KBLFramework for UiPath

By Kasper Bigum Larsen

# Disclaimer

* Work in progress!
* I’m aware that some of the names are quite similar to each other. Working on finding better names. Suggestions are welcome!

# Description/Purpose

* This framework is different from, but not necessarily better than UiPaths own Enterprise Framework (REF).   
  It was started before Uipath released REF.
* The core idea is the same as REF: Create some code up front to support general error handling that every process needs, along with the idea of when all developers are using the same framework, it’s easier to read and support each other's code.
* The main purpose of releasing this framework to the public is to get feedback for my own use, but also to give it to the public as I feel it is easier to understand and use than REF, especially for new developers.
* The framework is a Template, and templates are templates. They should remain as close to the original as possible, but if they need to be changed to fit the process, then change them!

If all processes need the same change, then the lead developer should change the template.

# Flowchart overview

For reference...



# Key features

Description of the key features in KBLFramework

## Load config / Config.xlsx

Out\_Config <dictionary<String,Object>>

This framework uses the same config file as REF, with any changes described below.

The config file (including orchestrator assets) is loaded into a dictionary<string,object>, at the start of Main, and can be used with “var value = in\_Config(key)” in any file you pass the config dictionary to.

Se UiPath documentation for more information.

## Status (Success/Application/Business)

Variable Status <String> Scope: MetaProcess, Default: S10000  
Variable StatusMessage <String> Scope: MetaProcess  
Variable StatusScreenshot <Image> Scope: MetaProcess

‘Status’ a concept of having 1 variable in change of controlling the flow of the process, and is defined in “MetaProcess.xaml”.

StatusMessage is a text version of Status pulled from the config.

StatusScreenshot is a screenshot taken by ExceptionToStatus when unknown exceptions happen.

If a process runs its course without business or application errors, it is considered a success. It has done the checks it needed, filled out the forms correctly or whatever else the process needs to handle.

A Success is normally ‘invisible’ to any end user, and no special followup is required (though a ‘Success notification’ or ‘Hand off’ could still be sent to end users).

Most processes would only have the default success status, but there is nothing wrong with using multiple success statuses, as long as using Business exceptions instead have been considered.

Status can be sorted into the 3 categories:

(S)uccess: S10xxx

(A)pplication (exception): A11xxx

(B)usiness (exception): B12xxx

Every status needs to be defined in the config file. This is done to keep a clean overview of what states/status the process can be in, and so Business can make changes to the process without major code change.

When you want to change the status of the process at runtime, you should do it using “ExceptionToStatus.xaml” (see below).

You could also set the status manually, but remember to put it into the config file to keep the overview, and to populate StatusMessage.

## Exceptions

Exceptions are sorted into 3 categories:

1. **Known (Application) exception. (A11xxx)**  
   Any expected application exceptions that you need to handle specially.

For example if your business application is closed for maintenance every month, and so you check if it's down and throw “new exception(“A11001”)” if it is.

1. **Business exception. (B12xxx)**  
   If any predefined deviations happen that redefines the ‘output’ for the process, it’s called a business exception and should be thrown with “new Exception(“B12001”)” (or UiPath.core.BusinessRuleException if you prefer, but the framework don’t care about the type).
2. **Unknown Exception.**  
   Any exception that is not pre defined, is an unknown exception and will be logged as an Application Exception. This would most of the time come directly from a selectorNotFoundException or similar. Depending on the expected frequency of the exception, Business can decide to make it a Business exception (and define a reaction), Developer can decide to make it a Known exception or leave it as an unknown exception.

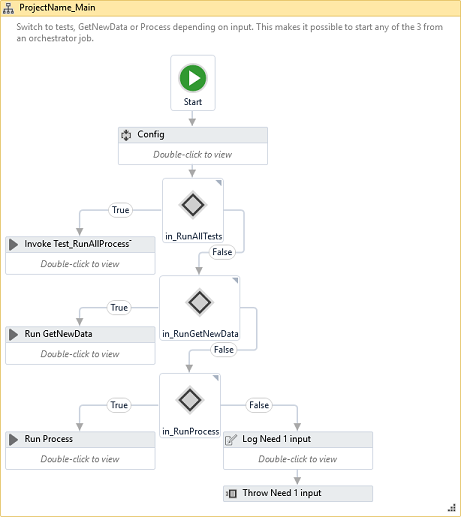
*Note: Extra messages can be given using “New exception(<status>****|****<extraMessage>), but remember the golden rule: Every state should have its own status ALMOST NO MATTER how similar they are, to make troubleshooting easier.*

By default, the framework catches exceptions from PreProcess and Process, and converts them to a status (A11001 and A11002) using ExceptionToStatus.xaml, before continuing to ‘PostProcess’, but if you need to change the status and do more process before continuing to PostProcess, you can build in ExceptionToStatus in try catches as you see fit.

# Structure details

## Main

Args: in\_RunAllTests (Bool), in\_RunProcess (Bool), in\_OverwriteEnvironment (String), in\_RunGetNewData (Bool).



The structure of Main is relatively simple.

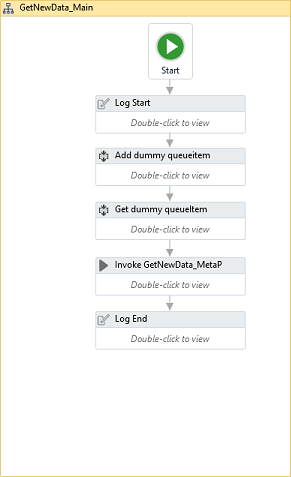
It loads the config file into a dictionary, and depending on which one of the inputs given, it either starts GetNewData to create queueItems (AKA Dispatcher) or starts the Process to process the queueItems (AKA Performer).

It is also possible to start all ProcessTests, that is, all test files in the “Test” directory.

*Note: There is no check to see if you give it multiple input as it is expected that the person creating the job has read this guide. But you can make an extra check if you want to :)*

## GetNewData\_Main (Dispatcher)

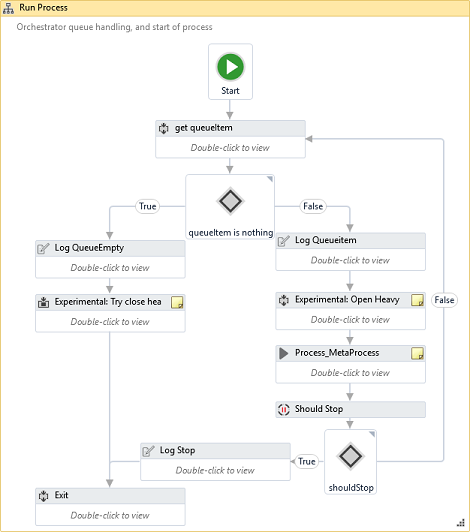
Args: In\_Config (Dictionary<String,Object>).



GetNewData\_Main is called from Main. (Yes. I’m sorry about the naming confusion)

The only purpose of GetNewData\_Main, is to generate a dummy queueItem in orchestrator, to be able to see the status of the run from orchestrator, and then run GetNewData\_MetaProcess to create the queueItems.

## Process\_Main (Performer)



Process\_Main is in charge of queue handling. It takes the first item from the queue, and passes it onto Process\_MetaProcess, which is where the queueItem gets processed.

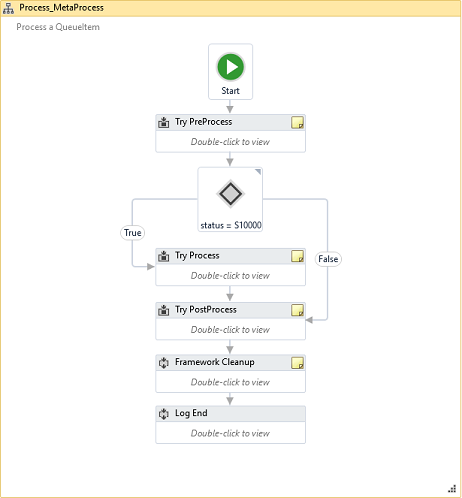
It is recommended to open and close applications inside Process\_MetaProcess, to make sure every queueitem is run from a clean environment. This makes the framework more resilient to unexpected application errors.

It also have a built in ShouldStop, to stop processing queue items if given the command from orchestrator

*Experimental: There is room for opening and closing ‘heavy’ applications, but this is still under development.*

## MetaProcess (both GetNewData and Process)

Args: In\_Config (Dictionary<String,Object>), in\_Queueitem (QueueItem).  
Previously called MetaMagic.

**

*Note: GetNewData\_MetaProcess is not to be confused with Process\_MetaProcess.*

*They are both called MetaProcess as they build from the same MetaProcess template, however* ***Process****\_MetaProcess is the process of* ***running QueueItems****, where* ***GetNewData****\_MetaProcess is the process of* ***creating QueueItems****.*

The structure of both GetNewData\_MetaProcess and Process\_MetaProcess, consists of 3 parts wrapped in their own TryCatch:

* **PreProcess**
* **Process**
* **PostProcess**

### PreProcess

**..**is in charge of doing stuff that needs to happen before the process can start. The only thing here to begin with, is assigning runtime variables.

Other uses for it could be input validation, checking system availability or whatever else you can check up front.

### Process

..is where the magic happens (previously called Magic). The core of the process goes here.

This includes opening applications and cleanup. If the process uses heavy applications, or (through analysing the process before development) you know that the queue always has many items waiting, consider opening and closing the application in GetNewData\_Main. The reason for letting the process be in charge of the applications, is to make it more resilient to unknown application errors and start from scratch each queueitem.

**GetNewData\_**Process also comes populated with some code, to help with the creation of the individual queueItems with predefined runtime variables. UniqID and NetworkSave are both explained separately in “[Other features](#_vu2u9ja2a6ga)”.

Use “Get data”-sequence to access what you need to get the basis for the data needed for the queueItems. Put all this data into “dataToQueue” (list/array/datatable), and let the loop handle dropping of unwanted elements based on criteria, along with assigning the correct values to the queueItem.

### PostProcess

..is where reactions to the status of the process can be made.

This would normally be sending an email to someone, with the information that the process has run into a problem and something needs to be done manually.

In theory could be anything, but you want this to be as simple as possible as PostProcess does not react to itself, meaning that if an error happens here it does not trigger PostProcess again.

If you need to do a reaction that is complex and may fail, consider doing it in “Process” instead

# Orchestrator setup

## Queues

Every project needs a total of 2x3 queues: <Projectname>\_DEV/TEST/PROD and <Projectname>\_GetNewData\_DEV/TEST/PROD.

The first is for the actual process queueItems, and the second is for the GetNewData dummy queueItems (only used for statistics).

## Assets

### Robot\_CurrentEnv (Required)

This “value pr robot”-asset should be set for each robot to either “DEV”,”TEST” or “PROD”. Using this asset, the developer can branch the workflow to do different activities depending on what environment the robot is in.  
Imagine you have a 3rd party system that you have no control over, and no test environment for that you want to test. It fills out a form in the system, and if it is running on the PROD robot it clicks ‘Submit’, but if it’s running on the TEST robot it takes a screenshot and clicks “Cancel” instead.

This means that no changes will need to be made to the code when moving the process between environments.

### Robot\_ID (Required)

The (simple) ID of the robot. A number to identify the robot. Used in UniqID

### RPA\_ErrorFolder (Required)

A (Network) location where error logs and screenshots are saved. It should be a network location where only a few selected people have access with automated cleanup, as (depending on the process) the screenshots can contain secret information.

By default the Framework only uses this when a fatal error (uncaught error in MetaProcess) or Default Application exception (Unknown exception) happens. It is recommended to use with all unknown errors for easier troubleshooting.

*The framework will generate unique directories inside this directory.*

### RPA\_LogFolder (Required)

A (Network) location for custom generated log files (See Log\_Dictionary). Per default, the log files contain no secret information so no access control is needed, but as the process starts putting data into the log file access control should be restricted to Process Manager (Business).

*The framework will generate unique directories inside this directory.*

### RPA\_NetworkSave (Required)

A (Network) location for robots to share information that can’t be saved in orchestrator.

For example, files that get pulled from an email by GetNewData, that Process then manipulates before sending back.

*The framework will generate unique folders inside this directory.*

### RPA\_FATALERROR\_EmailList (Required)

In case of a fatal error, this email list gets notified (Split emails by ‘;’).

### RPA\_Support (Optional)

This email list is used to send emails to the RPA-support team. This could be the developers, the developers manager, or someone from IT-support, depending on how your company runs RPA support.

It is not used by the framework per default, but it’s good use it from the beginning (with developers email), so it’s easy to change when you get a RPA support team.

### RPA\_TeamEmail (Optional)

Email for the RPA team. Used when you need a process to alert the RPA team directly instead of the RPA support.

As described under RPA\_Support, you should distinguish between which cases go to support and which go to the RPA team even if the two are the same persons, as it makes it easier to change later.

### Robot\_EmailAddress (Optional)

The robots email account. Used in SendEmail (Outlook activities).

# Other features

## UniqID<String>

UniqID (or Unique ID) is created by GetNewData, and is unique for each queueitem.

It consists of 3 elements:

1) A developer defined identifier, that should be information from the queueItem. In theory it can be anything but should be as unique as possible to make troubleshooting easier.

2) The ID of the robot (orchestrator Asset pr robot, Robot\_ID)

3) Timestamp (Now.tostring(“yyyyMMdd\_HHmmssff”))

Element 2 and 3 makes the variable unique, and 1 is for troubleshooting.

Every time the process creates an output of any time, the output should be stamped with the UniqID. This has 3 benefits:

1) We know that the output is unique and will never generate an overwrite error.

2) From any given output, you can see when the QueueItem was created and what robot created it.

3) When processes deliver outputs to different places, it’s easier to trace items from multiple places back to the same queueItem.

## ExceptionToStatus.xaml

This workflow is used to analyse exceptions, and convert them into statuses.

If the exception.message before any ‘|’ is defined in the config file it will convert that message into a status, pull the status message from the config file into StatusMessage, and add any message after | to the end of StatusMessage.

If the exception.message before any ‘|’ is NOT defined in the config (meaning it’s a system thrown exception), it will instead set the status to the ‘StatusIfUnknown’-argument and add the exception.message to the end.

If the status was not defined **or** the status is an A-status, a screenshot will be taken and put into StatusScreenshot. This screenshot is not automatically saved, but should be saved in PostProcess (by default an A-status will save the screenshot, using GenerateErrorFolder.xaml)

## NetworkSaveFolder\_path<String>

The NetworkSaveFolder\_path<String> is a path to a directory on a network location. It is used when robots need to have access to the same data, or the process needs to transfer data from GetNewData to Process that cannot be contained in queueItems.

The path uses the orchestrator asset RPA\_NetworkSave, and is always set in GetNewData, but the directory is not generated by default. The developer can put in a ‘create directory’-activity to generate and use it.

## WorkingFolder

A local folder that the robot can use to save temporary files. It uses UniqID (and therefore is always unique) and is generated in “Temp\”.

## CurrentEnv

See Orchestrator asset Robot\_CurrentEnv.

## Log\_Dictionary

Log\_Dictionary<String,Object> is saved to a .txt file at the end of the process. It is automatically populated with some runtime information (Time, status) and can be used to log important information during a run. The purpose is to have a log file that only contains important business information, so business can troubleshoot the logic without looking at code or UiPath log files.

It’s recommended to log all input and output from a process, but remember to consider GDPR.

## Testing (experimental)

*Note: This part of the framework is still under development, as i feel the time investment for creating tests is not paying off. Feedback is welcome :)*

The framework contains a Test folder with a test template. The developer can use this template (or make a better one!) to generate automated test cases that can test core features of the process. After any developer makes changes to the process, the tests can be run to check that the core features have not been affected by the change.

## Framework Cleanup

After PostProcess, the framework cleans up. By default this means checking that all processes that have been opened throughout the process have been closed again.

It is important that the developer remembers to add the process name of any opened application to the “OpenedApplication”<List<String>>. Cleanup will then check if processes with any name in the list are running, and kill them if they are.

*Note: Applications should still be closed properly during the process. This is only meant as a failsafe*

# Drawingboard

Planned features and features being considered

KBLFramework is developing as I need the features in my daily work, so the future is unknown, but here is some features currently being considered:

* **Project generator:** Projects need some setup, mainly Queues and code copy. This feature is an .xaml file to do those setup steps more automatically.
* **Standalone workflows/Live repository/other better name:** Workflows that can be called by any project directly (instead of containing them inside the project). This is another “bad practice”, but the benefit is that in an application, that is used by multiple projects, is changed you can change all of them at the same time very fast.

This could be “Login to application x” or “save documentation to application y”.

This is something i have used with success in an earlier version of the framework, but UiPath have since then released a “mass dependency update” option, that might be a better way to do it.

* **Change the type of Config dictionary** from <String,Object> to <String,String>. I see no reason why it can contain objects, when you have to convert the data when you pull it out anyways.

# Final notes

Along with some “deep” thoughts.

**Any feedback is highly appreciated!**

If you have any questions, have a reason why you think this framework is useless, or have any other positive or negative feedback, feel free to drop me an email at [KasperPublicLarsen@gmail.com](mailto:KasperPublicLarsen@gmail.com)

Some of the concepts in the framework are not following normal coding standards. This is because I think RPA is not to be considered “normal code”.

It’s important to remember the purpose and strengths of RPA:

To create an automated process, where “normal” code integrations either cannot be developed or takes too long to develop.

Using 2 week on creating a simple RPA project that correctly processes 80% of the given input and sends the rest to manual processing, is better than using 4 weeks on creating a project that correctly processes 90%!

Think less of what is “Correct” and more of the value you create!

In most cases, speed is not the problem, and so optimization of code should have focus on readability and customizability over speed.

UiPaths strongest feature is its easy to read interface, so remember to use this strength as much as possible, and be less concerned about doing it the smartest way.

It is better to use multiple activities (packed into a sequence) over using clever VB.net one-liners. Sure, the VB is faster and smarter, but if it’s less readable for your fellow developers, it’s a worse solution. If you do end up using long one-liners, make sure to split them into multiple steps to make them easier to read.