UiPath Automation Best Practice Guide

Table of Contents

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 the 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 describe 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_DanielDines
- 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 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 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.

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.

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.

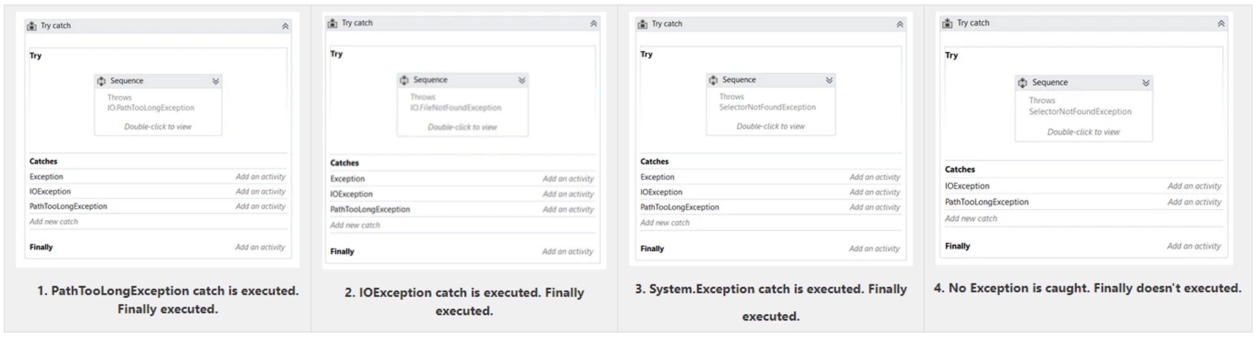
Try Catch

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 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 succesfully. 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.



1. PathTooLongException catch is executed. Finally executed.
2. IOException catch is executed. Finally executed.
3. System.Exception catch is executed. Finally executed.
4. No Exception is caught. Finally doesn't 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 (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

Exception object (+ Data dictionary)

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.

Debugging

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: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.

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 TFS & SVN - a tutorial explaining the connection steps and functionalities can be accessed here.

SVN

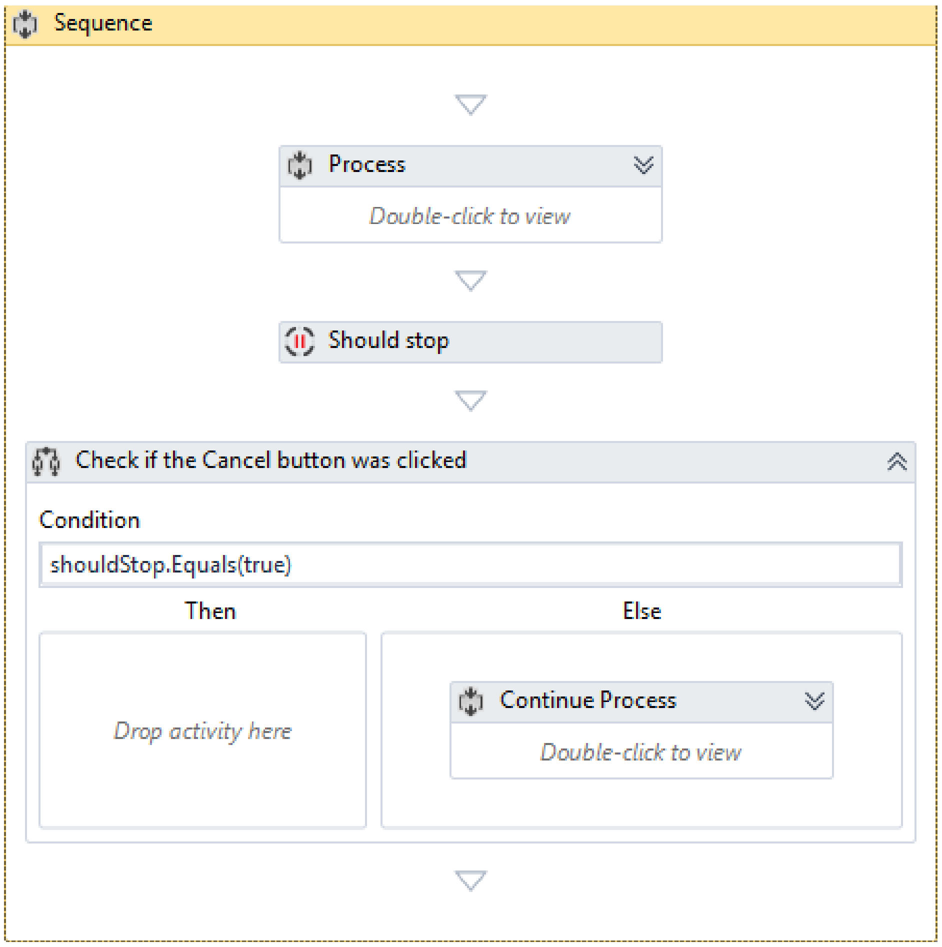
TFS

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, 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 structure.



shouldStop.Equals(true)

Principles

How it works:

- The Robot loads settings from the config file and Orchestrator assets, keeping them in a dictionary to be shared across workflows
- The Robot logs in to all applications, before each login fetching the credentials
- 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 (e.g. processing all the invoices from an Excel file) which are not executed through Orchestrator, local queues can be built (using .NET enqueue/ dequeue methods).

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

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

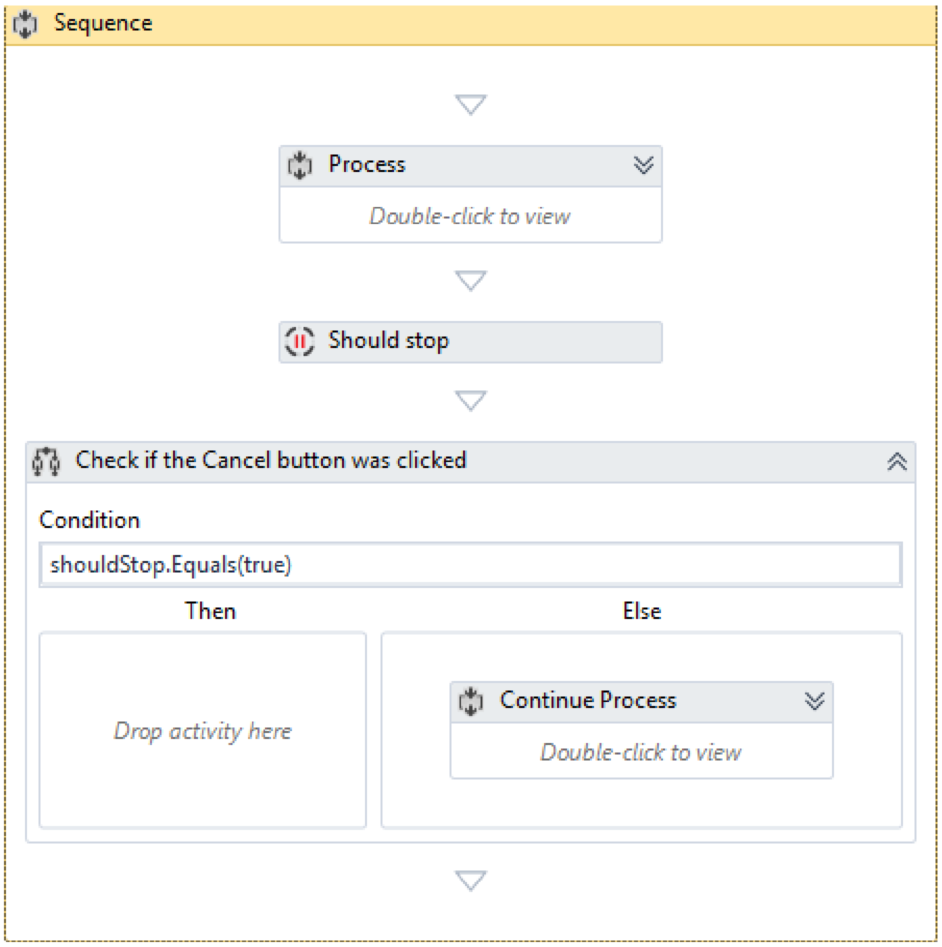
Refer to documentation

Enhanced REFrameWork Principles

Refer to documentation

Pilot Frameworks Principles

Refer to documentation



shouldStop.Equals(true)

Project related files (e.g. email templates) could be organized in **local folders** or **shared drives.**

Note: If placed inside the project folder, they will be replicated during the deployment process (together with the projects workflows) on all Robot machines under lib/net45 folder

These folders could be also stored on a **shared drive** - so all the Robots will connect to the same unique source. This way, the process related files could be checked and maintained by the business users entirely, without support from the RPA team. However, the decision (shared or local folders) is complex and should take into consideration various aspects related to the process and environment: size of the files, frequency of changes, concurrency for editing the same file, security policies etc.

Workflow design

Design Principles (from Frameworks)

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

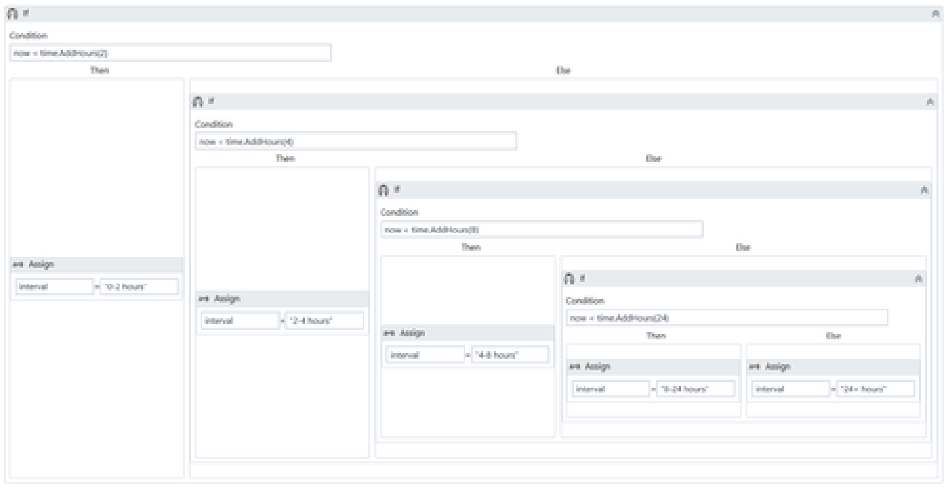
State Machine is a rather complex structure that can be seen as a flowchart with conditional arrows, called transitions. It enables a more compact representation of logic and we found it suitable for a standard high level process diagram of transactional business process template.

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. 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

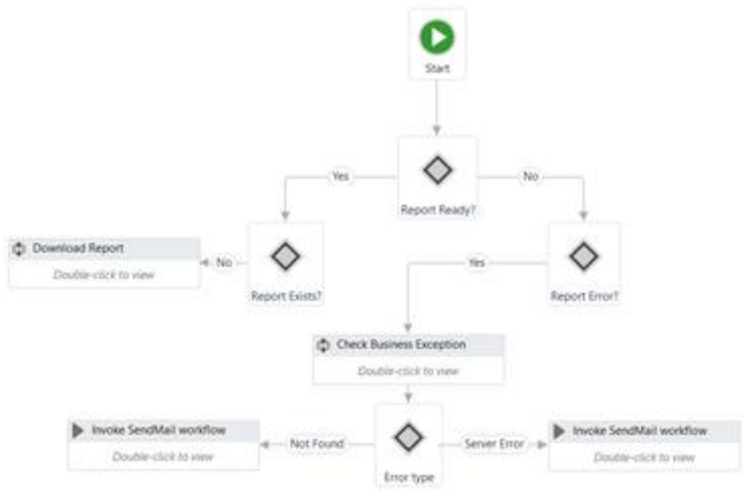
The **IF** activity splits a sequence vertically and is perfect for short balanced linear branches. Challenges come when more conditions need to be chained in an IF… ELSE IF manner, especially when branches exceed available screen size in either width or height. As a general guideline, nested If statements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



```
now < time.AddHours(2)
interval = "0-2 hours"
now < time.AddHours(4)
interval = "2-4 hours"
now < time.AddHours(8)
interval = "4-8 hours"
now < time.AddHours(24)
interval = "8-24 hours"
interval = "24+ hours"
```

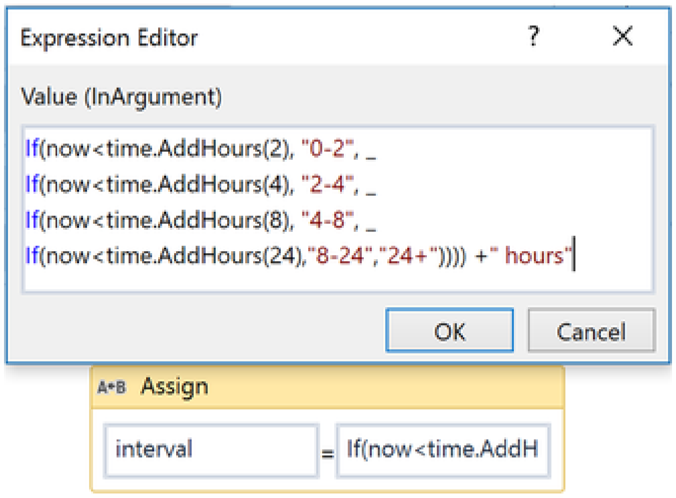
Flow Decision

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** 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 acieves 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.

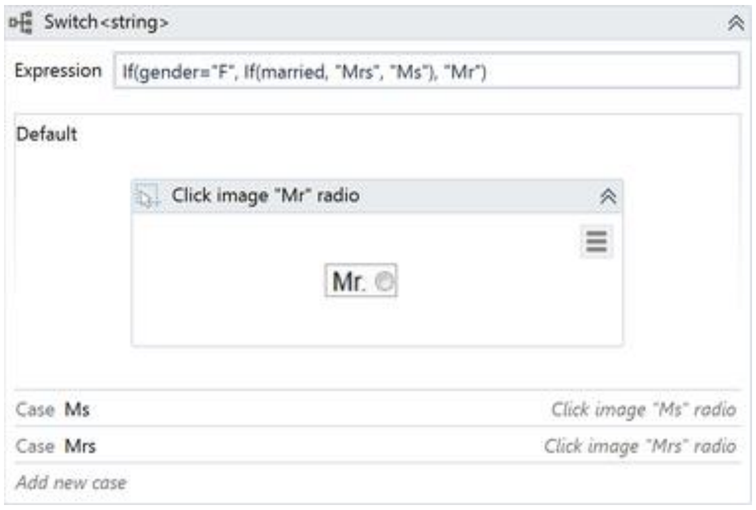


```
If(now<time.AddHours(2), "0-2", _
If(now<time.AddHours(4), "2-4", _
If(now<time.AddHours(8), "4-8", _
If(now<time.AddHours(24),"8-24","24+")))) +" hours"
```

interval = If(now<time.AddH

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.


Switch<string>

Expression If(gender="F", If(married, "Mrs", "Ms"), "Mr")

Default

Click image "Mr" radio

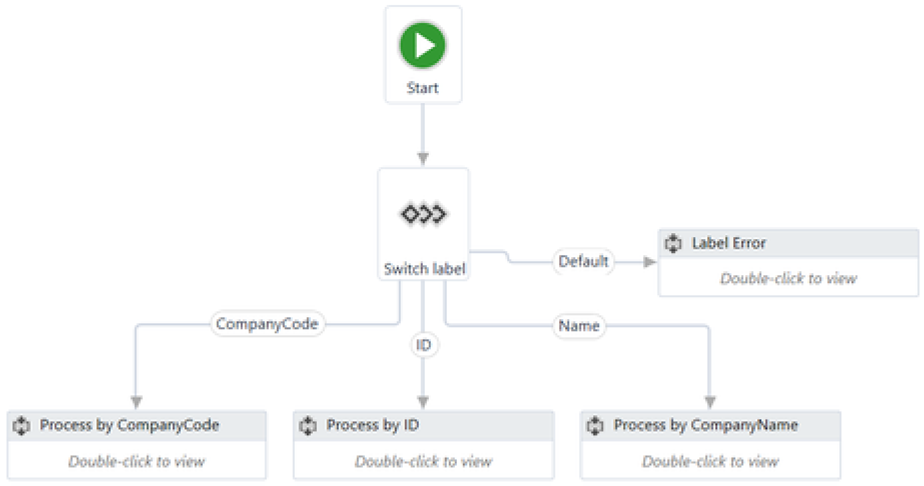
Case Ms — Click image "Ms" radio

Case Mrs — Click image "Mrs" radio

Add new case

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 matchmore 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.

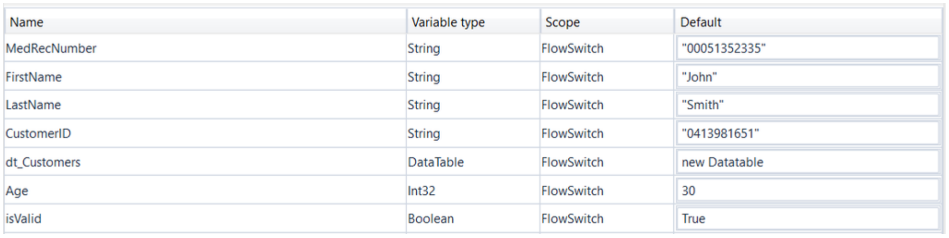
Variables should be kept in the innermost scope to reduce the clutter in the **Variables** panel and to show only, in autocomplete, what is relevant at a particular point in the workflow. Also, if two variables with the same name exist, the one defined in the most inner scope has priority.

Arguments

Keep in mind that when invoking workflows with the **Isolated** option (which starts running the workflow in a separate system process), 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.



Name	Variable type	Scope	Default
MedRecNumber	String	FlowSwitch	"00051352335"
FirstName	String	FlowSwitch	"John"
LastName	String	FlowSwitch	"Smith"
CustomerID	String	FlowSwitch	"0413981651"
dt_Customers	DataTable	FlowSwitch	new Datatable
Age	Int32	FlowSwitch	30
isValid	Boolean	FlowSwitch	True

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.

Workflow Abstraction Layers

Workflow templates

Workblocks

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.

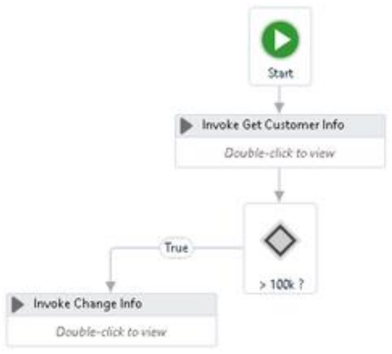
However, separation of **business logic** from the **automation components** is good principle that will help with building a code that can be reused effectively.

Example

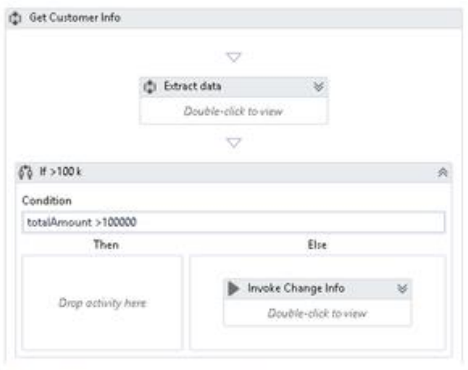
Let’s assume that a part of your process requires reading the customer info, then – based on that info and internal business rules - update the customer details.

"*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



Condition
totalAmount >100000

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.

Where to store reusable components

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. But make no confusion – once the code is added to the workflow, this will be static - if you update the workflow in the Library, it won't be reflected in the existing live processes.

Common (reusable) **components** (e.g. App Navigation, Log In, Initialization) 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.

An example of reusable content implementation - on release chapter

How to Code Review

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

UIAutomation

Sometimes the usual manual routine is not the optimal way for automation. Carefully explore the application’s behavior and UiPath’s integration/features before committing to a certain approach.

Relating to the UI

Context vs Structure

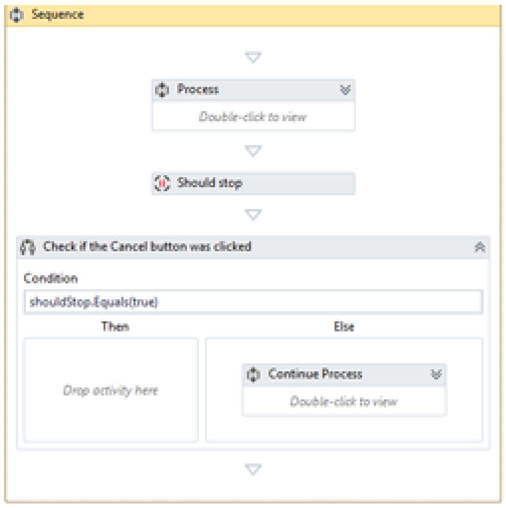
Desktop Automation

UI automation goes at its best when Robots and applications run on the same machine because UiPath can integrate directly with the technology behind the application to identify elements, trigger events and get data behind the scenes.

Input Methods

There are three methods UiPath uses for triggering a **Click** or a **Type Into** an application. These are displayed as properties in all activities that deal with UI automation.

- If **SimulateType** or **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 internallyconfluence
- 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 the order presented, as **Simulate** and **WindowMessages** are faster and also work in the background, but they depend mostly on the technology behind the application.

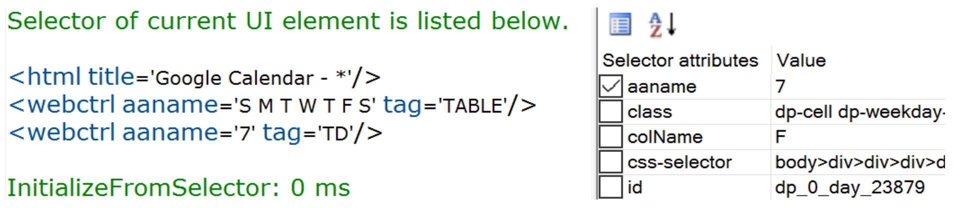
Hardware events work 100% as Studio performs actions just like a human operator (e.g. moving the mouse pointer and clicking at a particular location), but in this case, the application being automated needs to be visible on the screen. This can be seen as a drawback, since there is the risk that the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

Selectors

Sometimes the automatically generated selectors propose 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 (e.g., 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.

Here are some tips of 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.
- Avoid using *idx* attribute unless it is a very small number like 1 or 2


Selector of current UI element is listed below.

```
<html title='Google Calendar - *'/>
<webctrl aaname='S M T W T F S' tag='TABLE'/>
<webctrl aaname='7' tag='TD'/>
```

InitializeFromSelector: 0 ms

Selector attributes	Value
✓ aaname	7
class	dp-cell dp-weekday-
colName	F
css-selector	body>div>div>div>d
id	dp_0_day_23879

In the selector above, we notice the page title has a reference to the time when selector was recorded and also that some attributes have randomly looking IDs. Tweaking the attributes, we can come up with a better selector than UiPath recorder proposed.

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 May 28, 2018'*, 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*.

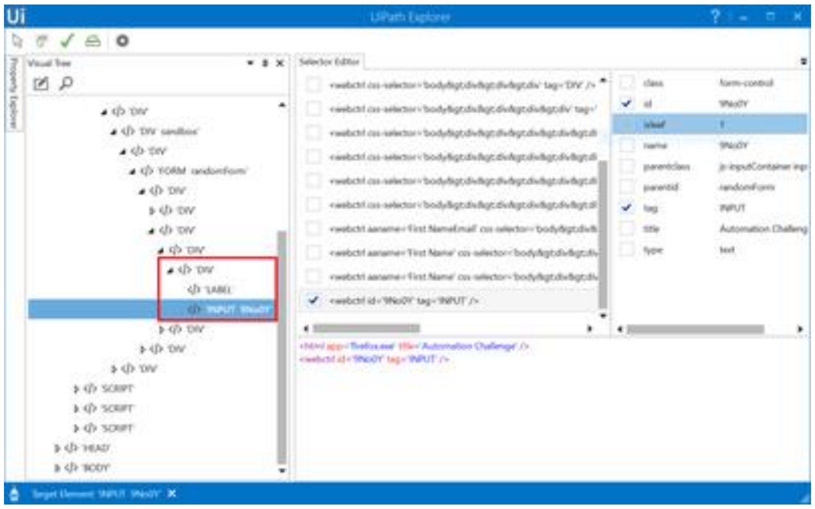
The excessive use of wildcards can make the selector too generic and match more than one element, so in some situations it is necessary to combine it with other attributes or other techniques to define selectors.

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.

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. We 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 *<br/>*) between the label and the tex tbox, so *<nav next='1'>* would not find the text box.

Dynamic elements

Finding UiElements

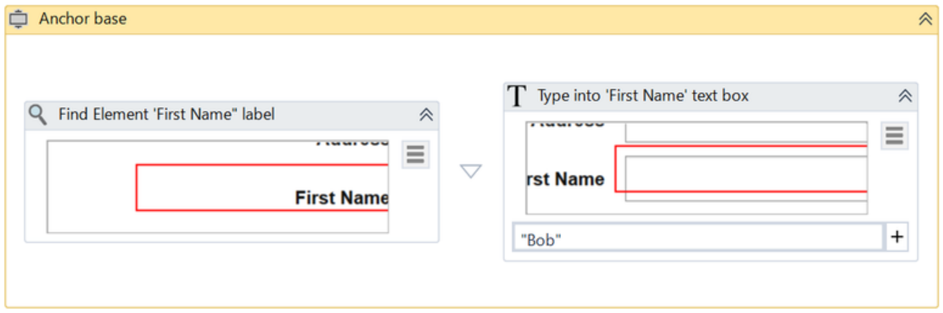
If it is not possible to define a reliable selector for an element, there are still other options that can retrieve it. They mainly work based on a relative element (for which a selector can be defined), but they still provide 100% accuracy as long as the relative element is the same.

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

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). It is important to understand how the Anchor Base activity works though: It will set a starting point on the screen (the center of the selected element to be the anchor) and will look 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.

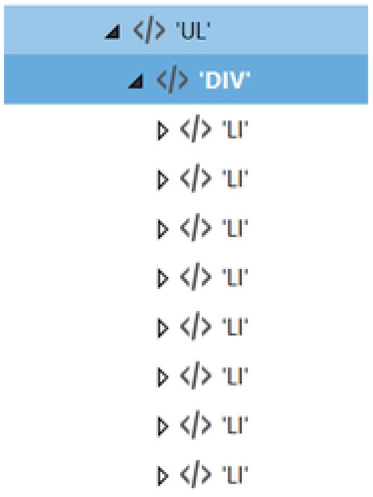

Anchor base
Find Element 'First Name" label
First Name
Type into 'First Name' text box
rst Name
"Bob"

Find Children Activity

In certain cases, it is necessary to work with a collection of elements that are relative to a particular one. The Find Children activity complements the feature offered by the *<nav>* tag and offers a way to access the children of an element.

For example, tabular data that cannot be extracted used using the Data 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 dropdown 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

Similar to file paths, selectors can be full or partial (relative). 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

Essential when working on two instances of the same application

Element scope

Image Automation

Image recognition is the last approach to automating applications if nothing else works to identify UI elements on the screen (like selectors or keyboard shortcuts). 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 like desktop theme or screen resolution. When the application runs in Citrix, the resolution should be kept greater or equal than when recording the workflows. 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 recordings can be placed inside a **PickBranch** activity and Robot will use either match.

Image Accuracy

OCR Engines

If OCR returns good results for the application, text automation is a good alternative to minimize environment influence. Google Tesseract engine works better for smaller areas and Microsoft MODI for larger ones.

Using the MODI engine in loop automations can sometimes create memory leaks. This is why it is recommended that scraping done with MODI be invoked via a separate workflow, using the **Isolated** property.

Image vs OCR

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 **DelayMS** 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)

OnElementAppear

WaitElementVanish

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

Automation Lifecycle

Process Understanding

Deciding between an automation for attended robots or back office unattended robots 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) a Robot working side by side with a human (so Attended) might be the only possible answer.

But 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, yet both sub-processes could be triggered automatically and run unattended.

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.

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. 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.

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.

Unit Test

Functional Test

Testing Framework

Release

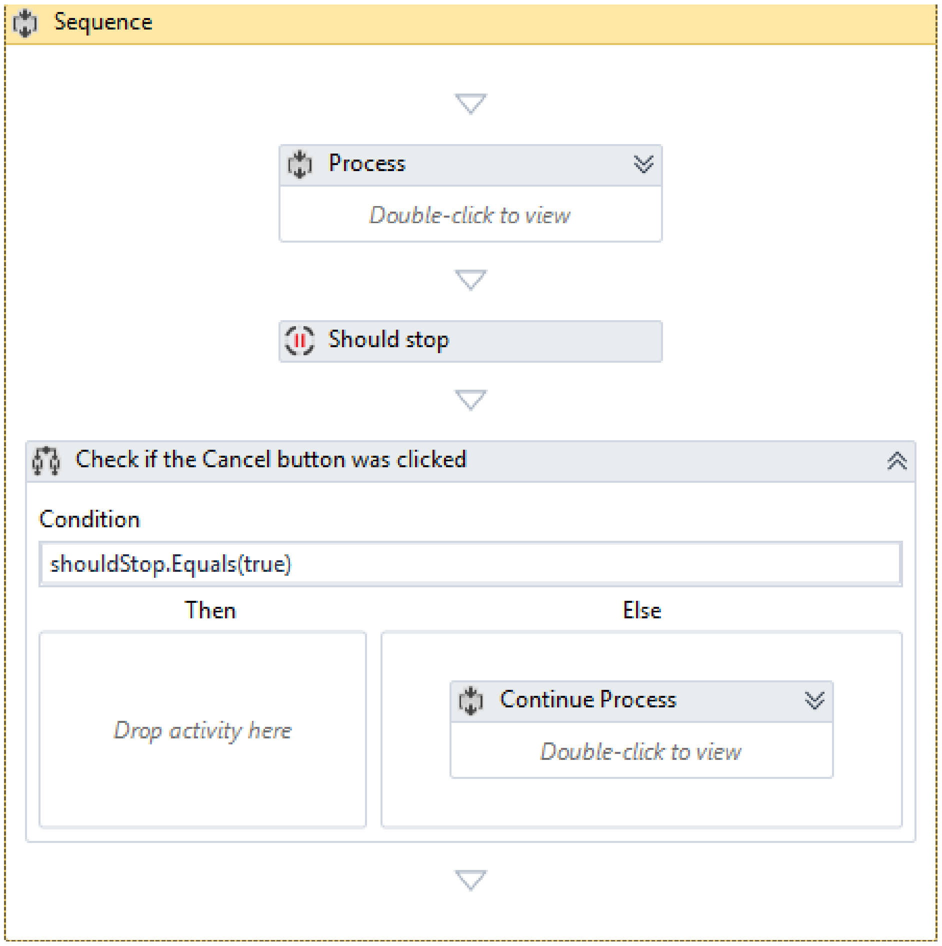
There are various ways of designing the architecture and release flow –considering the infrastructure setup, concerns about the segregation of roles etc.

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 VCS - 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 (eg IT).

The deployment paths on Orchestrator have been changed from default to folders managed by the VCS (by changing packagesPath value in web.config file under UiPath.Server.Deployment)

The model also contains a repository of reusable components.

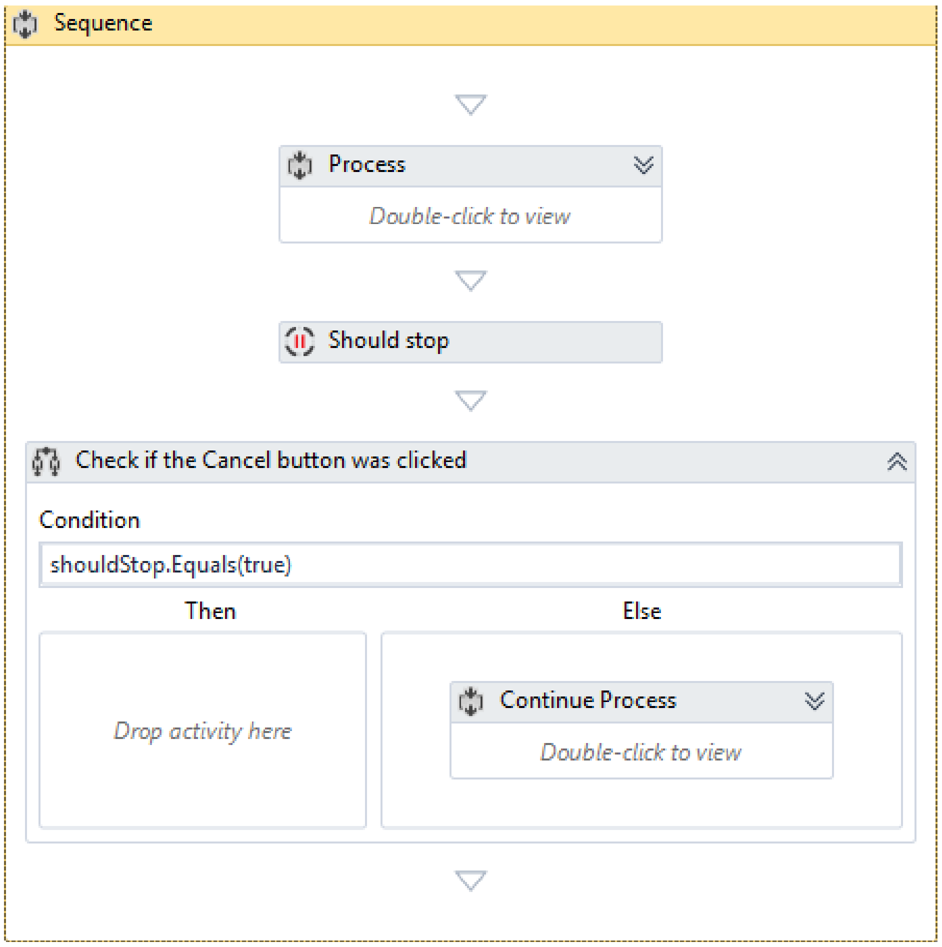


Here is the project publishing flow, step by step:

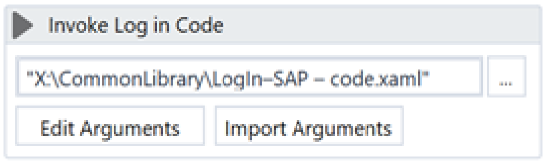
- Developers build the process in UiPath Studio and test it with the Development Orchestrator; Once done, they check in the workflows (not packaged) to a ***Master UiProcess*** Library folder (on VCS);
- The IT team will create the package for QA. This will be stored on a ***QA Package*** folder on VCS 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 (***P Package***)
- Process is going live, run by the production robots.

Reusable content is created and deployed separately – as UiPath code (Reusable Code Library) and Invokes (Invokes Repository).

So we distinguish here between the actual ***workflows with source code*** (.xaml files containing UiPath activities for automating a common process – eg Log in SAP) and ***invokes*** (workflows composed of only one UiPath invoke activity of the code workflows mentioned above).

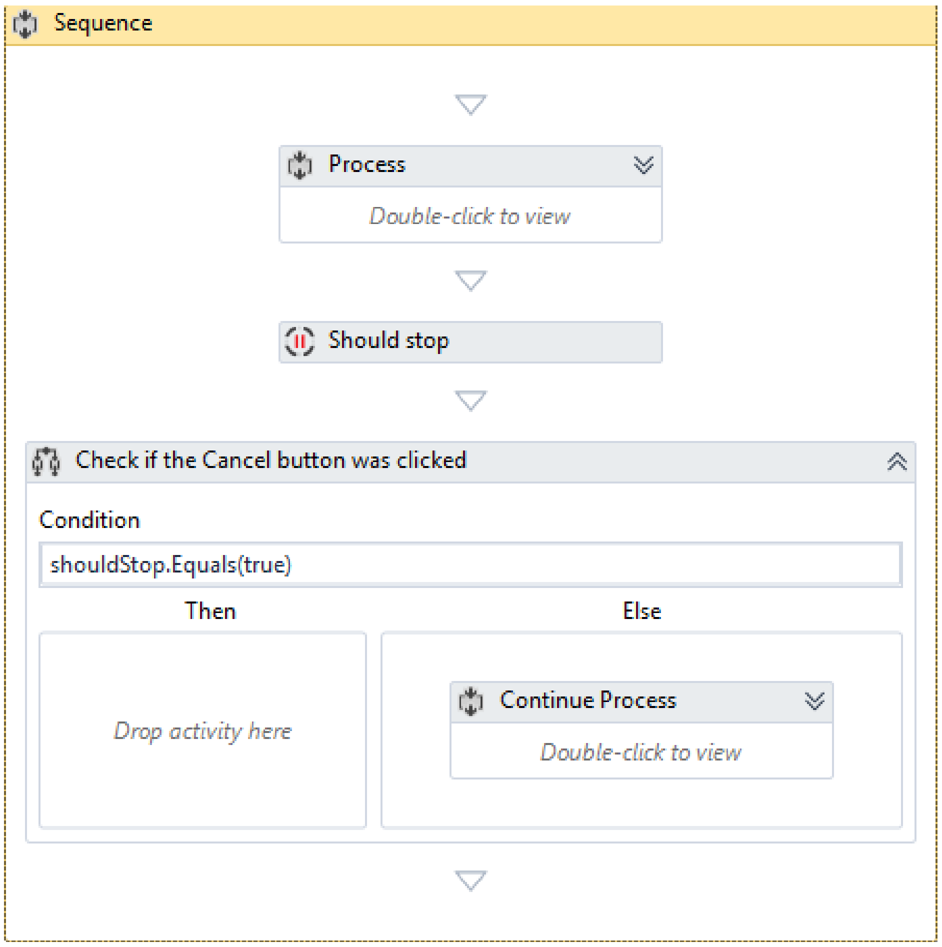


shouldStop.Equals(true)



"X:\CommonLibrary\LogIn-SAP – code.xaml"

The Library of developer Studio should point to this Invoke repository in order to provide easy access (drag & drop) to reusable content.



The software architect in charge with maintaining the reusable content will update (due to a change in process, for instance) the workflows with code. The invokes will remain unchanged.

The advantage of this approach (as opposed to work directly with the library of source code): when a change is done to a reusable component, all the running projects will reflect this change as well – as they only contain an invoke of the changed workflow.

Logging

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

Logging should be used:

- at the beginning and the end of every workflow - mai degraba la fiecare workblock - notiunea de tracing la nivel de business process.
- when data is coming in from external sources

- each time an exception is caught at the highest level

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

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.

Monitoring

Orchestrator

Multi tenant

UiPath Orchestrator offers a multi-tenant option. Using more than one tenant, users can split a single instance of Orchestrator to multiple environments, each one having their robots, processes, logs and so on.

This can be very useful when separated artifacts for different departments or different instances for clients are needed.

Robots

Use meaningful names and descriptions for each Robot provisioned.

For Back Office Robots, the Windows credentials are needed in order to run unattended jobs on these types of Robots. For Front Office Robots, credentials are not needed because the job will be triggered manually by a human agent, directly on the machine where the Robot is installed.

Every time a new Robot is provisioned, the type of the Robot should be chosen accordingly.

The next step after registering the Robot to Orchestrator is done is to check if its status is Available, in the **Robots** page.

Environments

Use meaningful names and descriptions for each environment created.

Orchestrator Environments should map the groups of process execution. Each environment should have a specific role in the company business logic.

If a Robot is going to execute two different roles, it can be assigned to multiple environments.

The access management of the Robots to the processes is done by using the Environments properly.

Processes

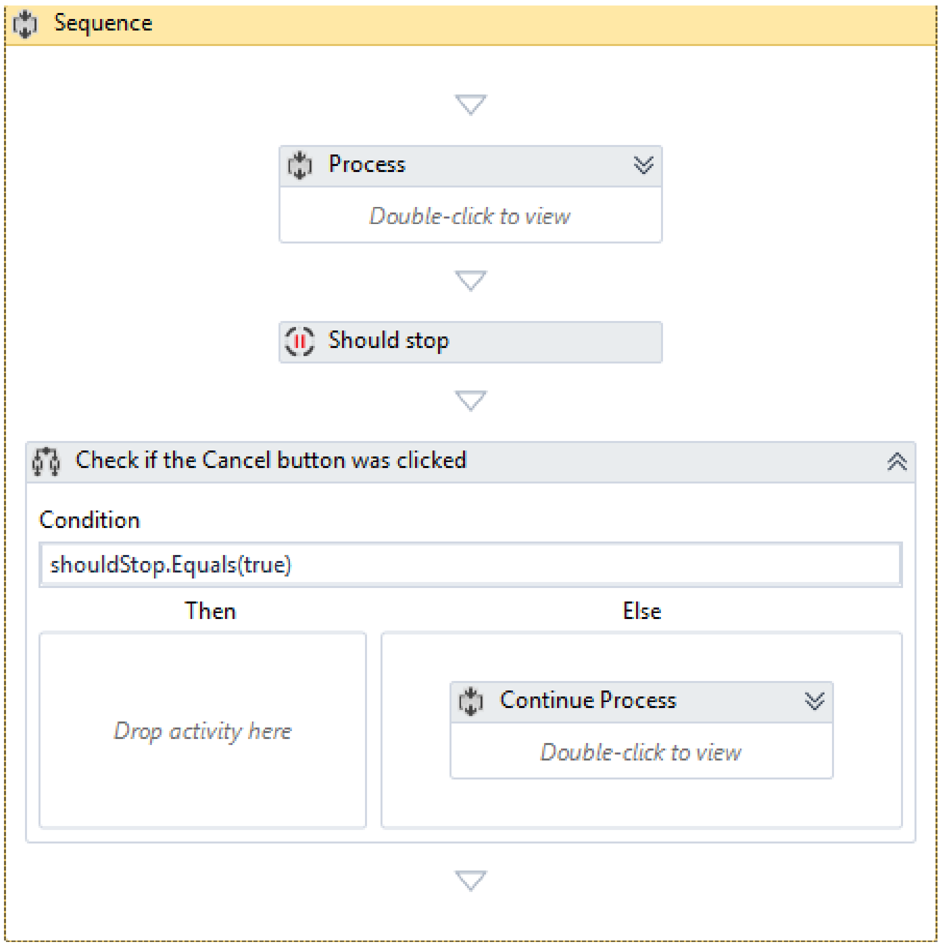
Once in a while, old versions of processes that are not used anymore should be deleted. Versions can be deleted one-by-one, by selecting them manually and clicking the **Delete** button or the **Delete Inactive**button, that deletes all the process versions that are not used by any Release.

Note: It’s recommended to keep at least one old version to be able to rollback if something is wrong with the latest process version.

Assigning Processes to Environments

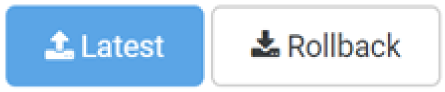
It is good practice to assign each process published to Orchestrator to an environment. In the **Processes** page, the deployment decision is taken. All the Robots from the environment will get access to the process version set for this Release.

When a new version of a process is available, an icon will inform the user.



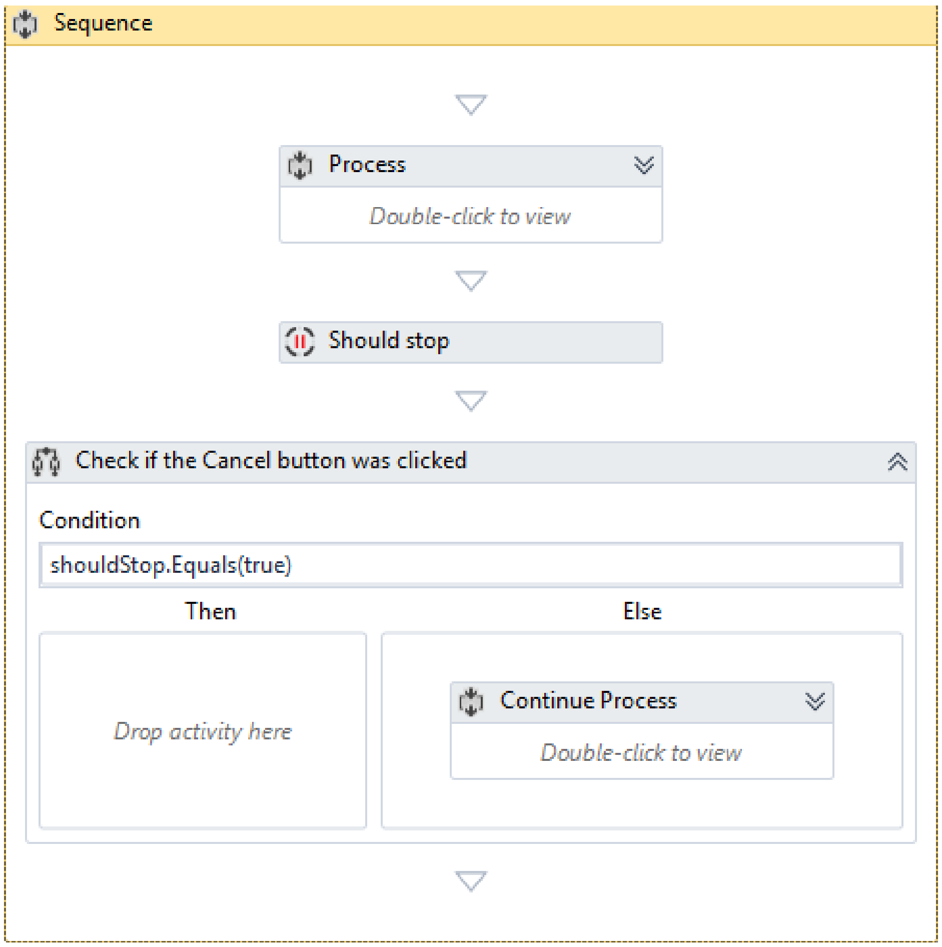
```
shouldStop.Equals(true)
```

Rolling back to the previous version is always an option if something goes wrong after updating. This can be done by pressing the **Rollback** button.



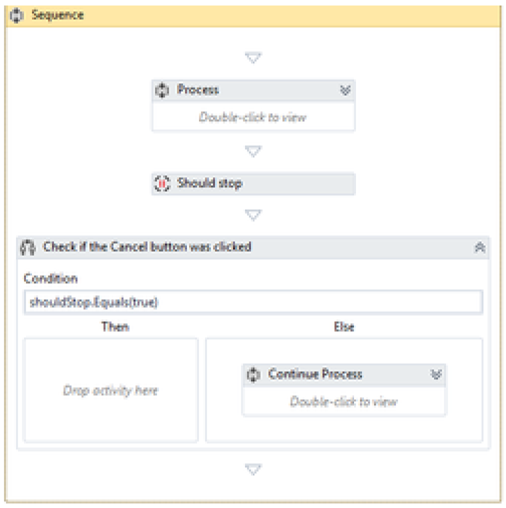
Jobs

If the robot should run multiple processes with no interruption, all the jobs should be triggered one after another even if the robot is busy. These jobs will go in a queue, with the Pending status, and when the Robot is available again, Orchestrator triggers the next job.



It’s better to cancel a job than to terminate it.

To be able to Cancel a job, the **Should Stop** activity is needed in the process workflow. This activity returns a boolean result that indicates if the Cancel button was clicked.



The **Terminate** button sends a Kill command to the Robot. This should be used only when needed, because the Robot might be right in the middle of an action.

Schedules

Besides the obvious functionality, schedules can be used to make a robot run 24/7. Jobs can be scheduled one after another (at least one minute distance) and if the Robot is not available when the process should start, it's going to be added to the jobs queue.

Cron expressions

Queues

Use meaningful name and description for each queue created.

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.

Using the **Set Transaction Status** activity, a queue item status can be set to **Successful** or **Failed**. Keep in mind that only the Failed items with Application ErrorType are going to be retried.

If there are two or more types of items that should be processed by the same Robots, there are at least two option of how these can be managed by the Queues.

1. Create multiple queues, one for each type and create a process that checks all the queues in a sequence and the one with new items should trigger the specific process.
2. Create a single queue for all the items and for each item, create an argument "Type" or "Process". By knowing this parameter, the robot should decide what process should be invoked.

Transactions

Transaction Item activity brings the option of getting all the Transactions functionalities without using a queue properly (a queue should still be created before). This activity adds an item to the queue and sets its status to InProgress. Start using the item right away and don't forget to use the **Set Transaction Status** activity at the end of your process.

Logs

The **Add Log Fields** activity adds more arguments to Robot logs for a better management. After using it in the workflow, **the Log Message** activity will also log the previously added fields.