Source code repository path: <https://github.com/mihhdu/UiPath_REFramework>

Goals of the Enhanced ReFramework:

* Provide a clear structure to the framework, both in project folder structure as well as code wise.
* Separate it in **layers,**in a way that the framework layer provides an easily configurable engine for running diverse processes with local or Queue data.
* Framework layer should provide a common interface to call business logic code and seamlessly provide top-level application environment recovery and transaction retry logic.
* Make the business logic components easily accessible in the code.
* Keep the same business logic components ("InitAllApplications.xaml", "GetSetTransactionData.xaml", "ProcessTransaction.xaml", "CloseAllApplications.xaml", "KillallProcesses.xaml"), making migration from previous version seamless.

Changelog:

* Achieve modularity and reusability by defining a new structure called **Workblock**designed to log runtime details of code running inside. Reuse structure for framework states and business code. (Note 1)
* Strive to Achieve Separation of Concerns by splitting the framework into layers. Thus, the business should only be concerned with the Business Logic code. (Note 2)
* State machine transitions is where the framework managed transaction iterators, the TransactionNumber and RetryNumber are incremented and system flags set. They are now kept in a dictionary called SystemReserved.
* SetTransactionStatus.xaml has now disappeared. The actual set of the transaction status is done in the Business Process layer.
* Process.xaml is moved/renamed to ProcessLayer\ProcessTransaction.xaml.
* Framework\GetTransactionData.xaml is moved/renamed ProcessLayer\GetSetTransactionData.xaml.
* Framework\InitAllApplications.xaml is moved/renamed ProcessLayer\InitAllApplications.xaml.
* Framework\CloseAllApplications.xaml is moved/renamed ProcessLayer\CloseAllApplications.xaml.
* Framework\KillAllProcesses.xaml is moved/renamed ProcessLayer\KillAllProcesses.xaml.
* ProcessLayer\CloseAllApplications.xaml called in case of recovery from System Error is now called in the InitLayer.
* Init state now retries a number of times upon exception, configurable from the Data\Config.xlsx file.
* ProcessLayer\KillAllProcesses.xaml is now called in the try catch of the ProcessLayer\CloseAllApplications.xaml file, allowing better dev control without touching the framework layer.
* When TransactionItem is a QueueItem, use GetQueueMaxRetries.xaml to query Orchestrator server and obtain maximum retry information. This function uses 2018.1 REST API authentication through the robot, and is not compatible to previous versions. If absolutely necessary, this can be made to work with older versions by using the generic HTTP Request that would require authentication.
* State machine now remembers consecutive application exceptions that occurred in the Process Transaction State and will abort process upon reaching configurable (Data\Config.xlsx) maximum. (Ex: Use this to prevent a robot consuming items)
* Implement services; each service is a state machine engine in its own right (same structure as the Enhanced ReFramework main engine). It consists of a Main.xaml file as well as its own ProcessLayer and Data folders. The ServicesLayer folder now stores the default services/service templates. (Note 4)
* WbLogging.xaml is a new workflow that does Workblock logging. Each workblock has this called twice, when successful and when failed.
* ListOfDictToDt.xaml is a new workflow that is used to convert the Audit object (the status of workblocks) from dictionary to dt and save it to disk in csv format. Component of test framework.
* Reorganize Data\Config.xlsx and provide some manner of in-file guidance. (Note 3)
* Added a Workblock Snippet folder containing a Workblock Template. Add this to the library folder and drag to create a workblock in a new Enhanced ReFramework project.

Notes:

1. Framework is now composed of **Workblocks**. Each workblock represents the minimal run and log structure. It is made up of a try-catch block. Within the Try section, the code we want to execute. After executing we log success, gathering runtime and hierarchical information from **child**Workblocks and passing it on to the parent Workblock. If an exception occurs within the code, the work block will catch the exception, log the failure, output the exception and act as instructed by the "wbHandleError" flag.

As such, the framework layer has 4 Workblocks. A master Workblock, let's call it "MainTask". This workblock has three child Workblocks: "Init", "GetSetData" and "Process".

The following log fields need to be logged, at a minimum level, in each workblock:

The following log fields are added:

Add Hierarchy Log Fields:

+ wbType - Name of Current Workblock

+ wbParentType - Name of Parent Workblock

+ wbLevel - Level of current workblock (typically level of Parent Workblock + 1)

Add Execution Log Fields:

+ wbStart - the timestamp of the start of execution

+ wbDurationSec - duration of execution in seconds

+ wbDurationHrs - duration of execution in hours

+ wbStatus - If the current workblock is successful and all child workblocks are also successful then set "Successful", If the current workblock is successful and some child workblocks have failed then set "Finished with Exceptions", if current workblock failed then "Failed", else "SoftFailed"

+ wbFinalExec - If the current workblock is successful or is at the last attempt and all parents are at the last attempt, wbFinalExec is true, else it is false

+ wbParentStart - The start time of the parent wb

+ wbHandleError - If the wb is logging in its Catch section, it can be instructed by its parent, through the wbHandleError flag, to "Rethrow" the exception, to "Failed" or to "SoftFail". Both "Failed" and "SoftFail" instructions mean do not rethrow. The difference between the two is "SoftFail" does not cause the parent to have a status of "Finished with Exceptions". Thus, the parent can try to solve the child's problem in another manner that could be successful.

+ wbPath - Workblock execution path to the current workblock, reflecting all ancestors and self

+ wbKey - md5Hash of wbPath+wbStart - pseudo unique key

The following rules are applied to Workblocks:

* wbType should be unique
* There should be exactly one log message containing the status of the Workblock execution, at the end of the Workblock.
* We should be wise in not generating a large quantity of log data by not logging successful work block executions for every single Workblock.

2. Separate the different components of the framework into layers:

1. Framework layer (the main dataflow representation, a state machine, calling different components of code). In this section the pieces of code that are needed for framework operation are encapsulated in a "System Reserved" sequence. This layer provides Transaction iterators and recovery mechanisms for Application exceptions by calling the user defined structure for reinitializing the environment.
2. Data layer (ProcessLayer\GetSetTransactionData.xaml), where working with transactional data is to take place. Clear separation between working with data and working with applications.
3. Business Process layer (the workflows that need to be modified to achieve the desired process behavior - InitAllApplications.xaml, CloseAllApplications.xaml, KillAllProcesses.xaml, ProcessTransaction.xaml)

3. Organize Config.xlsx better by creating the following extra sheets:

a) introduction sheet (write some documentation regarding the Config)

b) credentials sheet (place credentials here)

c) Workblock sheet (name Workblocks running in the framework layer here)

d) tasks sheet (details about services layer on)

4. Implement a pluggable services/tasks layer. Services are found within the ServiceLayer folder. Each task has its own ProcessLayer folder, with all the expected components, and its own Data\Config.xlsx file.

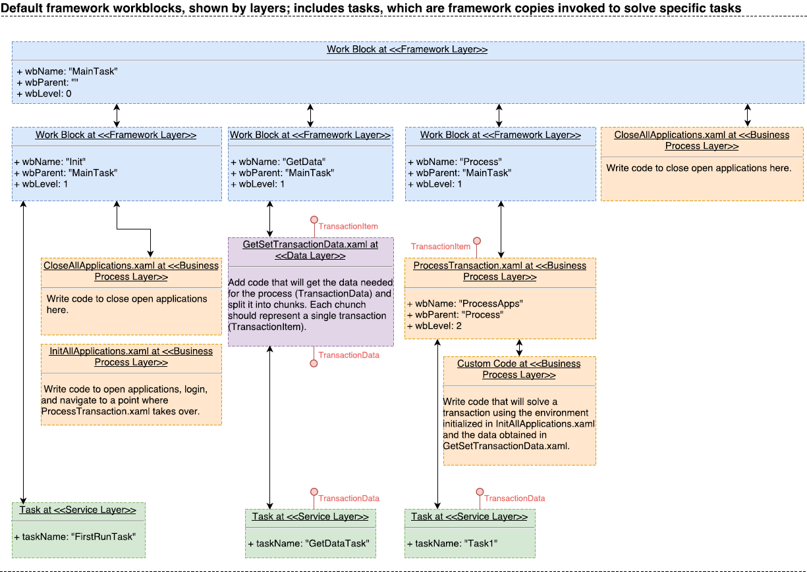
A task is a standard framework that will act like a slave running within the main workflow. The framework now implements two system services, "FirstRunTask" and "GetDataTask". Both are off by default. If enabled:

a) FirstRunTask will run only once, at the start of the process. Possible use case: queue dispatcher

b) GetDataTask will either run once, on the first process run, or on every process run. Possible use case: The data needed for processing is obtained from a resource that can fail (for example, a website). You simply then configure GetDataTask to independently get your data and output it in the main task.

We include two more services examples, "Task1" and "Task2", that could be called in the ProcessLayer\ProcessTransaction.xaml file. It's useful when, within a process, we need to work with multiple independent applications. Instead of implementing everything within the process section, we simply launch a service that solves said task.

It should be noted that, even though theoretically one could have service within service implementations, we highly discourage it and consider a horizontal development tends to yield better results.



**Framework WorkBlocks UML**