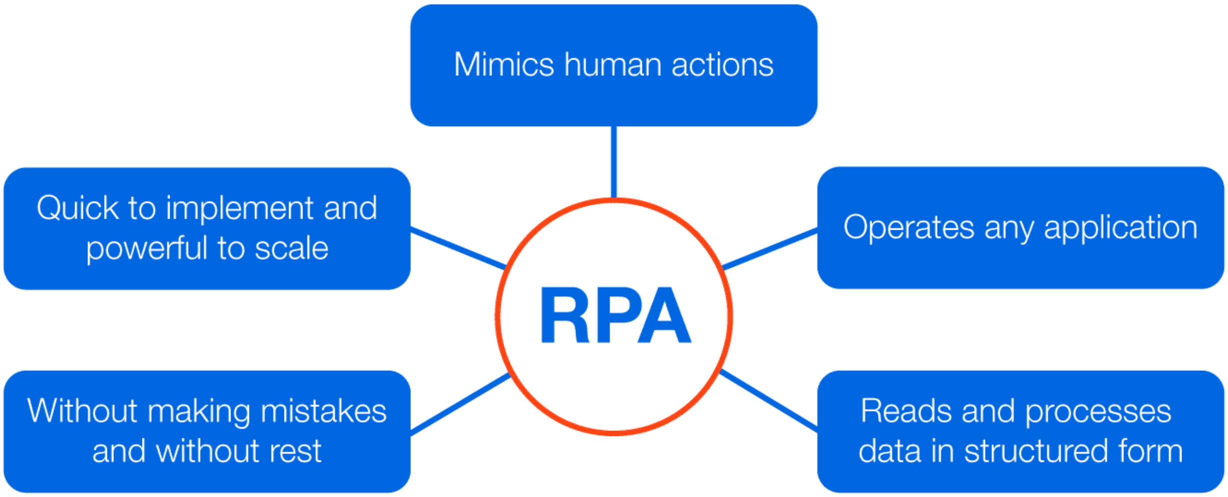
THE RPA JOURNEY

Robotic Process Automation, is the technology that enables computer software to emulate and integrate actions typically performed by humans interacting with digital systems.

Automating business processes using robots is known as Robotic Process Automation

RPA robots are able to capture data, run applications, trigger responses, take decisions based on predefined rules and communicate with other systems. RPA primarily targets processes which are highly manual, repetitive, rule-based, with low exceptions rate and standard electronic readable input.

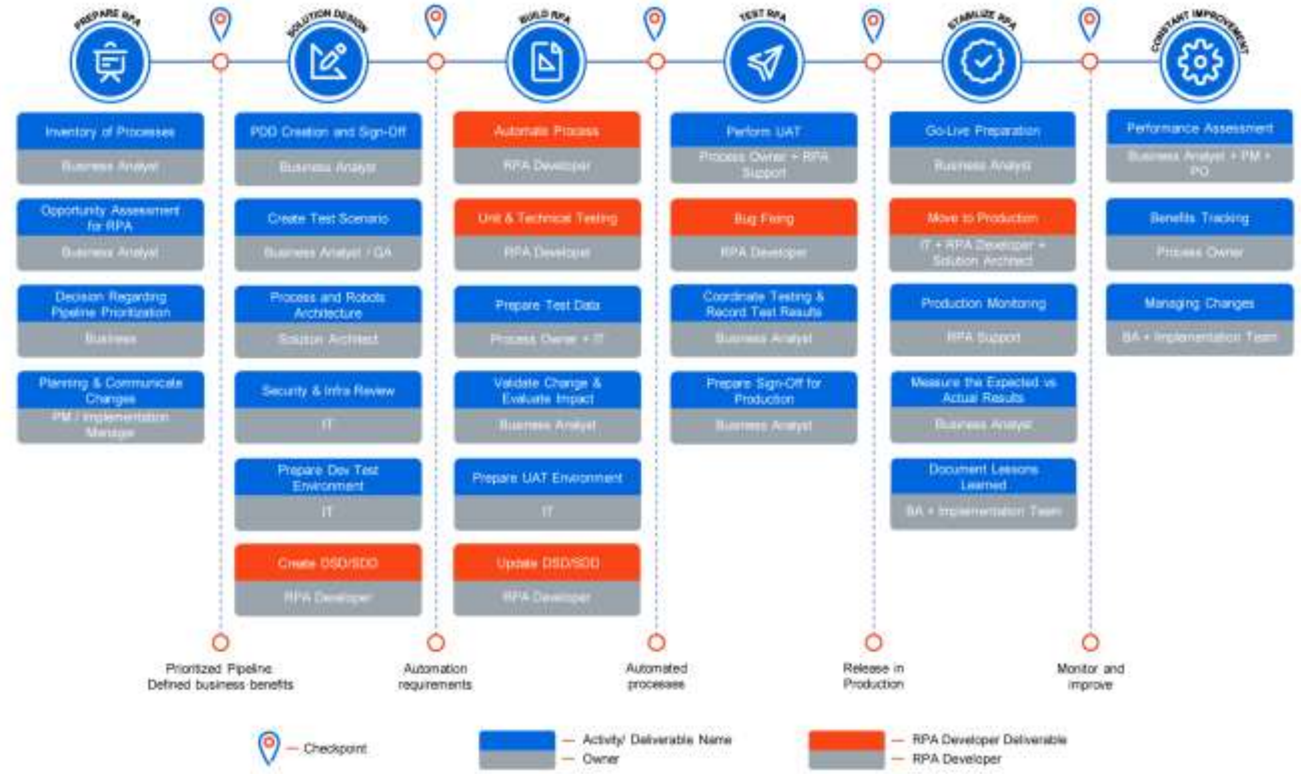
RPA solutions can be thought of as virtual robotic workforces, whose operational management is made by the business line (only supported by IT), just like for a human workforce.


Mimics human actions
Quick to implement and powerful to scale
Operates any application
RPA
Without making mistakes and without rest
Reads and processes data in structured form

Six stages in RPA implementations:-

1. **Prepare RPA** - the processes are defined, assessed, prioritized and the implementation is planned.
2. **Solution Design** - Each process to be automated is documented ("as is" and "to be"), the architecture is created and reviewed, the test scenarios and environments are prepared and the solution design is created and documented for each process.
3. **Build RPA** - The processes are automated, the workflow is tested and validated and the UAT prepared.
4. **Test RPA** - The UAT is performed, the workflow is debugged and the process is signed off.

5. **Stabilize RPA** - The Go-Live is prepared, the process is moved to production, monitored, measured and the lessons learned are documented.
6. **Constant Improvement** - The process automation performance is assessed, the benefits tracked and the changes managed.



Who will you be working with?

- Solution Architect - Is in charge of defining the Architecture of the RPA solution. The Solution Architect translates the requirements captured by the functional analysts, creating the architecture and design artifacts. They lead, advises, and is responsible for the developers' team delivery.
- Business Analyst - Is responsible for mapping of the AS IS and proposed TO BE processes. Business Analysts hold knowledge of the business process that gets automated, general business process theory and RPA capabilities. They are responsible with listing the process requirements for automation, clarifying the inputs and expected outputs, creating RPA documentation (Process Design Documents, Process maps.
- Implementation Manager/Project Manager - Forms and manages the RPA team, does resource planning and teams availability, in order to hit automation goals. Most of the times the PM is the Single Point of Contact (SPOC) for questions, RPA initiatives, or parallel RPA product projects.

- RPA Developers - On complex projects, several developers will collaborate to automate all processes.
- Infrastructure & IT Security admin - With good technical and security skills, they are responsible for setting up and maintaining hardware & software resources for UiPath product installations. They set up accounts for all the devs, end users and robots.
- Process Owner - Is the key stakeholder and beneficiaries of the RPA solution. Usually Senior Management level, with some 10-15+ years of experience, possibly split across domains. Multiple people can have this role, based on department (Finance, IT, HR, etc).
- RPA Support - Manage the robots after the processes have been moved to production, with support from the original RPA devs who have performed the automation. May have multiple levels of support: L1- Client, L2- client/ partner, (L0 – Super users; L3 – UiPath)

MEET THE UIPATH PLATFORM

The UiPath Platform offers you the components you need to design and develop automation projects, execute the instructions automatically and manage your robot workforce. The components of the UiPath Platform are **Studio** (the workflow designer), **Orchestrator** (the robot management platform) and the **Robot** (the agent executing the instructions).

Studio is at the heart of automation with UiPath products. **Activities** form into comprehensive workflows in Studio, which are then executed by the **Robot** and published to **Orchestrator**.

UiPath Studio

Helps you design automation workflows visually, quickly and with only basic programming knowledge. Studio is where the automated processes are built in a visual way, using the built-in recorder, drag & drop activities and best practice templates.

UiPath Orchestrator

Lets you control, manage and monitor the robots. It is also the place where libraries, reusable components, assets and processes used by the robots are stored. Orchestrator is a server application accessed via browser, through which the robotic workforce is controlled, managed and monitored:

- The connections with the robots are created and maintained, and the robots are grouped (control)

- The automated processes are distributed as tasks to the robots (management)

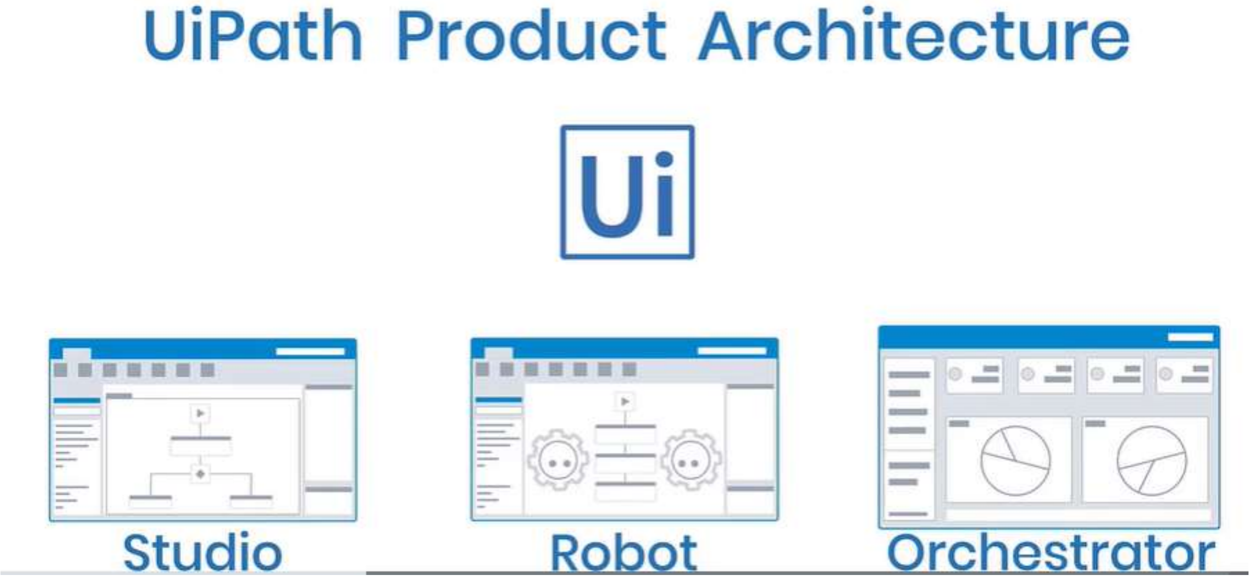
- The execution of tasks is logged and kept track of (monitoring)

The Robot

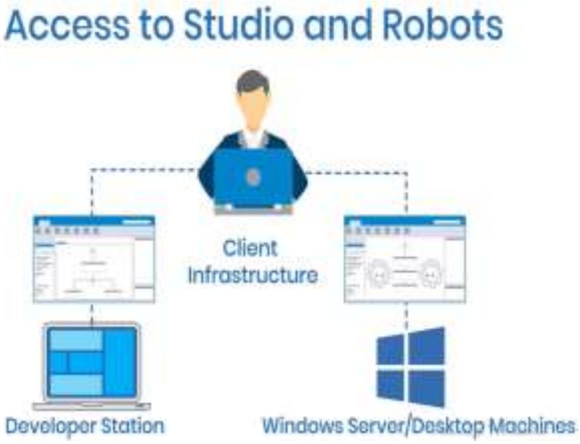
Executes the workflows and instructions sent locally or via Orchestrator. There are two types of robots:

- Attended – is triggered by user events, and operates alongside a human, on the same workstation

- Unattended - run unattended in virtual environments and can automate any number of processes



Studio and robots function within the client infrastructure. Studio is usually installed on the developer stations and robots function on windows server or desktop machines real or virtual depending upon the automation scenario**.**



Orchestrator is hosted on a web application server so the clients can access it through web browser with designated credentials.
Orchestrator comprises of several layers:

1) Presentation layer (interface to be accessed using browser)
2) Service layer (allowing for API calls)
3) Persistence layer (where relevant information and events are stored and locked. It is generally built using SQL server

Access to Orchestrator
Presentation layer
Service layer
Persistence layer

Installing UiPath Studio Community Edition

Community Edition	Enterprise Server Edition	Enterprise Cloud Edition
Always free. Upgrade to Enterprise any time.	On-premises enterprise deployments for large businesses.	Cloud enterprise deployments for businesses of any size. Currently in Preview.
2 Studios for designing automation	Unlimited Studios for designing automation	Unlimited Studios for designing automation
3 Robots	Unlimited Robots	Unlimited Robots
Cloud-hosted Orchestrator	On-premises Orchestrator	Cloud-hosted Orchestrator
Forum-only support	Premium Support	Premium Support
UiPath Academy access	Scale as you grow	Scale as you grow
	Self managed updates	Always up to date
	UiPath official training partners	Centralized user access management
		Secure and compliant
		UiPath official training partners

Business Process: A process is a set of interrelated or interacting activities that transforms inputs into outputs. Components of a process**:**

- **Inputs** - the data that goes in the process;
- **Process Flows** - the sequences of sub-processes or activities undertaken in the process;
- **Source Applications** - the applications or systems used to perform the sub-processes or activities of the process;
- **Outputs** - the result generated by the process;

NOTE: The outputs of a process can serve as inputs for other processes.

Organizations use processes because planning and executing them under controlled conditions can, among other things, improve compliance, ensure that operational needs are met, help in managing risks and drive improvement.

VARIABELS, DATA TYPES AND CONTROL FLOW

This lesson covers two separate constructs that are fundamental in any software process:

- **Variables and arguments**, or how data is collected, stored, processed and passed between various activities and workflows;
- **Control flow**, or how the activities, instructions and function calls are executed throughout the process.

Variables are containers that can hold multiple data entries (values) of the same data type. For example, emailAddress can be a variable that holds the value "rpadeveloper@uipath.com". The value of a variable can change through external input, data manipulation or passing from one activity to another.

Variables are configured through their properties. You can set them in the Variables panel. The main properties in UiPath are:

1. **Name**: It should be as descriptive as possible to make your automation easy to read by other developers and to save time.
2. **Type**: Defines what kind of data can be stored in the variable. In UiPath, the type is declared when the variable is created, however there are some specific types that are more generic and can accommodate different types of data. More about variables types right below.
3. **Default value**: In general, variables have initial values that change throughout the process. If no initial value is assigned at the creation of the variable, there is generally a default rule that assigns a value.
4. **Scope**: The part of the workflow in which the variable can be used. Some variables can be global, others local. In real automation scenarios, there are many variables in use. Making multiple variables unnecessarily global can cause efficiency issues as well as possibility for confusion.

Creating variables: There are 3 ways to create variables in UiPath: **Ctrl+K**

- From the Variables panel – Open the Variables panel, select the ‘Create new Variable’ option, and fill in the fields as needed. When you need it, provide its name in the Designer panel or in the desired Properties field.
- From the Designer panel – Drag an activity with a variable field visible (i.e. ‘Assign’) and press Ctrl+K. Name it and then check its properties in the Variables panel.
- From the Properties panel – In the Properties panel of the activity, place the cursor in the field in which the variable is needed (i.e. Output) and press Ctrl+K. Name it and then check its properties in the Variables panel.

NOTE: If variable is already created, no need to use within " ". If we want to type anything then we have to type within " ". Example is shown below.
Username is a variable of type **string**. Its value is **Archana**.
Message Box: Username + "is logged in."
Output: Archana is logged in.

Arguments

In UiPath, the scope of a variable cannot exceed the workflow in which it was defined. Since business automation projects rarely consist of single workflows, arguments have to be used.

Arguments are very similar to variables – they store data dynamically, they have the same data types and they support the same methods. The difference is that they pass data between workflows, and they have an additional property for this – the direction from/to which the data is passed. The direction can be In, Out and In/Out.

Data Types

The data types in UiPath are borrowed from **VB.Net**. Below are some of the most common ones used:

1. Numeric (category): Used to store numbers. There are different sub-types of numerical variables:

- Int32 - System.Int32 (signed integers): 10, 299, -100, 0x69
- Long - System.Int64 (long integers): 54354353430, -11332424D
- Double - System.Double (allows decimals, 15-16 digits precision): 19.1234567891011

2. **Boolean: System.Boolean**: Used to store one of two values – true or false.

3. **Date and Time** (category)

- DateTime - **System.DateTime**: Used to store specific time coordinates (**mm/dd/yyyy hh:mm:ss**). This kind of variable provides a series of specific processing methods (subtracting days, calculating time remaining vs. today, and so on). For example, to get the current time, assign the expression **DateTime.Now** to a variable of type DateTime.
- TimeSpan - **System.TimeSpan**: Used to store information about a duration (**dd:hh:mm:ss**). You can use it to measure the duration between two variables of the type DateTime. For example, you can save the time at the start of the process in one variable (of type DateTime), the time at the end in another (of type DateTime) and store the difference in a variable of type TimeSpan

4. **String**: **System.String**: Used to store text. This type of data comes with many specific methods of processing, and will be addressed in depth in another lesson, namely Data Manipulation.

5. **Collection** (category): This category reunites all the collections of objects, with each object being identified through its index in the collection. Collections are largely used for handling and processing complex data. Some of the most encountered collections are:

- Array - **ArrayOf<T>** or **System.DataType[]:** used to store multiple values of the same data type. The size (number of objects) is defined at creation;
- List - **System.Collections.Generic.List<T>:** used to store multiple values of the same data type, just like Arrays. Unlike Arrays, their size is dynamic;
- Dictionary - **System.Collections.Generic.Dictionary<TKey, TValue>:** used to store objects in the form of (key, value) pairs, where each of the two can be of a separate data type.

6. **GenericValue**: This is a UiPath proprietary variable type that can store any kind of data, including text, numbers, dates, and arrays. This type is mainly used in activities in which we are not sure what type of data we will receive, yet in general the use of this is temporary.

NOTE: Keep in mind that the list of types presented above is not a complete list, but the list of the most common types used. Other types may be used in specific situations. When browsing or searching, you will find most of them under the System and **System.Collections** categories.

In some cases, variables are generated automatically by activities, and their types may vary – for example, an activity that locates and stores a graphic element will automatically generate a variable of UiElement type.

Array Variables

The array variable is a type of variable that enables storing multiple values of the same data type. Think of it as a group of elements with a size that is defined at creation, and each item can be identified by its index.
In UiPath Studio, you can create arrays of numbers, of strings, of Boolean values and so on.

What are some business scenarios in which I will use arrays?

- When we want to save the names of the months to a variable
- When a fixed collection of bank accounts has to be stored and used in the payment process
- When all the invoices paid in the previous month have to be processed
- When the names of the employees in a certain unit have to be verified in a database

NOTE: As a good case practice, **arrays** are used for **defined sets of data** (for example, the months of the year or a predefined list of files in a folder). Whenever the collection might require **size changes**, a **List** is probably the better option.

GenericValue Variables

The GenericValue (UiPath.Core.GenericValue) variable is a type of variable particular to UiPath that can store any kind of data, including text, numbers, dates, and arrays.
While developing an automation process, there are situations where you are **not sure what type of data will be retrieved**. This is where we recommend temporarily using GenericValue Variables.

UiPath Studio has an automatic conversion mechanism of GenericValue variables, which you can guide towards the desired outcome by carefully defining their expressions. Please note that the first element in your expression is used as a guideline for what operation Studio performs. For example, when you try to add two GenericValue variables, if the first one in the expression is defined as a String, the result is the concatenation of the two. If it is defined as an Integer, the result is their sum.
Example: Below variables are of generic type.
A="111"
B="222"
C= 111
Output:

1. A+B=111222 (Concatenation has happened here as 2 variables are declared in string form " ")
2. A+C=111111 (Concatenation has happened giving preference to 1st data type. Means A is string and C is considered as Int)
3. C+A=222 (Addition has happened giving preference to 1st data type. Means C is Int and A is string)

What are some business scenarios in which I will use GenericValue variables?

- Data is extracted from a UI field and forwarded to another workflow without processing
- Two versions of the same Excel file are being compared column by column. The columns are different in terms of data type, the only relevant thing is which entries have changes

Control Flow

It is the order in which **individual statements**, **instructions** or **function calls** are executed or evaluated in a software project.

There are 2 concepts through which the control flow is enacted

The type of automation project
There are 4 predefined types of workflows – Sequence, Flowchart, State Machine and Global Exception Handler.

We are covering them in depth in the “Project Organization” lesson, for now let’s focus on the difference between sequences and flowcharts, as we will use both extensively in our examples throughout the entire course.

1. In **sequences**, the process steps flow in a clear succession. Decision trees are rarely used. Activities in sequences are easier to read and maintain, thus, they are highly recommended for simple, linear workflows.
2. In **flowcharts**, the individual activities are a bit more difficult to read and edit, but the flows between them are much clearer. Use flowcharts when decision points and branching are needed in order to accommodate complex scenarios, workarounds and decision mechanisms.

The control flow statements: The activities and methods used to define the decisions to be made during the execution of a workflow. The most common control flow statements are the if/else decision, the loops and the switch

IF Statement

- The condition that is verified (with 2 potential outcomes – true or false)
- The set of actions to be executed when the condition is **true** (the **Then** branch)
- The set of actions to be executed when the condition is **false** (the **Else** branch)

If decision can be used as an operator inside activities. Based on the chosen type of automation project, there are 2 corresponding activities that fulfill the If statement role:

- The If Statement in sequences
- The Flow Decision in flowcharts

What are some business scenarios in which I will use the If statement?
Whenever there are two courses of action that are not arbitrary, an If statement will most likely be used:

- Checking the status of a payment (done/not done) and performing a set of operations in each case
- Making sure that the outcome of the previous operation in the sequence is successful
- Checking the balance of an account to ensure that there is enough money to pay an invoice

- Checking if something has happened in a system, like if an element or an image exists and performing an action based on that.

Loops: Loops are repetitions of a set of operations based on a given condition. In UiPath, the most important loops are:

Do While
It executes a specific sequence while a condition is met. The condition is evaluated after each execution of the statements.

For example, a robot could perform a refresh command over a website and then check if a relevant element was loaded. It will continue the refresh - check cycle until the element is loaded.

While
It executes a specific sequence while a condition is met. The condition is evaluated before each execution of the statements.

In many cases, it is interchangeable with Do While, the only difference being when the condition verification is made. But in some cases, one is preferable over the other. For example, if a Robot would play Blackjack, it should calculate the hand before deciding whether to draw another card.

For Each
It performs an activity or a series of activities on each element of a collection.

This is very useful in data processing. Consider an Array of integers. For Each would enable the robot to check whether each numeric item fulfills a certain condition.

Switch: It is a type of control flow statement that executes a set of statements out of multiple, based on the value of a specific expression. In other words, we use it instead of an If statement when we need at least 3 potential courses of action. This is done through the condition, which is not Boolean like in the case of If statement, but multiple.

What are some business scenarios in which I will use Switch?

- An invoice that has 3 potential statuses (not started, pending, approved) and 3 sets of actions for each one
- A process of automatically ordering raw materials to 4 suppliers based on certain conditions

DATA MANIPULATION

Data manipulation is the process of modifying, structuring, formatting, or sorting through data in order to facilitate its usage and increase its management capabilities

What are some business scenarios in which I will use data manipulation?

- Data manipulation methods are frequently employed by website owners to extract and view specific information from their web server logs. That allows for watching their most popular pages, as well as their traffic sources.
- Consider the process of interrogating public financial or legal databases. Data manipulation provides the means for the credit analysts to extract only the relevant data and use it in other documents or correlate it with information from other sources.

Strings

Strings are the data type corresponding to text. It is difficult to imagine an automation scenario that doesn’t involve the use of strings

What are some business scenarios in which you will most likely encounter dictionaries?

- Getting the status of an operation
- Extracting the relevant piece from a larger text portion
- Displaying information to the human user

String manipulation: String manipulation is done by using String Methods borrowed from VB.Net. Below are some of the most common methods used in RPA:

1. **Concat**: Concatenates the string representations of 2 specified objects
   Expression: **String.Concat(Varname1,VArname2).** Here it concatenates 2 variables
2. **Contains**: Checks whether a specified substring occurs within a string. Returns true or false.
   Expression: **Varname.Contains(“text”)**
   Example: Message=Cat is small.
   IF Message.Contains(“cat”), then display true
   Output: True**.**
3. **Format**: Converts the value of objects to strings (and inserts the in to another text)
   Expression: **String.Format(“{0} is {1}”,Varname1,Varname2).** Here in place of {0}, Varname1 is printed and in place of {1}, Varname2 is printed.
4. **IndexOf**: Returns the zero-based index of the first occurrence of a character in string.
   Expression: **Varname1.IndexOf(“a”)**
5. **Join**: Concatenates the elements in a collection and displays them as string.

Expression: **String.Join(",",cities).** Here cities is variable. It joins all the strings present inside cities and each of them are separated by ,

6. **Replace:** Replaces all the occurrences of a substring in a string.
   Expression**: Varname.Replace("original","replaced")**
7. **Split:** Splits a string into 2 substrings using a given separator.
   Expression**: Varname.Split(","c)(index)**
   Example: Cities.Split("."c)(0). Here cities will be split by . and only 1st sentence before . will be taken and treated as (0).
8. **Substring:** Extracts a substring from a string using the starting index and length.
   Expression: **Varname.Substring(StartIndex,Length)**

Lists

Lists (or List<T>, as you will encounter them) are data structures consisting of objects of the same data type (for example string or integer). Each object has a fixed position in the list; thus, it can be accessed by index. While arrays are fixed-size structures for storing multiple objects, lists allow us to add, insert and remove items.

Lists can store large numbers of elements - names, numbers, time coordinates and many others. Lists provide specific methods of manipulation, such as:

- Adding and removing items
- Searching for an element
- Looping through the items (and performing certain actions on each)
- Sorting the objects
- Extracting items and converting them to other data types.

What are some business scenarios in which you will most likely encounter lists?

- Storing the computer names of the members of a project team for a certain configuration that needs to be made
- Collecting and storing the numbers of invoices that meet certain criteria
- Keeping track of the ticket numbers created in a certain period on a certain issue;

Uipath Methods applicable to Collections: Manipulating lists can be done by using .NET methods or using the collections methods that Uipath Studio offers:

- **Add to Collection**: Adds an item to a specified collection. It is equivalent to List.Add(). It can be used, for example, to add a new name to a list of company names.
- **Remove from Collection**: Removes an item from a specified collection and can output a Boolean variable that confirms the success of the removal operation. This activity can be used, for example, to remove an invoice number from a list of invoices to be processed.

- **Exists in Collection**: Indicates whether a given item is present in a given collection by outputting a Boolean as the result. We can use this activity to check whether a list of clients contains a specific name.
- **Clear Collection**: Clears a specified collection of all items. One possible use is to empty a collection before starting a new phase of a process that will populate it again

Dictionaries

Dictionaries (or Dictionary<TKey, TValue>, as you will encounter them) are collections of (key, value) pairs, in which the keys are unique. Think of the Address Book in your mobile phone, where each name has corresponding data (phone number(s), email).

The data types for both keys and values have to be chosen when the variable is instantiated. Data types in Dictionaries can be any of the supported variables (including Dictionaries, for example).

The operations that are most often associated with Dictionaries are:

- Adding and deleting (key, value) pairs
- Retrieving the value associated with a key
- Re-assigning new values to existing keys

What are some business scenarios in which you will most likely encounter dictionaries?

- Storing configuration details or other information that needs to be accessed throughout a process
- Storing the job titles or other relevant information of employees
- Storing the bank accounts of suppliers

Methods for working with Dictionaries:

1. **Initialization**:
   Just like in the example of Lists, Dictionaries have to be initialized. In the following example, the initialization is done inside an 'Assign' activity. However, as you may remember from the Lists chapter, it can be done from the Variables Panel.
2. **Adding**
   VarName.Add(Key, Value) – adds an item to an existing Dictionary. Because Add does not return a value, use the Invoke Code activity.
3. **Removing**
   VarName.Remove(Key) – removes an item from the Dictionary. It can be used in an 'Assign' activity.
4. **Retrieving**

- **VarName.Item(Key)** – returns the Dictionary item by its key .
- **VarName.Count** – returns an Int32 value of the number of Dictionary items
- **VarName.ContainsKey(Key)** – checks if the item with the given key exists in the Dictionary and returns a Boolean result
- **VarName.TryGetValue(Key, Value)** – checks if an item with a given key exists in the Dictionary and returns a Boolean result and the value if found

RegEx: Regular Expression

Regular Expression (REGEX, or regexp) is a specific search pattern that can be used to easily match, locate and manage text. However, creating RegEx expressions may be challenging.

UiPath Studio contains a RegEx builder that simplifies the creation of regular expressions.

Typical uses of RegEx include:

- Input validation
- String parsing
- Data scraping
- String manipulation

What are some business scenarios in which I will use RegEx?

- Retrieving pieces of text that follow a certain pattern, for example:
- extracting phone numbers that start with a certain digit;
- collecting all the street names from a bulk text, even if they don't follow a specific pattern - some of them contain "Street", others "Rd.", and so on.

It would take much longer to build the same expression using the regular String methods – for example, RegEx has a predefined expression to locate all the URLs in a string.

Methods in Uipath that use the RegEx builder:

1. **Matches**: Searches an input string for all occurrences and return all the successful matches.
2. **IsMatch**: Indicates whether the specified regular expression finds a match in the specified input string.
3. **Replace**: Replace strings that match a regular expression pattern with a specified replacement string.

EXCEL AND DATA TABLES

Data Tables:

It is the type of variable that can store data as a simple spreadsheet with rows and columns, so that each piece of data can be identified based on their unique column and row coordinates. Think of it as the memory representation of an Excel worksheet.

In Data Tables, the regular convention of identifying the columns and the rows is applied - columns are identified through capital letters, and rows through numbers.

What is the difference between a worksheet and a Data table?

How are DataTables created?
The most common ways to create DataTables are:

1. **Build data table:**
   By using this activity, you get to choose the number of columns and the data type of each of them. Moreover, you get to configure each column with specific options like allow null values, unique values, auto-increment (for numbers), default value and length (for strings).
2. **Read Range:**
   This activity gets the content of a worksheet (or a selection from that worksheet) and stores it in a DataTable variable, which can be created from the Properties panel using Ctrl + K.
3. **Read CSV:**
   This activity captures the content of a CSV file and stores it in a DataTable variable. Although not commonly used anymore, there are still legacy or internal-built applications that work with this kind of documents.
4. **Data scraping:**
   This functionality of UiPath Studio enables you to extract structured data from your browser, application or document to a DataTable.

Data Table Activities:

UiPath offers a broad range of activities that can be used to work with DataTable variables:

1. **Add Data Column**
   Adds a column to an existing DataTable variable. The input data can be of DataColumn type or the column can be added empty, by specifying the data type and configuring the options (allowing null values, requesting unique values, auto-incrementing, default value and maximum length).

2. **Add Data Row**
   Adds a new row to an existing DataTable variable. The input data can be of DataRow type or can be entered as an Array Row, by matching each object with the data type of each column.
3. **Build Data Table**
   Is used to create a DataTable using a dedicated window. This activity allows the customization of number of columns and type of data for each column.
4. **Clear Data Table**
   Clears all the data in an existing DataTable variable.
5. **Filter Data Table**
   Allows filtering a DataTable through a Filter Wizard, using various conditions. This activity can be configured to create a new DataTable for the output of the activity, or to keep the existing one and filter out (delete) the entries that do not match the filtering conditions.
6. **For Each Row**
   Is used to perform a certain activity for each row of a DataTable (similar to a For Each loop).
7. **Generate Data Table**
   Can be used to create a DataTable from unstructured data, by letting the user indicate the row and column separators.
8. **Join Data Tables**
   Combines rows from two tables by using values common to each other, according to a Join rule that answers the question "What to do with the data that doesn't match?". It is one of the most useful activities in business scenarios, where working with more than one Data Table is very common. This is why we'll cover in more in depth below.
9. **Lookup Data Table**
   Is similar to vLookup in Excel, as it allows searching for a provided value in a specified DataTable and returns the RowIndex at which it was found, or can be configured to return the value from a cell with given coordinates (RowIndex and Target Column).
10. **Merge Data Table**
    Is used to append a specified DataTable to the current DataTable. The operation is more simple than the Join Data Type activity, as it has 4 predefined actions to perform over the missing schema.
11. **Output Data Table**
    Writes a DataTable to a string using the CSV format.
12. **Remove Data Column**
    Removes a certain column from a specified DataTable. The input may consist of the column index, column name or a Data Column variable.
13. **Remove Data Row**
    Removes a row from a specified DataTable. The input may consist of the row index or a Data Row variable.
14. **Remove Duplicate Rows**

Removes the duplicate rows from a specified DataTable variable, keeping only the first occurrence.

15. Sort Data Table

Can sort a DataTable ascending or descending based on the values in a specific column.

Join Data Tables:

How does it work?

1. 3 Data Table variables have to be specified - 2 Input DataTables and 1 Output DataTable. Please note that the order of the first 2 is very important, as there is one option that keeps the values from Data Table 1 and it cannot be changed.

2. The **Join Type** has to be chosen - there are 3 options:

- **Inner**: Keep all rows from both tables that meet the Join rule. Any rows that do not meet the rule are removed from the resulting table.
- **Left**: Keep all rows from DataTable1 and only the values from DataTable2 which meet the Join rule. Null values are inserted into the column for the rows from DataTable1 that don't have a match in the DataTable2 rows.
- **Full**: Keep all rows from DataTable1 and DataTable2, regardless of whether the join condition is met. Null values are added into the rows from both tables that don't have a match.

3. The **Join rules** have to be configured (there can be one or more rules):

- One column from each DataTable has to be specified by their names (String), by their index (Int32) or by ExcelColumn variables
- The operator has to be chosen: = (Equal to), != (Not equal to), > (Greater than), < (Less than), >= (Greater than or equal to), <= (Less than or equal to)

What are some business scenarios in which I will use Join Data Tables?

Join Data Table provides one of the easiest ways to bring data from two sources in one place:

- Bringing together 2 databases of employees extracted from 2 applications
- Checking which of the clients (database 1) have been contacted in a marketing campaign (database 2)
- Checking which of the suppliers of a company (internal database) have applied for public aid (public database)

Workbooks and Common Activities:

In many business scenarios, databases are stored in workbooks (generally known as Excel files or spreadsheets). From there, they can be inputted into DataTables and further processed using the methods presented in the previous chapter, as well as other available methods and tools. It's time to see how RPA deals with workbooks.

UiPath offers 2 separate ways of accessing and manipulating workbooks, each of them with advantages and limitations:

1. **File access level :**
   All workbook activities will be executed in the background.

- Doesn't require Microsoft Excel to be installed, can be faster and more reliable for some operations just by not opening the file;
- Works only for .xlsx files.

2. **Excel app integration:**
   UiPath will open Excel just like a human would.

- Works with .xls and .xlsm, and it has some specific activities for working with .csv. All activities can be set to either be visible to the user or run in the background;
- Microsoft Excel must be installed, even when the 'Visible' box is unchecked. If the file isn't open, it will be opened, saved and closed for each activity.

Both access levels share some activities, with Excel App Integration having several more. Please note that in UiPath there are two activities for each method presented below - one under 'App Integration > Excel', the other under 'System > File > Workbook'.

Let's start with the common activities:

1. **Append Range**: Adds the information from a DataTable to the end of a specified Excel spreadsheet. If the sheet does not exist, it creates it.
2. **Get Table Range**: Locates and extracts the range of an Excel table from a specified spreadsheet using the table name as input.
3. **Read Cell:** Reads the content of a given cell and stores as String.
4. **Read Cell Formula:** Reads the formula from a given cell and stores it as String.
5. **Read Column:** Reads a column starting with a cell inputted by the user and stores it as an IEnumerable <object> variable.
6. **Read Range:** Reads a specified range and stores it in a DataTable. If 'Use filter' is checked in the Read Range activity under 'Excel Application Scope', it will read only the filtered data. This option does not exist for the Read Range activity under 'Workbook'.
7. **Read Row:** Reads a row starting with a cell inputted by the user and stores it as an IEnumerable <object> variable

8. **Write Cell:** Writes a value into a specified cell. If the cell contains data, the activity will overwrite it. If the sheet specified doesn't exist, it will be created.
9. **Write Range:** Writes the data from a DataTable variable in a spreadsheet starting with the cell indicated in the StartingCell field.

Excel Application Scope and Specific activities:

The integration with Excel is enabled by using an Excel Application Scope activity. In fact, it is a container and all the other activities used to work with the specified Excel file have to be placed inside the container. Basically, it opens an Excel workbook and provides a scope for Excel activities. When the execution ends, the specified workbook and the Excel application are closed.

The Excel Application Scope can be configured to write the output of the activities in the container in a different file.

Important note: If the same workflow deals with information from two or more Excel files, an Excel Application Scope has to be used for each file.

Excel App Integration Specific Activities

1. **CSV:** These activities can read from and write to CSV files, using DataTable variables. Although found under Excel App Integration, they work even if they are not placed inside an Excel Application Scope container.

- **Append to CSV**: add the info from a DataTable to a CSV file, creating it if it doesn't exist. The activity does not overwrite existing data
- **Read CSV**: reads all entries from a CSV file and store them in a DataTable
- **Write CSV**: overwrites a CSV with the inforomation from a DataTable

2. **Range**: These activities can read data, insert and delete rows and columns, and even copy/paste entire ranges. They are similar to the corresponding activities under DataTable, but they work directly in the Excel file.

- **Delete Column**: removes a column from an Excel file based on the name.
- **Insert Column**: inserts a blank column in an Excel file, at a certain position.
- **Insert/Delete Columns**: either adds blank columns or removes existing columns, based on the specified change type.
- **Insert/Delet Rows**: either adds blank rows or removes existing rows, based on the specified change type.
- **Select Range**: selects a specific range in an Excel file. In general, it is paired with another activity that performs a certain manipulation over the selected data.
- **Get Selected Range**: outputs a given range as String.

- **Delete Range**: removes a specified range from an Excel file.
- **Auto Fill Range**: applies a given formula over a given range in an Excel file.
- **Copy Paste Range**: copies and pastes an entire range (values, formulas and formatting) from a source sheet to a destination sheet.
- **Lookup Range**: searches for a value in all the cells in a given range.
- **Remove Duplicate Range**: deletes all duplicate rows from a given range.

3. **Table**: These activities create, filter and sort tables directly in Excel files.

- **Filter Table**: applies a filter on all the values from a column in a table inside an Excel file. Once the file is saved, only the rows that meet the filter will be displayed. Please note that this activity does not remove the rows that do not meet the criteria, but only hides them. A good use of this method involves using a Read Range activity after this one, with 'Use filters' box checked. The output will be a DataTable containing only the entries that met the given criteria.
- **Sort Table**: sorts a table in an Excel file based on the values in a given column.
- **Create Table**: it creates a table (with name) in a range specified in the Properties panel.

4. **File**: These activities work directly with the Excel files, either by saving or closing them.

- **Close Workbook**
- **Save Workbook**

5. **Cell Color**: These activities are able to capture and modify the background color of cells in Excel files.

- **Get Cell Color**: reads the background color or a given cell in an Excel file and stores it as color variable output.
- **Set Range Color**: changes the background color of all the cells in a given range. The input is a color variable.

6. **Sheet**: These activities can perform various actions over the sheets in an Excel file.

- **Get Workbook Shee**t: reads the name of a sheet by its index.
- **Get Workbook Sheets**: extracts the sheet names and stores them ordered by index.
- **Copy Sheet**: copies a sheet in an Excel file and pastes either in the same Excel file or in a different one specified.

7. **Pivot Table**: These activities facilitate working with pivot tables in Excel files.

- **Refresh Pivot Table**: refreshes a pivot table in an Excel file. This is useful when pivot table source data changes, as the refresh is nout automatic.
- **Create Pivot Table**: creates a pivot table using a specified sheet and given parameters.

8. **Macro**: These activities can execute macros that where already defined in the Excel file, or can invoke macros from other files. Please note that these activities work with .xslm files.

- **Execute Macro**
- **Invoke VBA**: macro from another file

UI INTERACTIONS

Input actions and methods:

Every time we insert data into an application, or we send a command to a system to produce a change (or a continuation), we perform an input action. In UiPath, the main input actions are Click, Type into, Send Hotkey and Hover. These are also the main actions that a human user would perform to input data in an application.

Input methods: Input methods provide the input actions with the means to input data. Each input action lets the user switch between the 3 methods: 'Simulate Type/Click/Hover', 'SendWindowMessages' and default. ‘Simulate Type/Click/Hover’ and ‘SendWindowMessages’ can be chosen by checking the corresponding box from the Properties panel; if none of the two boxes is checked, the default input method is used.

1. **Default:**

How does it work?

- Clicks: the mouse cursor moves across the screen
- Typing: the keyboard driver is used to type individual characters

What are the implications?

- Attended User cannot touch the mouse or keyboard during the automation
- It has a lower speed and load times can impact accuracy

What are the strong points?

- Supports special keys like Enter, Tab, and other hotkeys
- 100% compatibility

What are the limitations?

- Does not automatically erase previously written text
- Does not work in the background

2. **Send Window Messages:**

How does it work?

- Replays the window messages that the target application receives when the mouse/keyboard is used
- Clicking and typing occur instantly

What are the implications?

- Works in the background
- Comparable to the Default method in terms of speed

What are the strong points?

- Supports special keys like Enter, Tab, and other hotkeys
- Users can work on other activities during the execution of the automated processes

What are the limitations?

- Does not automatically erase previously written text
- Works only with applications that respond to Window Messages

3. **Simulate:**

How does it work?

- Uses the technology of the target application (the API level) to send instructions
- Clicking and typing occur instantly

What are the implications?

- Works in the background
- Actions are a lot faster, but there are some compatibility limitations

What are the strong points?

- Can automatically erase previously written text
- Users can work on other activities during the execution of the automated processes

What are the limitations?

- Does not support special keys like Enter, Tab, and other hotkeys
- It has a lower compatibility than the other 2 methods

Input actions: **Click**, **Type into** and **Send hotkey** are simple in terms of what they primarily do. All the input actions share several properties:

- **Delay**: can be used to set a delay before or after the click;
- **WaitForReady**: can be configured to wait for the target to become ready by verifying certain application tags.

What is maybe more important to see is the options that each of them offer:

1. **Click**:
   - **ClickType**: can be single or double (changing the default to double makes it very similar to a separate activity – Double Click);
   - **MouseButton**: can be configured to left, middle or right button;
     Timeout: specifies the duration in which the activity is retried until an error is thrown,
   - **KeyModifiers**: can press the Alt, Ctrl, Shift and/or the Win key while performing the action.
2. **Type into:**
   - **Activate**: the field in which the text is typed is brought to the foreground and activated, before the typing;
   - **ClickBeforeTyping**: the UI element in which the text will be typed into will be clicked on before typing;
   - **DelayBetweenKeys**: the delay between each key is typed;
   - **EmptyField**: the UI element will be emptied before typing.

You can find out more about the **Type into** activity and its properties here.

For typing into protected fields, such as 'Password', there is a more suitable activity – **Type Secure Text**.

3. **Send Hot key:**
   - **Activate**: the field in which the text is typed is brought to the foreground and activated, before the typing;
   - **ClickBeforeTyping**: the UI element in which the text will be typed into will be clicked on before typing;
   - **DelayBetweenKeys**: the delay between each key is typed;
   - **EmptyField**: the UI element will be emptied before typing.

Output actions and methods:

Output actions are used in UiPath either to extract data (in general, as text) from a UI element. Output methods are what enables output actions to extract data from UI elements.

Output Methods:

1. **Full Text:** The FullText method is the default method and good enough in most cases. It is the fastest, it has 100% accuracy and can work in the background. Moreover, it can extract hidden text (for example, the options in a drop-down list). On the other hand, it doesn't support virtual environments and it doesn't capture text position and formatting. This method offers the option to ignore the hidden message and capture only the visible text.

2. **Native:** The Native method is compatible with applications that use Graphics Design Interface (GDI), the Microsoft API used for representing graphical objects. It can extract the text position and formatting (including text color) and has 100% accuracy on the applications that support GDI. Its speed is somewhat lower than FullText, it doesn't extract hidden text and it cannot work in the background; and just like FullText, it doesn't support virtual environments. By default, it can process all known characters as separators (comma, space, and so on), but when only certain separators are specified, it can ignore all the others.

3. **OCR:** OCR (or Optical Character Recognition) is the only output method that works with virtual environments and with "reading" text from images. Its technology relies on recognizing each character and its position. On the other hand, it cannot work in the background, it cannot extract hidden text, and its speed is by far the lowest. Its accuracy varies from one text to another and changing settings can also improve the results. Just like the Native method, it also captures the text position. The OCR method has two default engines that can be used alternatively - Google Tesseract and Microsoft MODI. There are additional OCR engines that can be installed free of charge (such as Omnipage and Abbyy Embedded) or paid (IntelligentOCR offered by Abbyy).

   You can switch between all these methods by accessing the **Screen Scraping** wizard, once you indicate an area to extract the text from and the Preview window is launched.

Below are the most important output actions. As a difference from the Input Actions, where all the Input Methods are available for each Input Action, the Output Actions are somewhat matched with the Output Methods, as you will see.

- **Get Text:**
  Extracts a text value from a specified UI element.
- **Get Full Text:**
  Extracts a string and its information from an indicated UI element using the FullText screen scraping method. Thus, the hidden text is also captured by default (although it provides the option to ignore hidden text). This activity is automatically generated when performing screen scraping with the FullText method, along with a container.
- **Get Visible Text:**
  Extracts a string and its information from an indicated UI element using the Native screen scraping method. This activity is automatically generated when performing screen scraping with the Native method chosen, along with a container.
- **Get OCR Text:** Extracts a string and its information from an indicated UI element using the OCR screen scraping method. This activity can also be automatically generated when performing screen scraping, along with a container. By default, the Tesseract OCR engine is used
- **Data Scraping wizard:** Data scraping is a functionality of UiPath Studio that allows the extraction of structured information from an application, browser or document to a DataTable variable. The functionality can be accessed directly from the Design ribbon of UiPath Studio, the 'Data Scraping' button. The first element chosen is used to populate the first column, and the option to extract the URL (where these exist) is also presented. The user may change the order of the columns and specify the maximum number of entries to be extracted (the default is 100 and leaving 0 means extracting all the results). In the Preview stage, the 'Extract Correlated Data' option can be used to extract other fields of data, by indicating the first and the second entry, just like for the main field.
- **Extract Attributes:** This is actually a category of activities that can be used when you don't want to extract the text out of the UI element, but maybe the color, the position or an ancestor. There are 3 different UiPath activities to do this:

1. **Get Ancestor**: UI elements are in parents-children structures (a text document has the Notepad app as a parent, who has the category of apps as a parent, and so on). Get Ancestor retrieves the ancestor (or the parent) of an UI element.
2. **Get Attribute**: UI elements have plenty of attributes. Think of a button on a website – it definitely has a color, a name, a state, and so on. Get Attribute allows the user to indicate an attribute, and the activity retrieves the value of that specific attribute.
3. **Get Position**: this activity retrieves the actual position on the screen of a specific element. This can be very useful when there are many similar elements on a screen; without having their actual position, it would become very difficult to identify each of them.

Working with UI Elements:

Input and Output Actions can be viewed as a succession of 2 micro-steps: first, recognizing UI elements, and then inputting (or extracting) data. In many cases, such as most of the examples that we covered in the previous chapters, the first step is easily sorted out either by including it in the activity, or by using the delay option. This is why, in many cases, it’s not at all necessary to have a separate activity for locating an UI element (like a Find Element activity) before an actual Click or Type activity.
In some business scenarios however, building reliable workflows means accommodating situations in which UI elements move or become visible at only at various times. Moreover, some solutions that seem handy, like setting up a big Delay before executing a Click activity, may trigger a significant increase in the overall duration of an automation (if, for example, there are 50 Click activities).

There are several actions that are meant to locate UI elements and it’s important to know when each of them is useful:

- **Find element:** Waits for the specified UI element to appear on the screen (to be in the foreground) and returns it as a UiElement variable. This is useful, for example, when a certain action needs to be performed on the Ui Element found.
- **Element Exists:** Enables you to verify if a UI element exists, even if it is not visible. It returns a Boolean variable, which makes it very useful in ‘If statement’ activities, for example.
- **Wait Element Vanish:** Waits for the specified UI element to disappear from the screen. It’s an alternative to Find Element, for example when the disappearance of an element (a loading sign) is more reliable then the appearance of another element.
- **On Element Appear:** A container that waits for a UI element to appear and enables you to perform multiple actions within it.
- **On Element Vanish:** A container that enables you to perform one or multiple actions after a specified UI element vanishes.
- **Text Exists:** Checks if a text is found in a given UI element. There’s an alternative version of this that uses OCR technology to check for a given UI element. This is useful when UI elements are not accessible otherwise than as images.

SELECTORS

What are Selectors?
A Selector in UiPath Studio is a feature that enables the **identification** of the User Interface elements through its **address and attributes** stored as XML fragments. The element identification is done to perform specific activities in an automation project. Selectors are generated automatically every time we use an activity that interacts with graphical user interface elements.

We can think of the **element identification process** achieved through Selectors as a postman that delivers letters to a certain address. In order for the postman to deliver the letters, a specific path is required and must contain **structured and hierarchized details** such as Country > City > Zip Code > Street Name > Street Number > Apartment Number. Similarly, UiPath Studio requires the detailed path to a specific element within the user interface.

What are some business scenarios in which Selectors are useful?
Most of the times when an automated process involves working with UI elements, selectors are used. Typical activities include:

- Clicking buttons
- Inputting or scraping text in/from a field on a website
- Choosing an option from a drop-down list

Brief Introduction: Introduction to UI Trees within the UI Explorer, selector tag structure and selector fine-tuning scenarios.

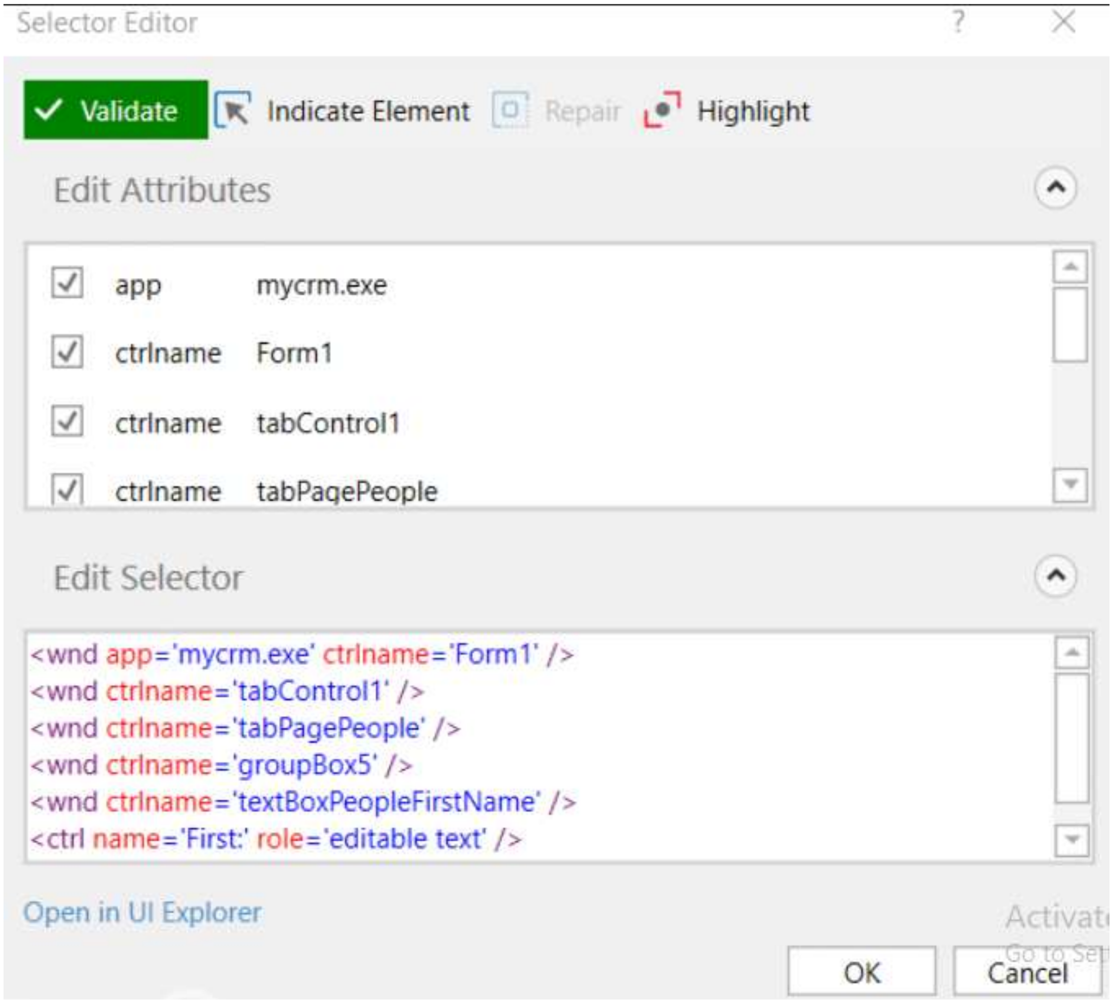
- They are properties of UI automation activities.
- They contain the attributes of each GUI elements and its parents as XML fragments.
- They are generated automatically when we use an activity that interacts with GUI elements.
- The GUI element where you click will be included in the last line of selector. The 1st line is usually referred to as **Root Node.**
- In the UI explorer window we can navigate the UI hierarchy tree by clicking arrows in front of each node to expand the corresponding sections. This is how each line in xml fragments is mapped with the application interface.
- When you **indicate element** any element on browser using UI explorer, the element ID is seen in the last line. However if we try to refresh the web page and then **validate** the selector, we can notice it becomes invalid.
  The value of ID attribute changes every time we refresh the page so we need to indicate element again
  Example: In 2 pdf files of invoices. Open UI explorer and indicate element on 1st pdf file. We can notice that the name of the file will be stores in **Title attribute.** And then when

we select 2nd pdf i.e., indicate element on 2nd file, the selection changes, means title changes. The file name will be different, so we need to change the selector to process further.

- Some selectors contain attributes named as **IDX.** It is recommended to replace IDX with more reliable one in order for the item to be detected even if web page is updated and index is different.
- When we indicate an element, example from rpa challenge web page you indicate a box named “Company name”. Now **if we deselect ID, the IDX attribute is displayed instead.** Now we can **highlight** the element to make sure that it’s the one we clicked on.
- But if we refresh the web page without updating the selector, we can notice a different field/column is highlighted this time because the order of field has changed. In such cases, IDX attribute is unreliable.

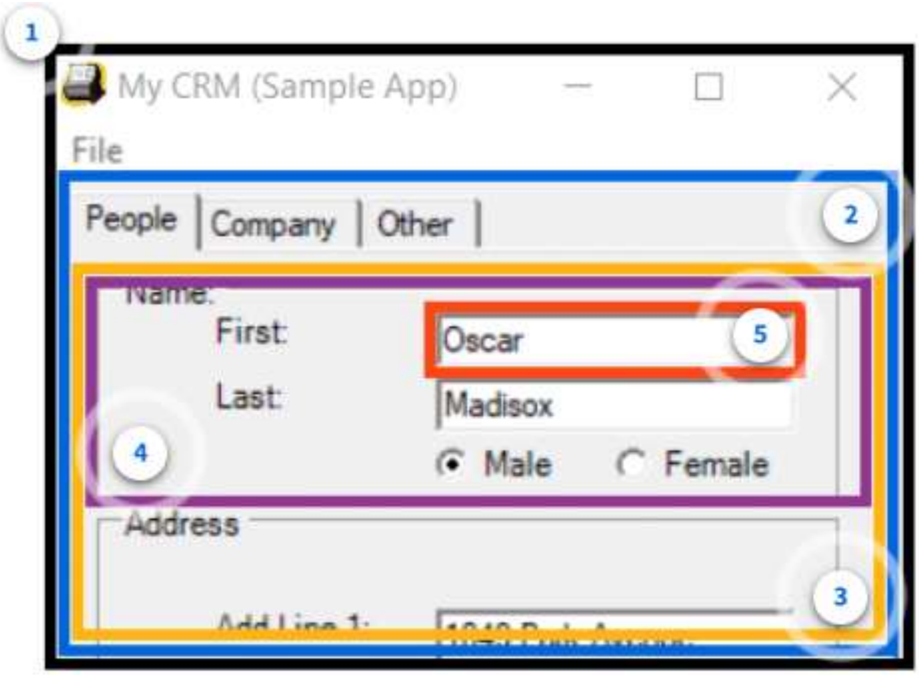
The Structure of Selectors

User interfaces (UIs) are built using a series of containers nested one inside the other. Let’s take the example of a selector for the First name input field in MyCRM Application, and try to understand the meaning of the structure.



☑	app	mycrm.exe
☑	ctrlname	Form1
☑	ctrlname	tabControl1
☑	ctrlname	tabPagePeople

```
<wnd app='mycrm.exe' ctrlname='Form1' />
<wnd ctrlname='tabControl1' />
<wnd ctrlname='tabPagePeople' />
<wnd ctrlname='groupBox5' />
<wnd ctrlname='textBoxPeopleFirstName' />
<ctrl name='First:' role='editable text' />
```



First: Oscar

Last: Madisox

● Male ○ Female

Root node

This corresponds to the highest-level container, for example an application.

<wnd app='mycrm.exe' ctrlname='Form1' />

Node 2

This identifies the entire area in which data can be edited.

<wnd ctrlname='tabControl1' />

Node 3

As you can see, the control area has 3 tabs. This node identifies the People tab.

<wnd ctrlname='tabPagePeople' />

Node 4

This identifies the Name category of the People tab. As you can see, there are other categories as well, and Address is visible.

<wnd ctrlname='groupBox5' />

Node 5

This is in fact the GUI element that interests us. It corresponds to the editable field of the First field.

<wnd ctrlname='textBoxPeopleFirstName' />
<ctrl name='First:' role='editable text' />

Tags & Attributes of Selectors

As you saw, selectors are made of nodes. And each node is made of tags and attributes. Let's take an example to explain the two. Below is a selector node.

<webctrl parentid='slide-list-container' tag='A' aaname='Details' class='btn-dwnl' />

Tags:

- Nodes in the selector XML fragment
- Correspond to a visual element on the screen
- First node is the app window
- Last node is the element itself

For example:

- wnd (window)
- html (web page)
- ctrl (control)
- webctrl (web page control)
- java (Java application control)

Attributes:

Every attribute has a name and a value. You should use only attributes with constant or known values.

For example:

- parentid='slide-list-container'
- tag='A'
- aaname='Details'
- class='btn-dwnl'

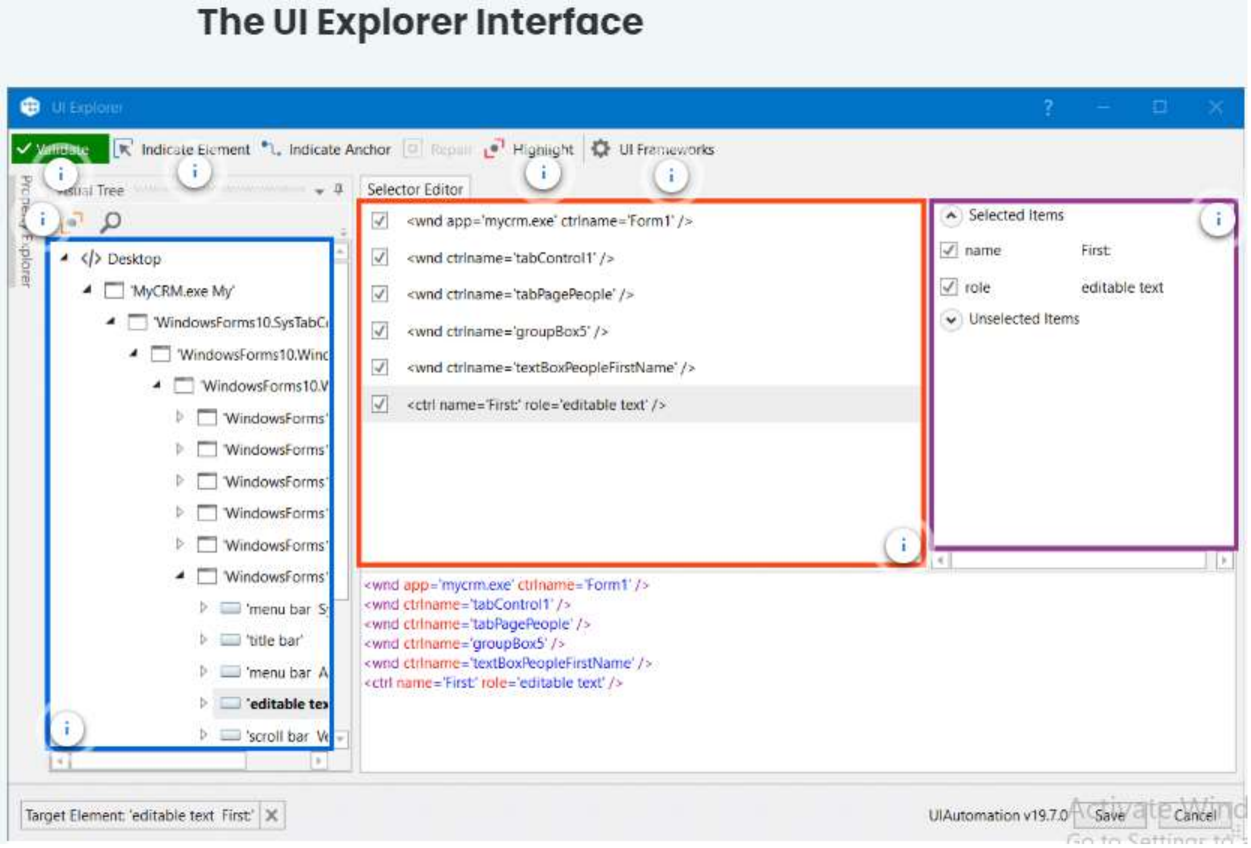
The UI Explorer

The UI Explorer is the functionality in UiPath Studio that allows analyzing and editing selectors. It contains a status button showing users the state of the selector, a Visual Tree Panel that displays a navigable UI of each application running at that moment, as well as the selected UI element. The UI Explorer displays all the available tags and attributes and gives the option to check them in or out.

What are some business scenarios in which I will use the UI Explorer?

Whenever the selectors that were automatically generated are not stable or adaptable enough, according to the issues previously highlighted:

- The selectors change from one execution to another
- The selectors might change following product updates
- The selectors use unreliable info, such as index.



The UI Explorer Interface

```
<wnd app='mycrm.exe' ctrlname='Form1' />
<wnd ctrlname='tabControl1' />
<wnd ctrlname='tabPagePeople' />
<wnd ctrlname='groupBox5' />
<wnd ctrlname='textBoxPeopleFirstName' />
<ctrl name='First' role='editable text' />
```

☑ name	First
☑ role	editable text

Target Element: 'editable text First'

UIAutomation v19.7.0

The UI Frameworks

In order to return the best selector for the element of interest, we can switch between the different UI Frameworks available in UiExplorer.

1. **Default:** This is the proprietary method which usually works correctly with all types of user interfaces.

2. **Active Accessibility :** This represents an earlier solution from Microsoft that makes apps accessible. It is recommended when using legacy software, if the default framework does not work as expected.
3. **UI Automation:** This is the improved accessibility model from Microsoft, which is recommended when using newer applications in case the default framework does not work as expected.

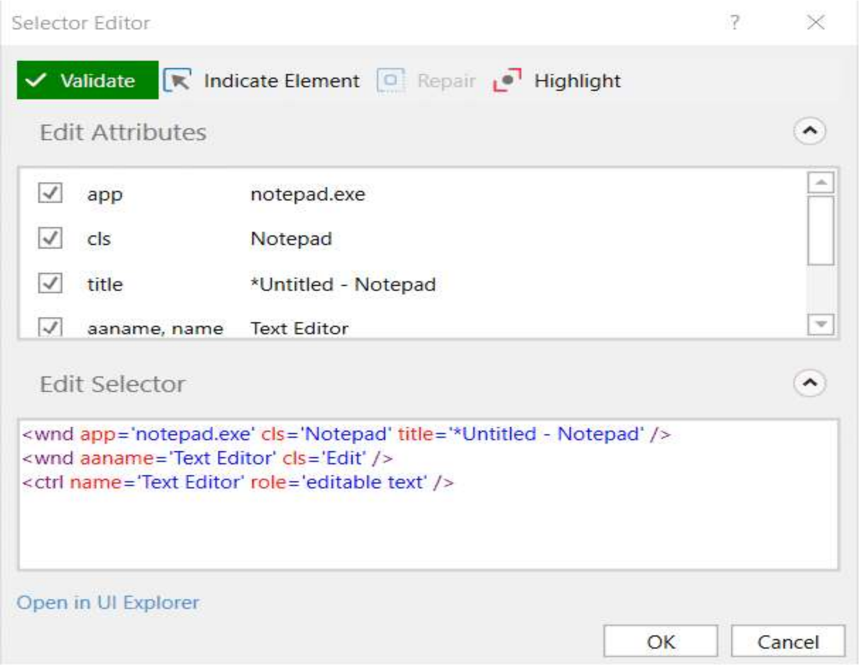
Types of Selectors:

As previously presented, selectors are automatically generated when UI elements are indicated inside activities or when the recorder is used. Knowing the difference between full and partial selectors is very important when using activities generated or added inside containers outside the containers, or the other way around.

The containers in UiPath are Attach Window, Attach Browser and Open Browser.

1. **Full Selectors:**

- Contain all the tags and attributes needed to identify a UI element, including the top-level window
- Generated by the Basic Recorder
- Best suited when the actions performed require switching between multiple windows.



Edit Attributes

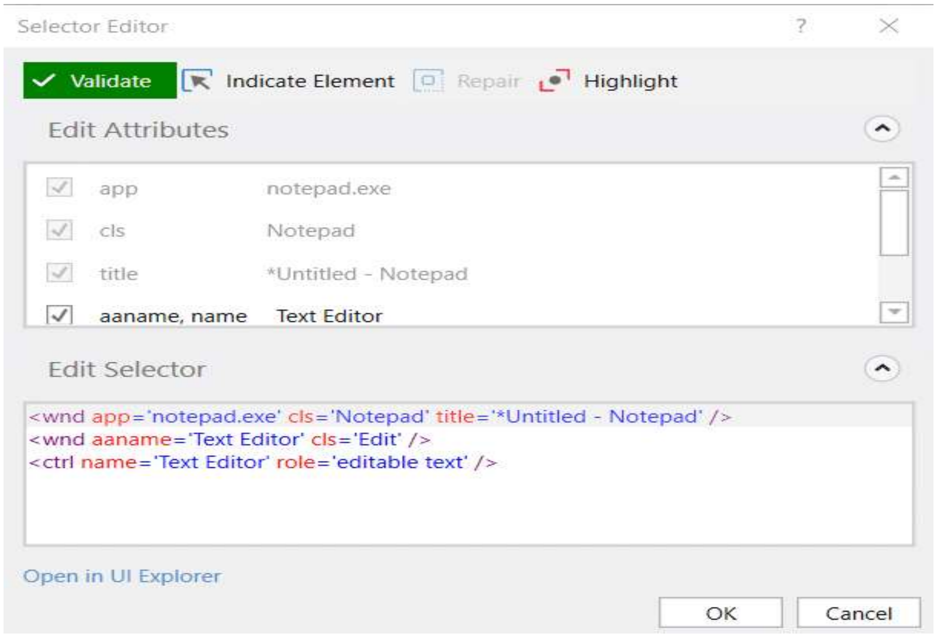
☑	app	notepad.exe
☑	cls	Notepad
☑	title	*Untitled - Notepad
☑	aaname, name	Text Editor

Edit Selector

```
<wnd app='notepad.exe' cls='Notepad' title='*Untitled - Notepad' />
<wnd aaname='Text Editor' cls='Edit' />
<ctrl name='Text Editor' role='editable text' />
```

2, Partial Selectors:

- Don’t contain the tags and attributes of the top-level window, thus the activities with partial selectors must be enclosed in containers
- Generated by the Desktop Recorder
- Best suited for performing multiple actions in the same window.



☑	app	notepad.exe
☑	cls	Notepad
☑	title	*Untitled - Notepad
☑	aaname, name	Text Editor

```
<wnd app='notepad.exe' cls='Notepad' title='*Untitled - Notepad' />
<wnd aaname='Text Editor' cls='Edit' />
<ctrl name='Text Editor' role='editable text' />
```

When are partial or full selectors used?

The best example of using a **Partial Selector** would be a simple automation where the deployed workflow only performs actions in the **same application** without shifting through multiple windows like a simple CRM.
On the other hand if the workflow would actually be required to interact with **multiple windows** like the same **CRM and a document**, which would make the UI elements required in this particular example dispersed in **multiple windows**, a **Full Selector** would be required.

Fine Tuning:

Fine-tuning is the process of refining selectors in order to have the workflow correctly executed in situations in which the generated selector is **unreliable**, **too specific** or **too sensitive** with regards to system changes.

It mainly consists of small simple changes that have a larger impact on the overall process, such as adding **wildcards**, using the **repair** function or using **variables** in selectors.

When is fine-tuning of selectors required?

- **Dynamically Selected selectors:** As it happens with some websites, the values of the attributes change with each visit.
- **Selectors being too specific**: Some selectors are automatically generated with the name of the file or with a value that changes. Here, placeholders are very useful.
- **System changes:** Some selectors contain the version of the application or another element that changes when the application is updated.
- **Selectors using IDX:** The IDX is the index of the current element in a container with multiple similar elements. This might change when a new element appears in the same container.

What are some business scenarios in which fine-tuning is required?

- The workflow uses files that have timestamp in the name;
- The environment in which a workflow was built has different parameters than the production environment (for example, the application version);
- The use of dynamic selectors would improve the reliability and robustness of the automation.

NOTE:

- We will use wildcards to replace values in selectors to improve their reliability.
- We will build a workflow and use the dynamic value of a variable in a selector to correctly identify the UI element.
- We will build a workflow in which we will use an incrementing index variable inside a Selector.

Managing difficult situations:

In most of the cases in which the selectors automatically generated are not reliable enough, fine-tuning will solve the issue. However, there are some other situations, that we call difficult.

Consider the example of a UI element that changes state, position or ID every time the workflow is run.

For these, there are other approaches like below:

- **Anchor Base:**
  This is very useful in cases in which the attribute values are not reliable (are generated at each execution, for example), but there is a UI element that is stable and is linked to the target UI element.

  The Anchor Base activity has two parts, one to locate the anchor UI element (like 'Find Element'), and the second to perform the desired activity

- **Relative Selector:**
  This activity will basically incorporate the information about the anchor's selector in the selector of the target UI element. However, the new selector will probably need additional editing, as some nodes of the first selector will still be in the new one. The solution is to have that part (like a dynamic ID) removed, and the selector will stabilize using the anchor's selector.

- **Visual Tree hierarchy:**
  The hierarchy in the Visual Tree can improve the reliability of a selector by including the tags and attributes of the element that is above in the hierarchy.

  This is very useful when the target UI element's selector is not reliable, but the selector of the UI element right above in the hierarchy is. However, again, the selector needs further editing and validation, as the dynamic part needs to be removed and, at the same time, you need to make sure that the target element can be identified with a unique attribute.

- **Find children:**
  This activity can identify all the children of an element that is more stable. Since its output is the collection of children, you will need to come up with a mechanism to identify only the target UI element (using one of its attributes, that makes is unique between the children, but wouldn't be enough to identify it universally).

PROJECT ORGANIZATION

Any UiPath implementation needs to follow clear principles.

Our purpose is to build automated processes that are:

- Reliable (robots should work as expected, with a minimum rate of unexpected errors)
- Efficient (the execution time should be minimized as much as possible)
- Maintainable (the workflows should be easy to understand and debug)
- Extensible (it should be easy to add new features)

How to choose the best project layout?

For small processes automated or parts of a larger automation project, there are 3 options of layout – Sequence, Flowchart and State Machine.

1. **Sequence**:

When to use it?

- When there’s clear succession of steps, without too many conditions (for example, a UI automation);
- Usually, Sequences are used to nest workflows and the high-level logic is handled through Flowcharts or State Machines.

What are the advantages?

- Easy to understand and follow, having a top to bottom approach;
- Great for simple logic, like searching for an item on the internet.

What are the disadvantages?

- Nesting too many conditions in the same Sequence makes the process hard to read;
- Not suitable for continuous flows.

2 **Flowchart**:

When to use it?

- When you have a complex flow with several conditions, a Flowchart is at least visually much easier to understand and follow;

- When you need a flow that runs continuously or that terminates only in several conditions.

What are the advantages?

- Easy to understand, as it is similar to logic diagrams in software computing;
- Can be used for continuous workflows.

What are the disadvantages?

- Flowcharts can be used only as the general workflow (with sequences nested inside), not for individual parts of projects (nested inside other workflows).

3 State Machine:

When to use it?
First of all, let’s understand what a State Machine is. It is an abstract machine consisting of a finite number of pre-defined states and transitions between these states. At any point, based on the external inputs and conditions verified, it can be in only one of the states.

State Machines can be used with a finite number of clear and stable states to go through. Some examples from your daily life include vending machines, elevators or traffic lights.

What are the advantages?

- Can be used for continuous workflows that are more complex;
- Transitions between states can be easily defined and offer flexibility;
- Can accommodate processes that are more complex and cannot be captured by simple loops and If statements;
- It is easier to cover all the possible cases/transitions with state machines.

What are the disadvantages?

- Longer development time due to their complexity: splitting the process into logical "states", figuring out transitions, and so on.

Note: State machines are not to be overused - they should define only the skeleton of the project

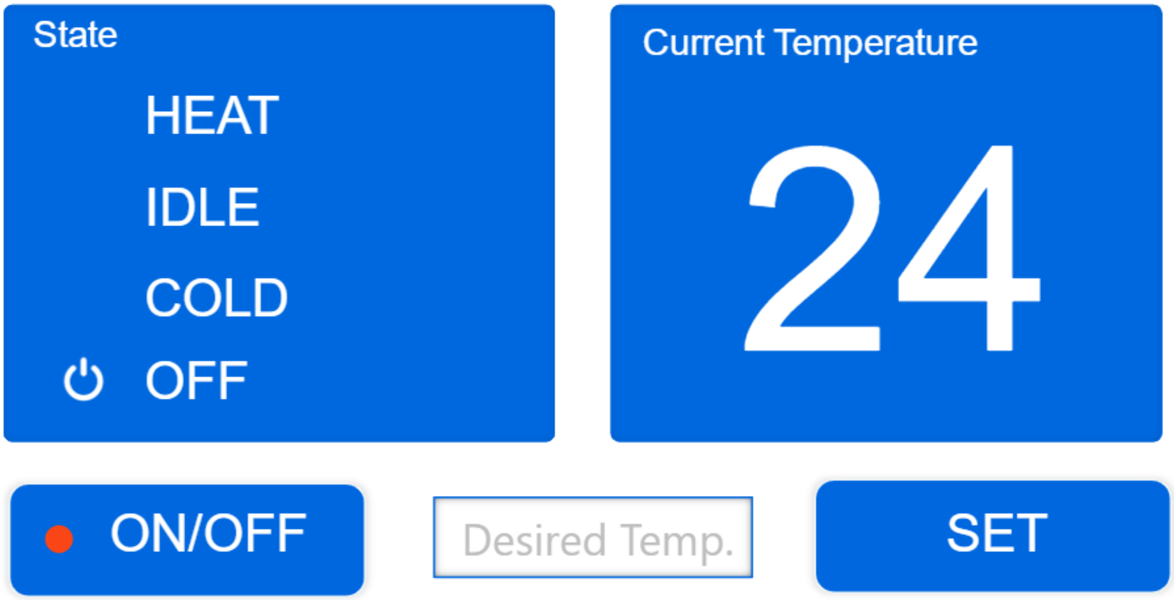
In fact, there are templates built upon State Machines especially designed to build large enterprise automations. The most commonly used is the Robotic Enterprise Framework - we have two separate lessons to cover it in depth later on in this course.

State Machine - A Real-Life Example

The representation of an air conditioner with thermostat is a good example of how a State Machine works and how it transitions between states. How to use it:

- Use the **ON/OFF button** to start or stop the air conditioner. When started, the air conditioner will go in the **IDLE state**;
- Write the desired temperature in the **'Desired Temp.' field** and click **'SET'**. Note that the working range is between 15 and 30 degrees, so any value below 15 or above 30 will be converted to 15 and 30, respectively;
- Once a temperature is set, the air conditioner will move to **HEAT** or **COLD**. Once the desired temperature is achieved, it is displayed under **'Current Temperature'** and the air conditioner moves to **IDLE**.

Navigate below to see the concepts of State Machines explained on this example.



1. IDLE, COLD, HEAT are **States** of the State Machine. OFF is the **Final State** of the State Machine; Similarly, in UiPath Studio, a process can be in a specific state at given time. This state is the one which contains the activities to be performed. If we go back to our example, in the state COLD, the air conditioning will perform the activity of cooling the air.

2. There are **Transitions** between these states:

- OFF to IDLE: clicking the ON/OFF button;
- All states (IDLE, COLD, HEAT) to OFF: clicking the ON/OFF button;

- IDLE to COLD: Desired Temperature < Current Temperature;
- IDLE to HEAT: Desired Temperature > Current Temperature;
- COLD/HEAT to IDLE: Desired Temperature = Current Temperature.

3. At any point, the Air Conditioner is either in a Transition or in a State.

4. The Air Conditioner process is **continuous**.

Important Notes about State Machines

- Naming the states in a State Machine is very important for maintenance and future development;
- In UiPath Studio, the order of the Transitions shown in each State is very important, as it matches the order in which they are evaluated.

How to break down a complex process?

Breaking an automation process in smaller workflows ensures speed of development and reliability, by allowing independent testing of components while encouraging team collaboration. Both are paramount for the success of such initiatives. Moreover, in complex automations (like most of the enterprise projects are), the question is not if it should be broken down, but how to do it.
There are multiple ways in which the workflows can be split and at least 3 factors should be considered as breakdown criteria:

The application that is being automated;

The purpose of a certain operation (login, processing, reading a document using OCR, filling in a template, and so on);

The length of each workflow.
For example, a complex process can be split into workflows for each application, and for each of those applications, split on input, processing or output. If any of these workflows are too long, they can be further split, having in mind also a purpose to do it.

Handling Data

Splitting a project into smaller workflows has an influence on how data is handled. Since variables work only inside the same workflow, having more than one workflow requires Arguments.

As you probably know, Arguments are very similar to variables – they store data dynamically, they have the same data types and they support the same methods. The main difference is the 'Direction' property, signifying the direction from/to which the data is passed. The direction can be In, Out and In/Out:

- In - if we want to use the value of the argument inside the invoked workflow;
- Out - if we want to pass the argument outside of the invoked workflow;
- In/Out - if we want to pass a variable from the parent workflow to invoked workflow, modify it in invoked workflow and then pass it on back to the parent folder.

When the 'Extract as Workflow' option is used, Variables are automatically turned into Arguments. Use 'Import Arguments' the first time you invoke the workflow to get all the available Arguments. Make sure you instantiate the Arguments with the corresponding values of the Variables using the Arguments panel.

Consider an invoked workflow that subtracts all the taxes from a gross salary. You will need at least 2 Arguments:

- IN Argument with the value of the variable containing the gross salary;
- OUT Argument to pass the value of the net salary to a variable.

Data stored in Arguments can be dynamic. Consider the following example of 2 workflows:

- A - invokes the workflow B
- B - has an IN Arguments of List data type

If the workflow B updates the list, it will also be updated in the workflow A.

Good Case Practices - Using Arguments
Use the direction as a prefix when naming and renaming arguments – i.e. in_argumentName1, out_argumentName2, io_argumentName3.

How to Re-use parts of a project?

Extracting workflows and reusing them across an automation project is a good habit for keeping a project organized, readable and sustainable. At the same time, there are cases in which workflows can be reused in different projects. Consider the sequence of logging into SAP. Every time an automation project deals with SAP, the same sequence will be needed.

Storing and reusing components in separate projects is done through process libraries. A process library is a package that contains multiple reusable components, which consist of one

or more workflows that act as individual activities. Libraries are saved as nupkg files, and then installed in different workflows using the Package Manager.
How do we identify if a part of a project is a reusable? Ideally, after the architect or the developer has separated the projects into sections, she or he will decide what section of the project can be reused. As a general practice, the graphical interaction with an application should be a reusable component because the processing of the data should not depend on how the data was obtained.

Good Case Practices - Libraries
In libraries, each workflow can be set as public or private. Public items can be used as activities in projects where the libraries are added, while the private ones can be used only inside the libraries.

How to manage versions of the same project?

Source control systems are particularly useful in larger projects, where multiple teams and individuals contribute in each stage. Source control systems allow users in different teams and locations to access the same resources and work on the same project fields. They are also used for versioning the code and maintaining the history of all the changes made during development.

Through the Team page in the backstage view, Studio supports the following source control systems:

Git
With the Git integration you can:

- Clone a remote repository
- Add a project
- Commit and push
- Copy a project to Git
- Create and manage branches
- Solve conflicts with File Diff option

TFS

- The following versions are supported:
    - 2012
    - 2013
    - 2015
    - Express 2012
    - Express 2013
    - Express 2015

- Firstly, you need to setup TFS in Studio.
- Then you can open a project or add a new project to the TFS.

SVN
With the SVN integration you can:

- Open a project from SVN
- Add a project to SVN

How to prevent and solve exceptions?

It is common for automation projects to encounter events that interrupt of interfere with the projected execution. Some of the these are identified in the development and testing phases, and handling mechanisms are implemented.

Consider an automation where the input data comes from a web form, and the application tries to match the data with the list of existing clients using the last name. But what if the user makes a mistake when spelling the name? Naturally, the name won't be recognized.

A good project design will include ways of recognizing and identifying the exception, and also patterns of action that are executed only when exceptions are caught. These can be simply stopping the execution, or explicit actions executed automatically within the workflow, or even escalating the issue to a human operator.

Predicting and treating exceptions can be done in two ways:

- At activity level, using Try/Catch blocks or Retry Scope;
- At a global level, using the Global Exception Handler.

Both will be covered in depth in the **Error & Exception Handling** lesson. At this point, it is important to be able to choose the correct type of exception, as the information will be used at a higher level later in the development to make other decisions. The categories of exceptions are:

- **Application exception:**
  The Application Exception describes an error rooted in a technical issue, such as an application that is not responding.

  Consider a project extracting phone numbers from an employee database and inserting them into a financial application. If, when the transaction is attempted, the financial application freezes, the Robot cannot find the field where it should insert the phone number, and eventually throws an error. These kinds of issues have a chance of being solved simply by retrying the transaction, as the application can unfreeze.

In managing application exceptions, it is extremely important to have good naming conventions for activities and workflows. This will help with tracking the activity that caused the exception.

- **Business Exception:**
  The Business Exception describes an error rooted in the fact that certain data which the automation project depends on is incomplete, missing, outside of set boundaries (like trying to extract more from the ATM than the daily limit) or does not pass other data validation criteria (like an invoice amount containing letters).

  Consider a robot that processes invoices, with a business rule set by the process owner that only invoices with the amount below 1 000$ must enter the automated process. For the others, a different flow is applicable, involving a human user to process them.

  In this case, retrying the transaction does not yield any chance of solving the issue, instead the business user should be notified about the pending invoice and the case should be treated as a business exception - because is an exception from the usual process flow and the validation is made explicitly by the developer inside the workflow.

  As a recommended practice, the text in the exception should contain enough information for a human user (business user or developer) to understand what happened and what actions need to be taken.

Best Practices:

Let's recap the main points to follow when aiming for a good project organization:

- Analyzing the process thoroughly, identifying the requirements and planning how the solution should look like before starting the actual development.
- Breaking the process into smaller workflows for a better understanding of the code, independent testing and reusability. This can also impact the effectiveness, as different team members can work on different (smaller) workflows.
- Grouping the workflows of your project into different folders based on the target application.
- Keeping a consistent naming convention across the project.
- Using the right type of argument (In/Out/InOut) when invoking a workflow based on the direction of information. For naming, our recommendation is CamelCase with the direction of the argument as a prefix (in_/out_/io_).
- Handling sensitive data responsibly: no credentials should be stored in the workflow directly, but rather loaded from safer places like local Windows Credential Store or Orchestrator assets and used with the Get Secure Credentials, Get Credentials and Type Secure Text activities

- Using Try/Catch blocks to predict and handle exceptions. At the same time, remember that simply using Try/Catch will only identify the error, not solve it. Thus, make sure you develop error handling mechanisms and integrate them.
- Using Global Exception Handler for situations that are global and/or less probable to happen (for example, a Windows Update pop-up).
- Using Libraries for creating and storing reusable components for your projects.
- Adding annotations to your workflows to clarify the purpose of each workflow.
- Using logs in production to get relevant information regarding critical moments or anytime specific data is needed

ERROR AND EXCEPTION HANDLING

We will start by going through the Common Exceptions encountered in UiPath, then we will cover the specific exception handling activities (**Try Catch, Retry Scope and Global Exception Handler**) and a property that many activities share and can be very valuable - **Continue on Error.**

But first, let's quickly distinguish between Errors and Exceptions:

Error:

Errors are events that a particular program can't normally deal with. There are different types of errors, based on what's causing them - for example:

- **Syntax errors**, where the compiler/interpreter cannot parse the written code into meaningful computer instructions;
- **User errors**, where the software determines that the user's input is not acceptable for some reason;
- **Programming errors**, where the program contains no syntax errors, but does not produce the expected results. This are often called **bugs**.

Exceptions:

Exceptions are events that are recognized (caught) by the program, categorized and handled. More specifically, there is a routine configured by the developer that is activated when an exception is caught. Sometimes, the handling mechanism can be simply stopping the execution.

Some of the exceptions are linked to the systems used, while others are linked to the logic of the business process.

Common Exceptions:

Below you can find the most common exceptions that you can encounter in projects developed with UiPath. As a general note, all exceptions are types derived from **System.Exception**, so using this generic type in a Try Catch, for example, will catch all types of errors.

- **NullReferenceException** - This error usually occurs when using a variable with no set value (not initialized).
- **IndexOutOfRangeException** - Occurs when the index of an object is out of the limits of the collection.
- **ArgumentException** - Is thrown when a method is invoked and at least one of the passed arguments does not meet the parameter specification of the called method.
- **SelectorNotFoundException** - Is thrown when the robot is unable to find the designated selector for an activity in the target app within the TimeOut period.

- **ImageOperationException** - Occurs when an image is not found within the TimeOut period.
- **TextNotFoundException** - Occurs when the indicated text is not found within the TimeOut period.
- **ApplicationException** - describes an error rooted in a technical issue, such as an application that is not responding.

NOTE: Business Rule Exceptions are separate from all the System Exceptions listed above. These describe errors rooted in the fact that certain data which the automation project depends on is incomplete, missing, outside of set boundaries (like trying to extract more from the ATM than the daily limit) or does not pass other data validation criteria (like an invoice amount containing letters).

The Business Rule Exceptions will not be thrown by using the generic System.Exception in a Try Catch activity. The mechanism of handling this exception has to be separately defined by the developer (based on the rules set by the process owner), or it can be reduced to stopping the execution of the process, by using a simple **Throw** activity, for example.

TRY Catch:

This activity catches a specified exception type in a sequence or activity, and either displays an error notification or dismisses it and continues the execution.
As a mechanism, Try Catch runs the activities in the Try block and, if an error takes place, executes the activities in the Catch block. The Finally block is only executed when no exceptions are thrown or when an exception is caught and handled in the Catch block (without being re-thrown).

TRY: The activities performed which have a chance of throwing an error.
Catch: The activity or set of activities to be performed when an error occurs. Please note that multiple errors and corresponding activities can be added in this block.
Finally: The activity or set of activities to be performed after the Try Catch block. This section is executed only when no exceptions are thrown or when an error occurs and is caught in the Catch section.

Retry Scope:

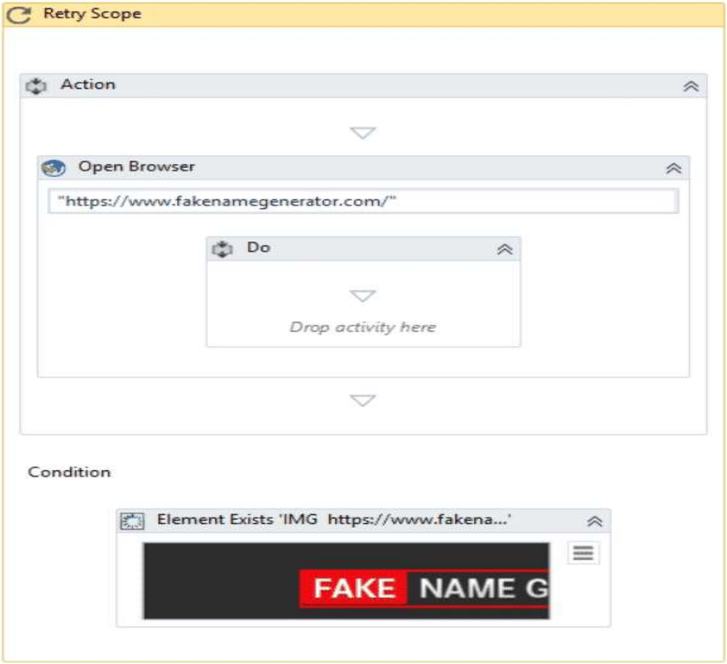
The Retry scope activity retries the contained activities as long as the condition is not met or an error is thrown.

The Retry Scope activity is used for catching and handling an error, which is why it's similar to Try Catch. The difference is that this activity simply retries the execution instead of providing a more complex handling mechanism.

It can be used without a termination condition, in which case it will retry the activities until no exception occurs (or the provided number of attempts is exceeded).

Additional Properties

- **NumberOfRetries -** The number of times that the sequence is to be retried.
- **RetryInterval -** Specifies the amount of time (in seconds) between each retry.



Retry Scope

Action

Open Browser

"https://www.fakenamegenerator.com/"

Do

Drop activity here

Condition

Element Exists 'IMG https://www.fakena...'

ContinueOnError Property:

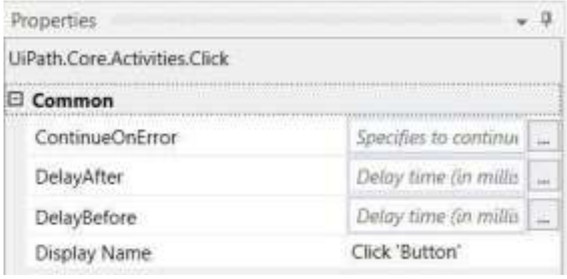
Continue on Error is a property that specifies if the execution should continue even when the activity throws an error.

Keep in mind that, if the ContinueOnError is set to True on an activity that has a scope (such as Attach Window or Attach Browser), then all the errors that occur in other activities inside that scope are also ignored.

Having this property set to true is not recommended in all situations, but there are some situations in which it makes sense, such as:

- while using data scraping - so the activity doesn't throw an error on the last page (when the selector of the 'Next' button is no longer found);
- when we are not interested in capturing the error, but simply in the execution of the activity.

This field only supports Boolean values (True, False). The default value is False, thus, if the field is blank and an error is thrown, the execution of the project stops. If the value is set to True, the execution of the project continues regardless of any error.



Properties

UiPath.Core.Activities.Click

⊟ **Common**

ContinueOnError	Specifies to continu
DelayAfter	Delay time (in millis
DelayBefore	Delay time (in millis
Display Name	Click 'Button'

Global Exception Handler

The Global Exception Handler is a type of workflow designed to determine the behavior when encountering an execution error at the project level. This is why only one Global Exception Handler can be set per automation project.

Only uncaught exceptions will reach the Exception Handler. If an exception occurs inside a Try Catch activity and it is successfully caught and treated inside the Catch block (and not re-thrown), it will not reach the Global Exception Handler.

A Global Exception Handler can be created either by starting a new project with this type, or by setting an existing project as Global Exception Handler from the Project panel.

How does is work?

The Global Exception Handler has 2 predefined arguments, that shouldn't be removed:

- **errorInfo**, with the In direction - contains the error that was thrown and the workflow that failed;
- **result**, with the Out direction - will store the next behavior of the process when it encounters the error.

The Global Exception Handler contains the predefined 2 actions below (that can be removed). Other actions can be added.

Log Error

This part simply logs the error. The developer gets to choose the logging level: Fatal, Error, Warning, Info, and so on.

Choose Next Behavior

Here the developer can choose the action to be taken when an error is encountered during execution:

- **Continue** - The exception is re-thrown;
- **Ignore** - The exception is ignored, and the execution continues from the next activity;
- **Retry** - The activity which threw the exception is retried;
- **Abort** - The execution stops after running the current handler.

DEBUGGING

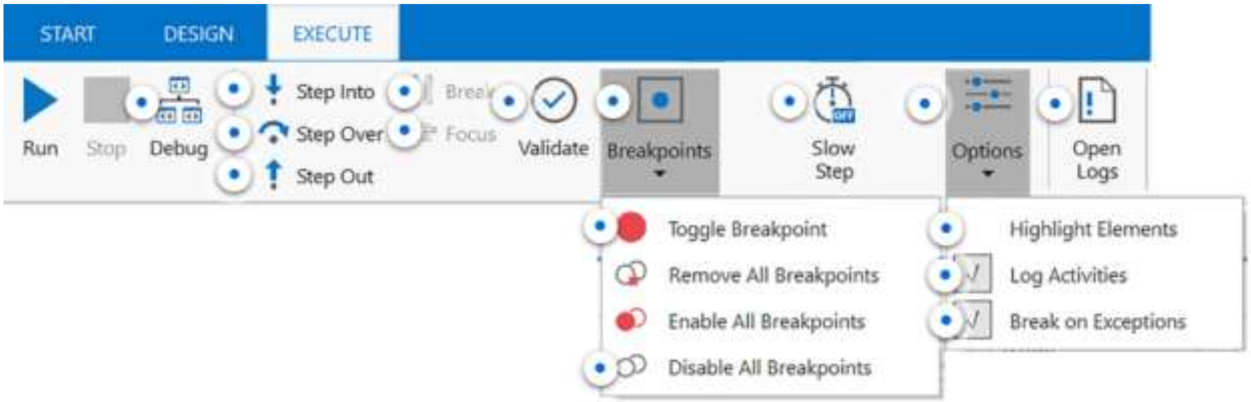
In this lesson we will cover one of the most important aspects of how to get your automation production-ready: how to anticipate, detect and resolve errors in your workflows.

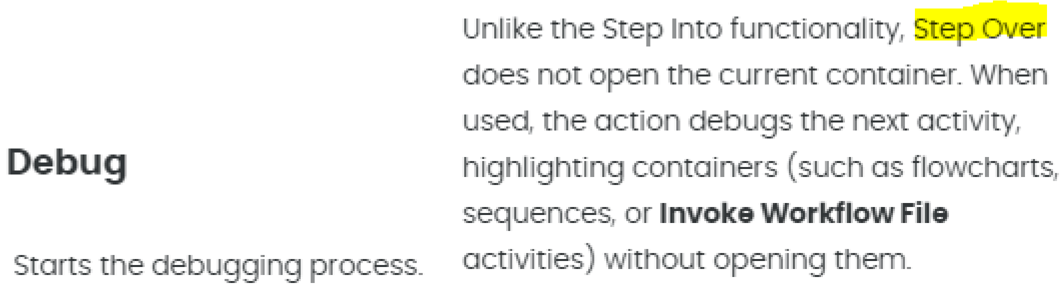
Debugging is the process of identifying and removing errors from a given project.

UiPath Studio comes with a **debug component** that helps find and locate problems easily in complex workflows. This is useful for viewing the execution of each activity, to verify what data it gets and in checking if there are errors in producing outputs.

The tool encapsulates a real-time engine that checks for errors while working with your workflow. Whenever an activity has errors, UiPath Studio Process Designer notifies and gives you details about the issues encountered.

Multiple debugging features are available under the **Execute** ribbon, select the markers to learn more about each feature:

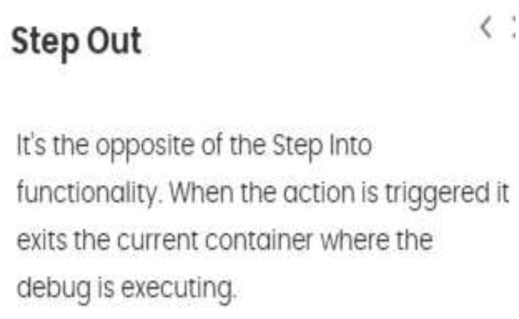




Debug

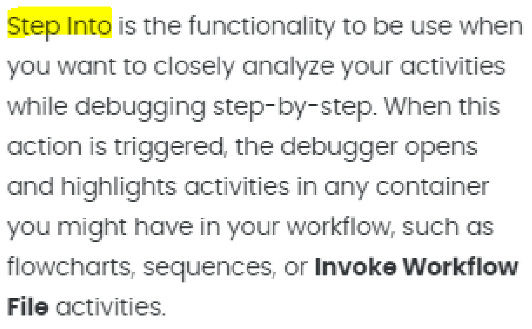
Starts the debugging process.

Unlike the Step Into functionality, Step Over does not open the current container. When used, the action debugs the next activity, highlighting containers (such as flowcharts, sequences, or **Invoke Workflow File** activities) without opening them.

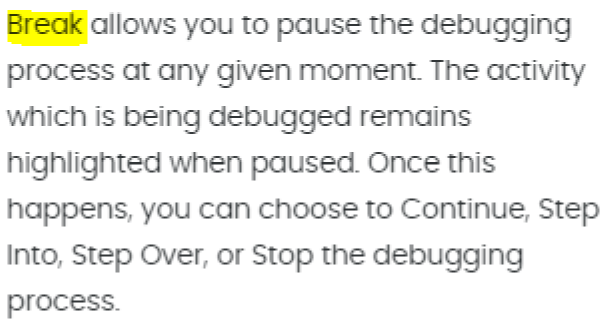


Step Out

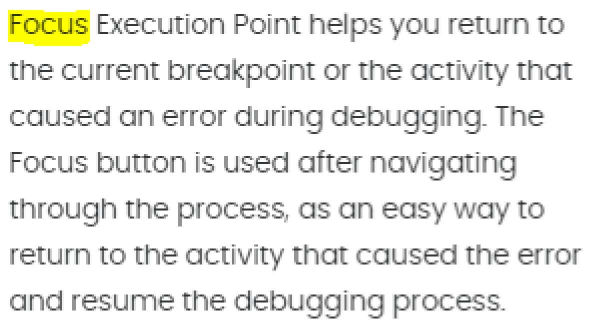
It's the opposite of the Step Into functionality. When the action is triggered it exits the current container where the debug is executing.



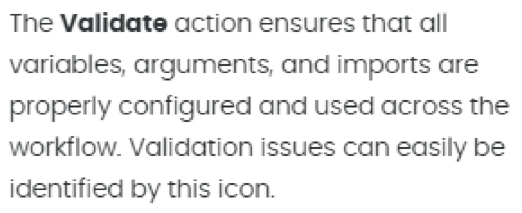
Step Into is the functionality to be use when you want to closely analyze your activities while debugging step-by-step. When this action is triggered, the debugger opens and highlights activities in any container you might have in your workflow, such as flowcharts, sequences, or **Invoke Workflow File** activities.



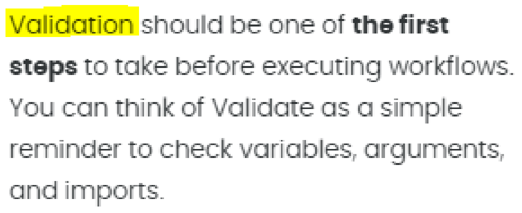
Break allows you to pause the debugging process at any given moment. The activity which is being debugged remains highlighted when paused. Once this happens, you can choose to Continue, Step Into, Step Over, or Stop the debugging process.



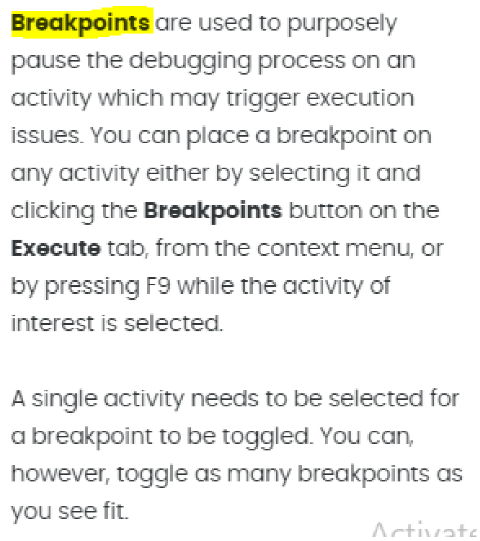
Focus Execution Point helps you return to the current breakpoint or the activity that caused an error during debugging. The Focus button is used after navigating through the process, as an easy way to return to the activity that caused the error and resume the debugging process.



The **Validate** action ensures that all variables, arguments, and imports are properly configured and used across the workflow. Validation issues can easily be identified by this icon.

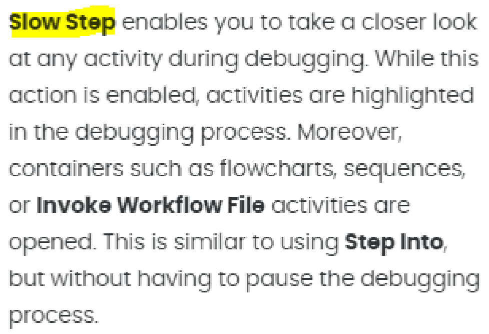


Validation should be one of **the first steps** to take before executing workflows. You can think of Validate as a simple reminder to check variables, arguments, and imports.

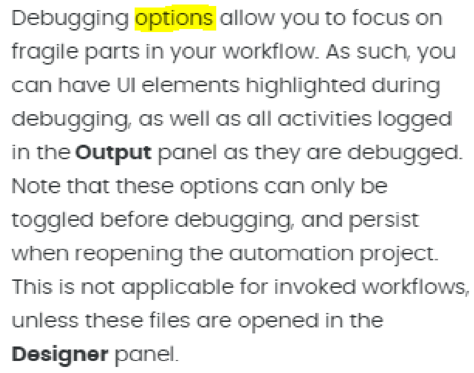


Breakpoints are used to purposely pause the debugging process on an activity which may trigger execution issues. You can place a breakpoint on any activity either by selecting it and clicking the **Breakpoints** button on the **Execute** tab, from the context menu, or by pressing F9 while the activity of interest is selected.

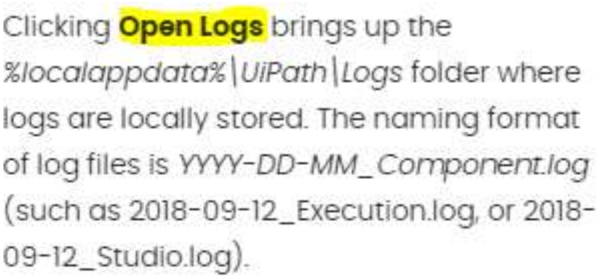
A single activity needs to be selected for a breakpoint to be toggled. You can, however, toggle as many breakpoints as you see fit.



Slow Step enables you to take a closer look at any activity during debugging. While this action is enabled, activities are highlighted in the debugging process. Moreover, containers such as flowcharts, sequences, or **Invoke Workflow File** activities are opened. This is similar to using **Step Into**, but without having to pause the debugging process.

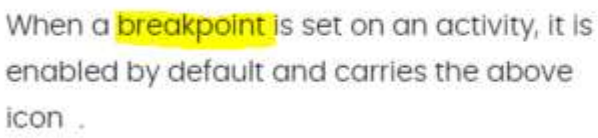


Debugging options allow you to focus on fragile parts in your workflow. As such, you can have UI elements highlighted during debugging, as well as all activities logged in the **Output** panel as they are debugged. Note that these options can only be toggled before debugging, and persist when reopening the automation project. This is not applicable for invoked workflows, unless these files are opened in the **Designer** panel.

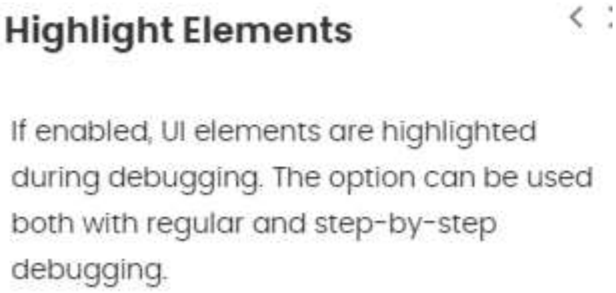


Clicking **Open Logs** brings up the *%localappdata%\UiPath\Logs* folder where logs are locally stored. The naming format of log files is *YYYY-DD-MM_Component.log* (such as 2018-09-12_Execution.log, or 2018-09-12_Studio.log).

Toggle breakpoint



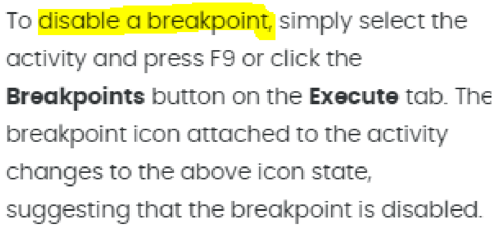
When a breakpoint is set on an activity, it is enabled by default and carries the above icon .



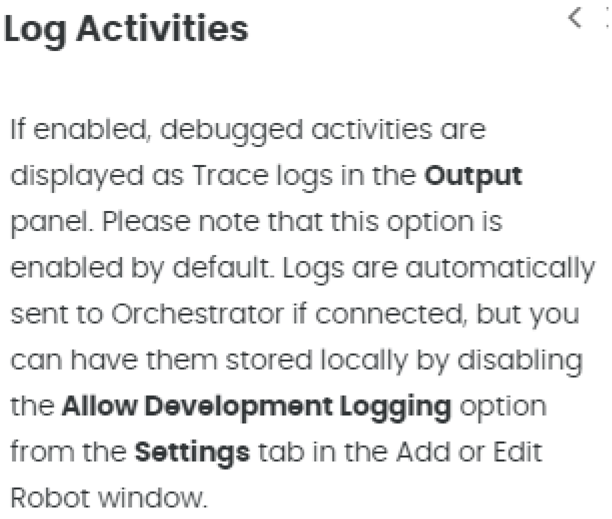
Highlight Elements

If enabled, UI elements are highlighted during debugging. The option can be used both with regular and step-by-step debugging.

Disable al breakpoints

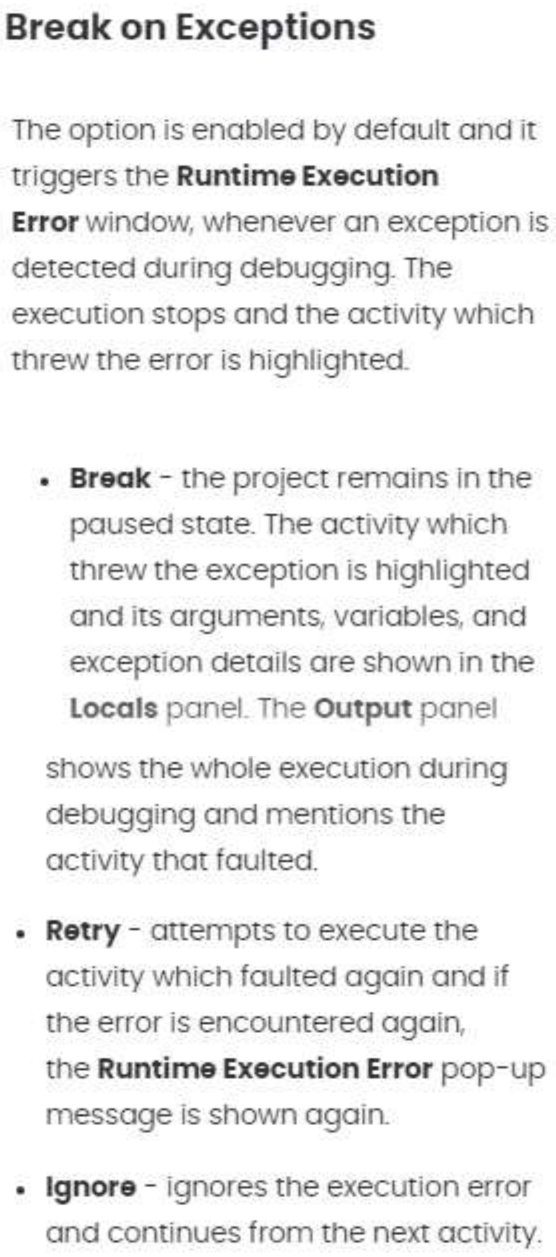


To disable a breakpoint, simply select the activity and press F9 or click the **Breakpoints** button on the **Execute** tab. The breakpoint icon attached to the activity changes to the above icon state, suggesting that the breakpoint is disabled.



Log Activities

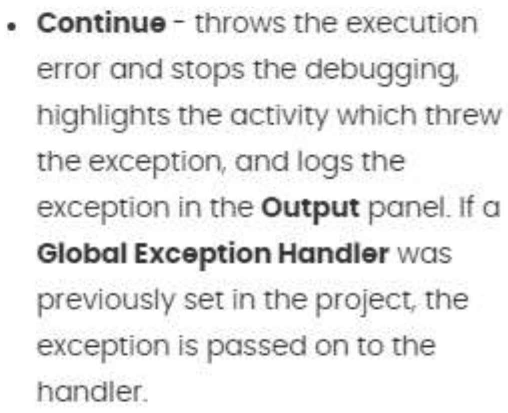
If enabled, debugged activities are displayed as Trace logs in the **Output** panel. Please note that this option is enabled by default. Logs are automatically sent to Orchestrator if connected, but you can have them stored locally by disabling the **Allow Development Logging** option from the **Settings** tab in the Add or Edit Robot window.



Break on Exceptions

The option is enabled by default and it triggers the **Runtime Execution Error** window, whenever an exception is detected during debugging. The execution stops and the activity which threw the error is highlighted.

- **Break** - the project remains in the paused state. The activity which threw the exception is highlighted and its arguments, variables, and exception details are shown in the **Locals** panel. The **Output** panel shows the whole execution during debugging and mentions the activity that faulted.
- **Retry** - attempts to execute the activity which faulted again and if the error is encountered again, the **Runtime Execution Error** pop-up message is shown again.
- **Ignore** - ignores the execution error and continues from the next activity.



- **Continue** - throws the execution error and stops the debugging, highlights the activity which threw the exception, and logs the exception in the **Output** panel. If a **Global Exception Handler** was previously set in the project, the exception is passed on to the handler.

Good Case Practices:

Here are a few good case practices when reviewing somebody else's workflow or preparing your own workflow for code review:

- Give activities and containers **descriptive names**. This will make it easier for other RPA developers to understand your workflows and easier for you to identify where an error is thrown.

- Give variables descriptive names and use **lower camel casing** (e.g. userName, valuesTable, cashIn). This will make them easier to read and it will make the workflow easier to understand.

- When necessary, use **Annotations** to help your future self and other RPA developers understand the workflow. To leave an annotation, right click an activity and select **Annotations > Add Annotation.**

- Make sure you don't use **variables** with the **same name** but **different scopes**. When this occurs, the variable defined in the most specific scope is used. Variables are always created in the most specific scope selected.

- Unless there's a very good reason for it, the **ContinueOnError** property on your activities should be set to blank or False.

- Break down a complex logic into several workflows and use Invoke Workflow. We'll learn more about this in the Project Organization lesson.

ORCHESTRATOR

Introduction:

What is Orchestrator?

Orchestrator is the component of UiPath Suite through which the automation workflows developed in Studio are published, assigned to robots and executed. It comes in the form of a web application that enables the management of robots, activity packages, data to be processed, execution schedules, as well as other assets.

Orchestrator is ideal for large deployments of robots covering complex processes, but it can also be deployed in scenarios dealing with short and repetitive processes and fewer robots.

What are Orchestrator's capabilities?

- **Provisioning** - creates and maintains the connection with the robots.
- **Deployment** - ensures the delivery of the workflows for execution, either immediately or using schedules.
- **Configuration** - enables the creation, configuration and maintenance of groups of robots and the execution of tasks.
- **Queues** - the data that needs to be processed is broken down to indivisible operations called transactions. Queues can store any number of transactions and they facilitate their distribution, execution and monitoring.
- **Monitoring** - keeps track of robot identification data and maintains user permissions.
- **Logging** - stores and indexes the logs to an SQL database and/or ElasticSearch.
- **Inter-connectivity** - acts as the centralized point of communication for 3rd party solutions or applications, and can be used for storing activity packages, libraries and other assets (such as credentials).

What are the benefits of Orchestrator?

As you know, workflows can be executed on a local machine to perform simple tasks. In business contexts though, all the capabilities listed in the previous section are needed for effectively managing a large number of robots that have to execute numerous tasks. All these capabilities translate into tangible benefits, such as:

1. **Accessibility and version control**

Workflows can be published as packages and stored in Orchestrator, at version level and with the developer's release notes. Different versions can be distributed to robots for execution.

Libraries can also be published and stored in Orchestrator. From there, they can be accessed at any time, used in development and distributed to robots.

2. Accessibility and version control

Workflows can be published as packages and stored in Orchestrator, at version level and with the developer's release notes. Different versions can be distributed to robots for execution.

Libraries can also be published and stored in Orchestrator. From there, they can be accessed at any time, used in development and distributed to robots.

3. Efficient planning and execution

Orchestrator allows execution of tasks according to schedules that can accommodate various scenarios (like non-working days), to choose execution targets from the available robots and even to create continuous execution loops even outside business hours.

4. Securely storing assets

As a good practice, configuration assets, credentials and data that change frequently must not be stored in the workflows. Orchestrator provides a secure way of storing assets and distributing them to robots according to different scenarios.

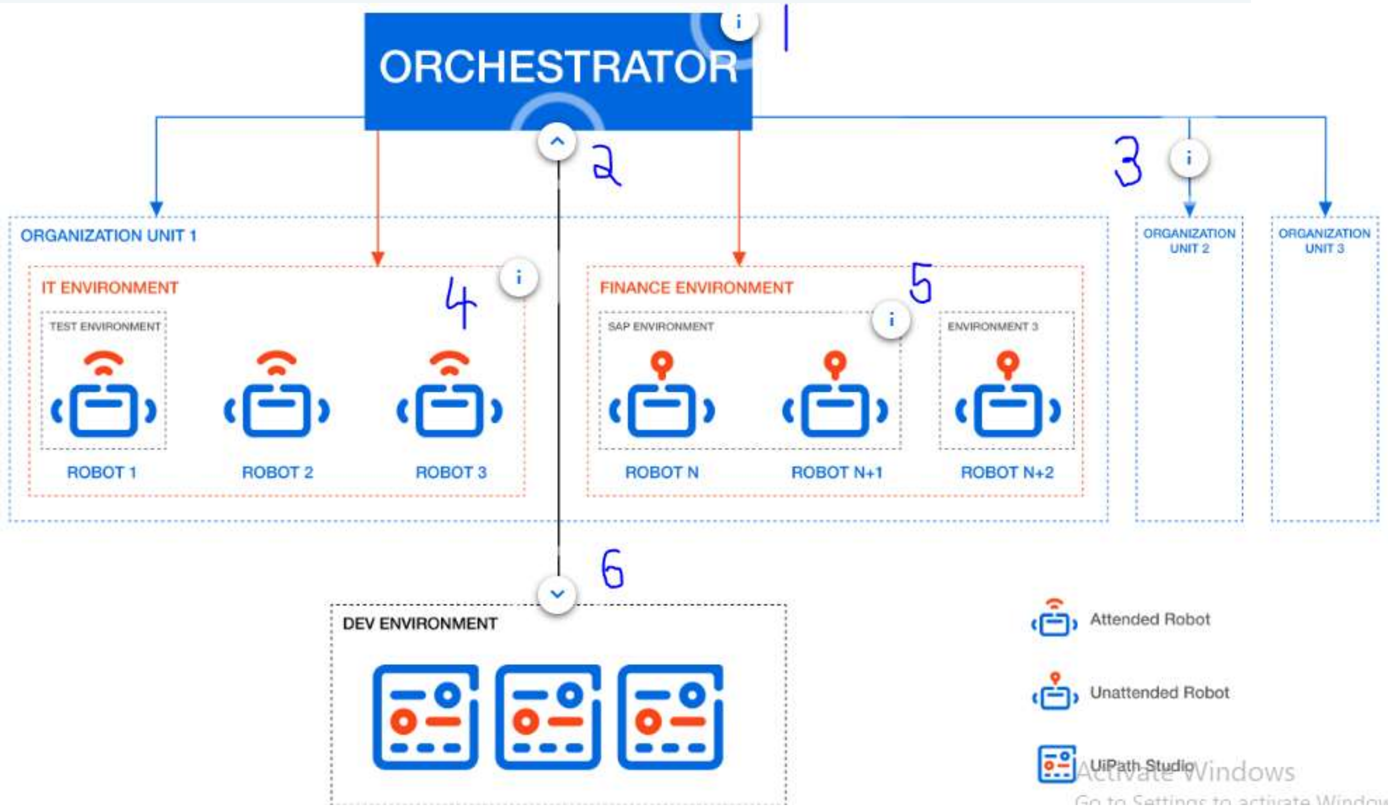
5. Monitoring

Robots, processes and task execution can be monitored via Orchestrator, which can enable quick reaction in case of any error, and also provides the means for accurate reporting and auditing of the robot work.

Orchestrator Overview:

A typical Orchestrator deployment can look like this:

1. This is where logs, queues, packages and assets are stored in a centralized way.
2. Publish packages to orchestrator.
3. One orchestrator instance can contain multiple organization units, allowing separation of queues, packages, jobs, schedules and so on.
4. Run a job manually on a specific environment.
5. Schedule processes to run on a specific environment.
   Get each transaction item from the queue and allocate to available robots in an environment.
6. Get packages from Orchestrator (i.e. libraries)



Below is a glossary of concepts related to Orchestrator.
We will cover each of them in depth in the following chapters.

1. **Robot**:
   The Robot is UiPath’s execution agent that enables you to run workflows built in Studio. It can be of several types, as you will see in the next chapter.
2. **Environment**:
   An environment is a group of robots configured in Orchestrator. Processes can be allocated to individual robots, but it's more effective to allocate them to environments. A robot can be part of more than one environment, provided they are in the same service.
3. **Organization Unit:**
   In general, it is an entity in Orchestrator that corresponds to a business unit. The same Orchestrator instance can have multiple organization units configured, each of them having separate robots, environments, queues, assets and so on.

The Community Edition plan won't allow the creation of other organization units than the default one created.

4. **Package:**
   A project developed in UiPath Studio that is published to Orchestrator. Multiple versions of the same project can be stored and used.

5. **Process**:
   It is a package that has been allocated to a certain environment.

6. **Job**:
   A job is a process that has been sent for execution to some or all of the robots in the environment.

7. **Schedule**:
   A process that is configured for an execution that is not immediate, but according to a schedule. Multiple configurations are possible when it comes to schedules.

8. **Asset**:

An asset is a piece of data stored in Orchestrator for the use of robots. There are four types of assets:

- **Text** - stores only strings (it is not required to add quotation marks);
- **Bool** - supports true or false values;
- **Integer** - stores only whole numbers;
- **Credential** - contains usernames and passwords that the Robot requires to execute particular processes, such as login details for ERP systems.

9. **Queue**:
   A queue is a sequence of transactions that is built in Orchestrator, and then used to dispatch to robots for processing.

Robots and Environments in Orchestrator:

Robots

What is a robot?

The robot is UiPath's execution agent that enables you to run workflows built in Studio. The installation of the Studio comes with a robot that is triggered when the 'Run' button is clicked when a project is open.

Provisioning robots in Orchestrator allows better package distribution capabilities, better scheduling, and enables the use of assets and queues. All these will take an implementation much closer to a business context.

Types of robots

Before covering robot provisioning, let's spend some time to talk about the types of robots that UiPath offers.

Based on how they can be used, there are 4 types of robots:

1. **Attended Robot:**

This type of Robot is triggered by user events, and operates alongside a human, on the same workstation. Attended Robots are used with Orchestrator for a centralized process deployment and logging medium.

2. **Unattended Robot:**

Robots run unattended in virtual environments and execute any number of processes. On top of the Attended Robot capabilities, Orchestrator covers execution, monitoring, scheduling and providing support for work queues.

3. **Development Robot:**

Has the features of an Unattended Robot, but it should be used only to connect Studio to Orchestrator, for development purposes.

4. **Non-Production Robot:**

Similar to Unattended Robots, but they should be used only for development and testing purposes.

According to the Robot/Machine interaction, there are two kinds of robots:

1. **Standard Robot**

Works on a single Standard Machine only. It is a good choice when the machine on which the robot runs is known and will never change.

2. **Floating Robot**

Works on any machine defined in Orchestrator. It is a good choice when human users work in shifts on the same machine or on different machines, and when virtual machines are being regenerated often.

Provisioning robots in Orchestrator

In order for robots to perform jobs, they have to be provisioned in Orchestrator, together with the machines that they will be using. It is important to remember that the machine name is mandatory for standard robots, and login credentials may be needed for unattended robots. Only the Orchestrator admins have rights to provision robots and machines.

Robots can be connected:

- directly from the Orchestrator Settings window
- from the Command Line
- or automatically enrolled

Usually, it is recommended to first try and connect the robot from the Orchestrator window. If issues are encountered when doing so, the next step would be to perform the connection from the Command Line.

Environments

An environment is a grouping of Robots, that is used to deploy processes. The dedicated tab in the Robots menu can be used to create, modify and delete environments.

A robot can be added in more than one environment, provided that they are under the same service.

In a real scenario, the Finance service could create environments at business unit level (Accounts Payable, Accounts Receivable), at application level (SAP, electronic banking, settlement application) or sub-process. It is very important to remember that services have the robots, environments, queues, processes and so on totally separated.

Important notes

- The Community Edition doesn't allow multiple services, but other plans do. Services allow separation of all data under the same Orchestrator instance.
- Template machines should be used instead of Standard Machines when the name changes every time (like in the case of Virtual Desktop Infrastructure)

Process Execution in Orchestrator

Packages

Packages consist of one or more automation workflows published from Studio. They can be published on the local machine or directly to the Orchestrator. Similarly, in Orchestrator, packages can be manually uploaded.

In Orchestrator, versions of the same package are stored automatically. The version and release notes can be accessed anytime by selecting Processes from the menu on the left in Orchestrator, and navigating to the Packages tab.

Package versions can be active or inactive:

- active: deployed to at least an environment;
- inactive: not deployed (this type can be deleted).

Packages can easily be updated following the same steps as for publishing. It will be automatically detected that it's a new version of an existing package.

Processes

A process represents the association between a package and an environment. Processes can be accessed from the menu on the left. In order to create a new Process, simply click on the '+', select a package available in Orchestrator, select the environment and give it a description.

If the Main.xaml of the process has In, Out, or In/Out arguments, they are displayed on the Parameters tab of the View Processes window. In Orchestrator, they become input and output parameters and they can be configured from the Parameters tab.

Jobs & Schedules

Once a process is created, its execution can be triggered. There are 3 ways to do it:

Using Jobs (immediately)

To start a job, simply navigate to Jobs from the menu on the left, choose a process and either select the robots you want to execute the job from the environment, or allocate the job dynamically. A job allocated directly to certain robots will have priority over jobs allocated dynamically, but jobs allocated dynamically are executed immediately when any robot in the environment becomes available.

When starting a job, you have the option to set input parameters or display the output parameters if they exist as arguments in the Main.xaml of the package.

There are 2 options to stop a job:

- using Stop Job, that will stop it when a Should Stop activity in the workflow is encountered
- using Kill Job, that will stop it immediately.

Using Schedules (planned)

Schedule offer multiple configuration options, such as:

- choosing a regular interval to execute the process
- setting up parameters (arguments in the Main.xaml in Studio)
- selecting the robots to execute (similarly to Jobs) - all robots, selected robots or dynamically
- setting up non-working days and other parameters.

To schedule a process execution, navigate to the Schedules page, create a new Schedule and configure its parameters. Once you are done, click 'Create'.

From the Robot Tray

Running processes from the Robot Tray can be very useful for testing purposes or simple automations created for parts of your job or personal matters.

To run a process directly from the Robot Tray:

- make sure the package is assigned to the environment in which your attended robot is part of;
- open the robot;
- if the package has an update available, you will see an up arrow from which you can update it in your robot;
- once this is done, you can run the process by clicking the 'Play icon'.

Assets:

What are Assets?

Assets are shared variables or credentials that are stored in the Orchestrator and used by the robots in different automation projects. They can be considered a data repository that the robots can access when running processes, based on clear instructions.

There are four types of assets:

- **Text** - the equivalent of String (it is not required to add quotation marks)
- **Bool** - supports true or false values
- **Integer** - stores only whole numbers
- **Credential** - contains usernames and passwords that the Robot requires to execute particular processes, such as login details for SAP or SalesForce.

What are some business scenarios in which I will use Assets?

- When data doesn't change from one execution of a process to another, so it doesn't make sense to assign it every time from the Parameters tab. At the same time, data that may change shouldn't be stored in the workflow;
- Whenever robots need credentials to access applications in an automation scenario. The Credentials stored as Assets are encrypted with the AES 256 algorithm.

How are Assets created and used?

In Orchestrator:

- Assets can be created from the dedicated area;
- The name and the data type have to be provided. By using the tabs, assets can be configured as follows:
  - Single Value - can be accessed and used by all Robots;
  - Value Per Robot - each value provided can be accessed only by the indicated Robot.
- Assets can be modified or deleted from the same menu.

In Studio:

- For **credentials**, the 'Get Credential' activity has to be used;
- For all the **other types of assets**, the 'Get Asset' activity is used.

Queues:

What are Queues?

Queues are containers that can hold an unlimited number of items. Queues in Orchestrator will store items and allow their distribution individually to robots for processing, and monitoring the status of the items based on the process outcomes.

Working with Queues brings a series of advantages, especially for large automations with multiple types of elements underlined by a complex logic:

- Centralized depository of work items;
- Reporting capabilities at individual item level, as well as queue level;
- Effectiveness of item distribution process - anytime a robot becomes available, the queue item in line is dispatched;
- Unitary logic of item distribution - such as First In, First Out.

What are some business scenarios in which I will use Queues?

Items in the Orchestrator Queues are known as transactions. They are meant to be indivisible units of work - a customer contract, an invoice, a complaint, and so on.

Working with Queues is very useful for large automations mainly, where the number of items is high and the distribution process may become problematic. Consider the following examples:

- New customer enrollment forms for a retail company - these may come from different sources (online, partners, own shops, call center) and having a smooth process of adding them to the processing line is crucial. Working with Queues will ensure that these are processed within the SLA time constraints;
- The complaint process of a worldwide retailer, having contact centers in many locations across the world - working with Queues will ensure that the items are centralized in a single depository and distributed to the available resources as they become available.

Working with Queues

Queues are easily created in Orchestrator. When they are created, queues are empty, but there are specific activities in the UiPath Studio to make the robots populate Queues. Bulk upload is also supported directly in Orchestrator, from .csv files.

When creating a queue, you set the **maximum number of retries** (the number of times you want a queue item to be retried) and the **Unique Reference** field (select Yes if you want the transaction references to be unique). Once a queue was created, these settings can not be modified

Queues are very important for the Dispatcher & Performer model, in which the two main stages of a process involving queues is separated:

- the stage in which data is taken and fed into a queue in Orchestrator, from where it can be taken and processed by the robots. This is called Dispatcher;
- the stage in which the data is processed, called Performer.

Below are the main activities used to program the robots to work with queues:

1. **ADD Queue item:**
   When encountering this activity in a workflow, the robot will send an item to the designated Queue and will configure the time frame and the other parameters.
2. **ADD transaction item:**
   The robot adds an item in the queue and starts the transaction with the status 'In progress'. The queue item cannot be sent for processing until the robot finalizes this activity and updates the status.
3. **Get Transaction item:**
   Gets an item from the queue to process it, setting the status to 'In progress'.
4. **Postpone Transaction item:**
   Adds time parameters between which a transaction must be processed.
5. **Set transaction progress:**
   Enables the creation of custom progress statuses for In Progress transactions. This can be useful for transactions that have a longer processing duration, and breaking down the workload will give valuable information.
6. **Set Transaction status:**

Changes the status of the transaction item to Failed (with an Application or Business Exception) or Successful. As a general approach, a transaction failed due to Application Exceptions will be retried, and a transactions failed due to Business Exceptions will not be retried.

A Queue item can have one of the following statuses:

- **New** - just added to the queue with Add Queue Item (or the item was postponed or a deadline was added to it).
- **In Progress** - the item was processed with the Get Transaction Item or the Add Transaction Item activity;
- **Failed** - the item did not meet a business or application requirement within the project;
- **Successful** - the item was processed;
- **Abandoned** - the item remained in the In Progress status for a long period of time (approx. 24 hours) without being processed;
- **Retried** - the item failed with an application exception and was retried (at the end of the process retried, the status will be updated to a final one - Successful or Failed)
- **Deleted** - the item has been manually deleted from the Transactions page

COMPLETE NOTES ON RE FRAMEWORK IS AVAILABLE IN THE 2nd DOCUMENT

INTRODUCTION TO ROBOTIC ENTERPRISE FRAMEWORK

Introduction

This lesson provides an introduction on UiPath Robotic Enterprise Framework (RE Framework). We will cover the notions of transaction processing, dispatcher, performer and we will give you an overview on what RE Framework can accomplish.

Learning Objectives

At the end of this course you will be able to:

- Explain what a transaction is and the purpose of using transactions.
- Provide an overview on what RE Framework is and how it works.
- Describe how the dispatcher and performer model can help in scaling processes.

Transaction Processing:

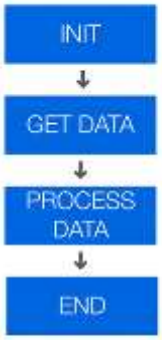
What is a transaction?

A transaction represents the minimum (atomic) amount of data and the necessary steps required to process the data, as to fulfill a section of a business process. A typical example would be a process that reads a single email from a mailbox and extracts data from it.

We call the data atomic because once it is processed, the assumption is that we no longer need it going forward with the business process.

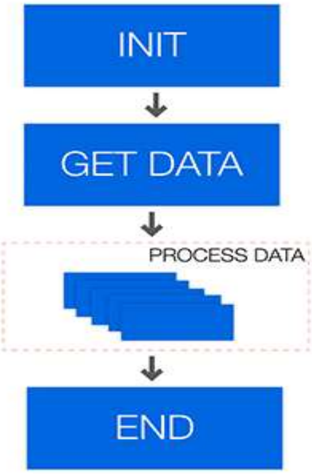
Considering the steps of a business process and how they are repeated, we can divide business processes into three categories:

1. **LINEAR:**



The steps of the process are performed only once and, if there is the need to process different data, the automation needs to be executed again. For example, if we go back to the email example from this chapter's introduction, and a new email arrives, the automation needs to be executed again in order to process it.
Linear processes are usually simple and easy to implement, but not very suitable to situations that require repetitions of steps using different data.

2. **ITERATIVE**:



The steps of the process are performed multiple times, but each time different data items are used. For example, instead of reading a single email on each execution, the automation can retrieve multiple emails and iterate through them doing the same steps.
This kind of process can be implemented with a simple loop, but it has the disadvantage that, if a problem happens when processing one item, the whole process is interrupted and the other items remain unprocessed.

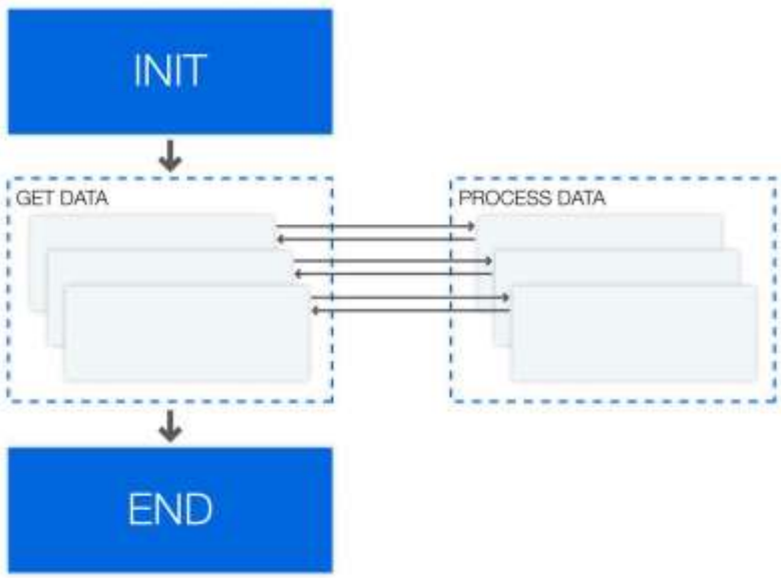
3. **<u>TRANSACTIONAL:</u>**

Similarly to iterative processes, the steps of transactional processes repeat multiple times over different data items. However, the automation is designed so that each repeatable part is processed independently.

These repeatable parts are then called transactions. Transactions are independent from each other, because they do not share any data or have any particular order to be processed.

The three categories of processes can be seen as maturity stages of an automation project, starting with simple linear tasks, which then are repeated multiple times and finally evolve into a transactional approach.

However, this is not a absolute rule for all cases, and the category should be chosen according to the characteristics of the process (e.g., data being processed and frequency of repetitions) and other relevant requirements (e.g., ease of use and robustness).



What are some business scenarios in which I will use transaction processing?

- You need to read data from several invoices that are in a folder and input that data into another system. Each invoice can be seen as a transaction, because there is a repetitive process for each of them (i.e. extract data and input somewhere else).
- There is a list of people and their email addresses in a spreadsheet, and an email needs to be sent to each of them along with a personalized message. The steps in this process (i.e., get data from spreadsheet, create personalized message and send email) are the same for each person, so each row in the spreadsheet can be considered a transaction.
- When looking for a new apartment, a robot can be used to make a search according to some criteria and, for each result of the search, the robot extracts the information

about the property and insert the data into a spreadsheet. In this case, the details page for each property constitutes a transaction.

The RE Framework:

Generally speaking, a framework is a template that helps you design (automation) processes. At a barebones minimum, a framework should offer a way to store, read, and easily modify project configuration data, a robust exception handling scheme, event logging for all exceptions and relevant transaction information.

The RE Framework is implemented as a state machine, which is a type of workflow that has two very useful features:

- States that define actions to be taken according to the specified input
- Transitions that move the execution between states depending on the outcomes of the states themselves.

You may remember state machines from the Project Organization lesson. One of the examples presented was the typical air conditioner:

- It has the OFF State, from where it moves to an IDLE State by pressing the ON/OFF button;
- From the IDLE State, it moves to either HEAT or COLD States when the temperature inputted by the user is lower and higher respectively than the current one. Once the desired temperature is achieved, it moves back to the IDLE State;
- From the IDLE State, it can move to the OFF State when the ON/OFF button is pressed;
- All the conditions that trigger the movements between the states are Transitions.

Based on a similar idea, REFramework has 4 main states that are usually common to business processes:

- 1

Initial State

This is where the process starts. It's an operation where the process initializes the settings and performs application checks in order to make sure all the prerequisites for the starting the process are in place.

- 2

Get Transaction Data State

Gets the next transaction item. This can be a queue item or any item of a collection.

By default, transaction items are queue *items*, but this can be easily changed to suit your needs. This is also the state in which the Developer should set up the condition to exit this state when there are no items to process.

- 3

Process Transaction State

Performs actions/applies logic in various applications for the transaction item obtained at the previous step. Once a transaction item is processed, the process continues with the next available transaction item.

- 4

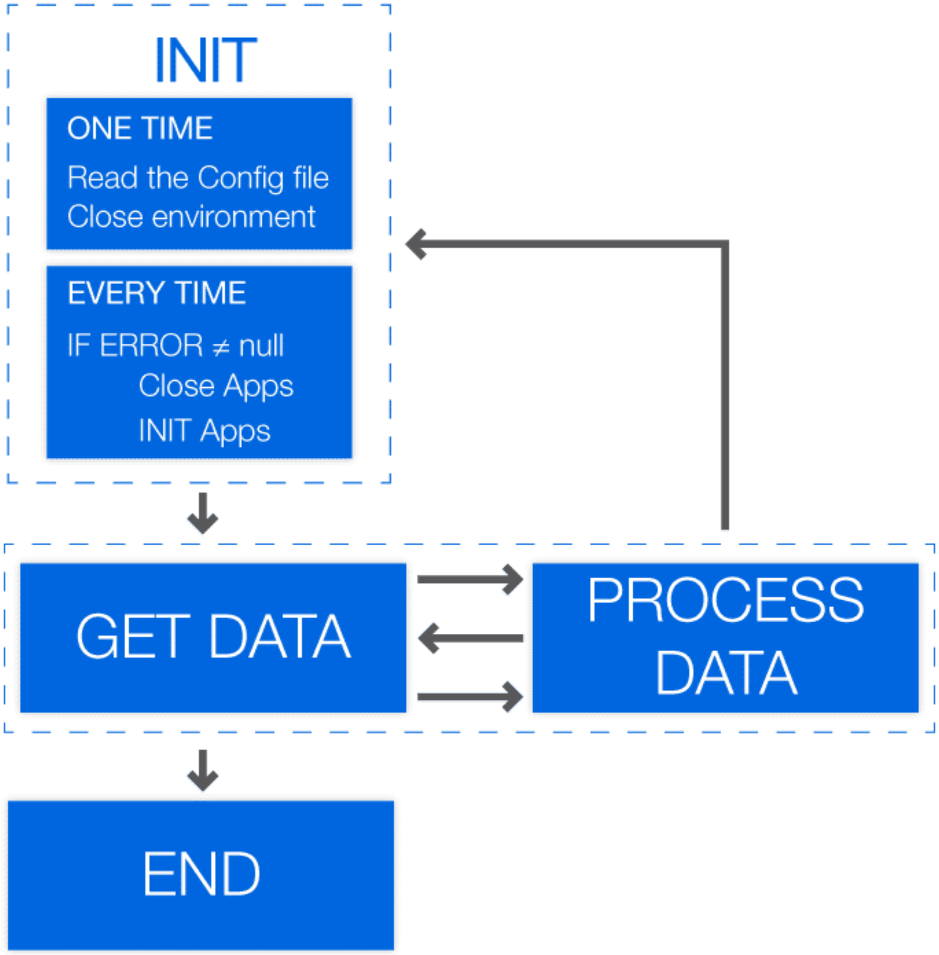
End Process State

The process ends (and the applications opened during the automation should be gracefully closed).

To have a better understanding of how we can use RE Framework, let's go through the following scenario:

There is a list of people and their email addresses in a spreadsheet. An email needs to be sent to each of them with a personalized message based on a template.

Select each marker below to see how the process can be implemented using the 4 states:



- **Initial State**: Read data from spreadsheet and read the email template from a text file.
- **Get transaction state**: Retrieve one row from the spreadsheet, which contains the name and the email of one person.
- **Process data**: Insert the person's name and email in to the email template to create a personalized message. After that, send the email using this message.
- **End process state:** After all rows are processed, finalize the execution. Since emails can be sent directly by the robot, there is no need to close any email applications.

RE Framework Features

- **Settings**

  In many processes, it is common to have certain settings and configuration values that are read during the initialization phase. Examples of settings include URLs to access web applications, Orchestrator queue names and default logging messages.

The RE Framework keeps track of such data by reading them from a configuration file (Config.xlsx) and storing them in a Dictionary object (Config) that is shared among the different states. This offers an easy way to maintain projects by changing values in the configuration file, instead of modifying workflows directly.

- **Logging**
  Another powerful feature of the RE Framework is the built-in logging mechanism. Most of the workflows that compose the framework use Log Message activities that output details of what is happening in each step of the execution.

  This can be used not only to find problems and help in the debugging process, but also to create visualizations and reports about the execution of the process (for example, how many invoices are processed each day, how many failures happen and what are the main causes of failures) and about the process itself (for example, what is the aggregated value of all reports processed in a month).

- **Business Exception & Application Exception**
  During the execution of most processes, there can be situations that are not part of the normal execution flow and they need to be addressed to achieve a more robust automation.

  As an example, consider a process that uses a few web applications, but at a certain point the web browser freezes. If an activity (for example, Click) tries to interact with a frozen application, it will probably fail and return an exception. The RE Framework was designed to enable recovery from exceptions by either attempting to process the transaction again (i.e., retrying) or by skipping that transaction.

  If the cause of the problem can be fixed by restarting applications, then the framework automatically does that and tries again to process the same transaction. These exceptions are called Application Exceptions.

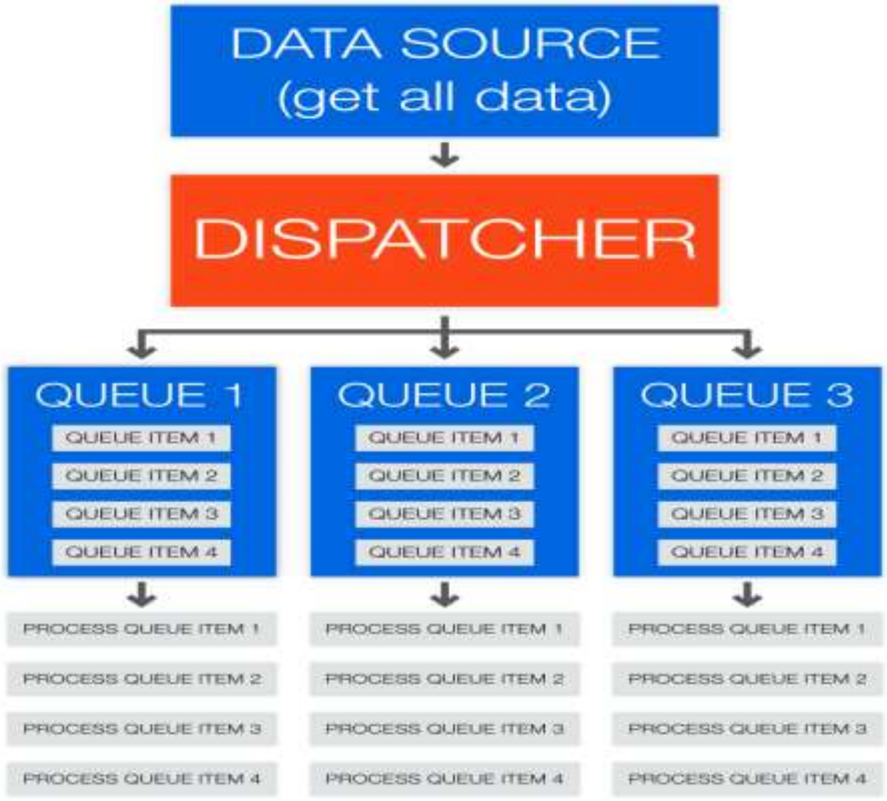
  If the problem is related to the data itself or an underlying business requirement (for example, don't process invoices with value more than a certain amount), then that transaction is skipped and the framework proceeds to the next transaction. These exceptions are called Business Rule Exceptions.

Dispatcher and Performer

Although the RE Framework can be used with different types of data sources, it provides a particularly strong integration with Orchestrator queues. When queues are used, it is possible to define priorities and deadlines for transaction items and to track retrying attempts of failed transactions.

The use of queues also enables an execution pattern called Dispatcher & Performer, which divides the process in two main phases: dispatching items to be processed and adding them to a queue, followed by retrieving an item from a queue and performing the process using that item. The second part of the process is generally built using the REFramework.

Dispatcher:

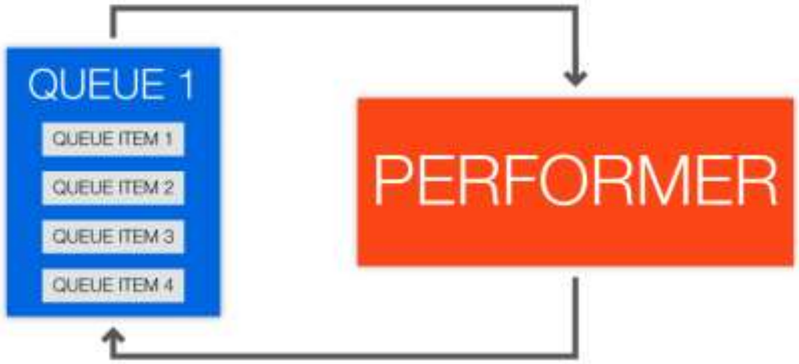


The dispatcher is a process used to push transaction items to an Orchestrator queue. It extracts data from one or multiple sources and uses it to create Queue items to be processed by Performer robots.

Information is pushed to one or more queues allowing the dispatcher to use a common format for all data stored in queue items.

The major advantage of using a dispatcher pattern is that you can split the processing of the items between multiple robots.

Performer:



The performer is a process used to pull transaction items from an orchestrator queue and process them as needed in the company. Queue items are processed one at a time.

It uses error handling and retry mechanisms for each processed item.

A major advantage of the performer is its scalability (multiple performers can be used with a single queue)

Let`s use the same scenario as previously where we have a list of people and their email addresses in a spreadsheet. An email needs to be sent to each of them with a personalized message based on a template.

We can use the Dispatcher & Performer pattern in the following manner:

1. The **Dispatcher** reads rows from the input spreadsheet and adds the data (i.e., name and email) to a queue; each queue item will have both name and email from one spreadsheet row.
2. The **Performer** retrieves an item from the same queue and does the necessary actions using that data, like replacing template values and sending an email.

Dispatcher & performer model advantages:

- Better separation of processes (between dispatcher and performer)
- Better separation & distinction between architecture and process layers
- Better error handling and retry mechanism
- Possibility to run processes across several machines (availability)
- Better re-usability within your project's created components
- Improved built-in configuration & Orchestrator integration
- Previous workflows created without RE Framework can be easily adapted and deployed in order to use RE Framework and the dispatcher/performer model