UiPath

Robotic Process Automation
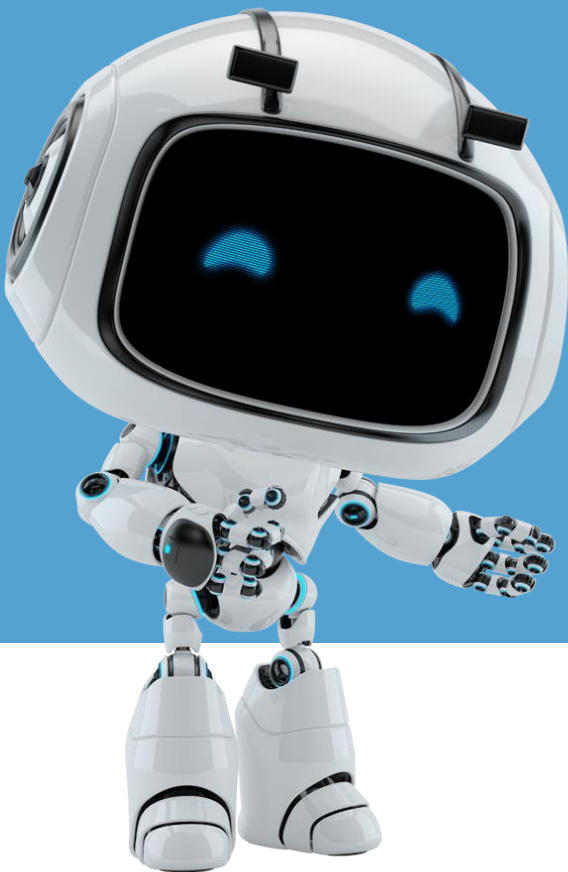
Best Practice Guide

Revision History

Date	Version	Author	Description
13th January 2017	0.1	A.C., T.H., A.S., M.B.	Created document
19th January 2017	0.2	A.S.	Added architecture & release flow
30th January 2017	0.3	A.S.	Changing doc structure (PO & AL)
24th February 2017	0.4	A.S.	Code reusability, standards
25th January 2018	0.5	A.C.	Added DSD

Table of Contents

This document outlines the standards and best practices to be considered by RPA developers when building automation processes with UiPath.

1. Workflow Design

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.

Choices

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.
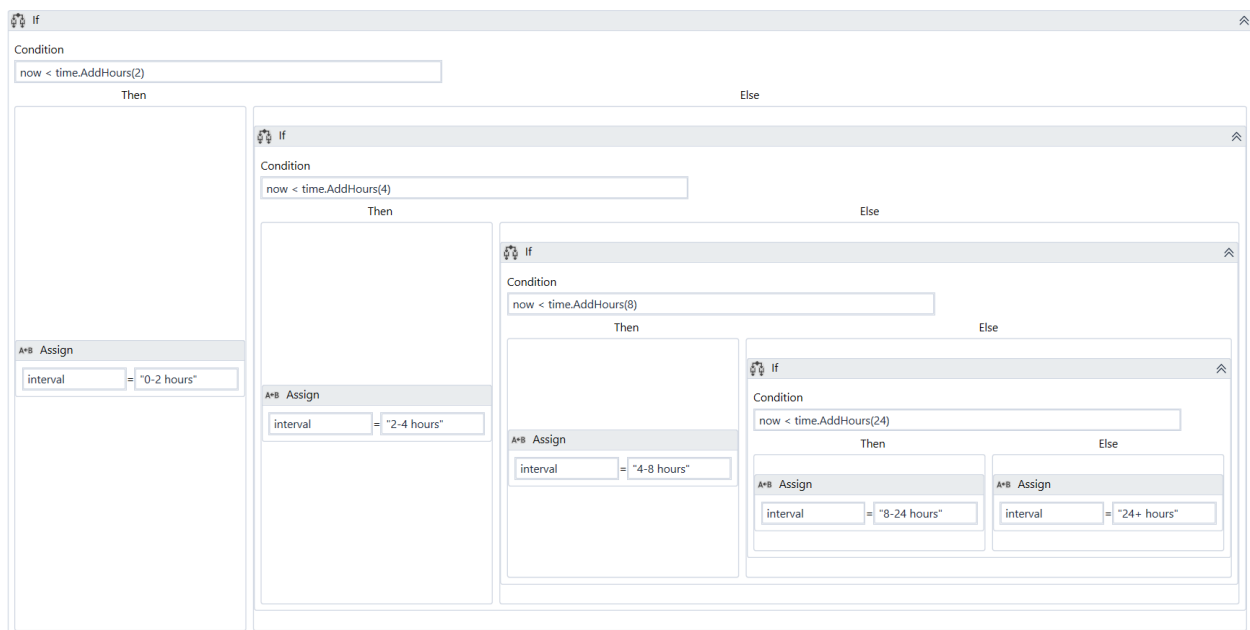
Flow Decision

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.
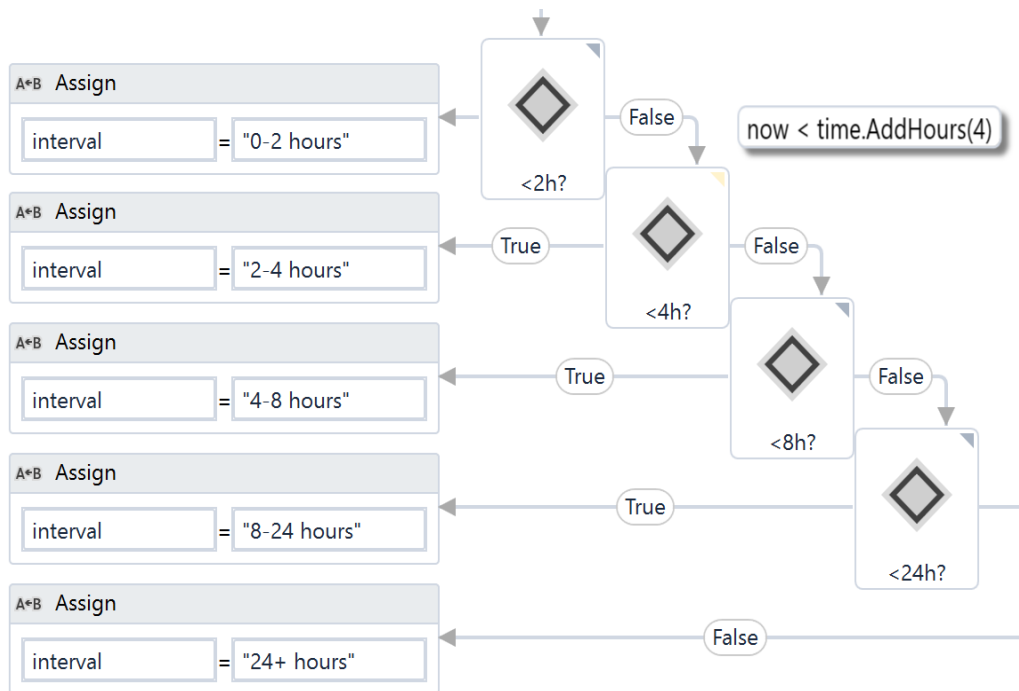
If Operator

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.
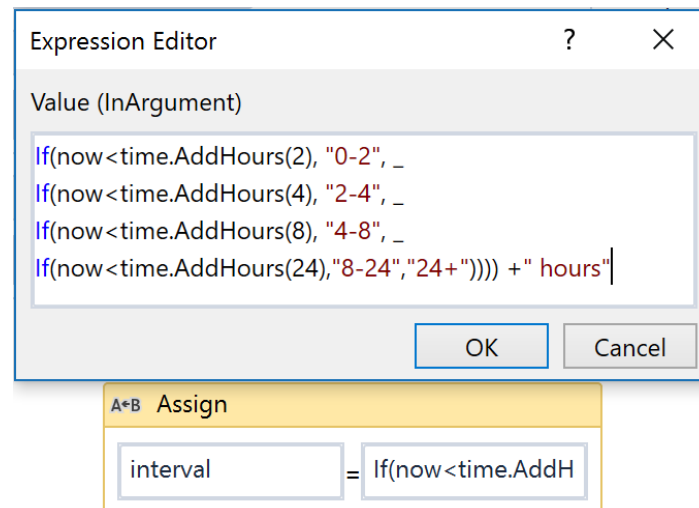
Switch Activity

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

Flow Switch

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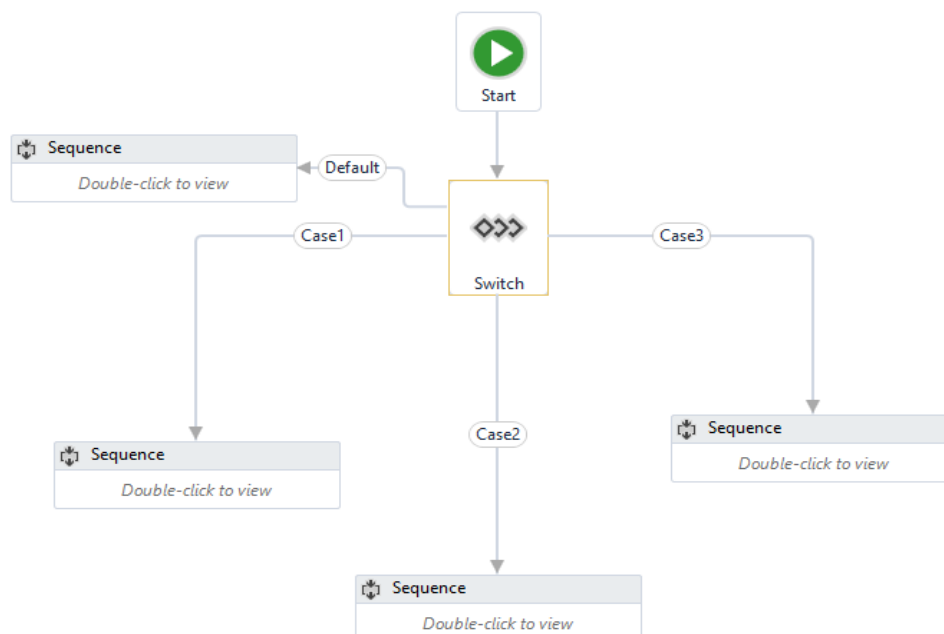
Data

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
medRecNum	String	Process Transaction	"000352686"
physician	String	Process Transaction	"BAILEY, JORDAN"
lastName	String	Process Transaction	"TEST"
firstName	String	Process Transaction	"GEORGE"

Naming Conventions

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 and might help in multi-user environments.

Only argument names are case sensitive, but to improve readability, variables also should align to a 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*

Comments and Annotations

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.
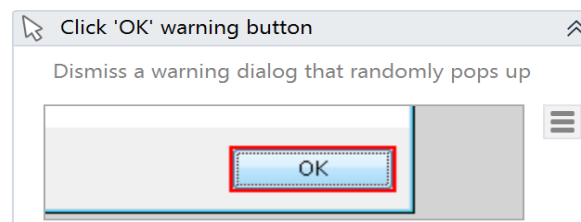
2. UI Automation

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.

2.1. 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.

2.1.1. 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 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

| SendWindowMessages | ☐ |
| SimulateClick | ☑ |

| SendWindowMessages | ☑ |
| SimulateType | ☐ |
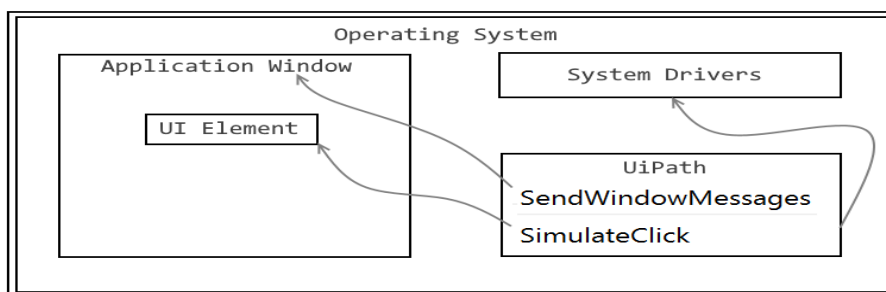
These methods should be tried in the order presented, as **Simulate** and **WindowMessages** are faster and also work in the background, but they depend mostly on the technology behind the application.

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.

2.1.2. 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, in dev, test and production environments and no matter the usernames logged on to the applications.

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
- If editing attributes doesn't help, try adding more intermediary containers
- Avoid using **idx** attribute unless it is a very small number like 1 or 2

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.

2.1.3. 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

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.

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.

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)

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

3. Project Organization

High-Level 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.
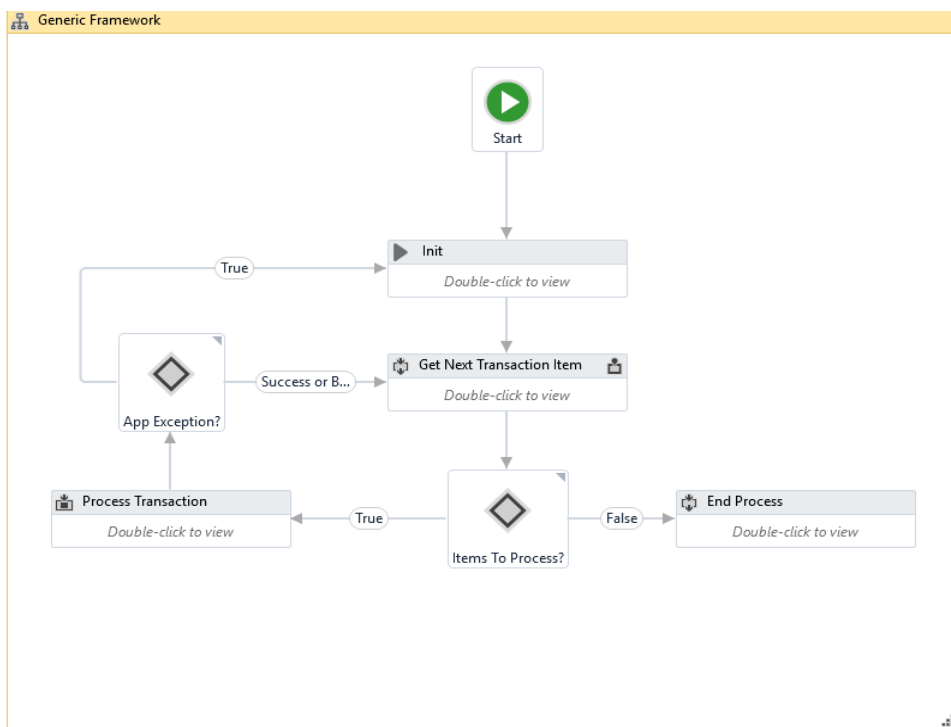
Robotic Enterprise FrameworkTemplate proposes a flexible high level overview of a repetitive process and includes a good set of practices described in this guide and can easily be used as a solid starting point for RPA development with UiPath. The template is built on a State Machine structure.
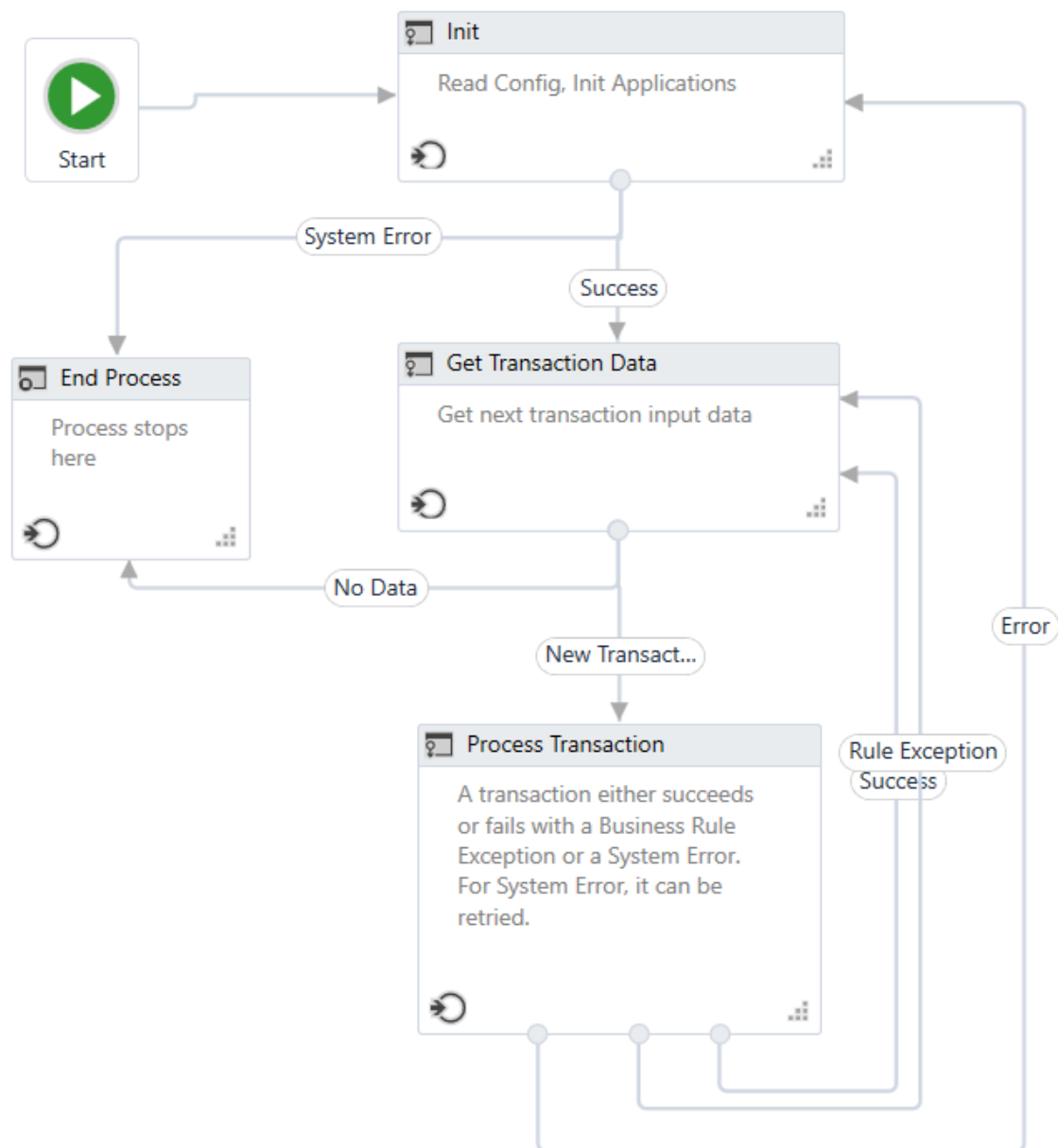
How it works:

- 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
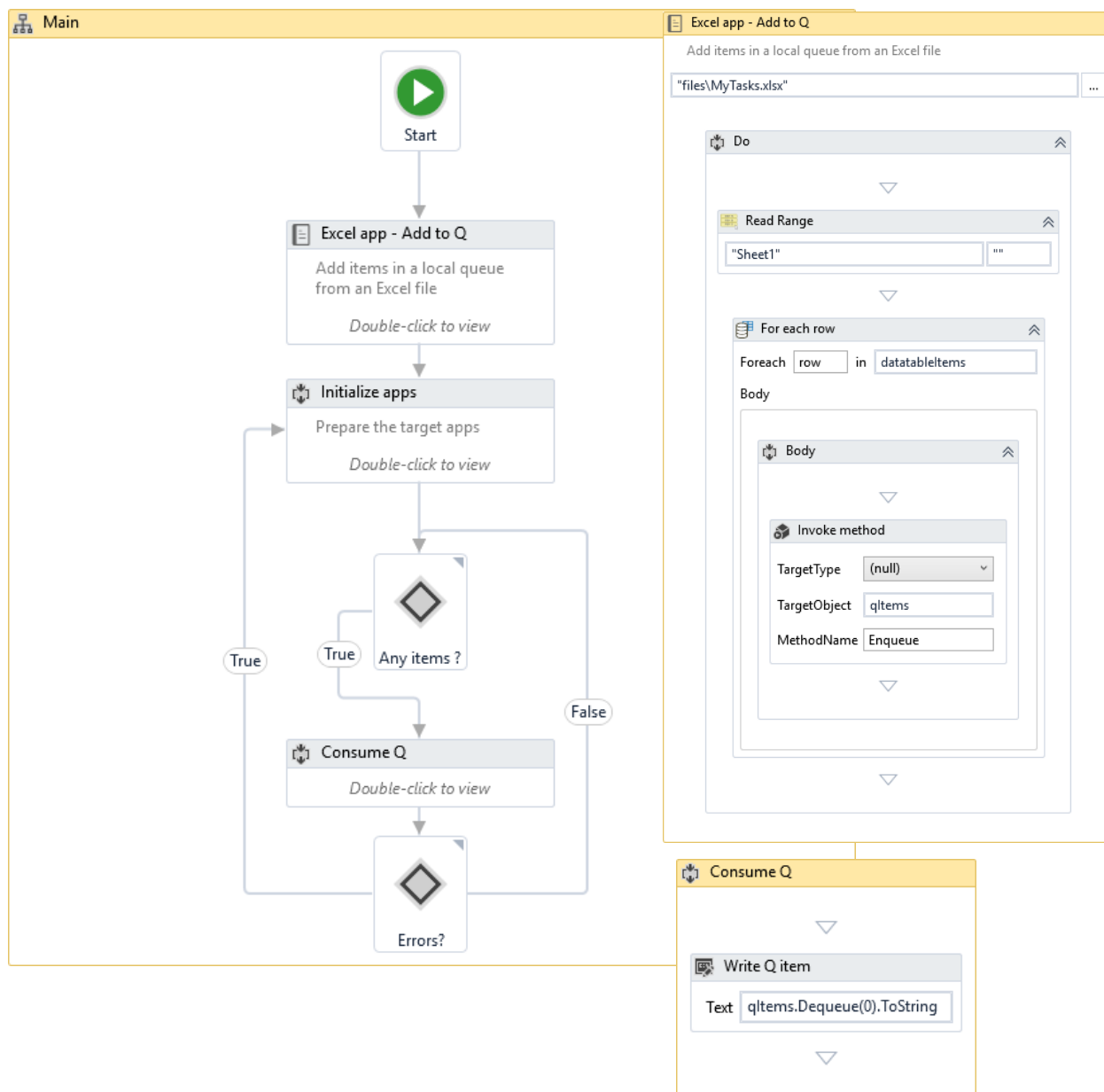
Design principles

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

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

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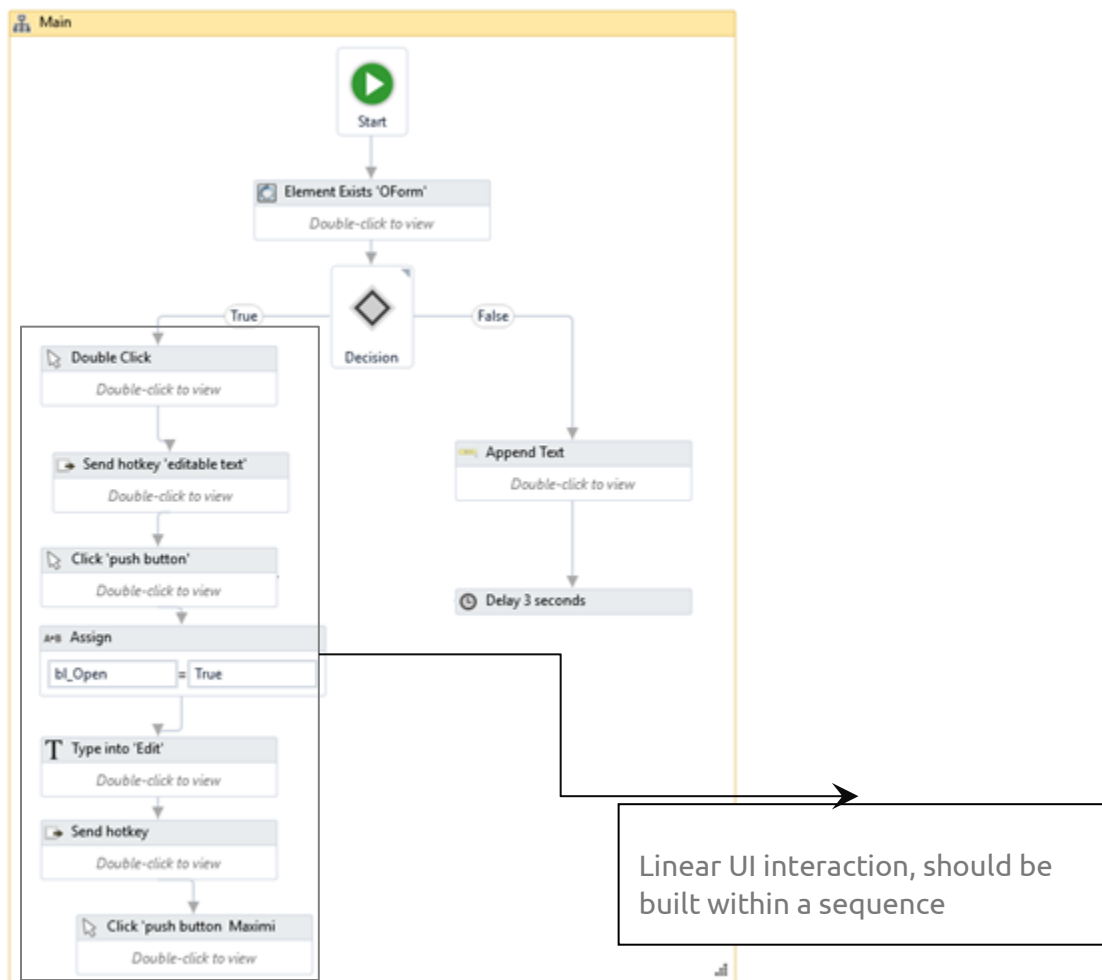
Project related files (e.g. email templates) could be organized in **local folders**or **shared drives.**

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.

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

Context Settings

To avoid hard coding external settings (like file paths, URLs) in the workflows, we recommend keeping them in a **config file** (.xlsx or .xml or .json) or in Orchestrator **assets** if they change often.

Generally speaking, the final solution should be extensible - allow variation and changes in input data without developer intervention. For example - lists with customers that are allowed for a certain type of transaction, emails of people to receive notifications etc. - should be stored in external files (like Excel) where business people or other departments can alter directly (add/remove/update).

For any repetitive process, all workflow invocations from the main loop should be marked with the Isolated option to defend against potential Robot crashes (e.g. Out of memory).

Credentials

No credentials should be stored in the workflow directly, but rather they should be loaded from safer places like local Windows Credential Store or Orchestrator assets. You can use them in workflows via the **GetCredential** activities.

Integration with some 3rd party password management solutions (e.gCyberArk) will be made possible in the next release of UiPath (2017.1)

Error Handling

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 human operators.

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.

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. 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**.

Enough details should be provided in the exception message for a human to understand it and take the necessary actions. Exception **messages** and **sources** are essential. The source property of the 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.

Invoke activities not renamed, in case of a crash the exception source will be meaningless (like *Invoke workflow file: Invoke workflow file: Invoke workflow file: Type Into*)
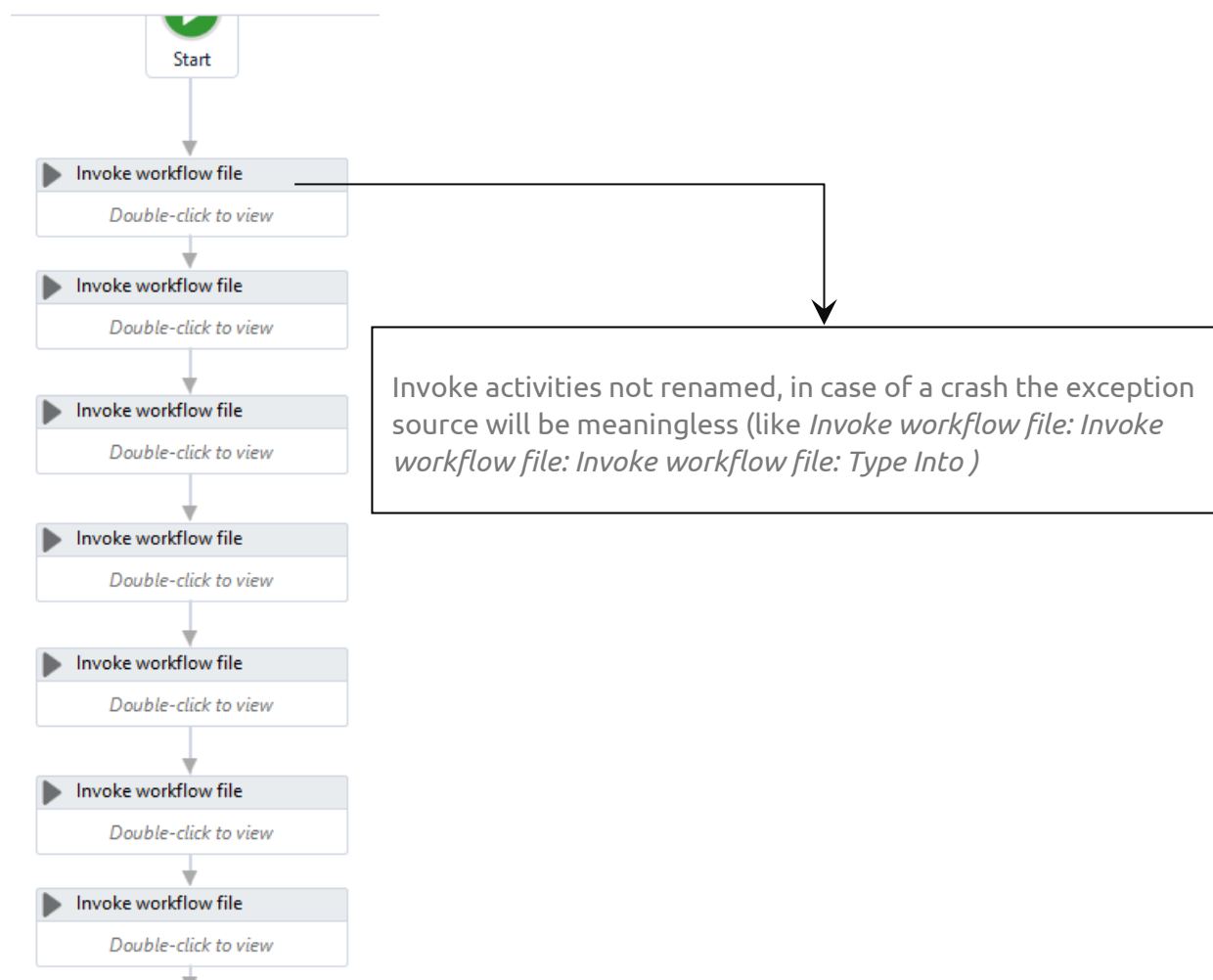
Keep it clean

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.

Code 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.
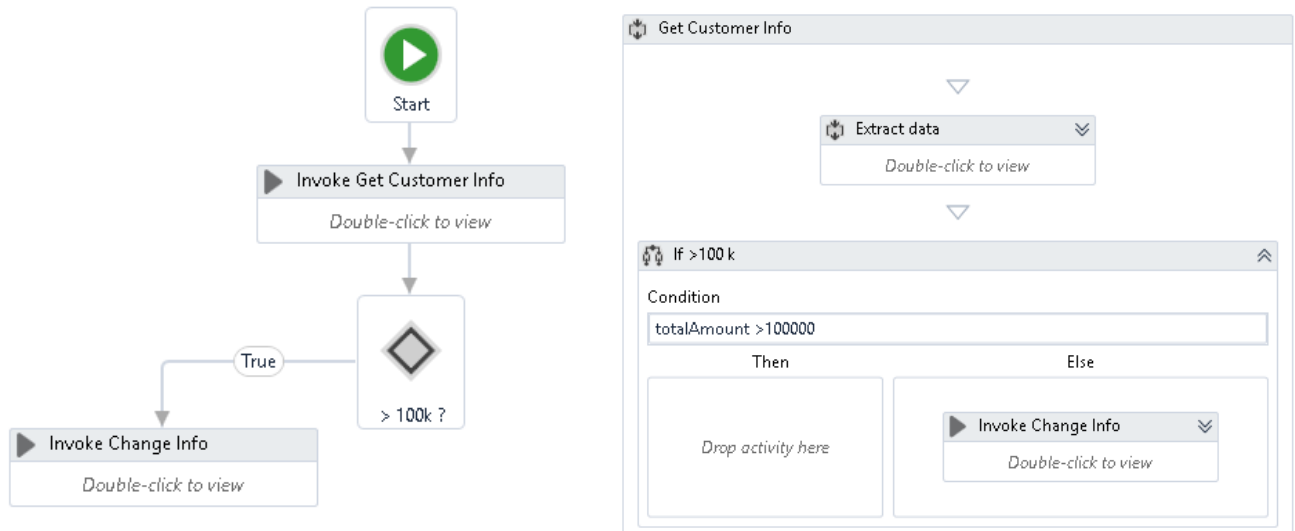
Separate components for *Get Info* and *Change Info* - OK   *Change Info* inside *Get Info* – Not OK

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 check quality automation

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

4. 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 & 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.

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.
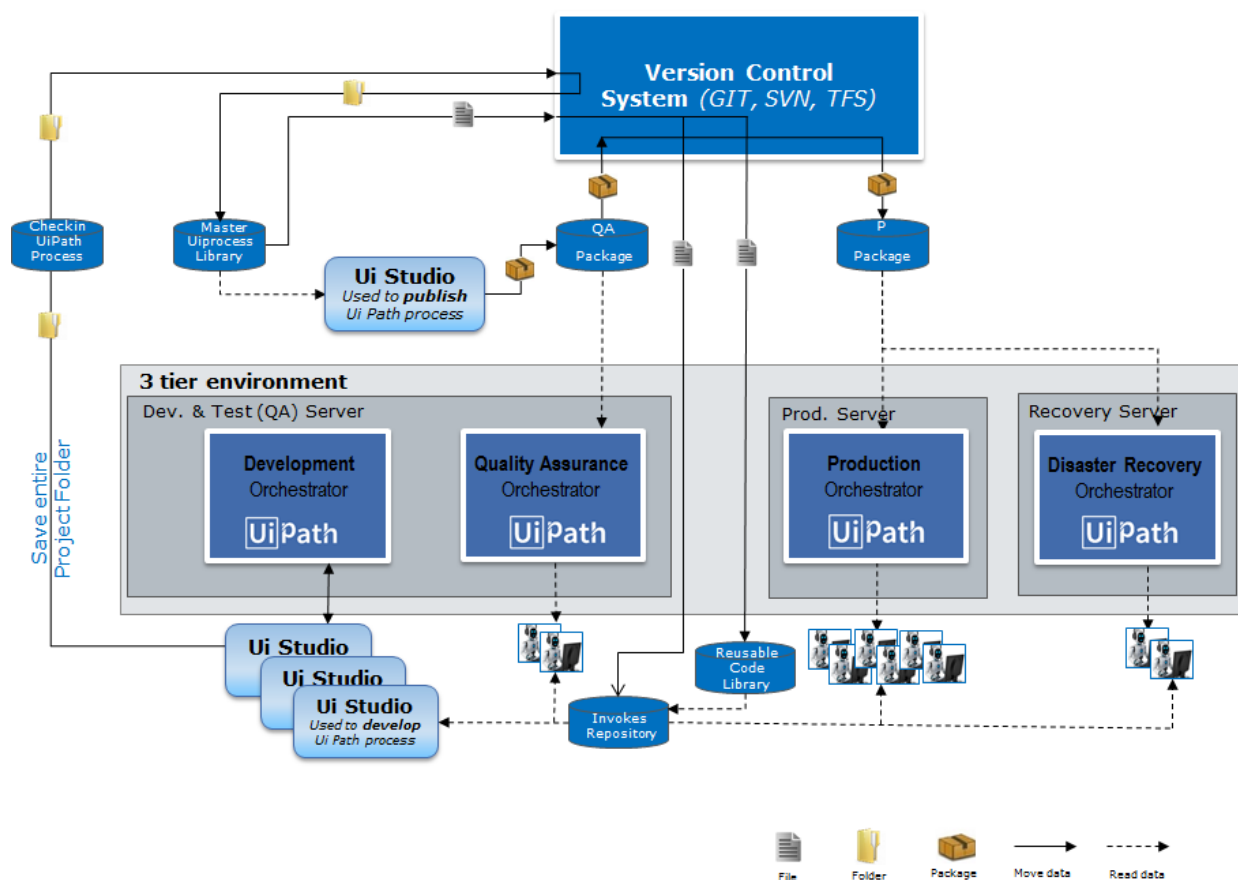
Here is the project publishing flow, step by step:

- 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).
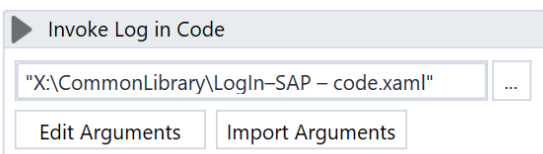
So we distinguish here between the actual *workflows with source code* (.xaml files containing UiPath activities for automating a common process – eg Log in SAP)
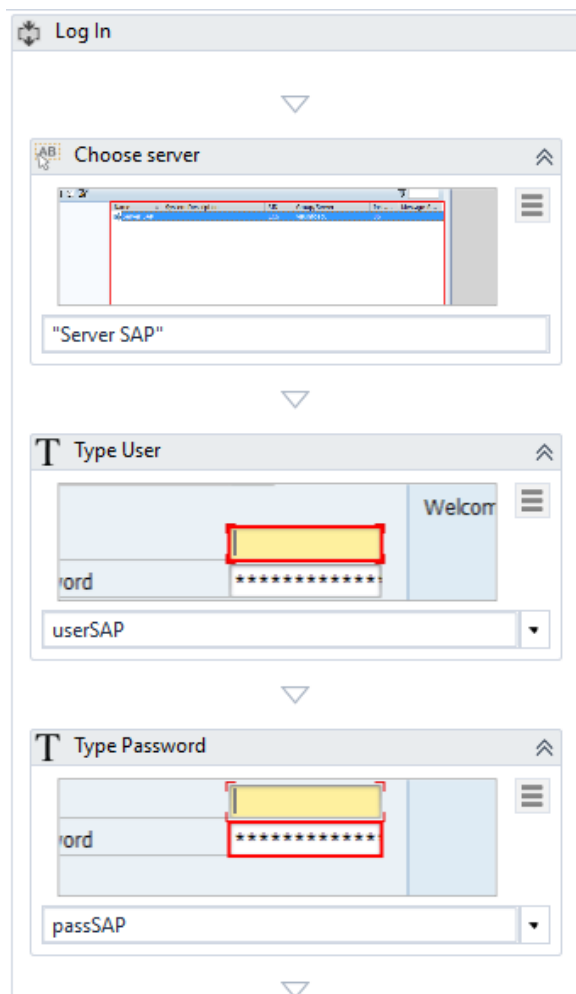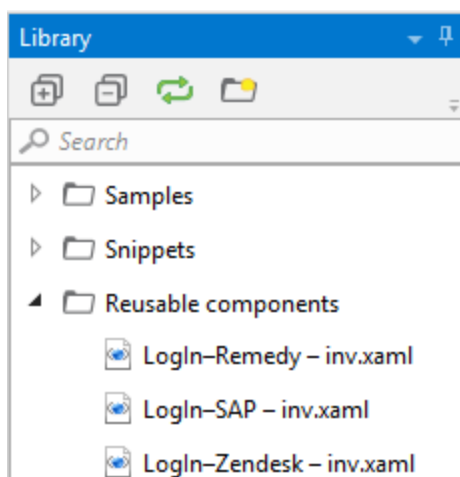

Ex: LogIn–SAP – code.xaml

and*invokes* (workflows composed of only one UiPath invoke activity of the code workflows mentioned above).

Ex: LogIn–SAP – inv.xaml

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

The local design authority 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.

Monitoring

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
- 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.
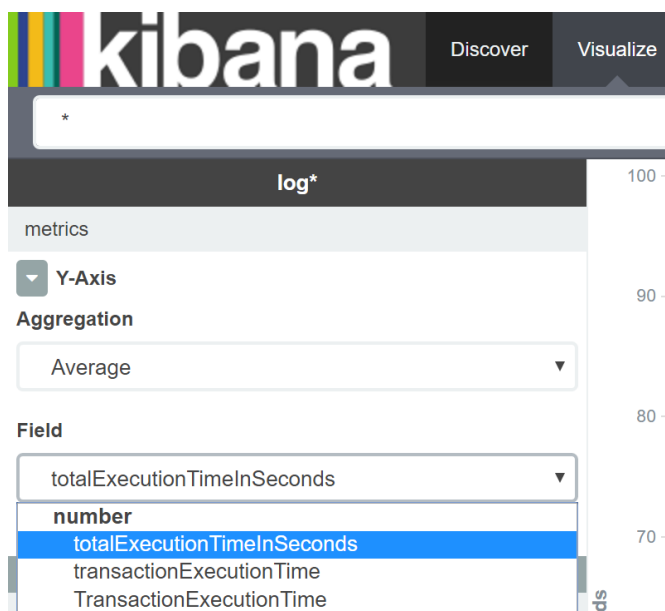
5. Orchestrator

Multi-tenant – Experimental

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.

	⚠	CristianProcess	1.0.6194.28074

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.

⬆ Latest   ⬇ Rollback

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.

		Process	Robot	State	▼
☐	•••	UiPath2	Teo-Robot	Pending	
☐	▶	UiPath1	Teo-Robot	Running	

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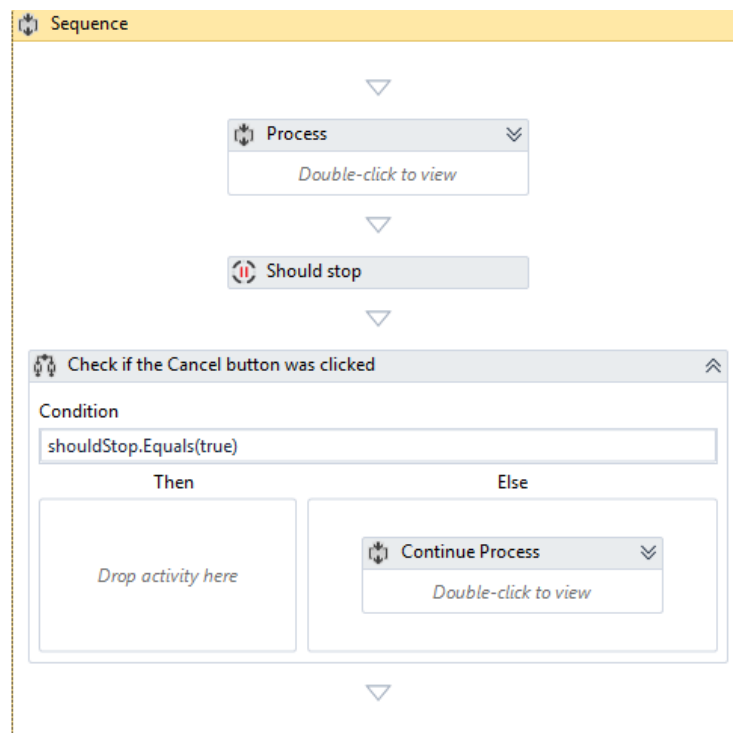
The **Terminate** button sends a Kill command to the Robot. This should be used only when needed, because the Robot might be right in the middle of an action.

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.

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

The **Add 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.