

Contents

[2 Revision History 4](#_Toc11834663)

[3 Coding Standards 5](#_Toc11834664)

[3.1 Naming convention and strategy 5](#_Toc11834665)

[3.1.1 Variables 5](#_Toc11834666)

[3.1.2 Arguments 6](#_Toc11834667)

[3.1.3 Activities 6](#_Toc11834668)

[3.1.4 Workflow files 6](#_Toc11834669)

[3.1.5 Projects and Sub-projects 6](#_Toc11834670)

[3.1.6 Orchestrator 7](#_Toc11834671)

[3.2 Configurations 8](#_Toc11834672)

[3.3 Credentials 9](#_Toc11834673)

[3.3.1 Robot Credentials 9](#_Toc11834674)

[3.3.2 Application Credentials 9](#_Toc11834675)

[3.3.3 Secure String 9](#_Toc11834676)

[3.4 Error Handling 11](#_Toc11834677)

[3.4.1 Try Catch 11](#_Toc11834678)

[3.4.2 Throw 12](#_Toc11834679)

[3.4.3 Rethrow 12](#_Toc11834680)

[3.4.4 Terminate Workflow 13](#_Toc11834681)

[3.4.5 Business Rule Exception 13](#_Toc11834682)

[3.4.6 Retry Scope 13](#_Toc11834683)

[4 Workflow dseign 14](#_Toc11834684)

[4.1 Design Principles 14](#_Toc11834685)

[4.2 Layout Diagrams 14](#_Toc11834686)

[4.2.1 Sequence 14](#_Toc11834687)

[4.2.2 Flowchart 14](#_Toc11834688)

[4.2.3 State Machine 14](#_Toc11834689)

[4.3 Decision 14](#_Toc11834690)

[4.3.1 If Activity 14](#_Toc11834691)

[4.3.2 Flow Decision 15](#_Toc11834692)

[4.3.3 If Operator 16](#_Toc11834693)

[4.3.4 Switch Activity 16](#_Toc11834694)

[4.3.5 Flow Switch 17](#_Toc11834695)

[4.4 Data 18](#_Toc11834696)

[4.4.1 Variable Scope 18](#_Toc11834697)

[4.4.2 Arguments 18](#_Toc11834698)

[4.4.3 Default values 18](#_Toc11834699)

[4.5 Annotations and Comments 18](#_Toc11834700)

[4.6 Reusability 19](#_Toc11834701)

[4.6.1 Intra-Project Reusability 19](#_Toc11834702)

[4.6.2 Reusable Components 20](#_Toc11834703)

[4.7 Keep it clean 21](#_Toc11834704)

[5 Unit/ Alpha Testing 22](#_Toc11834705)

[6 Code Review 23](#_Toc11834706)

[6.1 Review Guidelines 23](#_Toc11834707)

[7 Source Control 24](#_Toc11834708)

[8 Frameworks 25](#_Toc11834709)

[8.1 REFrameWork 25](#_Toc11834710)

[8.2 Enhanced REFramework 26](#_Toc11834711)

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Author | Description |
| 31 May 2019 | 1.0 | Kanad Mehta | Created Document |
| 17 June 2019 | 1.1 | Kanad Mehta | Added Decision and Data scope |
| 19 June 2019 | 1.2 | Kanad Mehta | Added Reusability, Unit testing and Code review guidelines |

# Coding Standards

## Naming convention and strategy

Meaningful names should be assigned to workflow files, activities, arguments and variables in  
order to accurately describe their usage throughout the project.

Firstly, projects should have meaningful descriptions, as they are also displayed in the  
Orchestrator user interface. Moreover, adopting a good naming strategy for environments, assets  
and queues makes the management of the virtual workforce in Orchestrator more manageable.

Only argument names are case sensitive, but to improve readability, variables and the other  
entities should also align to the same naming convention.

* Variables should be upper Camel Case, e.g. *FirstName*, *LastName*
* Arguments should be in upper Camel Case with a prefix stating the argument type, e.g. in\_*DefaultTimeout*, in\_*FileName, out\_TextResult, io\_RetryNumber*
* Activity names should concisely reflect the action taken, e.g. *Click ‘Save’ Button.* Keep the part of the title that describe the action (Click, Type Into, Element exists etc)

Except for Main, all workflow names should contain the verb describing what the workflow does, e.g. *GetTransactionData*, *ProcessTransation*, *TakeScreenshot*

### Variables

* Use one variable for one and only one purpose.
* Minimize the scope of each variable.
* Keep statements that work with the same variable(s) as close together as possible.
* Variables will always have **meaningful names**. The variable name should fully and accurately describe the entity the variable represents. State in words what the variable represents.
* We will use **upper Camel Case** (Pascal case) for naming variables. This practice used compound words, no other characters between the words, where each word will start with a capital letter. Ex: TransactionNumber, FilePath, ReportName etc.
* The length of the variable name should be between 6 and 20 characters long. If you feel that 20 characters are not enough, consider abbreviating longer words. Shorter variables names can be used when using a local scope (like: index, file, row)
* Datatable object: Start with dt \_ prefix followed by the normal name. Ex: dt\_Employees, dt\_Reports
* Boolean type: Give Boolean variables names that imply True or False. You can use the prefix is followed by the name. Ex. ApplicationExists, isRed, isFound etc. Always use positive names, negatives names (Ex: notFound) should be avoided if possible.

### Arguments

Same guidelines as for variables, with below differences:

* Each argument will have a prefix depending on the direction: in, out, io followed by the  
  underscore character ("\_"). Example: in\_Config, out\_InvoiceNumber, io\_RetryNumber,  
  in\_dt\_Employees
* Use default values for arguments either for testing individual workflow files, or, in case  
  of reusable components, for using default configuration. When invoking a workflow file,  
  UiPath allows the flexibility to pass a value to any of the defined arguments, having  
  essentially all possible signatures defined by the argument list. This allows for using the  
  default value of an argument for which no value is passed upon invoking the file. Specify  
  what is the default configuration in the description of the reusable workflow file.

### Activities

Activity names should concisely reflect the action taken, e.g. Click ‘Save’ Button. Keep the part  
of the title that describes the action (Click, Type Into, Element exists etc). In case an activity  
throws an exception the source of the exception will contain the activity name, so a proper name  
to each activity is advisable for an easy understanding of the exception. Take extra care in  
renaming activities that have a standard name, like Assign, If, For each or Sequence for which  
the name doesn't automatically change - one would need to manually rename them.

### Workflow files

* Upper Camel Case naming
* A workflow file starts with the prefix containing Application Name. E.g. for working in  
  SAP: **SAP\_**Login.xaml, or **SAP\_**ExtractClientReport. Typically, workflow files belonging to the same application or system will be grouped together in one folder under the project root folder. In case there are many files for one application, further categorizing by using subfolders can be used.
* When using a template framework - the framework files come already created and are  
  standard (including Main.xaml) - they should not be changed
* When using a test framework - for the Test\_Framework files – Use the prefix **Test\_** for a  
  workflow file that runs tests. Place these files in the Test\_Framework folder
* For an easier understanding, use number prefixes to emphasize the calling (invoking) hierarchy of the Project, where the root is always Process (in REFrameWork) if the hierarchy grows too large.

### Projects and Sub-projects

* Upper Camel Case naming
* Group by department using a prefix: E.g. AP\_, AR\_
* In case the process is automated using sub-processes (using multiple packages for the same business process, like using Dispatcher and Performer), use the business process code as the next prefix

### Orchestrator

A good and consistent naming strategy must be used when defining the Orchestrator entities.

#### Robots

* Development machines: DEV\_[Name of developer in upper Camel Case] E.g. DEV\_u6ic\_kme
* Test machines: TEST\_[Machine Name][Robot Number]
* Prod machines: [Machine Name][Robot Number]

#### Environments

An environment links together multiple robots that are running the same process. Hence, the naming will include a combination from robots and projects:

* Use prefix DEV\_ or TEST\_ or PROD\_
* Group by department using a prefix: E.g. AP\_, AR\_
* In case of sub-processes, use the business process code as the next prefix

#### Assets

* for normal assets: [Department]\_[Project code]\_[Asset Name] E.g. AR\_CA\_MappingTableURL
* for credentials: C\_[expiration period]\_[Department]\_[Project code]\_[Asset Name] E.g. C\_180\_AP\_SC\_SapCredentials

#### Queues

* [Department]\_[Project code]\_[Queue Name] E.g. AR\_CA\_ExcelItems

## **Configurations**

When automating processes, we will inevitably need to use configurations. We can categorize the configurations into the following groups:

* Configurations for which the values **never** change. Examples here would include a static selector, or a label in an application. These ones should be hardcoded in the workflows. There is not even a long term benefit from going through the trouble of storing them in a file.
* Configurations that are highly **unlikely** to change but are used into more than one place or settings that are important and are not meant to be changed by someone outside of the development team. To allow extensibility and also increase readability, we recommend storing these settings in a config file. Examples: Log messages, log fields, file or folder paths and patterns. This way, if during development there is a need to change one of these settings, they will be changed only in the config file. This technique also improves readability as the key in the dictionary will have a meaning attached to the actual value (E.g. using the "ReportID" key in the dictionary instead of the actual value: "12361223")
* Configurations that is **likely** to change from one environment to another. Into this category we have application paths, URLs, queue names, credential names etc. For these settings we recommend using Orchestrator assets. The main advantage in this case is that the values can be changed without modifying the code, so it allows the code developed only in the Dev environment to migrate without changes into Test and then Production.
* Runtime settings - This are required to be set during runtime. For Unattended robots we should use Orchestrator assets, queues or external callouts, while for Attended robots, this is achieved through input dialogs that request the necessary information.
* Configurations that have **different** values for different robots - Use Orchestrator assets with per robot value.

The final solution should be extensible, to allow variations and changes in the input data without an intervention from a developer, when required.

## **Credentials**

### **Robot Credentials**

Credentials are required by the Orchestrator to start an interactive Windows session on an unattended robot. They are defined in the Orchestrator Robot definitions. The password in stored encrypted with the 256 bit AES encryption algorithm and once set, the password cannot be displayed. There's also the possibility of storing the passwords in CyberArk which is integrated with Orchestrator.

### **Application Credentials**

Application credentials should not be stored in the workflows or Config files in plain text, but rather they should be loaded from safer places like local Orchestrator or Windows Credential Store.

There are three ways of dealing with credentials natively in UiPath. They are displayed in the order of recommendation:

* Orchestrator Credential assets: They are stored securely in the SQL Server DB, with 256 bit AES. Once set, the password can't be displayed. They are retrieved using the Get Credential activity under Orchestrator which returns a String Username and a SecureString Password. It also supports per robot values, like normal assets. Due to the increased security in the Orchestrator and global control, this is the recommended option.
* In case using Orchestrator Credential assets is not possible, the second best option is to use Windows Credential Store. Apart from getting the credentials, there's the possibility to Add and Delete a credential from the store. There's also a Request Credential activity for an Attended robot that creates a dialog at runtime designed to accept credentials. Using the Windows Credential Store will imply the credentials are stored locally on the robots and which means that in the case of deployment of a process on multiple robots, one needs to create the same credential on all robots.
* Using the Get Password activity - last resort option that stores the password encrypted in the xaml file. The encryption is linked to the machine, so, for a successful decryption in deployment it requires re-typing of the password and saving the xaml file. The code cannot migrate without changes in this case.

The **scope** of the credential related variables, i.e. username and password should be limited to  
where they are needed. Never use a larger scope for these variables.

### Secure String

The password output from the GetCredentials activities is returned as a SecureString datatype.

This is a special class in the .NET Framework that represents text that should be kept confidential. The password is not kept in plain text in memory, but rather obfuscated (not really encrypted) which makes it difficult to find the password if someone or something is just accessing the memory. Also, once the variable scope ends, the memory is immediately released, unlike normal Strings. Once a SecureString is retrieved, it should be used to log into the applications by using the Type Secure Text activity for normal applications or the Send Keys Secure activity for Terminals.

For other activities that require authentication, like email activities or HTTP and SOAP Request activities the password input type is String. In this case there's the following method to convert the SecureString to a String:

String UnsecurePassword

SecureString SecurePassword

Assign:

UnsecurePassword = new System.Net.NetworkCredential("abc", SecurePassword).Password

The scope for the new UnsecurePassword, together with the SecureString password and String username should be limited to where it's needed. The credential should not be used for any purpose other than the intended one.

## Error Handling

UiPath employs an exception handling mechanism very similar to what modern programming languages permit. It is mainly focused on the **Try Catch** activity and, together with the **Throw** activity, it enables an elegant error handling mechanism.

Two types of exceptions may happen when running an automated process: somewhat predictable or totally unexpected. Based on this distinction there are two ways of addressing exceptions, either by explicit actions executed automatically within the workflow, or by escalating the issue to a higher level.

Exception propagation can be controlled by placing susceptible code inside **Try Catch** blocks where situations can be appropriately handled. At the highest level, the main process diagram must define broad corrective measures to address all generic exceptions and to ensure system integrity. The REFrameWork has this exception handling mechanism in place and will recover from any unexpected error.

Contextual handlers offer more flexibility for Robots to adapt to various situations and they should be used for implementing alternative techniques, cleanup or customization of user/log messages. If a block catches an exception it cannot handle, it is recommended to log the exception and then rethrow the exception to the higher invoking level. Take advantage of the vertical propagation mechanism of exceptions to **avoid duplicate handlers** in catch sections by moving the handler up some levels where it may **cover all exceptions in a single place**. In the REFrameWork this is the place is the Main.xaml workflow file.

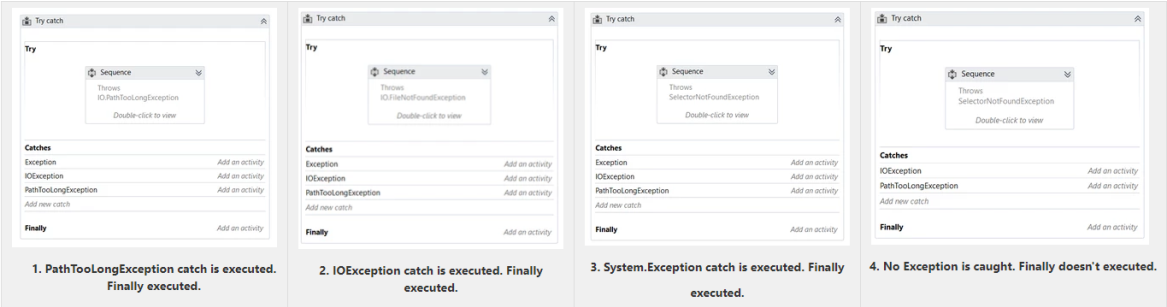
Enough details should be provided in the exception message for a human to understand it and take the necessary actions. The exception **message** and **source** are essential. The source property of an Exception object will indicate the name of the activity that failed (within an invoked workflow). Again, naming is vital - a poor naming will give no clear indication about the component that crashed or about the source of the problem.

### Try Catch

Any activity that may throw an exception should be part of the Try block in a Try Catch activity. It is not necessary to be directly in the Try, there can be standalone component that is not handling exceptions (no Try Catch in it), but, when invoking it, it should be placed in the Try block. There is only one exception from this rule: To set the status of a job as "Faulted" in the Orchestrator in the case of an unattended robot, the Main file must end with an exception, i.e. it should not finish the execution successfully. This only applies when the job is triggered from Orchestrator, otherwise the exception message popup is displayed on the screen. In this case there might be some logic to throw an exception in the Main file if the job is considered to be failed. In the REFrameWork, in the End Process state we have a Throw activity in case there's a fatal error - like failing to initialize.

There can be multiple **Catches** and, in case of an exception, only the most **specific** Exception will be caught and its handler executed. If the exception that is thrown in the Catch is not contained in any of the defined catches, the exception will not be caught and will propagate upwards. The **Finally** block will execute when the execution leaves the Try **and** the Catches block.

Consider the following three scenarios in which there are three catches: System.Exception, System.IO.IOException and System.IO.PathTooLongException:

1. In the first case PathTooLongException is thrown, so the catch that executes is PathTooLongException as it is the exact match (most specific). Assuming no exception is thrown in the catch, the **Finally** block will execute.
2. IO.FileNotFoundException is thrown, and the catch block executed is the IOException as FileNotFoundException inherits from the IOException class, so it is the most specific.
3. SelectorNotFoundException is thrown, and the most generic System.Exception executes. In fact, System.Exception will catch all exceptions, including custom defined ones. After that, the **Finally** block executes.
4. SelectorNotFoundException is thrown, but there is no Catch that can handle this exception. The exception is propagated upwards and Finally does not execute.

Despite their usefulness, do not overuse the Try Catch activity. You should not catch an exception unless you have a good reason for it. In most cases, the Catch will handle the exception and recover from the error. There are some cases however, when an exception is caught to perform some actions (like logging) and then the exception is rethrown to the upper levels. This is a standard mechanism in the Workblock components of the Enhanced REFrameWork (see below).

### Throw

The Throw activity is used when the intended action is to throw an exception. This activity takes an exception object input argument which can be created inline.

Another effect of using the Try Catch activity together with Throw is the reduction of decisions (If statements), as well as the subsequent increase of readability. This is because of the assumption that the code placed after the Try Catch activity will only be executed if no exception was triggered.

### Rethrow

In some cases, it may be necessary to return the exception to the normal flow by using the Rethrow activity. This activity can only be used inside the Catch block of a Try Catch activity and, as it does not receive any input, it uses the same exception that the Catch block caught.

A common use for Rethrow is when catching an exception for a particular action (for example, logging) and rethrowing it for processing in upper levels.

### Terminate Workflow

### Business Rule Exception

Business Rule exceptions can occur when an aspect of the process being automated does not follow the expected flow (for example, a Robot needs to download an invoice from an email, but the email has no attachments).

Differently from Application Exceptions, retrying Business Rule Exceptions automatically would not be a good idea, since they usually depend on some external action in order to be successful (for example, the invoice needs to be attached and the email resent). For this reason, the Orchestrator does not automatically retry transactions that failed due to a Business Rule exception. For more information, refer to the Orchestrator Guide.

### Retry Scope

The Retry Scope activity provides away to try a block for a predefined number of times in case there are any exceptions or a particular condition is not met. An important aspect of the Retry Scope activity is that it reattempts to execute its contents without ending the workflow. In addition, it does not throw exceptions unless the number of retries is reached. When checking whether a particular condition is met, the activities IsTrue and IsFalse can be used in the Condition block.

This activity is a powerful tool in cases where exceptions are thrown sporadically and other measures, like tuning selectors, already took place. For example, a particular selector is not found in a certain applications in less than 5% of the times the workflow runs, but no further selector improvements are possible. Using Retry Scope in this scenario will make the robot try to access the selector again in case a SelectorNotFoundException is thrown.

# Workflow dseign

## Design Principles

**Breaking the process in smaller workflows** is paramount to good project design. Dedicated workflows allow independent testing of components while encouraging team collaboration by developing working on separate files.

Choose wisely the layout type - flowcharts and sequences. Normally the **logic** of the process stays in flowcharts while the **navigation and data processing** is in sequences.

By developing complex logic within a sequence, you will end up with a labyrinth of containers and decisional blocks, very difficult to follow and update.

On the contrary, UI interactions in a flowchart will make it more difficult to build and maintain.

## Layout Diagrams

UiPath offers three diagrams for integrating activities into a working structure when developing a workflow file:

* Flowchart
* Sequence
* State Machine

### Sequence

Sequences have a simple linear representation that flows from top to bottom and are best suited for simple scenarios when activities follow each other. For example, they are useful in UI automation, when navigation and typing happens one click/keystroke at a time. Because sequences are easy to assemble and understand they are the preferred layout for most workflows.

### Flowchart

Flowcharts offer more flexibility for connecting activities and tend to lay out a workflow in a plane two-dimensional manner. Because of its free form and visual appeal, flowcharts are best suited for showcasing decision points within a process.

Arrows that can point anywhere closely resemble the unstructured GoTo programming statement and therefore make large workflows prone to chaotic interweaving of activities.

### State Machine

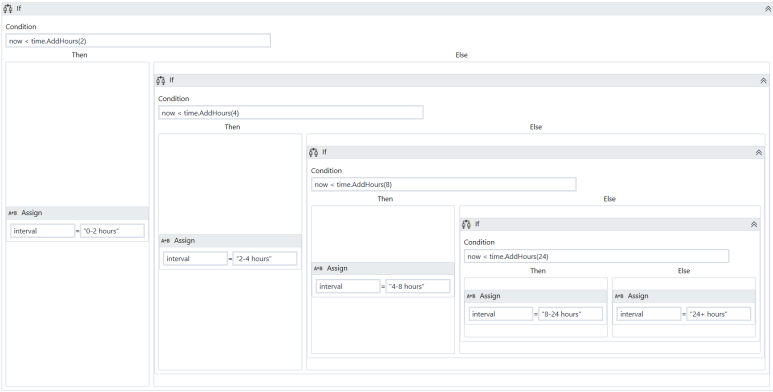
State Machine is a rather complex structure that can be seen as a flowchart with conditional arrows, called transitions. It enables a more compact representation of logic and we found it suitable for a standard high level process diagram of transactional business process template.

## Decision

Decisions need to be implemented in a workflow to enable the Robot to react differently in various conditions in data processing and application interaction. Picking the most appropriate representation of a condition and its subsequent branches has a big impact on the visual structure and readability of a workflow.

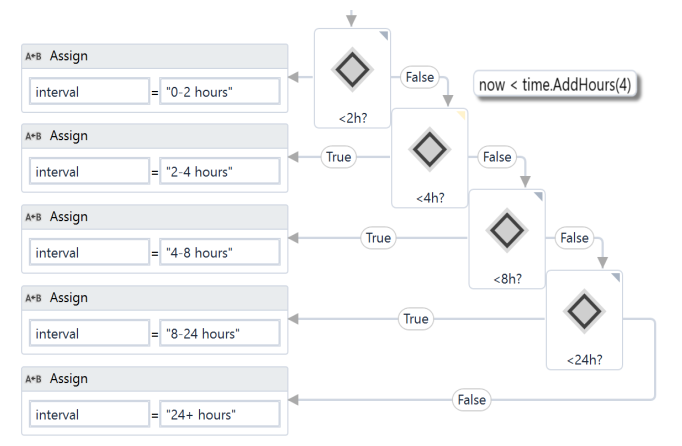
### If Activity

The **IF** activity splits a sequence vertically and is perfect for short balanced linear branches. Challenges come when more conditions need to be chained in an IF… ELSE IF manner, especially when branches exceed available screen size in either width or height. As a general guideline, nested If statements are to be avoided to keep the workflow simple/linear. Using the Collapse feature can help improve the readability.



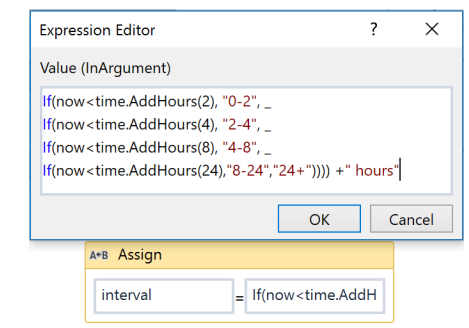
### Flow Decision

Flowchart layouts are good for showcasing important business logic and related conditions like nested IFs or IF… ELSE IF constructs. There are situations where a Flowchart may look good even inside a Sequence, e.g. the Robot Retry Flowchart in the SetTransactionStatus xaml file from the REFrameWork.



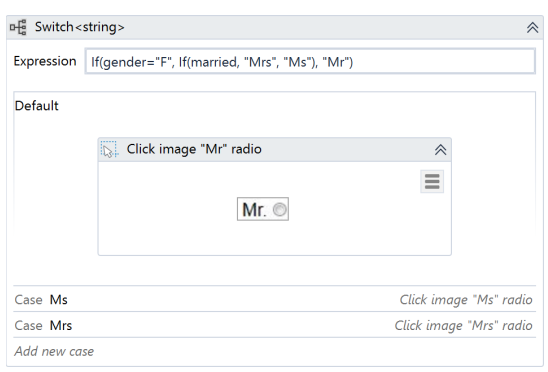
### If Operator

The [VB If operator](https://msdn.microsoft.com/en-us/library/bb513985.aspx)is very useful for minor local conditions or data computing, and it can sometimes reduce a whole block to a single activity. This might decrease the readability and should be used only for specialized code that achieves a certain function not necessarily important to the whole context. Make sure that the activity using the VB If operator is properly named or annotated.



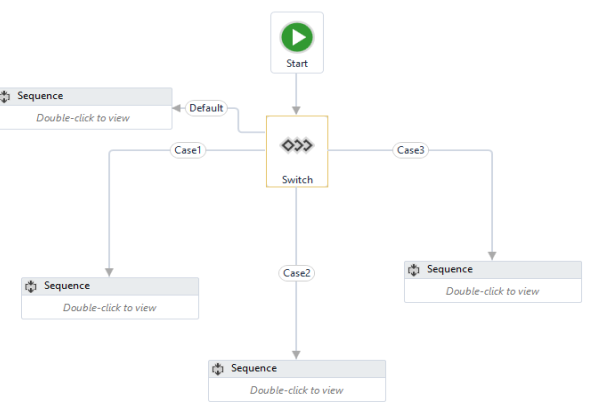
### Switch Activity

**Switch** activity may be sometimes used in convergence with the *If operator* to streamline and compact an IF… ELSE IF cascade with distinct conditions and activities per branch.



### Flow Switch

**Flow Switch** selects a next node depending on the value of an expression; **FlowSwitch** can be seen as the equivalent of the procedural **Switch** activity in the Flowchart world. It can match more than 12 cases by starting more connections from the same switch node.



## Data

Data comes in two flavors when it comes to visibility and life cycle: arguments and variables. While the purpose of arguments is to pass data from one workflow to another, variables are bound to a container inside a single workflow file and can only be used locally.

### **Variable Scope**

Unlike arguments, which are available everywhere in a workflow file, variables are only visible inside the container where they are defined, called scope.

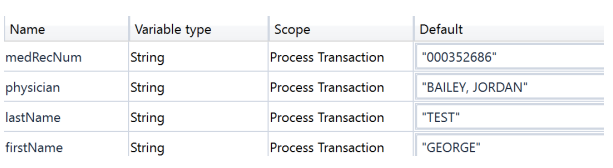
Variables should be kept in the innermost scope to reduce the clutter in the **Variables** panel and to show only, in autocomplete, what is relevant at a particular point in the workflow. Also, if two variables with the same name exist, the one defined in the most inner scope has priority.

### Arguments

Keep in mind that when invoking workflows with the **Isolated** option (which starts running the workflow in a separate [system process](https://en.wikipedia.org/wiki/Process_(computing))), only serializable types can be used as arguments to pass  
data from a process to another. For example, SecureString, Browser and Terminal Connection objects cannot safely cross the inter-process border.

### Default values

Variables and input arguments have the option to be initialized with some default static values. This comes in very handy when testing workflows individually, without requiring real input data  
from calling workflows or other external sources.



## Annotations and Comments

The **Comment** activity and **Annotations** should be used to describe in more detail a technique or particularities of a certain interaction or application behavior. Keep in mind that other people may, at some point, come across a robotic project and try to ease their understanding of the process.

## Reusability

When developing, we often need to automate the same steps in more than one workflow/ project,  
so it should be common practice to create workflows that contain small pieces of occurring  
automation and add them to the Library.  
There is no universal recipe that tells you how to split any given process.  
However, separation of **business logic** from the **automation components** is good principle that  
will help with building a code that can be reused effectively.

### Intra-Project Reusability

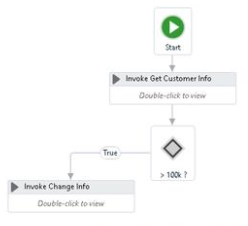
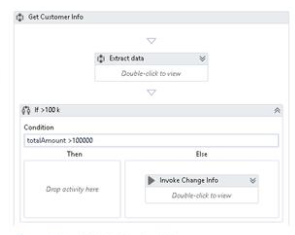
Identify flow which is/ can be repeatable within current use case.

Example:

Let’s assume that a part of your process requires reading the customer info, then – based on that info and internal business rules - update the customer details.

"*Get Customer Info*" and "*Change Customer Info*" should be two distinct automation components, completely agnostic of any process. The logic (e.g. update the customer type only when total amount is > 100k in the last 12 months) should be kept separated from automation. Both components could be used later, separately, in the same project or in a different one, with a different logic. If needed, specific data could be sent to these components through arguments.

"Change Customer Info" should not be invoked from within "Get Customer Info" - as this will  
make it more difficult to test, handle exceptions and reuse.



OK

Not OK

When separation between actions is not that obvious, copy - pasting existing code from one workflow to another (or from one project to another) – is also a good indication that you should build a separate component (workflow) for the code and invoke it when needed.

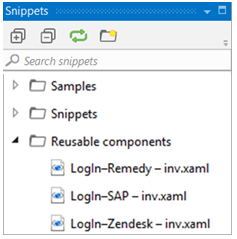
### Reusable Components

Reusable solutions **continuously accelerate** the implementation and reduce **development and deployment effort of RPA.**

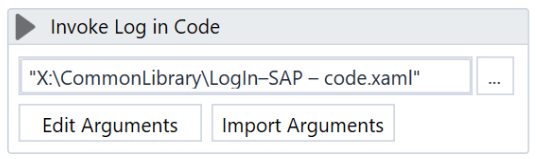
Dragging and dropping existing code from the **Library** to a workflow is easier than recreating the code from scratch, again and again. Dealing with data (Sorting, Filtering) or with text (Splitting, Regex patterns) are examples of what could be added to the sample library.

[https://go.uipath.com](https://go.uipath.com/) is a community marketplace by UiPath where a developer can find content that is highly diversified, varying between core UiPath product components and Add-Ons (e.g. Abbyy), custom activities, snippets, automation frameworks to connectors, dashboards, machine learning models and industry specific solutions.

#### Reusable Workflow/ Snippet

UiPath workflows created using UiPath Studio that can be shared and used across different projects, (e.g. SAP Login); updating it once will propagate the changes across all of your projects. These workflows use the same deployment, versioning and release management like the binary activities.

Files are better stored and maintained separately, on **network shared drives**. From that drive, they can be invoked by different Robots, from different processes. The biggest advantage of this approach – improved maintainability – is that any change made in the master component will be reflected instantly in all the processes that use it.



#### Custom activities

UiPath provides all the necessary activities to perform the required operations for automating any task. But there are scenarios where you want to add your custom code or want to use the existing .dll files. To perform these operations UiPath provides flexibility to add custom activity where you may add the required code that suits your requirement.

[Click here](https://activities.uipath.com/docs/creating-a-custom-activity) for steps to create custom activity.

## Keep it clean

In the process flow, make sure you close the target applications (browsers, apps) after the Robots interact with them. If left open, they will use the machine resources and may interfere with the other steps of automation.

Before publishing the project, take a final look through the workflows and do some cleanup: remove unreferenced variables, delete temporary **Write Line** outputs, delete disabled code, make sure the naming is meaningful and unique, remove unnecessary containers (Right-click >**Remove sequence**).

The project name is also important – this is how the process will be seen on Orchestrator, so it should be in line with your internal naming rules. By default, the project ID is the initial project name, but you can modify it from the *project.json* file.

The description of the project is also important (it is visible in Orchestrator) - it might help you differentiate easier between processes – so choose a meaningful description as well.

# Unit/ Alpha Testing

After each component is built, unit testing should be conducted. If every component is thoroughly tested, the integration runs more smoothly, and debugging lasts for a shorter period of time. The REFrameWork contains a Test\_Framework folder where all the test files should be placed. Using the RunAllTests.xaml, a developer can test a sequence containing a lot of xaml files automatically, thus being able to try out small integrations between components and to run stress tests. A report is generated at the end of each test. Typically, these kinds of tests should be run in testing environments.

The recommended UiPath architecture includes **Dev** and **Test** environments that will allow the processes to be tested outside the live production systems.

Sometimes applications look or behave differently between the dev/test and production environments and extra measures must be taken, sanitizing selectors or even conditional execution of some activities.

Use config file or Orchestrator assets to switch flags or settings for the current environment. A **test mode parameter** (Boolean) could be checked before interacting with live applications. This could be received as an asset (or argument) input. When it is set to True - during debug and integration testing, it will follow the test route – not execute the case fully i.e. it will not send notifications, will skip the OK/Save button or press the Cancel/Close button instead, etc. When set to False, the normal Production mode route will be followed.

This will allow you to make modifications and test them in processes that work directly in live systems.

# Code Review

This document defines rule that define development best practice. Peer review is essential at regular stages of the development phase to ensure development quality.

## Review Guidelines

**Modularity**

* Separation of concerns with dedicated workflows allows fine granular development and testing
* Extract and share reusable components/workflows between projects

**Maintainability**

* Good structure and development standards

**Readability**

* Standardized process structure encouraging clear development practices
* Meaningful names for workflow files, activities, arguments and variables

**Flexibility**

* Keep environment settings in external configuration files/Orchestrator making it easy to run automation in both testing and production environments

**Reliability**

* Exception handling and error reporting
* Real-time execution progress update

**Extensible**

* Ready for new use cases to be incorporated

**Flow Handling**

* Appropriate variable scope, check for duplicate variables
* Unknown exceptions to be handled gracefully
* Log for appropriate information
* Close application after use/ failure.

**Code Smell Checklist**

* Are any values hard-coded?
* Are variable values held in the appropriate place (e.g. as an Orchestrator asset or in a config file)
* Are activities appropriately named?
* Are useful annotations and comments used throughout?
* Are try-catch blocks used appropriately?
* Are logs used for appropriate information at the appropriate level?

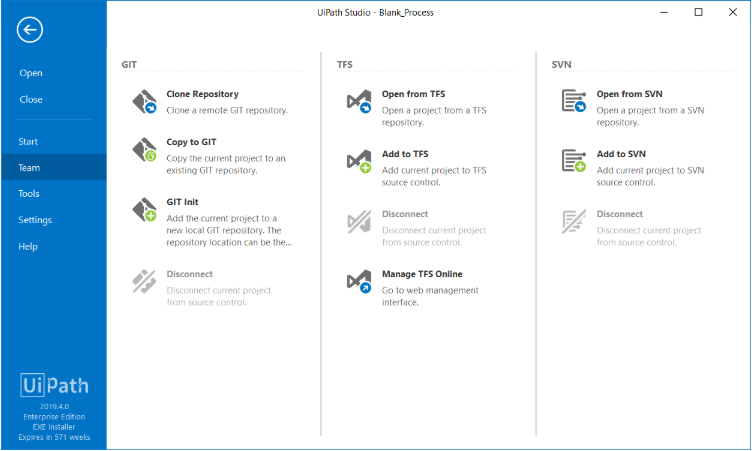
**Security Checklist**

* Is data handled securely, using the appropriate data type and stored in appropriately secured locations?
* Is it GDPR compliant?

# Source Control

In order to easily manage project versioning and sharing the work on more developers, we use a Version Control System.

[Click here](https://studio.uipath.com/docs/about-version-control) to view UiStudio integration with popular version control platforms (GIT, TFS, and SVN)



# Frameworks

Starting from a generic (and process agnostic) framework will ensure you deal in a consistent and structured way with any process. A framework will help you start with the high-level view, and then you go deeper into the specific details of each process.

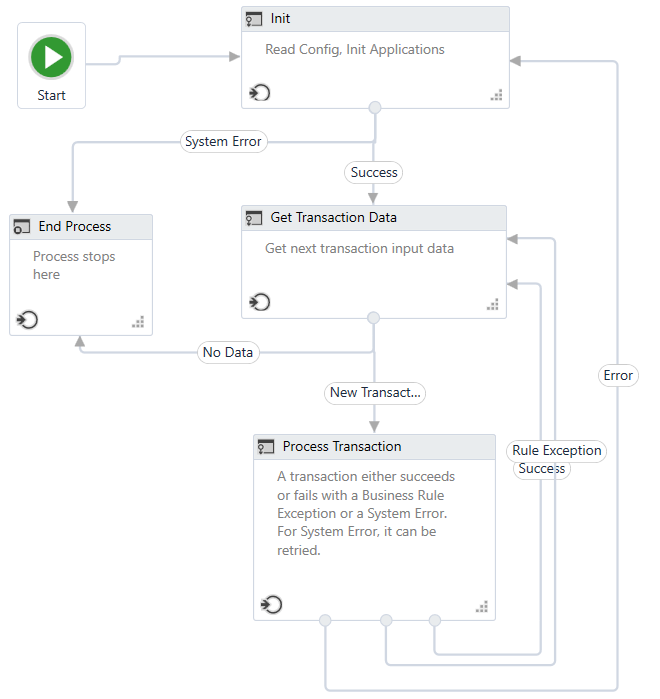
## REFrameWork

The Robotic Enterprise FrameworkTemplate proposes a flexible high level overview of a repetitive process and includes a good set of practices described in this guide and can easily be used as a solid starting point for RPA development with UiPath. The template is built on a [State Machine](https://studio.uipath.com/docs/state-machines) structure.

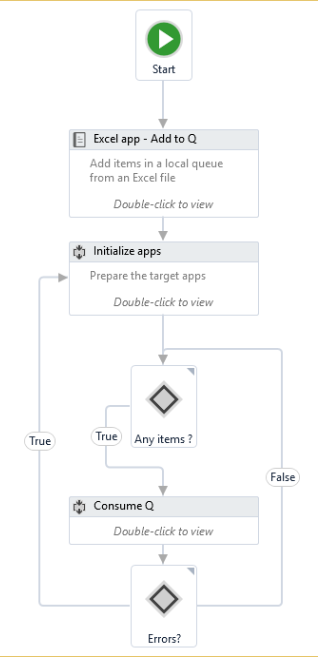
All the REFrameWork files, together with the documentation are found here: <https://github.com/UiPath/ReFrameWork>

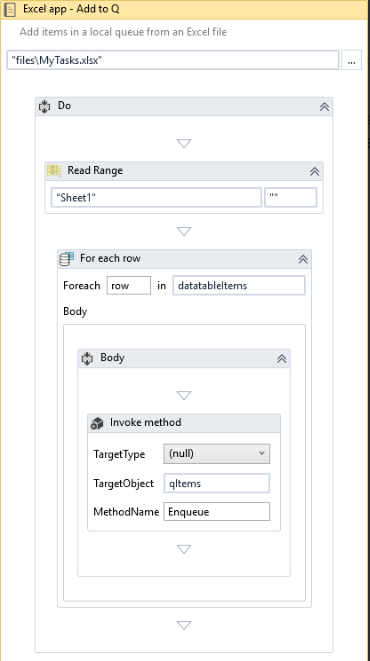
**How it works**:

* The Robot loads settings from the config file and Orchestrator assets, keeping them in a dictionary that is to be shared across workflows.
* The Robot fetches the required credentials and logs into all applications.
* It retries a few times if any errors are encountered, then succeeds or aborts.
* The Robot checks the input queue or other input sources to start a new transaction.
* If no (more) input data is available, configure the workflow to either wait and retry, or end the process.
* The UI interactions to process the transaction data are executed.
* If the transactions are processed successfully, the transaction status is updated and the Robot continues with the next transaction.
* If any validation errors are encountered, the transaction status is updated and the Robot moves to the next transaction.
* If any exceptions are encountered, the Robot either retries to process the transaction a few times (if configured), or it marks the item as a failure and restarts.
* At the end, an email is sent with the status of the process, if configured.

****

For transaction-based processes (such as processing all the invoices from an Excel file) which are not executed through Orchestrator, local queues can be built ([using .NET enqueue/ dequeue methods](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1.enqueue?redirectedfrom=MSDN&view=netframework-4.8#System_Collections_Generic_Queue_1_Enqueue__0_)).

Then, the flow of the high-level process (exception handling, retrial, and recovery) could be easily replicated - easier than by having the entire process grouped under a For Each Row loop.



## Enhanced REFramework

Enhanced framework is available at UiPath Marketplace.

Refer: <https://go.uipath.com/component/enhanced-reframework-57011>