UiPath Reboot Work.™

UiPath Automation Best Practice Guide

DISCLAIMER: Please keep in mind that this document should serve as a general guidance towards tailoring the best practices that fulfill your project and business specific needs.

Table of Contents

Naming conventions and strategies

It is important to set the naming conventions before starting a project, in order to ensure consistency and maintainability throughout the project(s). Workflow files, activities, arguments and variables should have meaningful names, in order to accurately describe their usage throughout the project.

Projects should also 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 in Orchestrator easier.

In the next sections, we will focus on specific conventions for different entities and artifacts in Studio and Orchestrator.

Variables

• 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** (also known as **PascalCase**). This practice uses compound words, no other characters between the words, where each word will start with a capital letter, for example FirstName, LastName, TransactionNumber, FilePath etc.
• Use one variable for one and only one purpose. Therefore, minimize the scope of each variable (avoid global variables).
• 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.
• 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.
• The length of the variable name should be between 6 and 20 characters long. If 20 characters are not enough to fully describe the variable, consider abbreviating longer words. Shorter variables names can be used when using a local scope (for example, om a Foreach activity - variable names index, file or row).
• **Boolean variables names** should imply True or False. The prefixes **Is** or **Has** can be used with this purpose, followed by the name. For example, ApplicationExists, IsRed, IsFound, HasRows, etc. Always use positive names. Negatives names (for example, NotFound) should be avoided.
• **Datatable variable names** should have the suffix Dt, for example: ItemsDt, ExtractedRowsDt.

Arguments

Same guidelines as for variables, with the below differences:

• Arguments should be in **Upper Camel Case** with a prefix stating the argument type, for example in_DefaultTimeout, in_FileName, out_TextResult, io_RetryNumber.
• Each argument will have a **prefix** depending on the direction: in, out, io followed by the underscore character ("_"). Examples: in_Config, out_InvoiceNumber, io_RetryNumber.
• Use default values for arguments either for testing individual workflow files, or, in case of reusable components, for using default configuration. **Delete the default values** for arguments in reusable components in the project before publishing the project.
• The length of an argument should be less than 30 characters.
• For readability and maintenance purposes, avoid having more than 20 arguments in a workflow.
• Keep in mind that when invoking workflows with the Isolated option (which starts running the workflow in a separate system process), 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.

Activities

• **Rename all the activities in a project** (including Log Message, Assign, If, Sequence). Do not leave the default name for activities.
• Activity names should concisely reflect the action taken, for example 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.

Workflow files

• For workflows, **Upper Camel Case** naming convention should be used.
• Except for Main, all workflow names should contain the verb describing what the workflow does, for example *GetTransactionData.xaml, ProcessTransa tion.xaml, Take-Screenshot.xaml*.
• A workflow file starts with the **prefix containing the application name**. For example, when working with SAP: **SAP**_Login.xaml, **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.

Projects and sub-projects

• **Upper Camel Case** naming
• Use a consistent naming convention for projects and sub-projects. For example, have a prefix for the department the process belongs to, an ID and the name. For example:
    o **FIN_001_DataCollection** (where FIN is the ID for Finance Department, 001 is the ID and DataCollection is the process name)
    o **RPA005_InvoiceProcessing** (if the department name/ID is not necessary)
• In case the process is automated using sub-processes (using multiple packages for the same business process, like using Dispatcher and Performer), append the name as a suffix. For example:
    o **FIN_001_DataCollection_Dispatcher**
    o **FIN_001_DataCollection_Performer**

Orchestrator

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

Robots
• Development machines: DEV_[Name of developer in upper Camel Case]. For example: **DEV_JohnDoe**
• Test/QA machines: **QA_[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 mix of robots and projects:
• Use prefix DEV_ or TEST_ or PROD_.
• Group by department using a prefix. For example, FIN_, AR_ etc.

Assets
• for usual 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

The configurations can be categorized 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, reusability and also increase readability, we recommend to store 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 are **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 to use Orchestrator assets. The main advantage in this case is that the values can be changed without actually 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 or queues, 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.

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

The configuration of a project should be stored in an Excel file (as it is currently in REFramework), in a json file or even in an asset. The configuration file can contain both constants and assets names, that are used to retrieve values from Orchestrator at runtime.

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 such as Orchestrator or Windows Credential Store.

• 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 retyping 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
v
Assign:

UnsecurePassword = new System.Net.NetworkCredential("abc", SecurePass-
word).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.

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 clean-up:
• add workflow annotations; use a template such as:

• for certain parts of the code that need to be explained, add annotations on activity level or comments
• remove unreferenced variables
• delete temporary Write Line outputs
• delete commented code
• make sure the naming is meaningful and unique for workflows, activities, variables, arguments
• remove unnecessary containers
• remove default arguments.

The project name is 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.

Source Control

In order to easily manage project versioning and sharing the work on more developers, we recommend using a Version Control System. UiPath Studio is directly integrated with Git, TFS and SVN - a tutorial explaining the connection steps and functionalities can be accessed here.

Workflow design

General design principles

Breaking the process in smaller workflows is mandatory for a 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 decision 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

Sequence
Sequences have a simple **linear representation** that flows from top to bottom. They are best suited for **simple scenarios when activities follow each other** and there are very few or no decisions. For example, they are useful in UI automation, when building a sequence of Type Into and Click activities. Because sequences are easy to assemble and understand, they are the preferred layout for most workflows.
Do not build lengthy sequences, since it will become difficult to understand, debug and maintain. **Aim instead to split the logic into atomic workflows, containing no more than 15 (maximum 20) activities**.
Each workflow should only contain one piece of logic. **Avoid mixing different logic in the same file** - for example, navigating to a page, downloading a file and filtering the data in the file. Instead, build 2 or 3 workflows, each tackling atomic pieces of logic (a group of 2 or more steps).

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 when multiple decisions are required in a process**.
When using REFramework, for a process with multiple decisions, the Process.xaml should be a flowchart. Then, sequences of steps must be grouped in workflows and invoked. For even better readability, the flowchart should be similar to the one in the TO BE diagram in the PDD.

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.
State machines shouldn’t be overused. Instead, they provide a template to start with (such as REFramework).

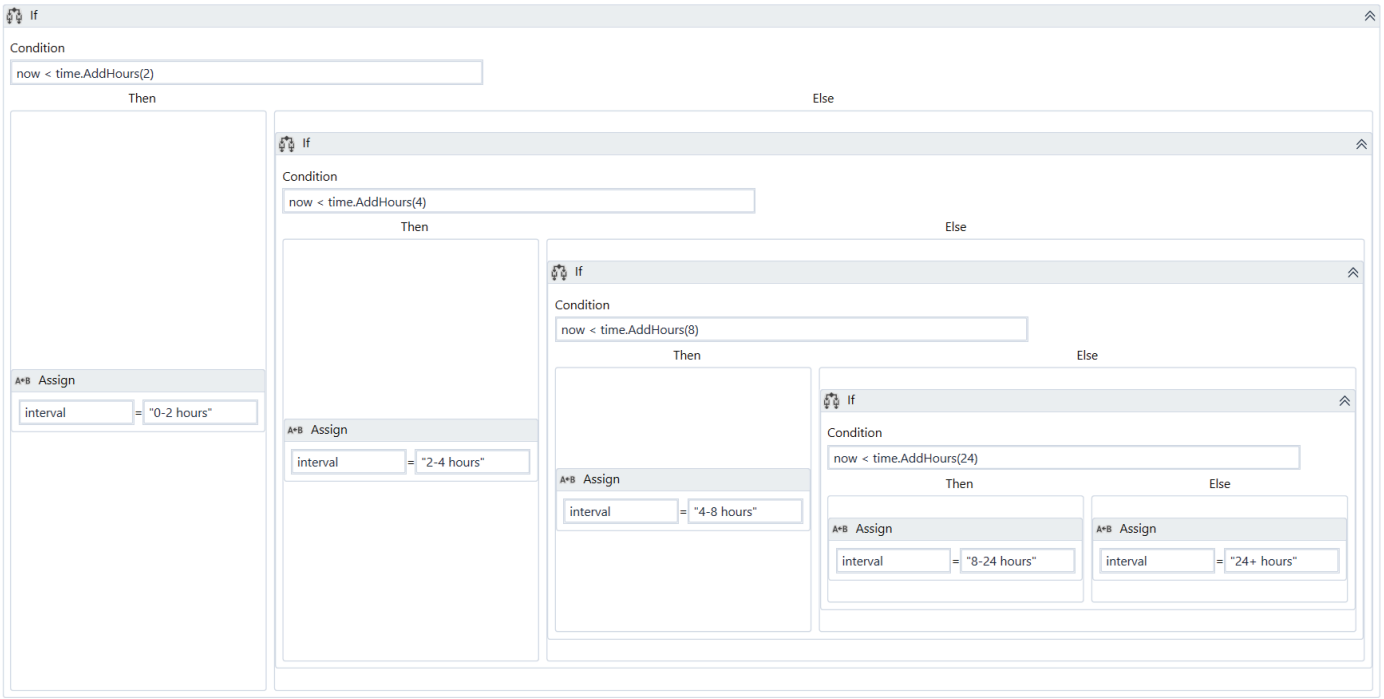
Decisions

Decisions need to be implemented in a workflow to enable the Robot to react differently in various conditions in data processing and application interaction. Choosing 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 should be avoided** to keep the workflow simple/-linear.

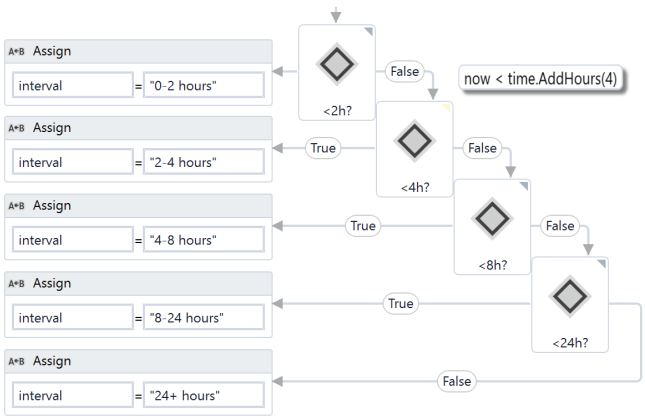
More than 3 imbricated IFs must be avoided. In these cases, use Invoke Workflow or flowchart instead.



Flow Decision

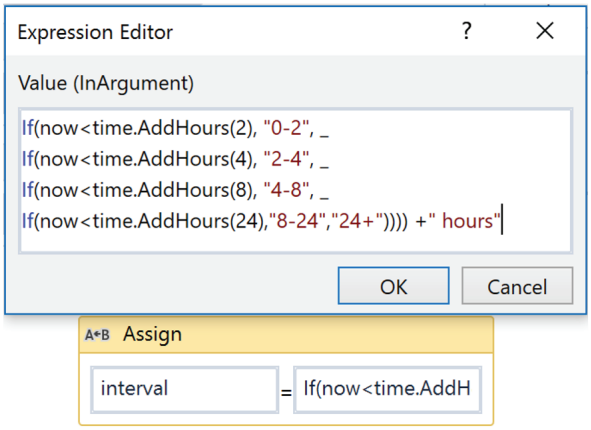
Flowchart layouts are good for showcasing important business logic and related conditions like nested IFs or IF... ELSE IF constructs.

Flowcharts should not be nested in sequences.



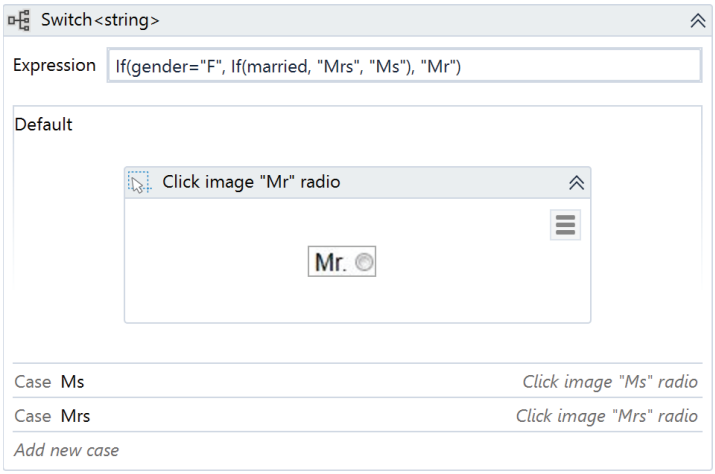
If Operator

The [VB If operator] 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.



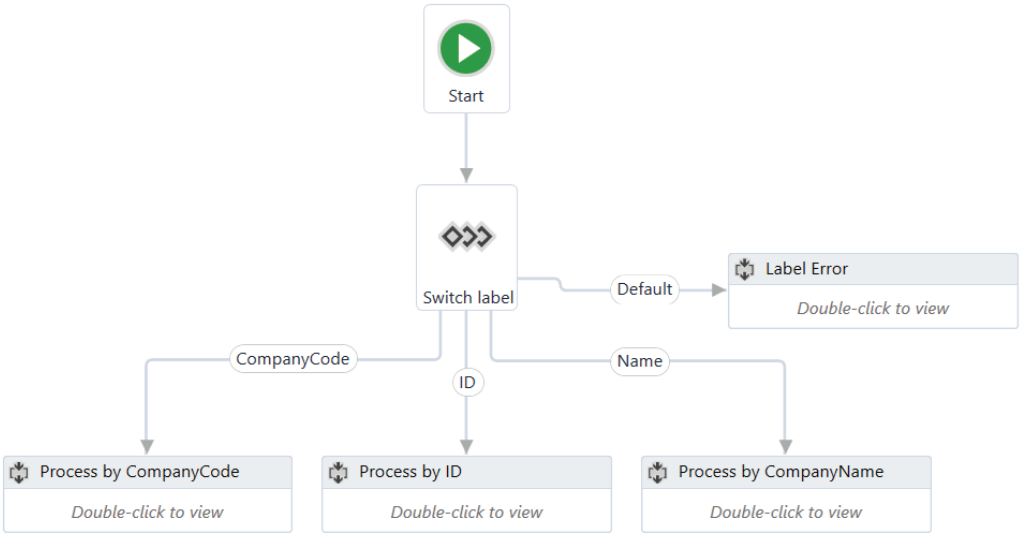
Switch Activity

The **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

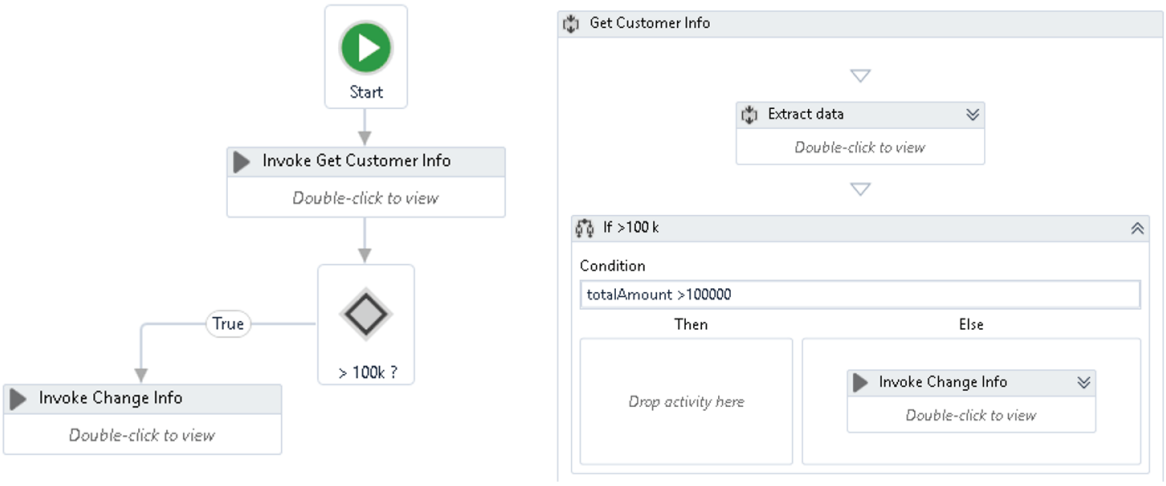
Flow Switch selects a next node depending on the value of an expression; **Flow Switch** can be seen as the equivalent of the procedural **Switch** activity in the Flowchart world. It can match more than 12 cases by starting more connections from the same switch node.



Reusability

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 a **library**.
However, separation of **business logic** from the **automation components** is good principle that will help with building a code that can be reused effectively.

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 (eg. 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.



Separate components for *Get Info* and *Change Info* - OK *Change Info* inside *Get Info* – 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.

Common (reusable) **components** (e.g. App Navigation, Login, Initialization) are better stored and maintained separately. The recommendation is to have a **library** for every application. The library can contain both workflows with UI interaction and Object Repository descriptors.

If libraries are not available due to some reasons (for example, security concerns), another approach is to store reusable workflows on **network shared drives**. From that drive, they can be invoked by different robots, from different processes. An advantage of this approach is that any change made in the master component will be reflected instantly in all the processes that use it. The disadvantage would be the tight dependency of another system; if the drive or the network fails, then all the robots using those workflows will also fail. We do not recommend this approach anymore in production.

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: *expected* or *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.

Business Rule Exceptions (BRE)

The **expected exceptions** are the one referred to as **Business Rule Exceptions**. They 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 **System 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.

The message of the business rule exception should be meaningful for the process. For example, if the business require only POs with value greater than 1000$ to be processed, we need to have an If statement in the code to check the value and if the value is <= 1000$, a BRE is thrown with the message "*The PO <id of PO> value is less or equal than 1000$. The transaction is not processed*".

The BREs are thrown by the developer using the Throw activity and their type is proprietary to UiPath: UiPath.Core.BusinessRuleException.

System Exceptions (SE)

On the other hand, all the other exceptions that can happen at runtime are considered **System Exceptions**. Another term used for them is **Application Exception (AE)**, but the concept is actually the same. All those exceptions are from .NET and fall under System.Exception class (i.e. they are either System.Exception or a class that inherits it). Proper handling should be in place for this kind of exceptions and they can cause the process to fail.

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 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**.

Try Catch

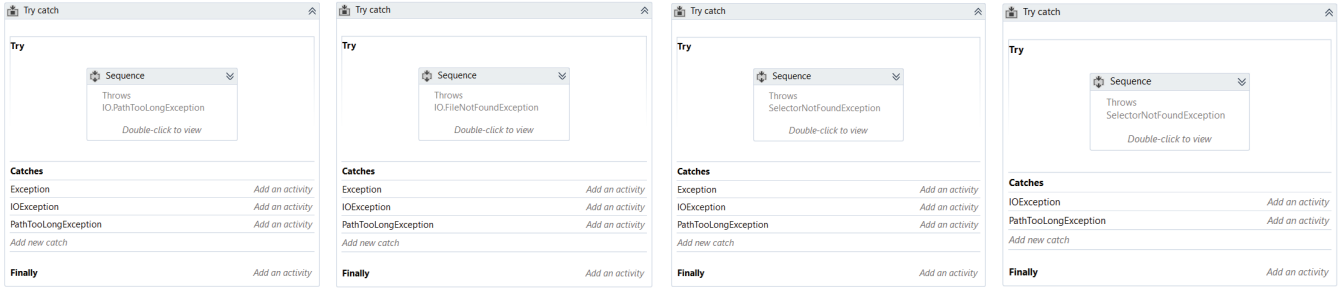
Any activity that may throw an exception should be part of the Try block in a Try Catch activity. It is not necessary to be directly in the Try, there can be stand alone component that is not handling exceptions (no Try Catch in it), but, when invoking it, it should be placed in the Try block. There 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 exits the Try and the Catches block.

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.

Example:

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.



PathTooLongException catch is executed. Finally is executed

IOException catch is executed. Finally is executed

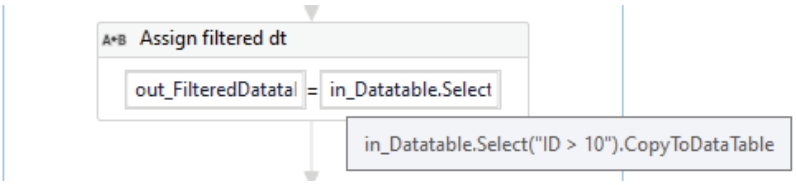
System.Exception catch is executed. Finally is executed

No Exception is caught. Finally is not executed

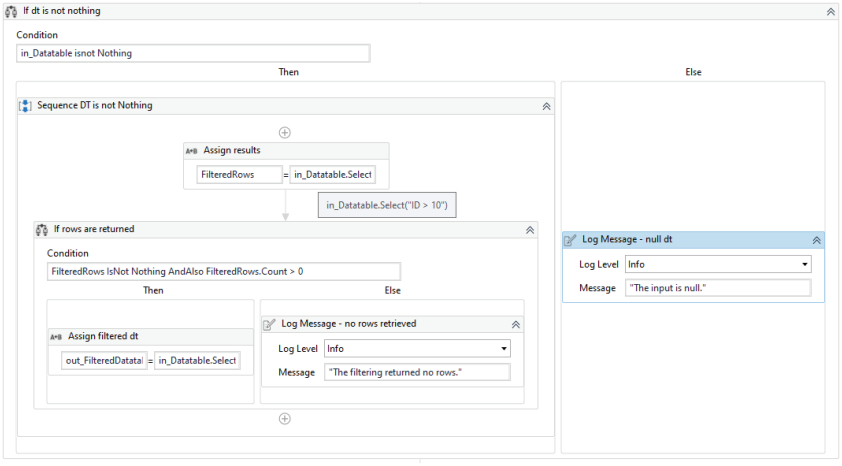
Despite their usefulness, **do not overuse the Try Catch activity**. You should not catch an exception unless you have a good reason for it. In most cases, the Catch will handle the exception and recover from the error. There are some cases however, when an exception is caught to perform some actions (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.

Examples in which Try Catch should not be used:

• an Assign activity in which we do a datatable filtering, as in the image below. This activity may throw an exception if in_Datatable is Nothing or if Select returns nothing.



Instead of placing this Assign activity in a Try, we should have an If in which we check the requirements:

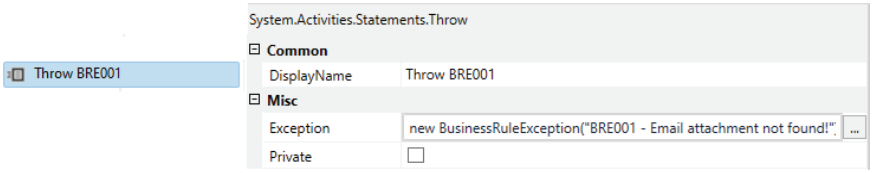


• In case of UI interaction, avoid placing Click or Type Into activities in Try Catch. Instead, use synchronization activities to check the availability of the target elements, such as: Element Exists, Find Element, Check App State etc.

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.

It should be used to throw the Business Rule Exceptions found in the PDD. Ideally, the message should be stored in the config file, for better management and maintainability.



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.

Retry Scope

The Retry Scope activity provides a way 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, only activities that return a Boolean value can be used in the Condition block. For example: IsTrue, IsFalse, Element Exists.

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.

Note: the Retry Scope will throw an exception if the retry number is exceeded. Therefore, it should be place is a Try Catch, with a proper Log Message in the Catch section.

Logging

Using **Log Message** activities to trace the evolution of a running process is essential for **supervising, diagnosing and debugging** a process. Messages should provide all relevant information to accurately identify a situation, including transaction ID and state.

As a best practice, logging should be used:

- at the beginning and the end of every workflow (Log level = Information) - the level can be downgraded to Trace in case a particular workflow is called many times (for example, in case or recursion), in order not to send too many logs to Orchestrator;
- each time an exception is caught in a Catch block (Log level = Error);
- each time a Business Rule Exception is thrown (Log Level = Error);
- when data is read from external sources (for example, log a message at Information level when an Excel file is read) (Log Level = Information);
- in Parallel or Pick activities, log messages on every branch, in order to trace the branch taken (Log Level = Information);
- in If/Flowchart Decision/Switch/Flow Switch activities (however, since processes might have a lot of these activities, we can decrease the Log Level from Information to Trace, in order not to have a lot of these logs in the database).

Messages are sent with the specified priority (e.g. Info, Trace, Warning) to the Orchestrator (if the robot is connected to it) and also saved in the local NLog file.

Logging levels and when to use them:

Fatal - The robot cannot or should not recover from this error. Something has gone critically wrong and the process needs to be stopped.

Error - An error occurred. The robot will attempt to recover and move on with the next item.

Warn - Any important data that we need to stand out from the rest of the log information.

Info - Information about robot progress. Usually includes when we enter/exit a workflow, enter a value etc.

Trace - Information useful while developing/debugging, however not useful and needed in production.

Custom Log fields
To make data easily available in Kibana for reporting purposes, the robot may tag log messages with extra values using the **Add Log Fields** activity. By default, any UiPath log output has several fields already, including message, timestamp, level, processName, fileName and the Robot's windowsIdentity. Log Fields are persistent so if we need not mark all messages with a tag, fields should be removed immediately after logging (**Remove Log Fields**). Do not to use a field name that already exists. It's important to specify the proper type of argument the first time when you add the field. This is how ElasticSearch will index it.

Frameworks

Starting from a generic framework will ensure consistency and structure. A framework will help the developer start with the high-level view, then dive into the specific details of each process.
Any process deployed in production, no matter its complexity, must follow the same agreed framework.

REFramework

The **Robotic Enterprise Framework Template** 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 structure.

All the REFramework files, together with the documentation are found here: https://github.com/UiPath/ReFrameWork or in Studio Templates.

General rules
• Follow the guidelines and best practices of REFramework in the Documentation PDF found in the template in Studio;
• Change the default annotation on Main, adding the process title, description, and any relevant information for a better understanding of the process;
• When creating process specific workflows, organize them in new folders and avoid creating them in the project's root folder or in already existing folders (Framework, Data etc.);
• Avoid deleting the framework's pre-defined workflows (if needed, just at the very end of development, you can delete the invocations of the unused workflows);
• Avoid changing the states and transitions from Main and if you do, add an annotation with the reason;

• Avoid changing the pre-defined names of the states;
• Avoid modifying the pre-defined logs; add additional ones if needed, or additional fields to the existing logs;
• Avoid adding code on transitions (it is not visible, therefore difficult to debug and maintain);
• Avoid creating new types of exceptions to treat in the framework. Use the System.Exception section to treat all the exceptions, except from the Business Rule Exceptions;
• Make sure the status set for a transaction emphasizes the correct outcome. Do not leave a successful status for an execution with errors. (For example, make sure the Catch section in Try Catches is not empty; log a message and use the Rethrow activity if you need to throw the exception to the workflow caller);
• If you need to add reporting functionalities to the framework (for example, an Excel report sent at the end of the execution), make sure you capture all the needed information in all cases including exceptions and failures.
• At the end of each transaction, setting the result of the item processing is mandatory. Otherwise, the transaction status will be set by default to Abandoned after 24 hours. This behavior is present by default in the framework and should not be changed.

Init State
• Config/Assets:
    o avoid creating multiple config files per environment (for example, “Config-qa.xlsx, Config-dev.xlsx);
    o avoid storing the config file outside of the process folder (for example, on the machine where it is deployed, outside of the UiPath folder);
    o add a description on each setting/asset in the configuration file.

• Add all the login workflows and applications opening actions in InitAllApplications.xaml (not in Get transaction data, or Process). For example, Open browser/application, Init connection for a database. Retrieve the variable corresponding to the open app (only for Classic UI activities) and/or database and use it throughout the process.

Get Transaction Data State
• Modify this section as advised in the REFramework documentation, in case of linear processes or tabular data;
• Do not remove this state, no matter if your process is a transactional one or not.

Process Transaction State

• In Process.xaml, avoid putting all the logic in a sequence/flowchart. Instead, use Invoke Workflow and split the logic into reusable/testable workflows;
• If the process' to be diagram has multiple decisions (more than 3), try to reproduce the same flowchart in the Process.xaml file – this will help the readability of the code. If the number of decisions is small, use a sequence of Invoke Workflow activities;
• If there are any predictable exceptions that can happen (i.e., selector not found, null value of a variable etc.), treat them in Process.xaml. The retry mechanism of the framework should only be used for unforeseen exceptions. Use Retry Scopes, Try Catches, UI Synchronization activities in order to mitigate exceptions;
• Throw Business Rule Exceptions with a proper message for all the BRE's that are captured in the PDD. Avoid throwing System Exceptions;
• For any custom actions required on BRE/System exceptions, create new workflows with the required logic and invoke them in the Catch section in Process Transaction.

End Process

• Include in this state any logic that needs to be executed at any end of the process. Avoid including the logic at the end of Process.xaml, for example, as the code may not be reached in every situation. A good example would be an email notification sent at the end of the processing, with the results.

UI Automation

The UI Automation techniques have evolved a lot in UiPath in the past releases. Therefore, there are more and more possibilities to automate the UI and the right method for the use case should be a well-informed decision.

How to choose the right approach

Before automating the steps in a certain application (through a virtual environment or local, no matter the type - web or desktop), the following 'recipe' needs to be applied in order to find the right approach:

Applications available only in virtual environments (Citrix, VMWare, RDP):

• Check if it is possible to install the **UiPath Remote Runtime** on both client and server. With this approach, the selectors will become available in UiPath.

• Investigate if the application exposes an **API** that can be used to automate at least a part of the process' steps. API calls are the most reliable for interaction with an application. Check also our integrations with that application (from our Integrations team or on marketplace) to see if we have activities for that application.
• If the first two approaches are not possible (for example, the client's policies prohibit them), the next step is to try automating with **Computer Vision**.
• If CV is not available (for example, the client does not have a Cloud API key for CV or they are not willing to send screenshots to the UiPath CV server), explore the possibility of using a **Local CV Server**.
• If neither Remote Runtime nor CV are possible or if they do not return satisfactory results, then go for **Image Automation**. Keep in mind that this should be the last resort! Image automation is sensitive to screen variations (for example, resolution, colors, fonts) and it is considered unreliable.
• Consider using also **keyboard shortcuts** in order to navigate and get data.

Applications available on the local environment (web, desktop):
• Investigate if the application exposes an **API** that can be used to automate at least a part of the process' steps. API calls are the most reliable for interaction with an application. Check also our integrations with that application (from our Integrations team or on marketplace) to see if we have activities for that application.
• Pick the UI Automation approach - **modern vs classic**:
o modern - if the Studio version >= 20.10 (note: you can still use the UIA package v. 20.10 on older Studio versions, but the experience is limited);
o classic - if the Studio version < 20.10.
• Test the application compatibility with UiPath: navigate through different screens in the application and get selectors with UIExplorer, try out a few Click and Type Into activities.
• In case the selectors generated by UIExplorer are not reliable:
    o try different **UI Frameworks** in UIExplorer: Default, Active Accessibility, UI Automation;
    o fine-tune the selectors, add anchors;
    o try to use alternative methods, such as **Find Relative Element**.
• Consider using also **keyboard shortcuts** in order to navigate and get data.
• If selectors are still not reliable, explore the option of using **Text Automation**. For example, if the text on a button is unique, we can try Click Text instead of Click.
• For legacy applications that do not expose selectors, **Computer Vision** is an option (see steps above).
• Similarly to the applications in virtual environments, the last resort to use is **Image Automation**. With the modern UIA, anchors can also be defined for images, making this approach a little more reliable.

General UIA best practices

Input Methods

There are three methods used for triggering a **Click** or a **Type Into** an application.
• If **SimulateType/SimulateClick** are selected, Studio hooks into the application and triggers the event handler of an indicated UI element (button, edit box);
• If **SendWindowMessages** is selected, Studio posts the event details to the application message loop and the application's window procedure dispatches it to the target UI element internally;
• Studio signals system drivers with **hardware events** if none of the above option are selected and lets the operating system dispatch the details towards the target element.

These methods should be tried in this order, as Simulate and WindowMessages are faster and also work in the background, but they depend mostly on the technology behind the application.

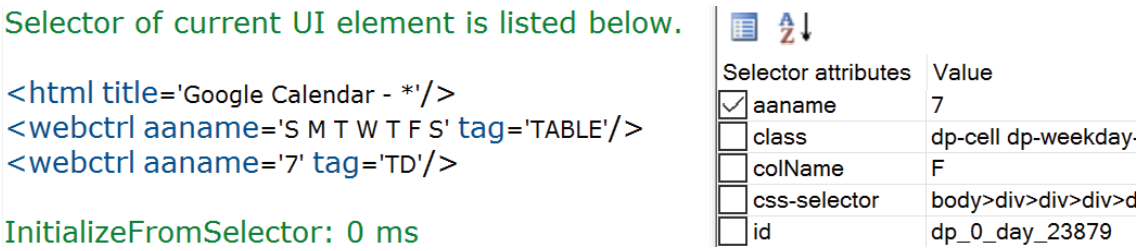
Hardware events work 100% as Studio performs actions just like a human operator (i.e. moving the mouse pointer and clicking at a particular location), but in this case, the application being automated needs to be visible on the screen. This can be seen as a drawback, since there is the risk that the user can interfere with the automation, in attended scenarios.

Timeouts and delays
Create variables/arguments for timeouts and stored the values in the config file. It this way, it is easier to maintain the values and have an overview of all the timeouts in the projects. Avoid as much as possible hardcoding values.

Avoid using DelayBefore and DelayAfter. Try to use UI synchronization methods instead.

Selectors
Sometimes the automatically generated selectors contain volatile attribute values to identify elements and manual intervention is required to calibrate the selectors. A reliable selector should successfully identify the same element every time in all conditions (development, test and production environments) and no matter the usernames logged on to the applications. In other words, it must be **specific enough to uniquely identify interface elements**, but also **generic enough to work even if there are a few changes** on the screen.

Tips on how to improve a selector in **Selector Editor** or **UIExplorer**:
• Replace attributes with volatile values with attributes that look steady and meaningful;
• Replace variable parts of an attribute value with wildcards (*);
• If an attribute's value is all wildcard (e.g. name='*') then attribute should be removed, since it would not contribute to restricting the search for the element;
• If editing attributes doesn't help, try adding more intermediary containers (e.g., Attach Browser and Attach Window) to help restricting the search for the element (*only for classic exprience*);
• Avoid using idx attribute unless it is a very small number like 1 or 2.



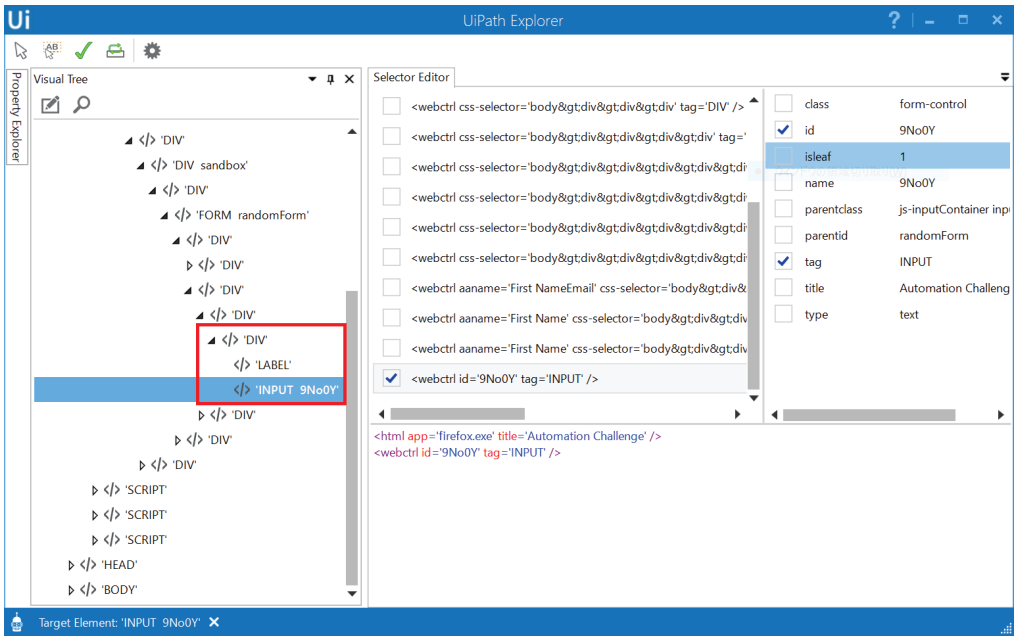
Wildcards
Wildcards can be used to make a part of an attribute more generic. For example, if the title of a window is represented by *title='Calendar March 4, 2021'*, the selector will not work if the date changes. In these cases, it is better to use a wildcard to replace the date part, which will indicate that the selectors should find all windows whose title begins with the word *Calendar*.

The excessive use of wildcards can make the selector too generic and match more than one element, so in some situations it is necessary to combine it with other attributes or other techniques to define selectors.

IDX
The *idx* attribute should be used carefully. The value of the *idx* attribute represents the index of a particular element that has the same selector as elements on the screen. There might be some undesired behavior when using the *idx* attribute since the index can change with the order of appearance of such elements. For this reason, it is recommended not to use this attribute, unless it is a small number like 1 or 2 or you specifically need to retrieve the N-th element that appears on the screen with that particular selector.

The <nav/> tag

Other than the attributes of related elements, it is also possible to use the element hierarchy (seen on the Visual Tree panel of UIExplorer) to construct selectors. For example, the following figure shows the result of inspecting the 'First Name' text box that appears in the form of the RPA Automation Challenge (**www.rpachallenge.com**). It is possible to see that the text box (the HTML element *INPUT*) is on the same hierarchy level as the *LABEL* element. In addition, we can see that the text box will follow the *LABEL* element. W e can take advantage of this hierarchy information to find the correct text box for a desired label.



To translate this logic into a selector, we make use of the tag <nav next='1'> to indicate that we are looking for an element that comes after the label on same hierarchy level (i.e., the next sibling):

```
<html title='Automation Challenge' />
<webctrl aaname='First Name' tag='LABEL' />
<nav next='1' />
```

There are a few points of attention when using this hierarchy navigation:

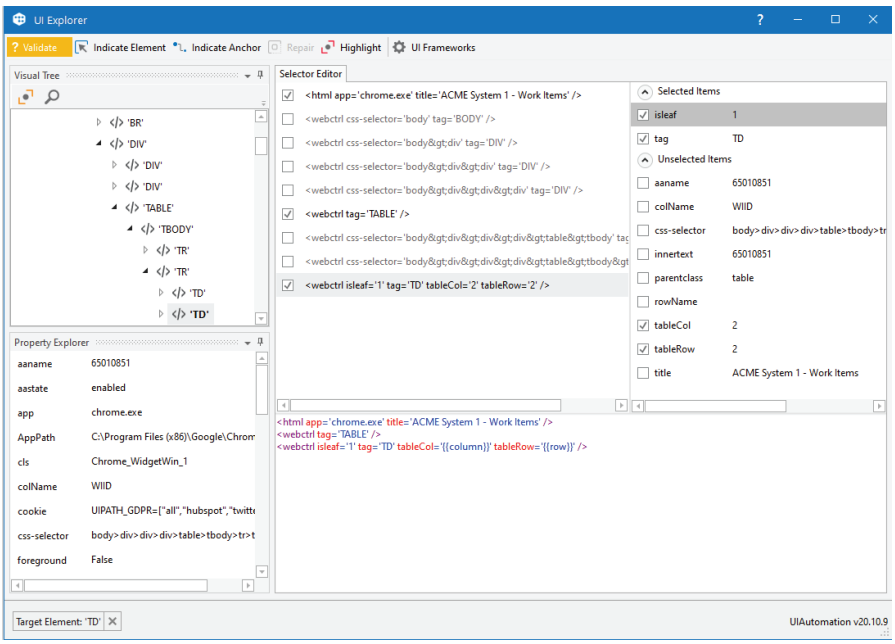
- Using this syntax, it is possible to navigate to next, previous and up. To go down the hierarchy, use the **Find Children** activity.
- The number associated to the direction tells how many times we should move in that direction. For example, *next='1'* refers to the first next sibling and *next='2'* refers to the second next sibling.
- If order of the elements varies, the selector might have an undesired behavior. For example, sometimes there will be a line break (HTML element
) between the label and the

• If order of the elements varies, the selector might have an undesired behavior. For example, sometimes there will be a line break (HTML element *
*) between the label and the tex tbox, so *<nav next='1'>* would not find the text box.

Dynamic Selectors

A dynamic selector uses a variable or an argument as a property for the attribute of your target tag. Use this approach whenever you need to match the selector with a value only known at runtime and stored in a variable.

For example, using the same selector, all of the data in a table can be identified, using the **column** and the **row** variables.



Finding UiElements

If it is not possible to define a reliable selector for an element, there are still other options that can retrieve it. They mainly work based on a relative element (for which a selector can be defined), but they still provide 100% accuracy as long as the relative element is the same.

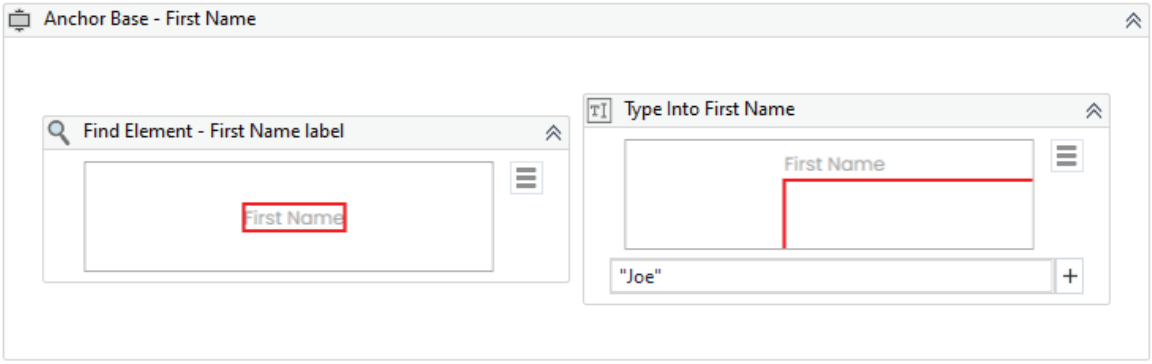
Find Relative Element Activity

One way to use one element to find another is by taking advantage of the Find Relative Element activity. This activity returns an UiElement object that is located in a certain offset from a selected element. We can then pass this object to another activity that could not access it directly. This method can be useful in situations having static elements that cannot be directly accessed (for example, checkboxes in a few SAP screens).

Anchor Base Activity

Note: this activity exists only in the classic UIA package. For modern, anchors are now embedded in the unified target.

Another activity that also takes in consideration another element is the Anchor Base activity. This activity works by defining an element (like the label "First Name") to be the anchor that will help find the desired text box (see figure below). The Anchor Base activity sets a starting point on the screen (the center of the selected element to be the anchor) and looks for a match based on the direction specified by the AnchorPosition property. Since the element is searched based on what is shown on screen, this activity cannot work on the background.

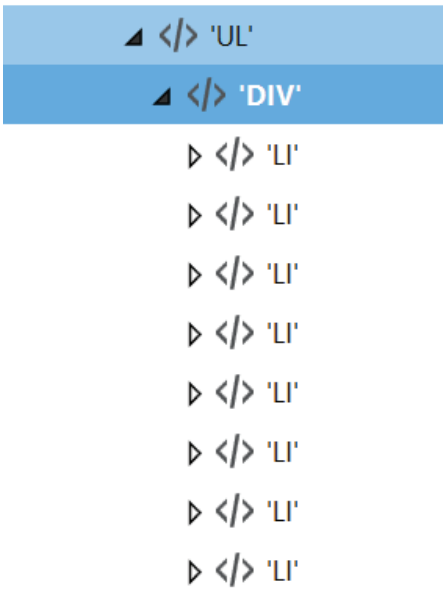


Find Children Activity

In certain cases, it is necessary to work with a collection of elements that are relative to a particular one.

For example, tabular data that cannot be extracted used using the Scraping Wizard may be retrieved using the Find Children Activity. More concretely, considering an HTML table, we may use the Find Children activity to retrieve all of its rows (i.e., the children of the table's body). In another situation, this activity can be used to retrieve all entries of a drop-down menu (or combo box) in order to verify whether a particular entry exists before trying to select it with the Select Item activity.

When using this activity, the UIExplorer is an useful tool to understand the structure of the interface and identify the parent-child relationships among elements. The following figure shows the result of such an inspection: an element having the DIV tag and its children having the tags LI.



Containers
Selectors can be full or partial. Full selectors start with a window or html identifier and have all necessary information to find an element on the whole desktop, while partial selectors work only inside an attach/container that specifies the top-level window where elements belong: **OpenBrowser, OpenApplication, AttachBrowser, AttachWindow.**

There are several advantages to using containers with partial selectors instead of full selectors:

- Visually groups activities that work on the same application;
- Is slightly faster, not seeking for the top window every time;
- Makes it easier to manage top level selectors in case manual updates are necessary.

Note: full selectors exist only in the classic UIA package. In the modern experience, all UI activities need to be placed inside a *Use Application/Browser* container. All the containers are now merged into a single one (*Use Application/Browser*).

Modern UIA best practices

The modern experience brings some changes to the UI activities and therefore some changes in mindset are required.

1. **Element Exists** was replaced by **Check App State**.
2. Full selectors are no longer used, therefore all the interaction has to happen in containers.
3. Open **application/browser** were deprecated and **Use application/browser** are now the only container activities. Therefore, they need to be used for both opening and attaching a browser/window. Make use of Open and Close properties on Use application/browser activity, to make sure we have the proper expected behavior.
4. Timeouts are now in seconds instead of milliseconds. Also, the data type has changed from Int to Double.
5. Leverage Object Repository (OR) when working with the new activities. Have the descriptors in a different library project.

UI Synchronization

Unexpected behavior is likely to occur when the application is not in the state the workflow assumes it to be. The first thing to watch for is the time the application takes to respond to Robot interactions.

The **Delay** property of input enables you to wait a while for the application to respond. However, there are situations when an application's state must be validated before proceeding with certain steps in a process. Measures may include using extra activities that wait for the desired application state before other interactions. Activities that might help include:

- **ElementExists, ImageExists, Text Exists, OCR Text Exists**
- **FindElement, Find Image, Find Text**
- **WaitElementVanish, WaitImageVanish**
- **WaitScreenText** (in terminals)

Background Automation

If an automation is intended to share the desktop with a human user, all UI interaction must be implemented in the background. This means that the automation has to work with UI element objects directly, thus allowing the application window to be hidden or minimized during the process.

- Use the **SimulateType**, **SimulateClick** and **SendWindowMessages** options for navigation and data entry via the **Click** and **TypeInto** activities
- Use the **SetText**, **Check** and **SelectItem** activities for background data entry
- **GetText, GetFullText** and **WebScraping** are the output activities that run in the background
- Use **ElementExists** to verify application state

Image Automation

Image recognition is the last approach to automating applications if nothing else works to identify UI elements on the screen (as we previously mentioned, selectors, keyboard shortcuts or Computer Vision). Because image matching requires elements to be fully visible on the screen and that all visible details are the same at runtime as during development, when resorting to image automation extra care should be taken to ensure the reliability of the process. Selecting more/less of an image than needed might lead to an image not found or a false positive match.

Resolution Considerations
Image matching is sensitive to environment variations such as desktop theme or screen resolution. When the application runs in in a virtual environment, it is important that the resolution is the same as in development. Otherwise, small image distortions can be compensated by slightly lowering the captured image Accuracy factor.

Check how the application layout adjusts itself to different resolutions to ensure visual elements proximity, especially in the case of coordinate based techniques like relative click and relative scrape.

If the automation supports different resolutions, parallel workflows can be placed inside a **PickBranch** activity and the robot will use either match.
OCR Engines
If OCR returns good results for the application, text automation is a good alternative to minimize environment

OCR Engines
If OCR returns good results for the application, text automation is a good alternative to minimize environment

Integrations

Email

Because of Outlook's caching mechanism, automating emailing steps with the Outlook activities may cause issues (especially in unattended use cases). For example, Outlook not synchronizing, emails not received or stuck in Outbox etc. This may happen especially for repeated operations on Outlook and when the robot sends an email and tries to read it right away. For example: read an email from Inbox, move it to a new folder called 'InProgress', then try to read it again from the new folder and get the UUID; in this case, the email might take a long time (or even require a refresh) in order to be seen in the 'InProgress' folder.

The solution is to choose, whenever possible, other email activities, that interact with the server, for example: O365, Exchange, SMTP, IMAP.

Excel

Usually, choose Workbook activities instead of Excel Scope, since they do not require Excel to be installed on the machine. However, depending on the operations needed, the Excel Scope might be more helpful, since it provides more activities. For Workbook related activities, the Excel workbook file should be closed at runtime otherwise it will throw and error that the file is being used by other process.

Avoid using Excel files as databases. If Excel reporting is required, avoid updating it after each transaction and choose to have a reporter process to gather the data at the end of the performer, instead. Another good practices would be to use a database, if it is available. For example: **Data Service** from UiPath, a local SQLite database or any other RDBMS/NoSQL database available on the servers of the customer.

Excel may cause problems such as freezing, not responding or corruption of file, therefore we need to avoid performing a lot repeated operations on files.

Automation Lifecycle

Process Understanding

Deciding between an automation for **attended or unattended automations** is the first important decision that impacts how developers will build the code. The general running framework (robot triggering, interaction, exception handling) will differ. Switching to the other type of robots later may be cumbersome.

For time critical, live, humanly triggered processes (e.g. in a call center) an attended robot working side by side with a human might a the possible answer. UiPath Apps can also be a viable solution.

Not all processes that need human input are supposed to run with attended robots. Even if a purely judgmental decision (not rule-based) during the process could not be avoided, evaluate if a change of flow is possible - like splitting the bigger process in two smaller sub-processes, when the output of the first sub-process becomes the input for the second one. Human intervention (validation/modifying the output of the first sub-process) takes places in between. This use case could make use of long running workflows and Actions (human in the loop). A typical case would be a process that requires a manual step somewhere during the process (e.g. checking the unstructured comments section of a ticket and - based on that - assign the ticket to certain categories).

Generally speaking, going with an unattended robot will ensure a more efficient usage of the robot load and a higher ROI, a better management and tracking of robotic capacities.

But these calculations should take into consideration various aspects (an attended robot could run usually only in the normal working hours, it may keep the machine and user busy until the execution is finished etc.). Input types, transaction volumes, time restrictions, the number of robots available etc. will play a role in this decision.

Documenting the process - DSD

The process documentation guides the developer's work and provides help in tracking the requests and the application maintenance. Of course, there might be lots of other technical documents, but one is critical for a smooth implementation - DSD (Development Specification Document).

The **Development Specification Document (DSD)** should contain the automated process details and focus on two main categories: **Runtime Guide** and **Development Details**.

The Runtime Guide should contain a high-level runtime diagram, as well as details about the functionality of the robot, such as sub-processes, schedules, configuration settings, input files, output files, temporary files, and performed actions. Additional details about the master process should be specified - prerequisites, automatic and manual error handling, process resuming in case of failure, Orchestrator usage, logging and reporting, credential management, and any other relevant information related to security or function.

The Development Details should contain information about the packages in use, the development environment, the logging level, the source code repository and versioning, a list of workflow components with their description and argument list, a list of reusable components, the workflow invoke tree, defined custom logs and log fields, relevant snapshots of the process flowchart, the level of background vs foreground automation, and any other relevant or outstanding development items.

Development and Code review

The **RPA Solution Architect** is responsible for continuously coaching developers on the best practices. Hence, frequent and thorough code reviews are a must, to enforce a very high quality of the developed workflows. This way, the developers are motivated to build robust workflows and to follow the best practices guide.

Workflow Analyzer is a powerful tool for both the developer and the SA. The standard rules that come embedded with the Studio should be up to date for the organization and, if required, new rules can also be added.

The developer should run Workflow Analyzer periodically during the development in order to ensure that the code meets the required standards. In this way, the time necessary for any modifications or feedback during code review may decrease.

The SA can use the tool for help during the code review phase, to gather feedback on naming conventions and best practices used. However, the SA should also have a checklist and should make sure they look over every workflow and understand the logic. Any improvement should be documented an implemented at this point, to ensure the high quality of the implementation and also the possibility to scale the process in the future.

Test

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.

There are two methods for unit testing:

1. 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 outside office hours, in testing environments, to optimize the developer's time.
2. Using **Test Suite** and creating **Test Sets** (group of unit and integration tests) that can be run after every modification in the code.

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.

Release

There are various ways of designing the architecture and release flow – considering the infrastructure setup, concerns about the segregation of roles etc.

In this proposed model UiPath developers can build their projects and test them on Development Orchestrator. They will be allowed to check in the project to a drive managed by a version control system (GIT, SVN, TFS etc).

Publishing the package and making it available for QA and Prod environments will be the work of a different team (operations, IT, support).

Here is the project publishing flow, step by step:

- Developers build the process in UiPath Studio and test it with the Development Orchestrator; once done, they check in the workflows to a repository (on VCS).
- The IT team will create the package for QA. This will be stored on a ***QA Package*** folder on VCS QA run the process on dedicated machines.
- If any issue revealed during the tests, steps above are repeated.
- Once all QA tests are passed, the package is copied to a the production environment.
- Process is going live, run by the production robots.

Reusable libraries are created and deployed separately.