Transaction Processing

What is a transaction?

A transaction 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.

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 state on each transaction. The States section gives more details about specifying a source of transactions and how each one is processed.

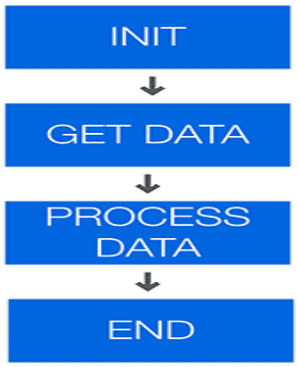
Considering the steps of a business process and how they are repeated, we can divide business processes into three categories:

1. Linear
2. Iterative
3. Transactional

1. Linear:

The steps of the process are performed only once and, if there is the need to process different data, the automation needs to be executed again. For example, if we go back to the email example from this chapter's introduction, and a new email arrives, the automation needs to be executed again in order to process it.

Linear processes are usually simple and easy to implement, but not very suitable to situations that require repetitions of steps using different data.



2. Iterative:

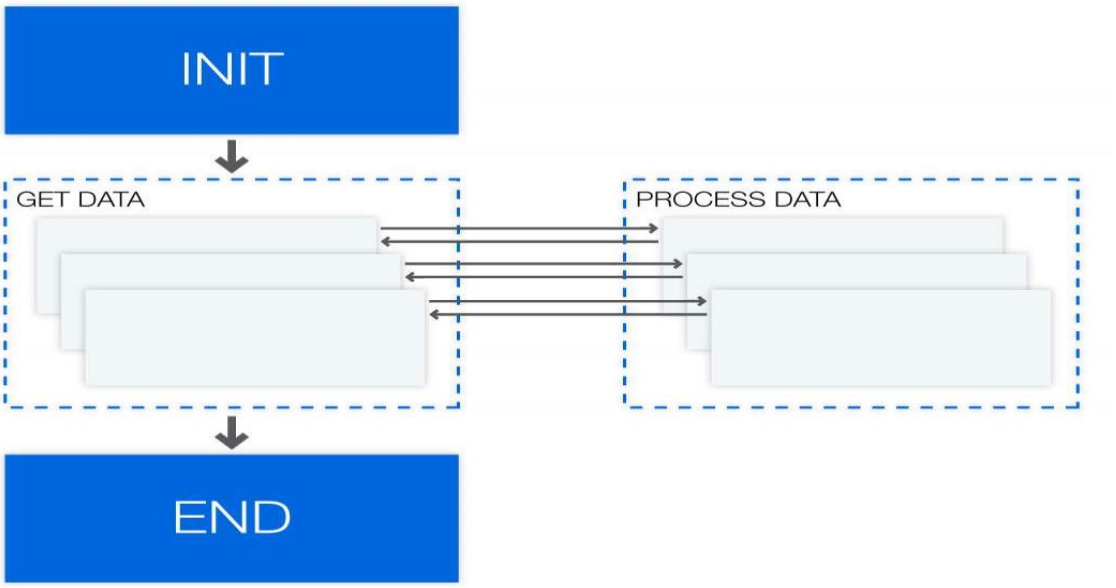
The steps of the process are performed multiple times, but each time different data items are used. For example, instead of reading a single email on each execution, the automation can retrieve multiple emails and iterate through them doing the same steps.

This kind of process can be implemented with a simple loop, but it has the disadvantage that, if a problem happens when processing one item, the whole process is interrupted and the other items remain unprocessed.

3. Transactional:

Similarly to iterative processes, the steps of transactional processes repeat multiple times over different data items. However, the automation is designed so that each repeatable part is processed independently.

These repeatable parts are then called transactions. Transactions are independent from each other, because they do not share any data or have any particular order to be processed.



The three categories of processes can be seen as maturity stages of an automation project, starting with simple linear tasks, which then are repeated multiple times and finally evolve into a transactional approach.

However, this is not a absolute rule for all cases, and the category should be chosen according to the characteristics of the process (e.g., data being processed and frequency of repetitions) and other relevant requirements (e.g., ease of use and robustness).

What are some business scenarios in which I will use transaction processing?

- You need to read data from several invoices that are in a folder and input that data into another system. Each invoice can be seen as a transaction, because there is a repetitive process for each of them (i.e. extract data and input somewhere else).
- There is a list of people and their email addresses in a spreadsheet, and an email needs to be sent to each of them along with a personalized message. The steps in this process (i.e., get data from spreadsheet, create personalized message and send email) are the same for each person, so each row in the spreadsheet can be considered a transaction.
- When looking for a new apartment, a robot can be used to make a search according to some criteria and, for each result of the search, the robot extracts the information about the property and insert the data into a spreadsheet. In this case, the details page for each property constitutes a transaction.

The REFramework

What is it?

Generally speaking, a framework is a template that helps you design (automation) processes. At a barebones minimum, a framework should offer a way to store, read, and easily modify project configuration data, a robust exception handling scheme, event logging for all exceptions and relevant transaction information.

REFramework is a template which provides all the required basic needs for any process automation, like

- Reading and storing the Configuration data
- Closing all the unnecessary applications
- Killing all the processes
- Open's all the required applications
- Get the transaction and Process it
- Retrying the transactions if required, i.e. Getting an Application exception
- Logging the status of all processed transactions; i.e. Successful or Failed
- After processing of all the transactions then closing the opened applications successfully. Suppose if we are not able to close the opened applications the we can kill those applications based on process name

Why we are using the REFramework template for any Business Process Automation?

REFramework helps developers to build processes quicker, robust, and apply the best practices.

The **REFramework** is implemented as a **State Machine**, which is a type of workflow that has two very useful features:

- States that define actions to be taken according to the specified input
- Transitions that move the execution between states depending on the outcomes of the states themselves.

You may remember state machines from the Project Organization course. One of the examples presented was the typical air conditioner:

- It has the OFF State, from where it moves to an IDLE State by pressing the ON/OFF button;

- From the IDLE State, it moves to either HEAT or COLD States when the temperature inputted by the user is lower and higher respectively than the current one. Once the desired temperature is achieved, it moves back to the IDLE State;
- From the IDLE State, it can move to the OFF State when the ON/OFF button is pressed;
- All the conditions that trigger the movements between the states are Transitions.

REFramework Architecture

What is Transition?

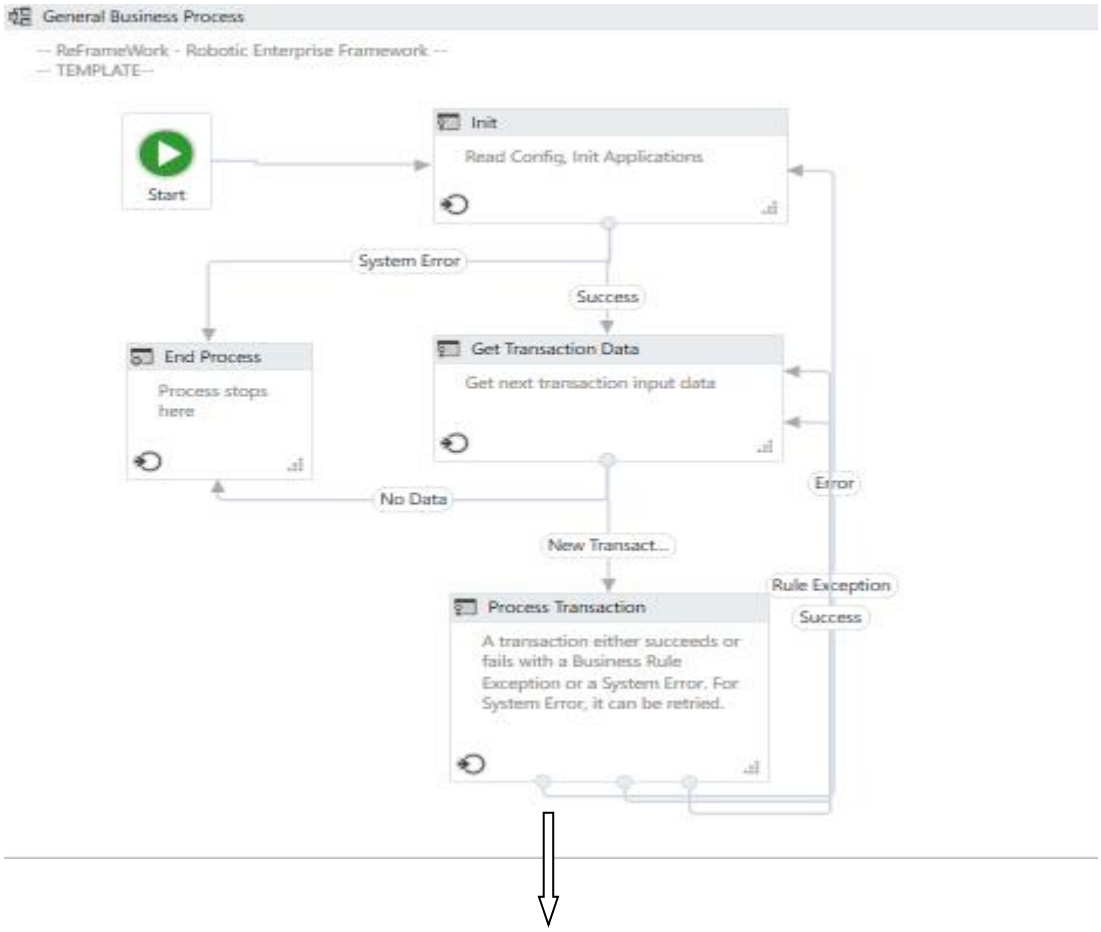
A directed relationship between two states which represents the complete response of a state machine to an occurrence of an event of a particular type.

Transition enable you to add conditions based on which to jump from one state to another. These are represented by arrows or branches between states.

Transition Activity:

An activity which is executed when performing a certain transition

Based on a similar idea, REFramework has 4 main states that are usually common to business processes:



State Machine with the States of the REFramework

1. Initial State

2. Get Transaction State
3. Process Transaction State
4. End Process State

1. Initial State:

This is where the process starts. It's an operation where the process initializes the settings and performs application checks in order to make sure all the prerequisites for the starting the process are in place.

Generally, this is the Initial State which is used to read and store the configuration data in a Dictionary, close all the unnecessary applications and Kill the processed if its required.
Based on outcome from the "**Init**" state, here Robot will be moving to another state.

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.

2. Get Transaction State:

Gets the next transaction item. This can be a queue item or any item of a collection.
By **default**, **Transaction Items** are **Queue Items**, but this can be easily changed to suit your needs. This is also the state in which the Developer should set up the condition to exit this state when there are no items to process.

Generally, it is the **data retrieval mechanism**. It's used to **get the transaction item** either from **Queue**, or **Data Table**, or **Mail Box**, or **Database**, or **Folder**, or **String**.

Based on the outcome from the "**Get Transaction Data**" state, here Robot will moving to the another state.

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**, its processed in the **Process Transaction** state.

3. Process Transaction State:

Processing of a Single Transaction. It is used to process the transaction which Robot has retrieved from the previous state; i.e. "**Get Transaction Data**".

The result of the processing can be Success, Rule Exception (Business Exception) or Error (System Exception).

Based on the outcome from the "**Process Transaction**" state, here Robot will moving to the another state.

Suppose while processing the transaction item, if we are getting any "**Rule Exception** (**Business Exception)**", then the current transaction is skipped, and the Framework tries to retrieve a new transaction in the "**Get Transaction Data**" state.

The execution is also returns to the "Get Transaction Data" state to retrieve a new transaction if the processing of the current transaction is successful.

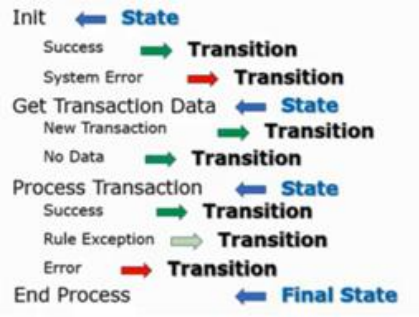
Otherwise while processing the transaction item, if we are getting any "**Error** (**Application/System Exception**)" then it will be moving to the "**Init**" state and the processing of the current transaction can be automatically retried.

4. End Process State:

Finalize the process and close all applications used.

REFramework has "Main.xaml" file which uses the State Machine workflow.

REFramework is prepared with 3 State Activities, one Final State Activity and 7 Transitions. Each state has connected with each other with each other with Transitions.



Below is the list of the Workflows Invoked in States:

State	Invoked Workflows
Initialization	I. InitAllSettings.xaml II. KillAllProcesses.xaml III. InitAllApplications.xaml
Get Transaction Data	GetTransactionData.xaml
Process Transaction	I. Process.xaml II. SetTransactionStatus.xaml • RetryCurrentTransaction.xaml • TakeScreenshot.xaml • CloseAllApplications.xaml • KillAllProcesses.xaml
End Process	I. CloseAllApplications.xaml II. KillAllProcesses.xaml

Shared Variables in REFramework:

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

One important variable that is passed to almost all 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

S.No.	Name	Default Type	Description

1	**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 Practical Example 2: Using Tabular Data for a sample). In another scenario, if transactions are paths to image files to be processed, this variable's type can be changed to **String**.
2	**SystemException**	**Exception**	Used during transitions between states to represent exceptions other than **BusinessRuleException**.
3	**BusinessException**	**BusinessRuleException**	Used during transitions between states and represents a situation that does not conform to the rules of the process being automated.
4	**TransactionNumber**	**Int32**	Sequential counter of transaction items.
5	**Config**	**Dictionary(Of String, Object)**	Dictionary structure to store configuration data of the process (settings, constants and assets).
6	**RetryNumber**	**Int32**	Used to control the number of attempts of retrying the transaction processes in case of system exceptions.
7	**TransactionField1**	**String**	Optionally used to include additional information about the transaction item.
8	**TransactionField2**	**String**	Optionally used to include additional information about the transaction item.
9	**TransactionID**	**String**	Used for information and logging purposes. Ideally, the ID should be unique for each transaction.
10	**TransactionData**	**DataTable**	Used in case transactions are stored in a **DataTable**, for example, after being retrieved from a spreadsheet

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

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

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).	No default value
out_TransactionItem	Transaction item to be processed.	No default value
out_TransactionField1	Optionally used to include additional information about the transaction item.	No default value
out_TransactionField2	Optionally used to include additional information about the transaction item.	No default value
out_TransactionID	Used for information and logging purposes. Ideally, the ID should be unique for each transaction.	No default value
io_TransactionData	Used in case transactions are stored in a DataTable, for example, after being retrieved from a spreadsheet.	No default value

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

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

Arguments of RetryCurrentTransaction.xaml:

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

Arguments of TakeScreenshot.xaml:

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

REFramework Main Features

Settings:

In many processes, it is common to have certain settings and configuration values that are read during the initialization phase.

Examples of settings include URLs to access web applications, Orchestrator Queue Names, Folder Path's and default logging messages.

The REFramework keeps track of such data by reading them from a configuration file (**Config.xlsx**) and storing them in a Dictionary object (Config) that is shared among the different states. This offers an easy way to maintain projects by changing values in the configuration file, instead of modifying workflows directly.

Constants:

Values that are supposed to be the same across all deployments of the workflow. For example, department name or bank name to be input in a certain screen.

Assets:

Values defined as assets in Orchestrator.

Rows in 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 specifies 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**.

Below table provides an example of how to define application settings in the Settings sheet:

Name	Value	Description
ACME_URL	https://acme-test.uipath.com/account/login	ACME Login URL
ACME_Credential	ACME_Login	ACME Login Credentials
Result	Data\Result.xlsx	This file having the result of the Filtered Records of WorkItems

Then, during the implementation of workflows, developers can use **Config("ACME_URL")** to retrieve the value **https://acme-test.uipath.com/account/login**. Below illustrates this relationship between the configuration file Config.xlsx and the Config dictionary.

Config.xlsx Configuration File

Name	Value	Description
ACME_URL	https://acme-test.uipath.com/account/login	ACME Login URL
ACME_Credential	ACME_Login	ACME Login Credentials
Result	Data\Result.xlsx	This file having the result of the Filtered Records of WorkItems

Config Dictionary

Key	Value	Usage
ACME_URL	https://acme-test.uipath.com/account/login	Config("ACME_URL").ToString
ACME_Credential	ACME_Login	Config("ACME_Credential").ToString
Result	Data\Result.xlsx	Config("Result").ToString

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**.

If an **Orchestrator Queue** is being used as a **source of transactions**, then the **value of MaxRetryNumbe**r 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**.

Note:

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:

Another powerful feature of the REFramework is the built-in **logging mechanism**. Most of the workflows that compose the framework use Log Message activities that output details of what is happening in each step of the execution.

This can be used not only to find problems and help in the debugging process, but also to create visualizations and reports about the execution of the process (for example, how many invoices are processed each day, how many failures happen and what are the main causes of failures) and about the process itself (for example, what is the aggregated value of all reports processed in a month).

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. 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**.

Business Exception & Application Exception:

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

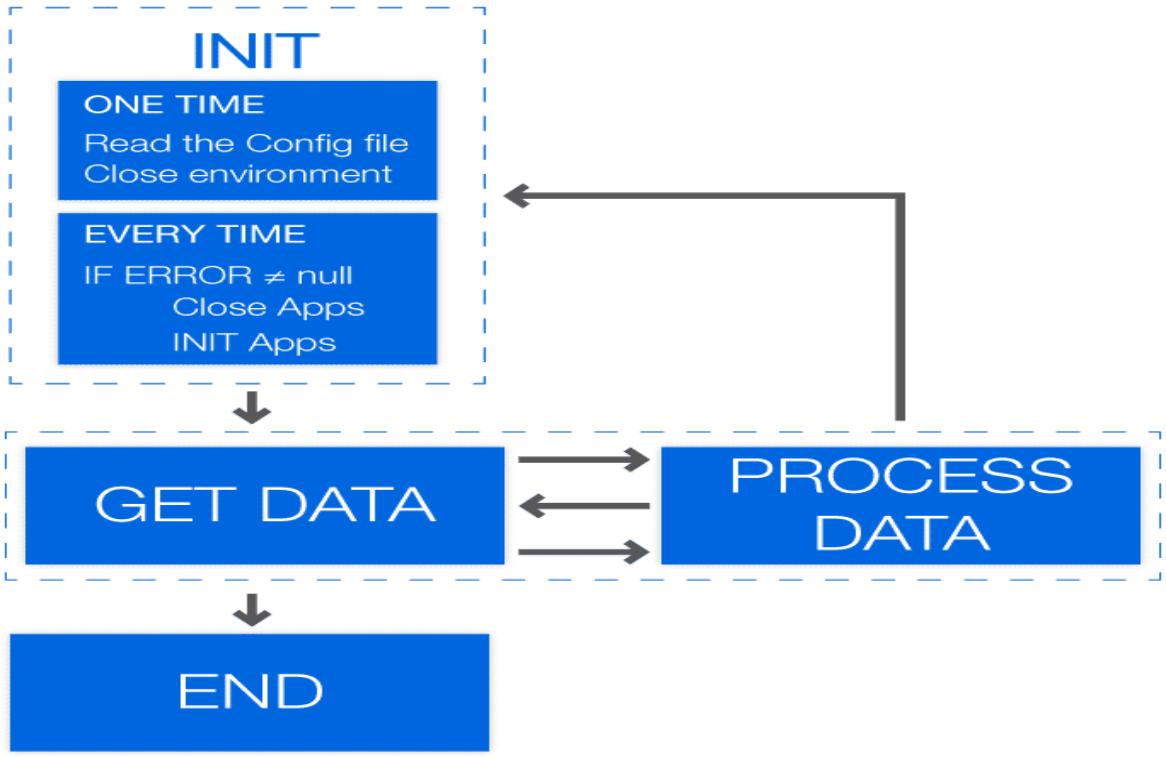
Business Exceptions: This kind of exception is implemented by the class BusinessRuleException and it should be thrown when there are problems related to rules of the business process being automated. For example, if a process expects to receive an email with an attachment, but the attachment does not 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 to be followed. Note that BusinessRuleExceptions must be explicitly thrown by the developer of the workflow, and they are not automatically thrown by the framework or activities.

Application (System Exceptions): If an exception is not related to rules of the process itself, it is considered Application/System exception. Examples of Application/System exceptions include an activity that timed-out due to slow network connection or a selector not found because of a browser crash.

REFramework Sample Scenario

To have a better understanding of how we can use REFramework, let's go through the following scenario:

There is a list of people and their email addresses in a spreadsheet. An email needs to be sent to each of them with a personalized message based on a template.



State	Description
Init	Read data from spreadsheet and read the email template from a text file
Get Transaction Data	Retrieve one row from spreadsheet, which contains the name and the email of one person.
Process Transaction	Insert the person's name and email into the email template to create a personalized message. After that, send the email using this message.
End Process	After all rows are processed, finalize the execution. Since emails can be sent directly by the robot, there is no need to close any email applications.

Dispatcher & Performer

Although the REFramework can be used with different types of data sources, it provides a particularly **strong integration** with **Orchestrator queues**. When queues are used, it is possible to define priorities and deadlines for transaction items and to track retrying attempts of failed transactions.

The use of queues also enables an **execution pattern** called **Dispatcher** & **Performer**, which divides the process in **two main phases**: **dispatching items to be processed** and **adding them to a queue**, followed by **retrieving an item from a queue** and **performing the process** using that item. The **second part** of the process is generally built using the **REFramework**.

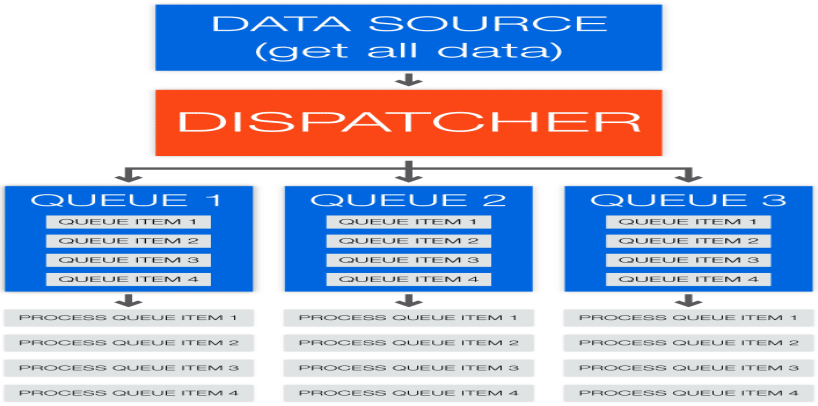
Dispatcher:

The dispatcher is a process used to push transaction items to an Orchestrator queue. It extracts data from one or multiple sources and uses it to create Queue items to be processed by Performer robots.

Information is pushed to one or more queues allowing the dispatcher to use a common format for all data stored in queue items.

The major advantage of using a dispatcher pattern is that you can split the processing of the items between multiple robots.

Dispatcher Architecture:



Performer:

The performer is a process used to pull transaction items from an orchestrator queue and process them as needed in the company. Queue items are processed one at a time.

It uses error handling and retry mechanisms for each processed item.

A major advantage of the performer is its scalability (multiple performers can be used with a single queue)

Performer Architecture:

Let`s use the same scenario as previously where we have a list of people and their email addresses in a spreadsheet. An email needs to be sent to each of them with a personalized message based on a template.

We can use the Dispatcher & Performer pattern in the following manner:

The **Dispatcher reads rows from the input spreadsheet** and **adds the data** (i.e., name and email) **to a queue**; each queue item will have both name and email from one spreadsheet row.

The **Performer** retrieves **an item from the same queue** and **does the necessary actions** using that data, like replacing template values and sending an email.

Dispatcher & Performer model advantages:

1. Better separation of processes (between dispatcher and performer)
2. Better separation & distinction between architecture and process layers
3. Better error handling and retry mechanism
4. Possibility to run processes across several machines (availability)
5. Better re-usability within your project's created components
6. Improved built-in configuration & Orchestrator integration

7. Previous workflows created without REFramework can be easily adapted and deployed in order to use REFramework and the dispatcher/performer model