFolder	Description
CountryAsset	CountryName		Default name for country.

Config Dictionary

Key	Value	Usage
CountryAsset	Romania	Config("CountryAsset").ToString

For example, if there is an asset in Orchestrator called *CountryName*, there can be a row in the **Assets** sheet whose **Name** is *CountryAsset* and whose **Value** is *CountryName*. During the initialization phase, the framework retrieves the contents of the *CountryName* asset and inserts it as a value corresponding to the key *CountryAsset* in the **Config** dictionary.

The above example uses different names for the asset name in Orchestrator (*CountryName*) and the corresponding dictionary key (*CountryAsset*), but it is common to use the same name for both. By doing so, it becomes easier to maintain the configuration file and to reduce naming mistakes during development.

The Assets sheet from the **Config.xlsx** file contains the **OrchestratorAssetFolder** column. This column stores the path of the Orchestrator folder where the asset is located and must be retrieved from.

In case one of the Assets is not found in Orchestrator, an exception will be thrown, resulting in the Job being stopped.

Although the **Assets** sheet can be used for most types of assets, it cannot be used for assets of type credential, since credentials have two values: username and password. To use credential assets defined in Orchestrator, include them in the **Settings** sheet instead (Figure 3): the **Name** column defines the key in the **Config** dictionary, the **Value** column determines the name of the credential asset, and the **Description** column provides an explanation about the credential. During the implementation, use the **Get Credential** activity to retrieve the credential from Orchestrator.

If the credential asset is stored in a different Orchestrator folder than the one where the process is running, an additional row in the **Settings** sheet might be needed in order to store the folder name.

Orchestrator Credential Asset

Asset Name	Type	Username	Password	Description
System1Credential	Credential	UserABC	Pass123	Credential to access System1.

Settings Sheet in Config.xlsx Configuration File

Name	Value	Description
System1Credential	System1Credential	Credential for ACME System 1.

Config Dictionary

Key	Value	Usage
System1Credential	System1Credential	<i>Config("System1Credential").ToString</i>

Figure 3 - Relationship Between Orchestrator Credential Assets, Config.xlsx and Config Dictionary.

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.

Logging

The proper use of logging in an automation project has several benefits, such as better visibility of actions and events, easier debugging and more meaningful auditing.

The REFramework has a comprehensive logging structure that uses different levels of the **Log Message** activity to output the statuses for the transactions, the exceptions, and the transition between states. Most of the used log messages have static parts that are configured in the **Constants** sheet of the **Config.xlsx** file.

Other than the regular log fields included in messages generated by robots (e.g., robot name and timestamp), the REFramework uses additional custom log fields to add more data about each transaction. When retrieving a new transaction to be processed, in the file **GetTransactionData.xaml**, it is possible to define values for the custom log fields **TransactionId**, **TransactionField1** and **TransactionField2**.

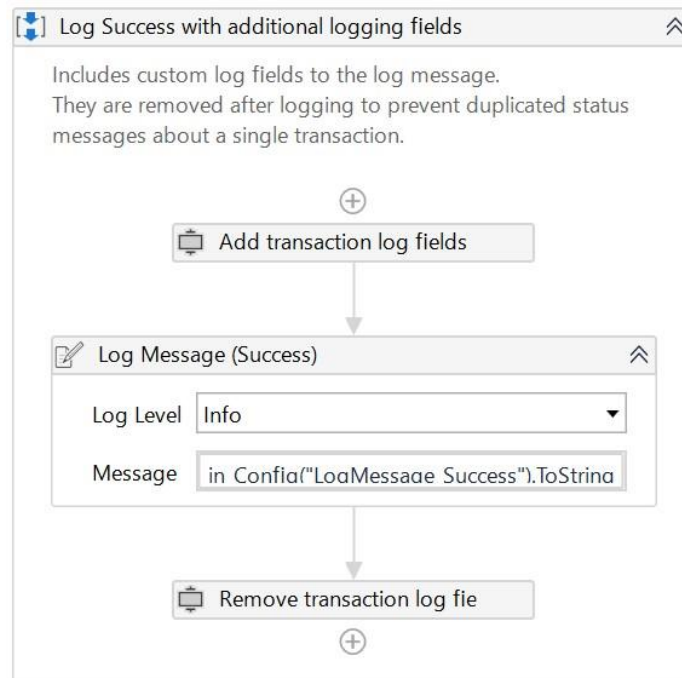


Figure 4 - Addition and Removal of Custom Log Fields.

Figure 4 shows part of the **SetTransactionStatus.xaml** file, which adds custom fields to log messages using the **Add Log Fields** activity. Note that, after the **Log Message** activity is used, the added fields are removed using the **Remove Log Fields** activity. This guarantees that the custom fields previously defined are used just inside the desired log message activity and not in all the following ones, as well.

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.

Additionally, these custom log fields can be leveraged for business reporting purposes. For example, in a process which considers invoices as transactions, the invoice number can be assigned to the **TransactionId** field, the invoice date to **TransactionField1** and the total amount to **TransactionField2**. By using logs generated with such data, it is possible to construct visualizations displaying the days in a month in which a large number of invoices were processed or showing the aggregated total amount processed during a certain period of time (Figure 5).

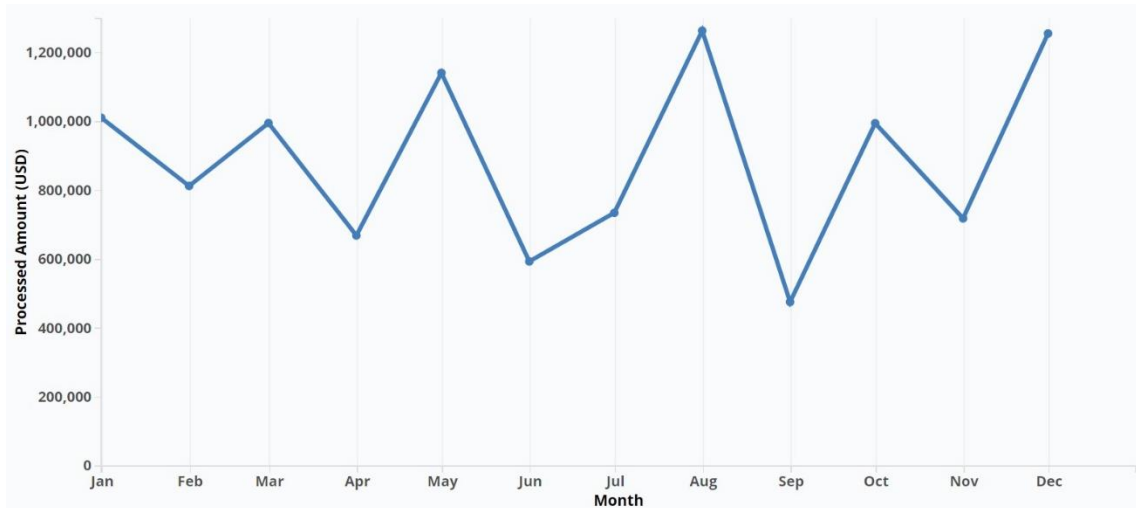


Figure 5 - Example of Reporting Using Custom Log Fields.

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

Exception Handling and Recovery

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. This feature is closely related to the logging capabilities, so that all information about exceptions is properly logged and available for analysis and investigation.

Exceptions that happen during the framework's execution are divided into two categories:

- **Business Exceptions:** This kind of exception is implemented by the class **BusinessRuleException** and it should be thrown when there are problems related to the rules of the business process being automated. For example, if a process expects to receive an email with an attachment, but the attachment does not exist, the process would not be able to continue. In this case, a developer can use the **Throw** activity to throw a **BusinessRuleException**, which indicates that there was a problem that prevented the rules of the process from being followed. Note that **BusinessRuleExceptions** must be explicitly thrown by the developer of the workflow, and they are not automatically thrown by the framework or by the activities.
- **System Exceptions:** If an exception is not related to the rules of the process itself, it is considered a system exception. Examples of system exceptions include an



activity that timed-out due to slow network connection or a selector not found because of a browser crash.

Depending on the category of exception, business exception or system exception, the REFramework decides whether the transaction should be retried or not. In the case of a business exception, the transaction is not automatically retried, since issues related to business rules usually require human intervention. On the other hand, in the case of a system exception, the error might have been caused by a temporary problem and retrying the same transaction can make it succeed without human intervention.

Note that both business exceptions and system exceptions are concepts that also exist in Orchestrator under the names **Business Exceptions** and **Application Exceptions**. In fact, if the source of transactions is an Orchestrator queue, then the number of retries in the case of system exceptions can be set directly on Orchestrator. If the Orchestrator is not used, the configuration for retries is done in the **Config.xlsx** file, as mentioned in *Settings* section.

To avoid consuming all queue items when a persistent error occurs (e.g. application is unavailable), the constant **MaxConsecutiveSystemExceptions** should be used. If the **MaxConsecutiveSystemExceptions** setting is different than 0 and the number of consecutive system exceptions is reached, the job is stopped. To enable this feature, the value of the constant should be set to an integer greater than 0.

By default, all Exceptions are handled by the REFramework, resulting in the Job being marked as **Successful** even when an exception occurs in the **Init State**. However, if the value of **ShouldMarkJobAsFaulted** constant is changed to TRUE and an error occurs in the Initialization state or the **MaxConsecutiveSystemExceptions** is reached, the job is marked as **Faulted**. This enables organizations to make use of the Media Recording feature from Orchestrator to easily identify the error cause.

Architecture

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

States

The states of the REFramework can be seen in Figure 6, and they are detailed as follows:

- **Initialization:** Read the configuration file and initialize the applications used in the process. If the initialization is successful, the execution moves to the **Get Transaction Data** state; in case of failure, it moves to the **End Process** state. If a system exception occurs during the processing of a transaction, the framework attempts to recover from the error by closing all applications used and returning to the Initialization state so the applications can be initialized again.
- **Get Transaction Data:** Get the next transaction to be processed. If there is no data to be processed or any errors occur, the execution goes to the **End Process** state. If a new transaction is successfully retrieved, it is processed in the **Process Transaction** state.
- **Process Transaction:** Process a single transaction. The result of the processing can be *Success*, *Business Exception* or *System Exception*. In the case of a *System Exception*, the processing of the current transaction can be automatically retried. If the result is a *Business Exception*, the transaction is skipped, and the framework tries to retrieve a new transaction in the **Get Transaction Data** state. The execution also returns to the **Get Transaction Data** state to retrieve a new transaction if the processing of the current one is successful.
- **End Process:** Finalize the process and close all applications used.

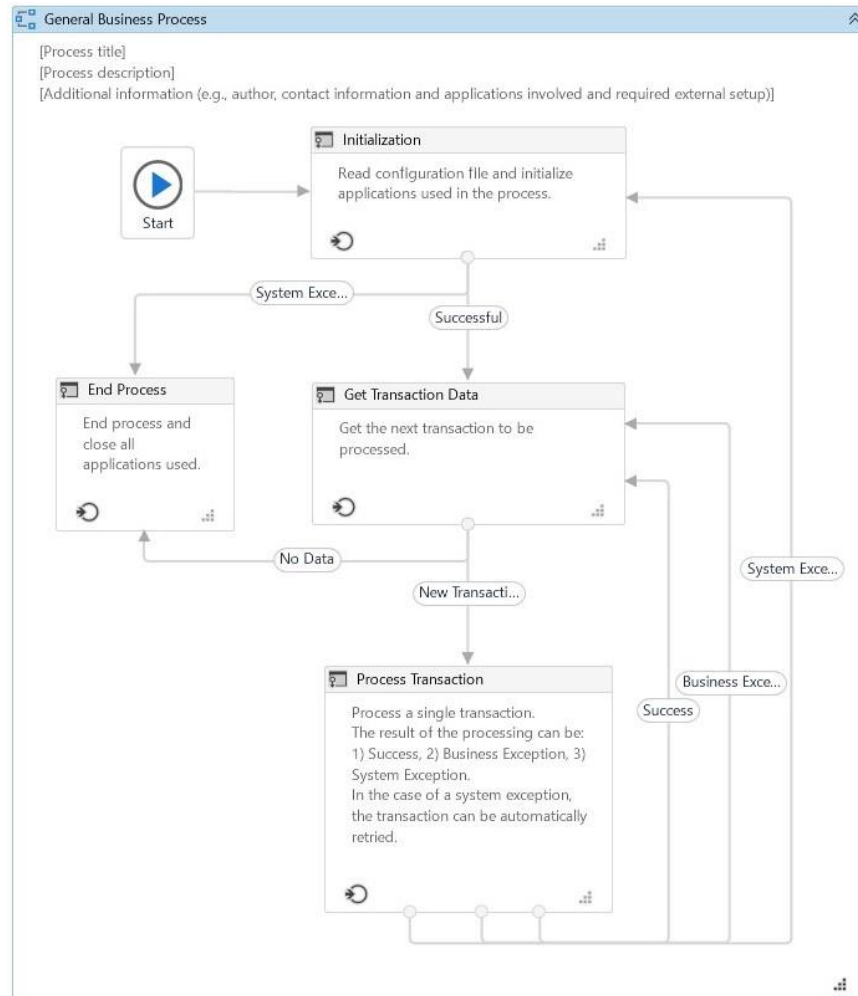


Figure 6 - State Machine with the States of the REFramework.

Table 2 - Workflows Invoked in States.

State	Invoked Workflows
Initialization	InitAllSettings.xaml KillAllProcesses.xaml InitAllApplications.xaml
Get Transaction Data	GetTransactionData.xaml

Process Transaction	Process.xaml SetTransactionStatus.xaml <ul style="list-style-type: none"> • RetryCurrentTransaction.xaml • TakeScreenshot.xaml • CloseAllApplications.xaml • KillAllProcesses.xaml
End Process	CloseAllApplications.xaml KillAllProcesses.xaml

Each state invokes one or more workflows, which are listed in Table 2 and detailed in the section Workflows.

Shared Variables

Table 3 shows the variables declared in the **Main.xaml** file and which are passed as arguments to the workflows invoked in different states.

One important variable that is passed to almost all the workflows invoked in **Main.xaml** is the **Config** dictionary. This variable is initialized by the **InitAllSettings.xaml** workflow in the **Initialization** state, and it contains all the configuration declared in the **Config.xlsx** file. Since it is a dictionary, the values in **Config** can be accessed by its keys, like *Config("Department")* or *Config("System1_URL")*. Note that, although it is present in the **Config.xlsx** file, the **Description** of each value is not included in the dictionary.

Table 3 - Shared Variables.

Name	Default Type	Description
------	--------------	-------------

TransactionItem	QueueItem	Transaction item to be processed. The type of this variable can be changed to match the transaction type in the process. For example, when processing data from a spreadsheet that is read into a DataTable , this type can be changed to DataRow (refer to section <i>Practical Example 2: Using Tabular Data</i> for a sample). Taking the scenario in which the transactions are paths to image files to be processed, the type of the variable can be set to String.
SystemException	Exception	Used during transitions between states to represent exceptions other than BusinessRuleException .
BusinessException	BusinessRuleException	Used during transitions between states and represents a situation that does not conform to the rules of the process being automated.
TransactionNumber	Int32	Sequential counter of transaction items.
Config	Dictionary(Of String, Object)	Dictionary structure to store configuration data of the process (settings, constants and assets).

RetryNumber	Int32	Used to control the number of attempts when it comes to retrying the transaction, in case of a system exception.
TransactionField1	String	Optionally used to include additional information about the transaction item.
TransactionField2	String	Optionally used to include additional information about the transaction item.
TransactionID	String	Used for information and logging purposes. Ideally, the ID should be unique for each transaction.
TransactionData	DataTable	Used in case transactions are stored in a DataTable, for example, after being retrieved from a spreadsheet.
ConsecutiveSystemExceptions	Int32	Used to control the number of consecutive system exceptions allowed before stopping the job.

Workflows

This section details the workflows that compose the REFramework, including overview, purpose and arguments. When applicable, it is also mentioned what parts need to be modified if the transaction type is not **QueueItem**.

Framework\InitAllSettings.xaml

This workflow, located in the **Framework** folder, initializes, populates and outputs a configuration dictionary, **Config**, to be used throughout the project. Settings and constants are read from the local configuration file, **Data\Config.xlsx**, and assets are

fetches from Orchestrator. Asset values overwrite settings and constant values if they are defined with the same name. Table 4 shows the arguments used by **InitAllSettings.xaml**.

Table 4 - Arguments of *InitAllSettings.xaml*.

Argument	Description	Default Value
in_ConfigFile	Path to the configuration file that defines settings, constants and assets.	"Data\Config.xlsx"
in_ConfigSheets	Names of the sheets corresponding to settings and constants in the configuration file.	{"Settings","Constants"}
out_Config	Dictionary structure to store configuration data of the process (settings, constants and assets).	No default value

If an exception occurs during the execution of this workflow - for example, if the configuration file is not found, it is caught by the **Try Catch** activity in the **Initialization** state and the execution transitions into the **End Process** state.

Framework\KillAllProcesses.xaml

After the initialization of settings, the framework can perform actions to make sure that the system is in a clean state before the main process starts. This can be done by using the **Kill Process** activity, which forces the termination of a Windows process representing an application used in the business process. Note that killing processes might have undesirable outcomes, such as losing unsaved changes to files. The **KillAllProcesses.xaml** workflow, located in the **Framework** folder, can be used to implement such cleanup steps.

Also, despite the name of this workflow, it is not mandatory to always kill all the processes used, and other steps might be more appropriate to return the system to a clean state. Ultimately, such steps depend on the requirements of the business process.

Table 5 - Argument of *InitAllApplications.xaml*.

Argument	Description	Default Value
in_Config	Dictionary structure to store configuration data of the process (settings, constants and assets).	No default value

Framework\InitAllApplications.xaml

The **InitAllApplications.xaml** workflow, located in the **Framework** folder, can be used to initialize applications operated during the execution of the process. It can contain activities like **Open Application** activities and **Open Browser**, or it can also invoke other workflows that implement actions like login and authentication.

Table 5 shows that this workflow receives only one argument, the configuration dictionary, **Config**, which can contain data necessary to start certain applications (e.g., URL of a web application).

Framework\GetTransactionData.xaml

This workflow, located in the **Framework** folder, attempts to retrieve a transaction item from a specified source (e.g., Orchestrator queues, spreadsheets, databases, mailboxes or web APIs).

If there are no transaction items remaining to be processed, the argument **out_TransactionItem** is set to *Nothing*, which leads to the **End Process** state. All arguments used are detailed in Table 6.

For cases in which there is only a single transaction (i.e., a linear process), the developer should add an **If** activity to check whether the argument **in_TransactionNumber** has the value *1* (meaning it is the first and only transaction) and assign the transaction item to **out_TransactionItem**. In such cases, for any other value of **in_TransactionNumber**, **out_TransactionItem** should be set to *Nothing* (Figure 7).

Table 6 - Arguments of GetTransactionData.xaml

Argument	Description	Default Value
in_TransactionNumber	Sequential counter of transaction items.	No default value

in_Config	Dictionary structure to store configuration data of the process (settings, constants and assets).	<i>No default value</i>
out_TransactionItem	Transaction item to be processed.	<i>No default value</i>
out_TransactionField1	Optionally used to include additional information about the transaction item.	<i>No default value</i>
out_TransactionField2	Optionally used to include additional information about the transaction item.	<i>No default value</i>
out_TransactionID	Used for information and logging purposes. Ideally, the ID should be unique for each transaction.	<i>No default value</i>
io_TransactionData	Used in case transactions are stored in a DataTable, for example, after being retrieved from a spreadsheet.	<i>No default value</i>

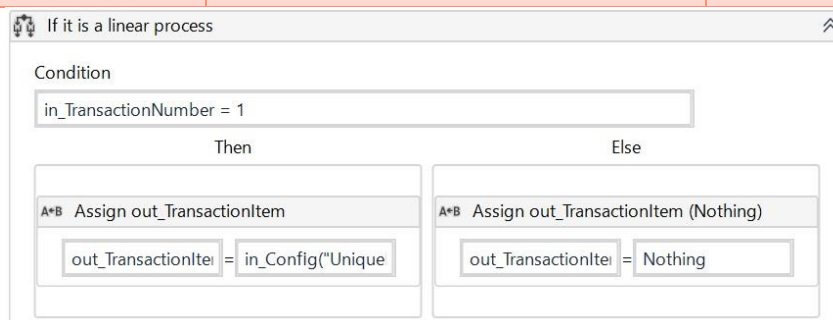


Figure 7 - Configuration of GetTransactionData.xaml in Case of Linear Processes.
Table 7 - Arguments of Process.xaml

Argument	Description	Default Value
----------	-------------	---------------

in_TransactionItem	Transaction item to be processed.	No default value
in_Config	Dictionary structure to store configuration data of the process (settings, constants and assets).	No default value

If there are multiple transactions from a source other than an Orchestrator queue, use the argument **in_TransactionNumber** as an index to retrieve the correct transaction to be processed. If there are no more transactions left, it is necessary to set **out_TransactionItem** to *Nothing*, thus leading to the end of the process.

The **GetTransactionData.xaml** workflow assumes the use of Orchestrator queues by default, and the first activity tries to retrieve a new transaction item from an Orchestrator queue. This situation is illustrated by the example in section *Practical Example 1: Using Queues*.

If Orchestrator queues are not used, replace the **Get Transaction Item** activity with the appropriate logic to retrieve the transaction items. For example, if the transactions are rows from a **DataTable**, the row corresponding to the current transaction is retrieved at this point. Section *Practical Example 2: Using Tabular Data* offers an example of this case.

Lastly, note that this workflow contains an optional step that can be used to include more information about a transaction item, and it is used mainly for logging and visualization purposes. For example, if the transaction items for a given process 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. The *Logging* section offers more information about logging with custom log fields.

Framework\Process.xaml

The **Process.xaml** workflow is used to invoke the major steps of the business process, which are commonly implemented by multiple sub-workflows. Its main argument is **in_TransactionItem**, which represents the piece of data to be processed. The default type for the argument **in_TransactionItem** is **QueueItem** (Table 7), and it should be changed in case other types are used (e.g., **DataRow**, **String** or **MailMessage**).

If a **BusinessRuleException** is thrown during the processing, the current transaction is skipped. If another kind of exception occurs, the current transaction is retried according to the retry configurations.

Framework\SetTransactionStatus.xaml

The **SetTransactionStatus.xaml** workflow, located in the Framework folder, sets and logs each transaction's status. There can be three possible statuses: *Success*, *Business Exception* and *System Exception*.

A business exception, represented by a **BusinessRuleException** object, characterizes an irregular situation according to the process's rules and prevents the transaction from being processed. The transaction is not retried in this case, since the result would be the same until the problem that caused the exception has been solved. For example, it can be considered a business exception if a process expects to read an email's attachment, but the sender did not attach any file. In this case, immediate retries of the transaction would not give a different result.

On the other hand, system exceptions are characterized by exceptions whose types are different than **BusinessRuleException**. When this kind of exception happens, the transaction item can be retried after closing and reopening the applications involved in the process. The idea behind this behavior is that the exception was caused by a problem in the applications being automated (e.g., a system that freezes), which might be solved by restarting them.

If an Orchestrator queue is the source of transactions, the **Set Transaction Status** activity is used to update their status. In addition, the retry mechanism is also managed by Orchestrator.

If Orchestrator queues are not used, the status can be set, for example, by writing to a specific column in a spreadsheet. In such cases, the retry mechanism is managed by the framework itself and the number of retries is defined in the configuration file.

At the end of the **SetTransactionStatus.xaml** workflow, **io_TransactionNumber** is incremented, which makes the framework get the next transaction to be processed.

Table 8 provides details about other arguments of **SetTransactionStatus.xaml**.

Table 8 - Arguments of SetTransactionStatus.xaml

Argument	Description	Default Value
in_Config	Dictionary structure to store configuration data.	No default value

in_SystemException	Exception variable that is used during transitions between states.	No default value
in_BusinessException	Exception variable that is used during transitions between states.	No default value
in_TransactionItem	Transaction item to be processed.	No default value
io_RetryNumber	This variable controls the number of attempts of retrying the process in case of system error.	No default value
io_TransactionNumber	Sequential counter of transaction items.	No default value
in_TransactionField1	Allow the optional addition of information about the transaction item.	No default value
in_TransactionField2	Allow the optional addition of information about the transaction item.	No default value
in_TransactionID	Transaction ID used for information and logging purposes.	No default value
io_ConsecutiveSystemExceptions	This variable controls the number of consecutive system exceptions.	No default value

Table 9 - Arguments of RetryCurrentTransaction.xaml

Argument	Description	Default Value
----------	-------------	---------------

in_Config	Dictionary structure to store configuration data of the process (settings, constants and assets).	<i>No default value</i>
io_RetryNumber	Used to control the number of attempts of retrying the transaction processing in case of system exceptions.	<i>No default value</i>
io_TransactionNumber	Sequential counter of transaction items.	<i>No default value</i>
in_SystemException	Used during transitions between states to represent exceptions other than business exceptions.	<i>No default value</i>
in_QueryRetry	Used to indicate whether the retry procedure is managed by an Orchestrator queue.	<i>No default value</i>

Framework\RetryCurrentTransaction.xaml

Table 9 provides details about the arguments of **RetryCurrentTransaction.xaml**, located in the **Framework** folder. This workflow manages 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**. As mentioned in the *Settings* section, if the **MaxRetryNumber** constant is zero, the management of retries is handled by Orchestrator. If **MaxRetryNumber** is greater than zero, the management of retries is handled locally by the framework.

Table 10 - Arguments of TakeScreenshot.xaml

Argument	Description	Default Value
in_Folder	Path to the folder where the screenshot should be saved.	<i>No default value</i>
io_FilePath	Optional argument that specifies the path and the name of the screenshot to be taken.	<i>No default value</i>

Framework\TakeScreenshot.xaml

This workflow, located in the **Framework** folder, captures a screenshot of the whole screen and saves it with the PNG extension in a folder specified by the argument **in_Folder** (Table 10).

TakeScreenshot.xaml is invoked when there are exceptions during the processing of a transaction. Although it is used for all processes by default, this feature is particularly helpful when debugging issues that happen during the execution of unattended processes, providing clues even when there is no human supervising the robot and seeing the problem happen live.

If an Orchestrator queue is used and a transaction item fails due to a **System Exception**, the path of the screenshot is saved in the **Transaction Item Details**.

Framework\CloseAllApplications.xaml

This workflow, located in the **Framework** folder, does the necessary procedures for ending the process and closing the used applications. Similar to **OpenAllApplications.xaml**, activities can be placed directly in this workflow or, preferably, sub-workflows can be invoked to perform more complex steps, such as logging out of a system.

The REFramework is available as a UiPath Studio project template (Figure 8), and when creating a new project starting from the REFramework template, it will automatically include all the files explained above.

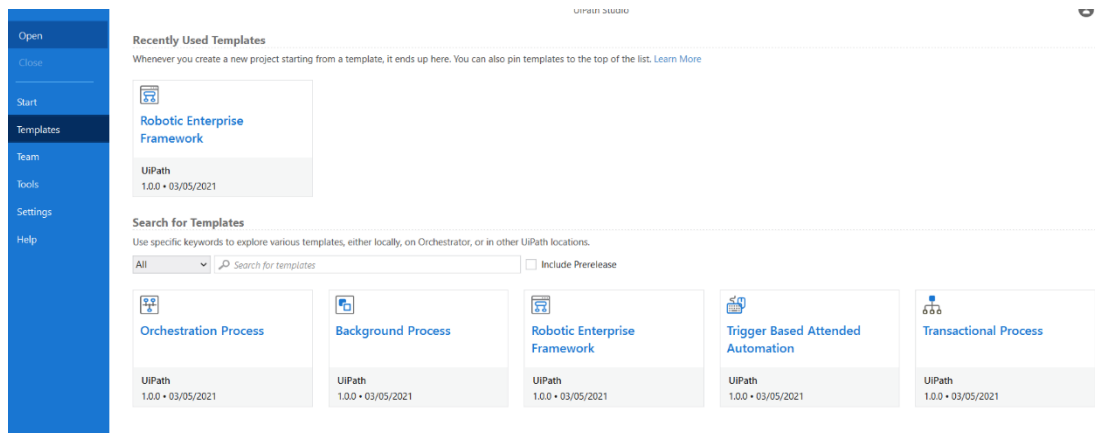


Figure 8 - Template Menu in UiPath Studio's Home Screen.

Changes to Framework Files

After the project is created, the following files need to be modified according to the requirements of the process to be automated.

Data\Config.xlsx

Other than adding the necessary settings, constants and assets that depend on the process, make the following modifications:

1. Change the value of the **logF_BusinessProcessName** setting to match the name of the process. This value is used for logging purposes, and it is going to be included in all the log messages generated by the framework when this process is executed.
2. If the source of transactions is an Orchestrator queue, change the value of the **OrchestratorQueueName** setting to match the name of the queue as defined in Orchestrator. If the process does not use a queue, then it is safe to delete this row and change the value of **MaxRetryNumber** in the **Constants** sheet to an integer greater than zero. This indicates the number of times a robot should retry a transaction that fails with a system exception (refer to the Settings section for details).

3. If the source of transactions is an Orchestrator queue and the Orchestrator folder of the queue is different than the one where the process will be running, specify the Orchestrator folder of the queue in the **OrchestratorQueueFolder** setting.

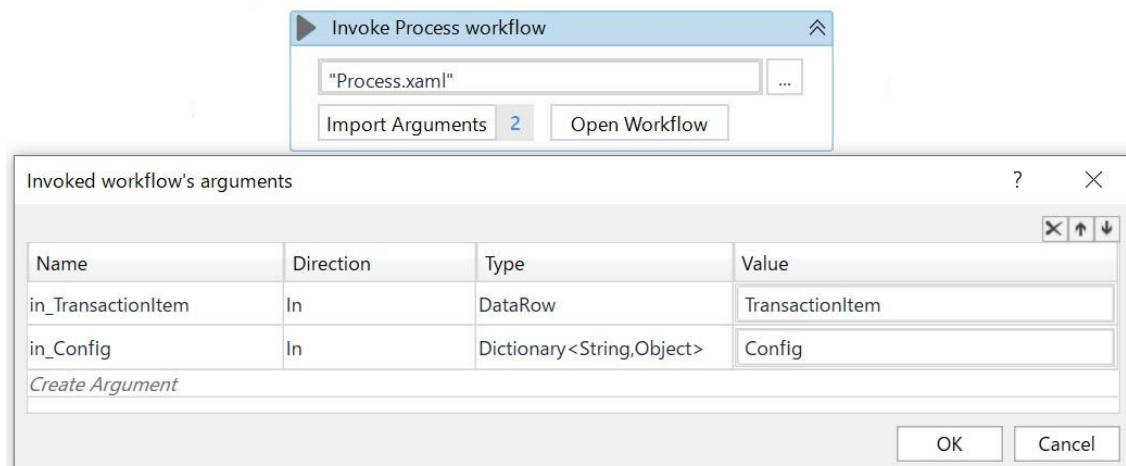


Figure 9 - Updating Arguments of Invoked Workflows.

Main.xaml

First, set the type of the **TransactionItem** variable according to the type of the process transaction. The default type is **QueueItem**, but it can be changed, for example, to **DataRow** in case rows are being read from an Excel file or to **MailMessage** in case emails are retrieved from an email account.

If queues are used, there is no need for further modifications. However, if the type is changed, the following workflows should also be updated, since they expect the variable **TransactionItem** to be of type **QueueItem**: **GetTransactionData.xaml**, **Process.xaml** and **SetTransactionStatus.xaml**. Section Practical Example 2: Using Tabular Data provides an example of how to do such updates.

After the above workflows are adjusted, it is also necessary to update the arguments passed by the corresponding Invoke Workflow File activities: **GetTransactionData.xaml** is invoked in the **Get Transaction Data** state, and both **Process.xaml** and **SetTransactionStatus.xaml** are invoked in the **Process Transaction** state. Updating arguments can be done by clicking the Import Arguments button of the Invoke Workflow File activity and entering the variables that are passed to the adjusted arguments, as shown in Figure 9.

If Orchestrator queues are used, the transaction retrieval is handled by the **Get Transaction Item** activity included by default, and it is not necessary to make any modifications to the **GetTransactionData.xaml** workflow.



Figure 10 - Configuration for Processes with a Single Transaction.

If transactions are of types other than **QueueItem**, change the type of the **out_TransactionItem** argument to match the process's transaction type (for example, **DataRow** or **MailMessage**). To define a new data source, replace the first activity of this workflow, **Get Transaction Item**, with appropriate data retrieval. For example, use the **Read Range** activity to retrieve data from a spreadsheet and save it to the **io_TransactionData** argument. After that, make sure that the new type of **out_TransactionItem** is reflected in the Invoke Workflow File activity that invokes this workflow in the **Get Transaction Data** state of **Main.xaml**.

Once the data source is defined, it is necessary to include steps to get transaction items. For cases in which there is only a single transaction, check whether the argument **in_TransactionNumber** has the value 1 (meaning it's the first and only transaction) and assign the transaction item to **out_TransactionItem**. For any other value of **in_TransactionNumber**, **out_TransactionItem** should be set to Nothing (Figure 10).

If there are multiple transactions, use the argument **in_TransactionNumber** as an index to retrieve the correct transaction to be processed. If there are no more transactions left, it is necessary to set **out_TransactionItem** to Nothing, thus ending the process (Figure 11).

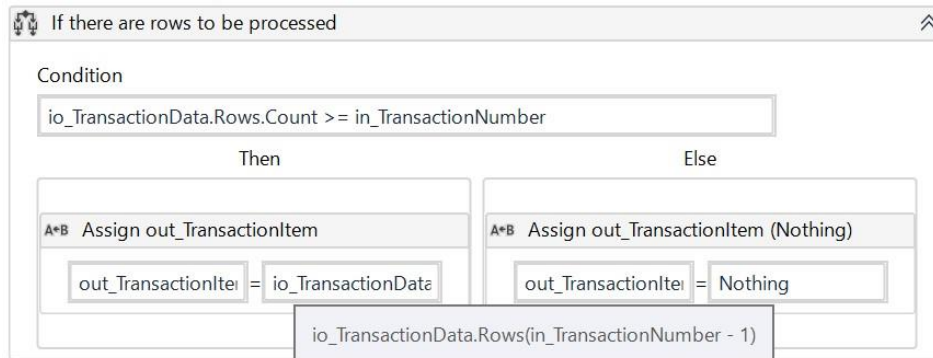


Figure 11 - Configuration for Transactional Process (Multiple Transactions).

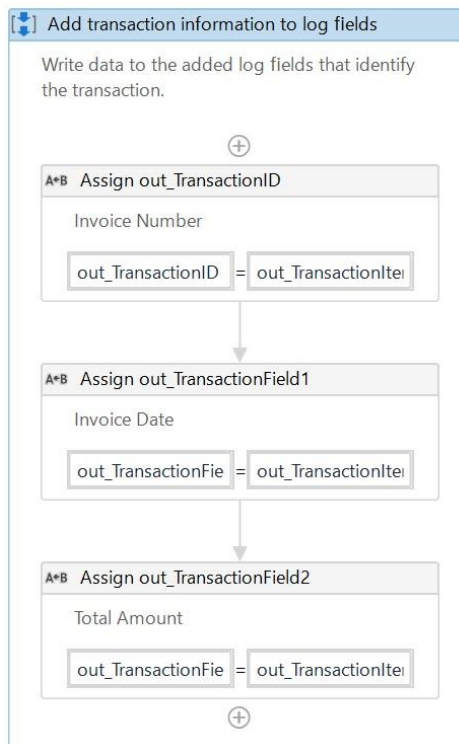


Figure 12 - Configuration of Custom Log Fields.

Optionally, it is possible to add information about the transaction item using the **Assign** activities in the sequence named **Add transaction information** to log fields at the end of this workflow. For example, for creating reports about an invoice processing automation, one might use **out_TransactionID** to store invoice number, **out_TransactionField1** to store invoice date and **out_TransactionField2** to store the total amount, as mentioned in section Logging and illustrated in Figure 12.

Framework\Process.xaml

No special changes need to be made to **Process.xaml** if Orchestrator queues are used. Each transaction item is accessible in this workflow via the argument



in_TransactionItem. For instance, in an invoice processing automation project, `in_TransactionItem.SpecificContent("InvoiceNumber")` can be used to retrieve the invoice number and `in_TransactionItem.SpecificContent("TotalAmount")` may be used to obtain the total amount.

If Orchestrator queues are not used, set the type of the **in_TransactionItem** argument to match the type defined for the variable **TransactionItem** in **Main.xaml**. After that, make sure that the new type of **in_TransactionItem** is reflected in the **Invoke Workflow File** activity that invokes this workflow in the **Process Transaction** state of **Main.xaml**.

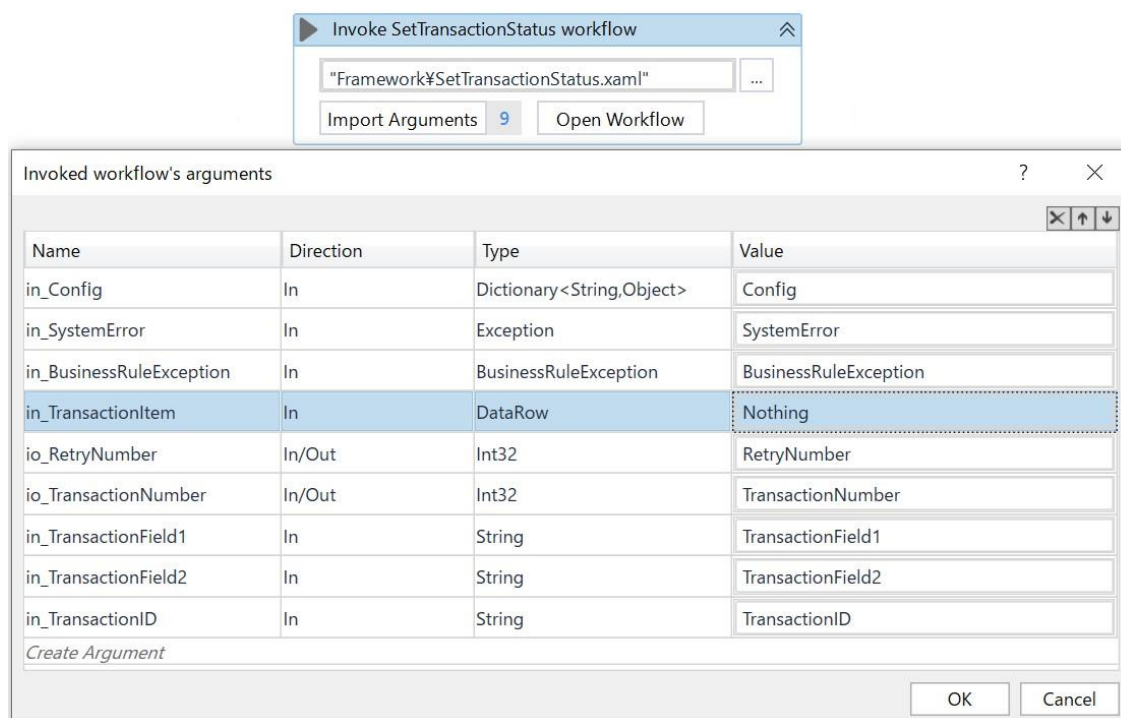
Framework\SetTransactionStatus.xaml

As mentioned in section **Framework\SetTransactionStatus.xaml**, this workflow is called after the **Process.xaml** workflow is executed, and it sets the status of the transaction according to the result of the processing step.

If the process's data source is an Orchestrator queue, the status of the queue item is updated by the **Set Transaction Status** activity by default and no further changes are necessary.

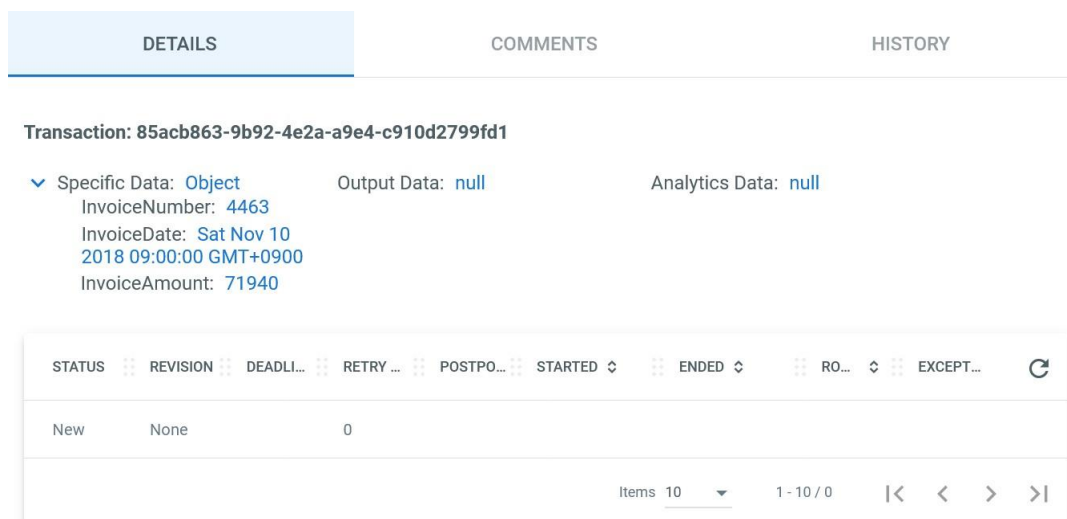
For processes that do not use an Orchestrator queue, in addition to adjusting the type of the **in_TransactionItem** argument, the appropriate steps must be implemented to set the transaction status. After that, make sure that the new type of **in_TransactionItem** is reflected on the **Invoke Workflow File** activity that invokes this workflow in the **Process Transaction** state of **Main.xaml**.

If it is not desirable to track statuses of transactions, then it is possible to keep the type of the **in_TransactionItem** argument as it is (i.e., **QueueItem**) and simply pass the value **Nothing** to the corresponding argument of the **Invoke Workflow File** activity in the **Process Transaction** state of **Main.xaml**, as illustrated by Figure 13.



Name	Direction	Type	Value
in_Config	In	Dictionary<String,Object>	Config
in_SystemError	In	Exception	SystemError
in_BusinessRuleException	In	BusinessRuleException	BusinessRuleException
in_TransactionItem	In	DataRow	Nothing
io_RetryNumber	In/Out	Int32	RetryNumber
io_TransactionNumber	In/Out	Int32	TransactionNumber
in_TransactionField1	In	String	TransactionField1
in_TransactionField2	In	String	TransactionField2
in_TransactionID	In	String	TransactionID

Figure 13 - Configuring Arguments when Invoking SetTransactionStatus.xaml.



Transaction: 85acb863-9b92-4e2a-a9e4-c910d2799fd1

Specific Data: **Object**
 InvoiceNumber: 4463
 InvoiceDate: Sat Nov 10 2018 09:00:00 GMT+0900
 InvoiceAmount: 71940

Output Data: null Analytics Data: null

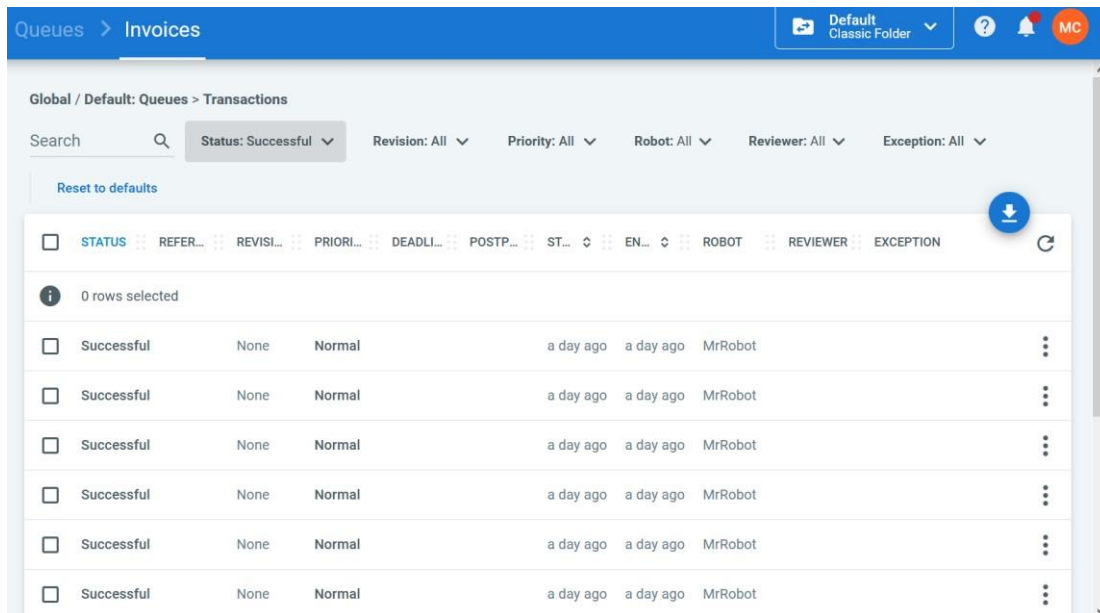
STATUS	REVISION	DEADLI...	RETRY ...	POSTPO...	STARTED	ENDED	RO...	EXCEPT...
New	None		0					

Items 10 1 - 10 / 0 |< < > >|

Figure 14 - Details of a Queue Item from the "Invoices" Queue.

Practical Example 1: Using Queues

For the first practical example, consider that an Orchestrator queue named *Invoices* is created and populated with data about invoices, such as invoice number, date and total amount (Figure 14).



Global / Default: Queues > Transactions

Search Status: Successful Revision: All Priority: All Robot: All Reviewer: All Exception: All

Reset to defaults

STATUS	REFER...	REVISI...	PRIORI...	DEADLI...	POSTP...	ST...	EN...	ROBOT	REVIEWER	EXCEPTION
0 rows selected										
<input type="checkbox"/>	Successful	None	Normal		a day ago	a day ago		MrRobot		
<input type="checkbox"/>	Successful	None	Normal		a day ago	a day ago		MrRobot		
<input type="checkbox"/>	Successful	None	Normal		a day ago	a day ago		MrRobot		
<input type="checkbox"/>	Successful	None	Normal		a day ago	a day ago		MrRobot		
<input type="checkbox"/>	Successful	None	Normal		a day ago	a day ago		MrRobot		
<input type="checkbox"/>	Successful	None	Normal		a day ago	a day ago		MrRobot		

Figure 15 - Updated Statuses of Queue Items.

As detailed in the *Changes to Framework Files* section, the processing of invoices can be implemented by the following steps:

1. In the **Settings** sheet of the file **Data\Config.xlsx**, change the value of the **OrchestratorQueueName** parameter to **Invoices** and the value of **logF_BusinessProcessName** to **InvoiceProcessingSample**.
2. In the **InitAllApplications.xaml** file, invoke workflows that implement opening and logging into applications used in the process, such as an invoice registration system.
3. In **CloseAllApplications.xaml**, invoke workflows that carry out the logging out and the closing of the used applications. Optionally, use the **KillAllProcesses.xaml** file for additional cleanup steps.
4. In **Process.xaml**, invoke the necessary workflows to implement the actual invoice processing steps, like accessing the appropriate screens of the registration system and using activities like **Click** and **Type Into** to register each invoice.

Note that in case an Orchestrator queue is used as data source, only a few modifications are necessary to be implemented into the framework. It automatically communicates with the queue set in the configuration file, retrieves one transaction item at a time and updates the status of the item according to the result of the processing (Figure 15).

	A	B	C	D
1	Number	Date	Total	Processed
2	1009	2018/04/05	2952	
3	8824	2018/10/21	64125	
4	3803	2018/01/04	80269	
5	6550	2018/01/10	91526	
6	4433	2018/10/01	28177	
7	3867	2018/04/04	17614	
8	7329	2018/05/31	99886	
9	4162	2018/07/30	65745	
10	5222	2018/08/09	46538	
11	8523	2018/04/26	49064	
12	5850	2018/09/19	98002	
13	9663	2018/04/05	45369	
14	3360	2018/12/12	2294	
15	1018	2018/06/10	55155	
16	4327	2018/09/05	46983	
17	1398	2018/08/07	6911	
18	9568	2018/05/16	94112	
19	2961	2018/12/13	43388	

Figure 16 - Invoice Data in a Spreadsheet.

A	B
Name	Value
logF_BusinessProcessName	InvoiceProcessingSample
SampleDataFilepath	Data\Input\InvoiceSampleData.xlsx

Figure 17 - Settings sheet in Config.xlsx.

Practical Example 2: Using Tabular Data

The second example uses invoice data stored in an Excel spreadsheet, as shown in Figure 16. Each row of the spreadsheet contains data about a single invoice, so the data should be loaded into the **TransactionData** variable and the type of **TransactionItem** should be changed to **DataRow**. To do so, make the following changes:

1. Similar to the first example, in the **Settings** sheet of the file **Data\Config.xlsx**, change the value of **logF_BusinessProcessName** to *InvoiceProcessingSample*. However, since the data source is not an Orchestrator queue, delete the row corresponding to **OrchestratorQueueName**. Add a new setting parameter by using *SampleDataFilepath* as the name and, as the value, specify the path for the input Excel file that has data about invoices to be processed, such as *Data\Input\InvoiceSampleData.xlsx* (Figure 17).

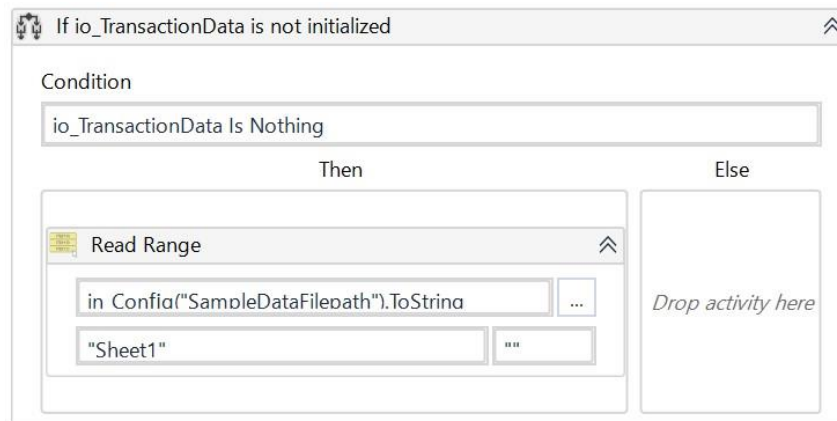


Figure 18 - Initializing Data Source (*io_TransactionData*).

2. In the **Constants** sheet of the file **Data\Config.xlsx**, change the value of **MaxRetryNumber** to an integer greater than zero. As detailed in the Settings section, this value indicates the number of times the processing should be retried in case of system exceptions. For this example, change it to 2.
3. In the **Process.xaml** workflow, change the type of the argument **in_TransactionItem** to **DataRow** instead of **QueueItem**. Also, invoke the necessary workflows to implement the actual invoice processing steps, such as accessing the appropriate screens of the registration system and using activities like **Click** and **Type Into** to register each invoice.
4. In **GetTransactionData.xaml**, other than changing the type of **out_TransactionItem** to **DataRow**, delete the existing **Get Transaction Item** activity, since this example does not use Orchestrator queues. Two checks are necessary to correctly retrieve transaction items in this case, and they are implemented as follows:
 - a. Add an **If** activity that checks whether the data source was initialized with the condition *io_TransactionData Is Nothing* (Figure 18). If it was not initialized, read the spreadsheet from the designated Excel file by using the **Read Range** activity and the path defined in the configuration file: *in_Config("SampleDataFilepath").ToString*.
 - b. After that, it is necessary to implement the logic to retrieve one row each time the **GetTransactionData.xaml** is executed. To do so, add another **If** activity and use the condition *io_TransactionData.Rows.Count >= in_TransactionNumber*, which verifies whether there are rows to be processed. If there are unprocessed rows, use an **Assign** activity to set the appropriate row to be the current transaction item:

`io_TransactionData.Rows(in_TransactionNumber - 1)`. Note that the argument **in_TransactionNumber** is used to track the row currently being processed. If there is no unprocessed row left, set **out_TransactionItem** to **Nothing** (Figure 11). This action is necessary to prevent the framework from attempting to retrieve new transactions.

5. Make the following modifications to the **SetTransactionStatus.xaml** file so that the statuses of transactions are tracked in the **Processed** column (Figure 16) of the input spreadsheet. First, change the type of the argument **in_TransactionItem** to **DataRow** instead of **QueueItem** and implement the following steps to update the **Processed** column of the input Excel file with the result of the processing:

- a. In the sequence called *Success*, delete the **If** activity named *If TransactionItem is a QueueItem (Success)* and add a **Write Cell** activity instead. In the properties of this activity, assign "Yes" to the **Text** property, `in_Config("SampleDataFilepath").ToString` to the **WorkbookPath** property and `"D" + (io_TransactionNumber + 1).ToString` to the **Cell** property (Figure 19). "D" refers to the **Processed** column and `io_TransactionNumber + 1` skips the table header and writes to the correct row.

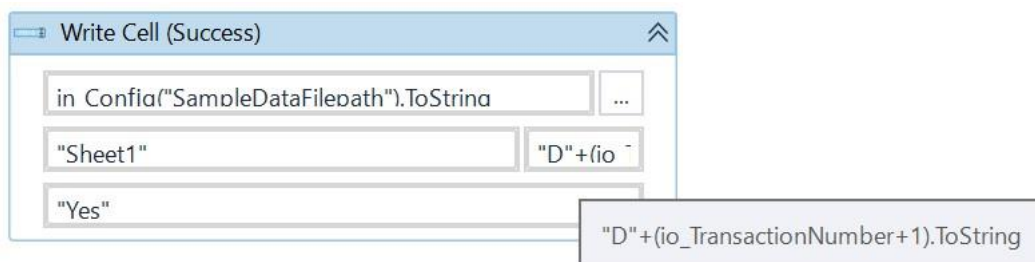


Figure 19 - Configuration of Write Cell (Success).

- b. Similar to the previous step, in the sequence called *Business Exception*, delete the **If** activity named *If TransactionItem is a QueueItem (Business Exception)* and add a **Write Cell** activity instead. The values of the properties of this activity are the same as the success case, except that the property **Text** should have the value "No (Business Rule Exception)".
- c. Finally, in the sequence called *System Exception*, delete the **If** activity named *If TransactionItem is a QueueItem (System Exception)* and add a **Write Cell** activity instead. Once again, use the same values for the properties set in case of success, except for the **Text** property, which should be set to "No (System Exception)".

6. In **Main.xaml**, change the type of the variable **TransactionItem** to **DataRow** instead of **QueueItem**. Notice that this change raises alerts in different parts of the workflow, indicating that the new type is not compatible with the type of the arguments previously defined in the **Invoke Workflow File** activities. Click the **Import Arguments** button of the following **Invoke Workflow File** activities and set the **TransactionItem** variable to the corresponding argument:
 - a. *Invoke GetTransactionData workflow* in the **Get Transaction Data** state.
 - b. *Invoke Process workflow* in the **Process Transaction** state (Try block of Try Catch activity).
 - c. *Invoke SetTransactionStatus workflow* in the **Process Transaction** state (Try and Catch blocks of Try Catch activity).
7. In the **InitAllApplications.xaml** file, invoke workflows that implement opening and logging into applications used in the process, such as an invoice registration system.
8. In **CloseAllApplications.xaml**, invoke workflows that perform the logout and the closing of the used applications. Optionally, use the **KillAllProcesses.xaml** file for additional cleanup steps.
9. In **Process.xaml**, invoke the necessary workflows to implement the actual invoice processing steps, like accessing the appropriate screens of the registration system and using activities like **Click** and **Type Into** to register each invoice.

To summarize, the steps above read data about invoices from an Excel file and use each row of the file as a transaction. After processing a transaction, the framework updates the **Processed** column according to the result of the processing (i.e., success, business exception and system exception).

Lastly, note that the same steps can be applied for other types of transactions, such as emails (**MailMessage**) and paths to files (**String**).

Test Framework

The REFramework also includes a testing feature that makes it easier to do automatic testing of workflows. Instead of testing them one by one and checking the results manually, it is possible to specify the predicted outcome of a workflow (i.e., successful execution, business exception and system exception) and see whether the actual results matched the expected results.



The TestSuite integration with REFramework provides an easy method of implementing unit testing of REFramework components.

It contains several files related to testing different parts of the framework:

Tests\Tests.xlsx - an Excel file that contains a sheet: Tests. In the Tests sheet, the developer will write the workflow paths of the workflows to be tested and the expected exception - SystemException, BusinessRuleException or Success.

After running **GeneralTestCase.xaml**, there is going to be a clear result of comparing the expected result with the actual result after passing through this list of workflows. There are two possible statuses - **PASS** or **FAIL** for each workflow tested. The status is PASS if the actual exception caught is the one previously defined in the Tests sheet and FAIL otherwise.

We identified the need to test each part of the REFramework individually with the right context set, therefore a set of unit tests per part of process were created:

Tests\InitAllSettingsTestCase.xaml - Verifies if the Config dictionary was created and contains information. May verify if a certain key is present in dictionary (e.g., MaxRetryNumber)

Tests\InitAllApplicationsTestCase.xaml - Having the Config set, we can now test if the initialization of applications works as expected. The Verify Control Attribute should be used to check if an element that is supposed to be on the screen after the initialization of the applications is indeed there.

Tests\ProcessTestCase.xaml - Having the Config set and applications open, this workflow should be modified such as the TransactionItem is created/obtained from a test queue and provide it as input for the processing part. Also, an output argument should be returned if the processing part worked as expected, otherwise it will just assume that if the workflow did not throw an exception, the test case should pass.

Tests\MainTestCase.xaml - This test case assumes that the process has a reporting system in place. It should be configured to verify if the obtained output is the same as the expected output.

The Test Cases presented above should be treated as samples. Please modify them and add new ones as required by your process.

UiPath team highly recommends having a CI/CD pipeline in place for each project.



Distribution and Support to Extensions

The REFramework is available under the MIT License and distributed as a template in UiPath Studio or via <https://github.com/UiPath-Services/StudioTemplates>.

Regarding the adaptation of the framework to particular use cases and transaction types, it is encouraged that customers and partners understand the steps for extensions and implement such modifications to better suit their needs. For an example of extension using spreadsheet data, refer to section Practical Example 2: Using Tabular Data. Alternatively, templates based on the REFramework can be downloaded from the UiPath Marketplace (<https://marketplace.uipath.com/>).