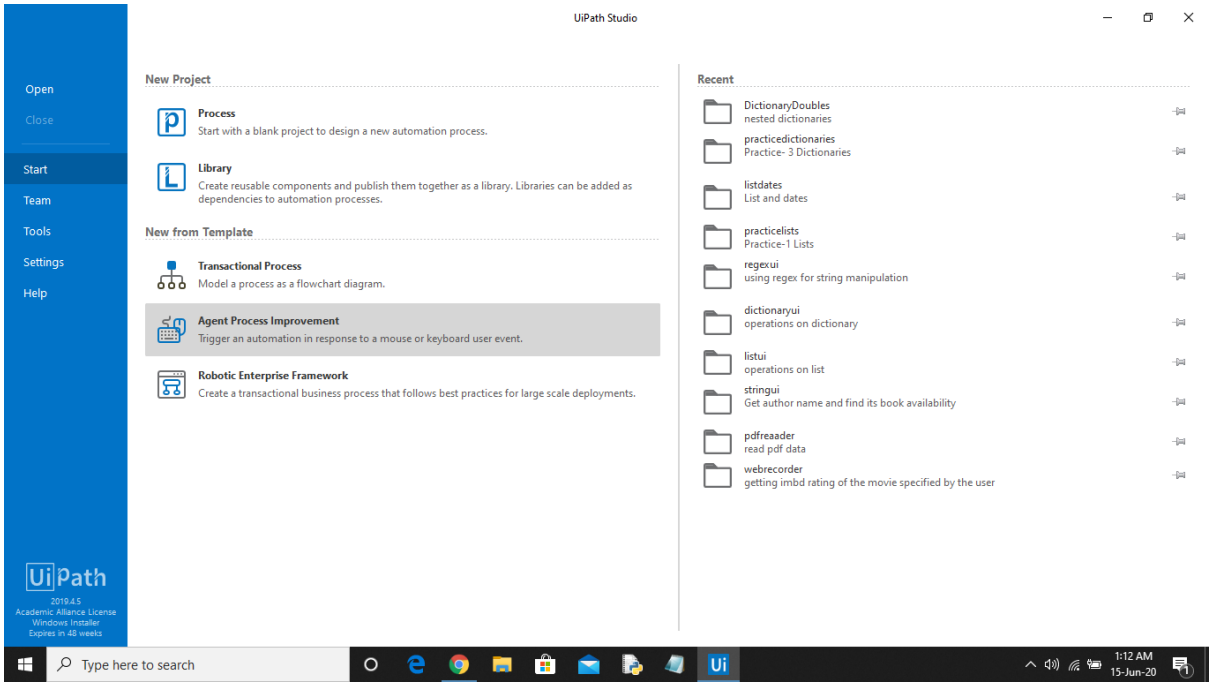
UNIT-V

Robotic Process Automation (RPA) – Email Automation using UIPath

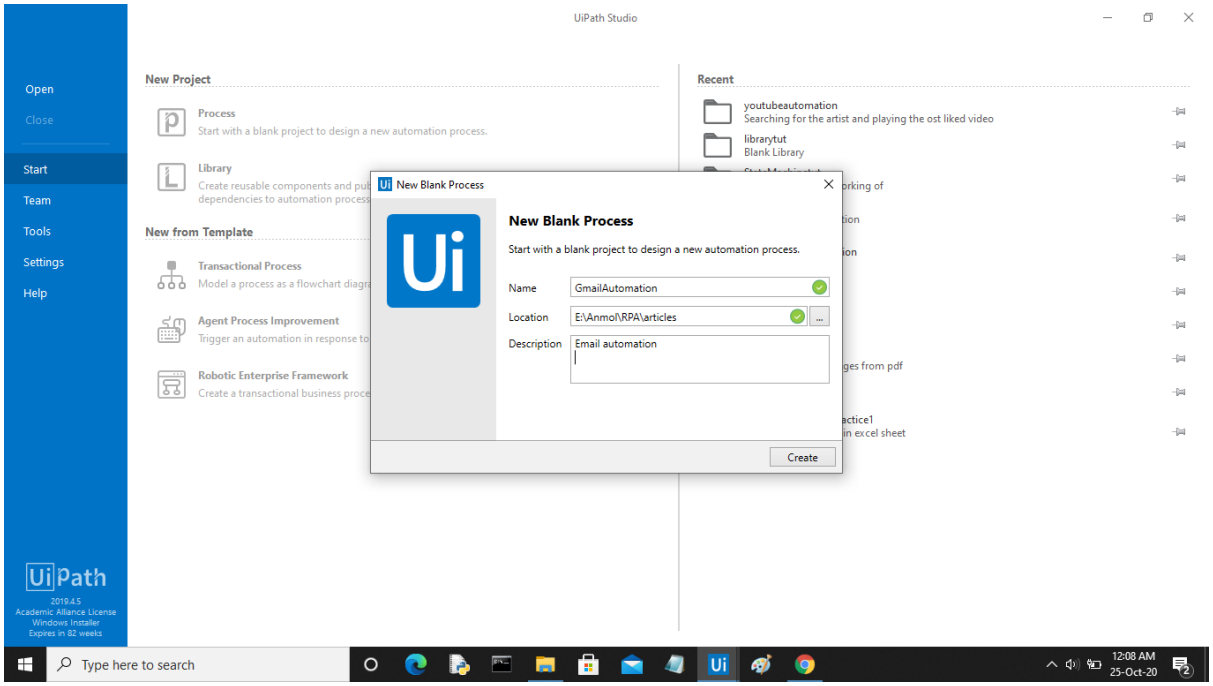
We are going to learn how we can automate sending emails using Uipath Studio. This project is a basic application of Robotic Process Automation (RPA). The user just needs to provide their login credentials and the user email they want to send the email and that is it, rest of the work will be done by the bot that we are going to create using the steps mentioned below –

Note : For this automation to work you need to enable access from *Less Secure apps* on your Gmail account.

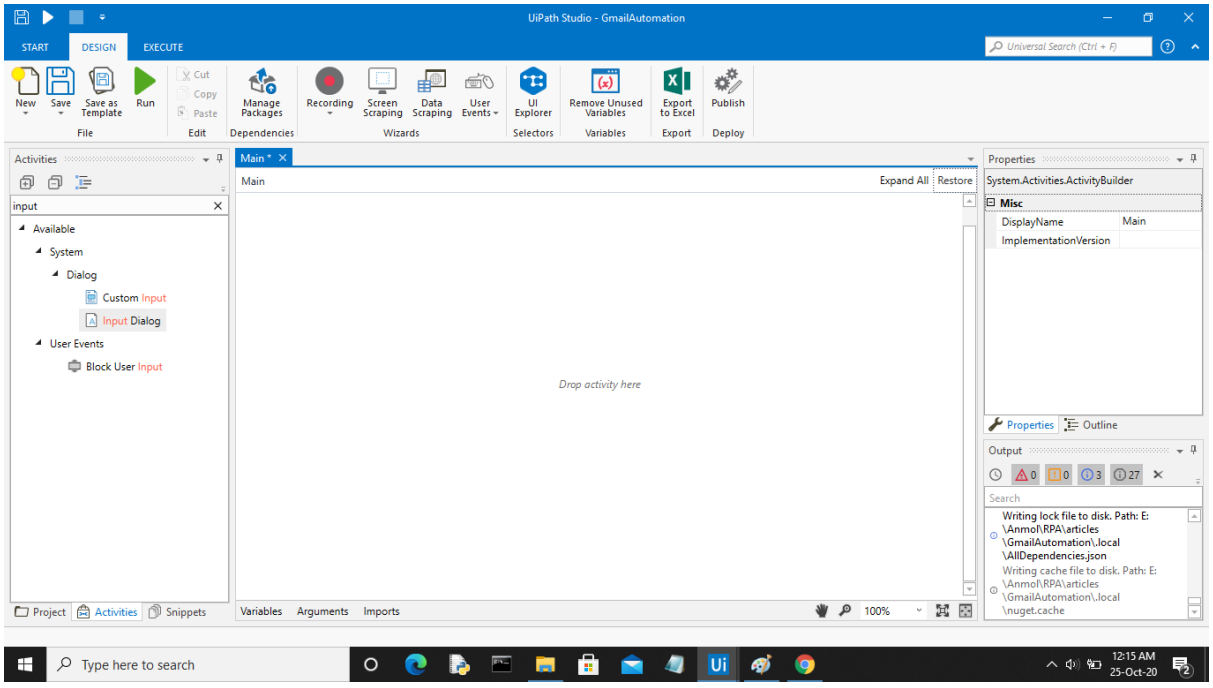
Step-1 : Create a new process in Uipath Studio by clicking on the ***Process Tab***.



Step-2 : Set the name of the process, give a brief description, and click on ***Create***. The Uipath studio will automatically load and add all the dependencies of the project.



The following screen will be opened.



Step-3 : Now in the activities panel search for ***Sequence*** activity. Drag and drop it in the designer window.

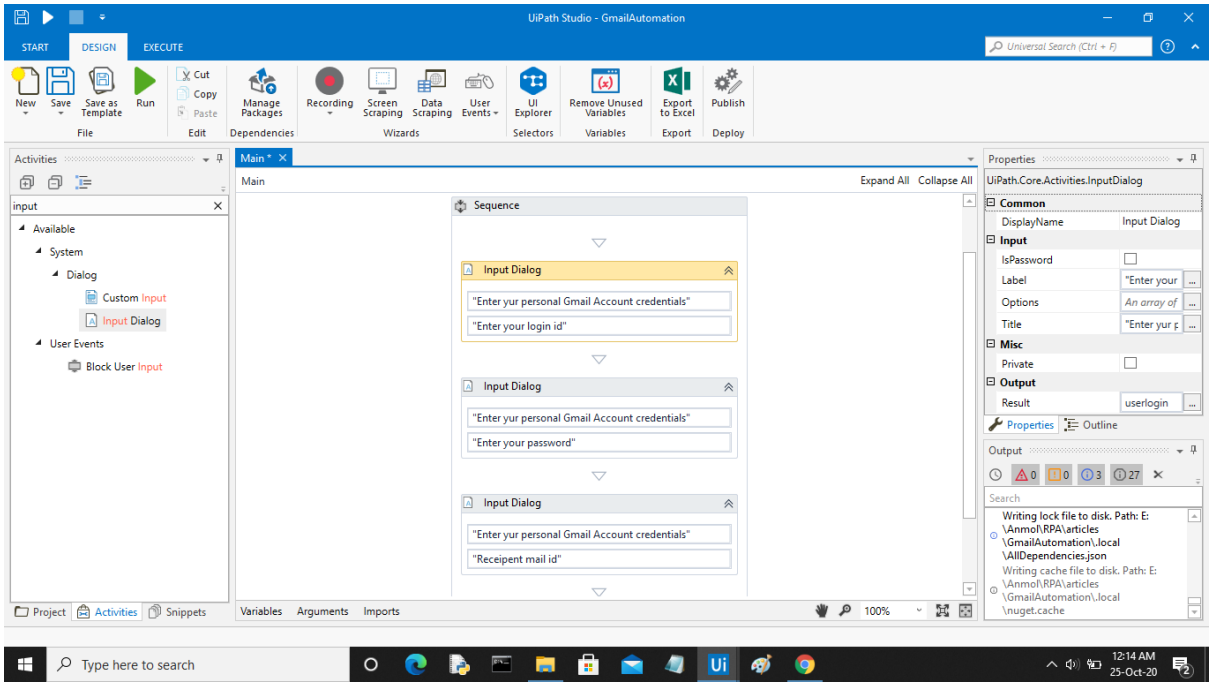
Step-4 : Now in the sequence add an ***Input dialog box*** using the search activity panel.

Step-5 : Now click on the ***Variables*** tab to create a variable of string type that will store the personal email id entered by the user. In the properties panel of the Input Dialog activity add the variable in the ***Output area***.

Create two more string variables, one for storing the password of the user and the second for storing the recipient email id.

Step-6 : Now add another *Input dialog box* and pass the variable created earlier to store the password in the ***Output Area*** of the properties panel. Click the checkbox ***IsPassword*** for this activity.

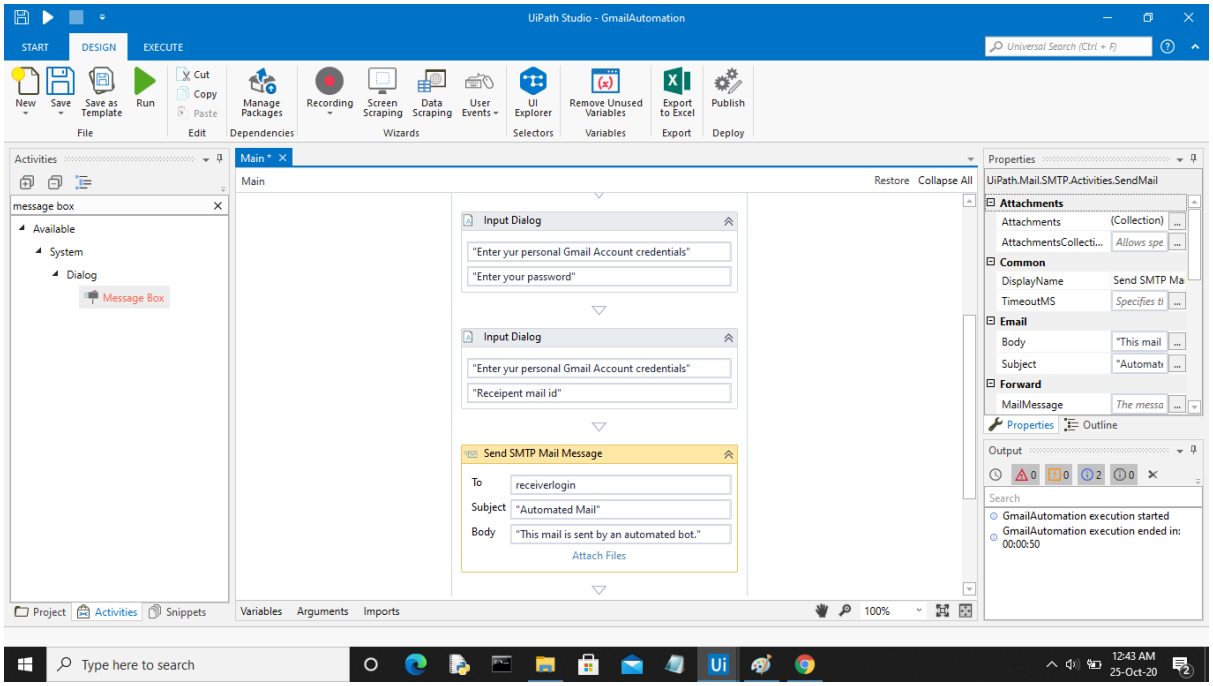
Step-7 : Finally add another ***Input dialog box*** for taking the recipient email id from the user. Pass the variable created earlier in the ***Output Area***.



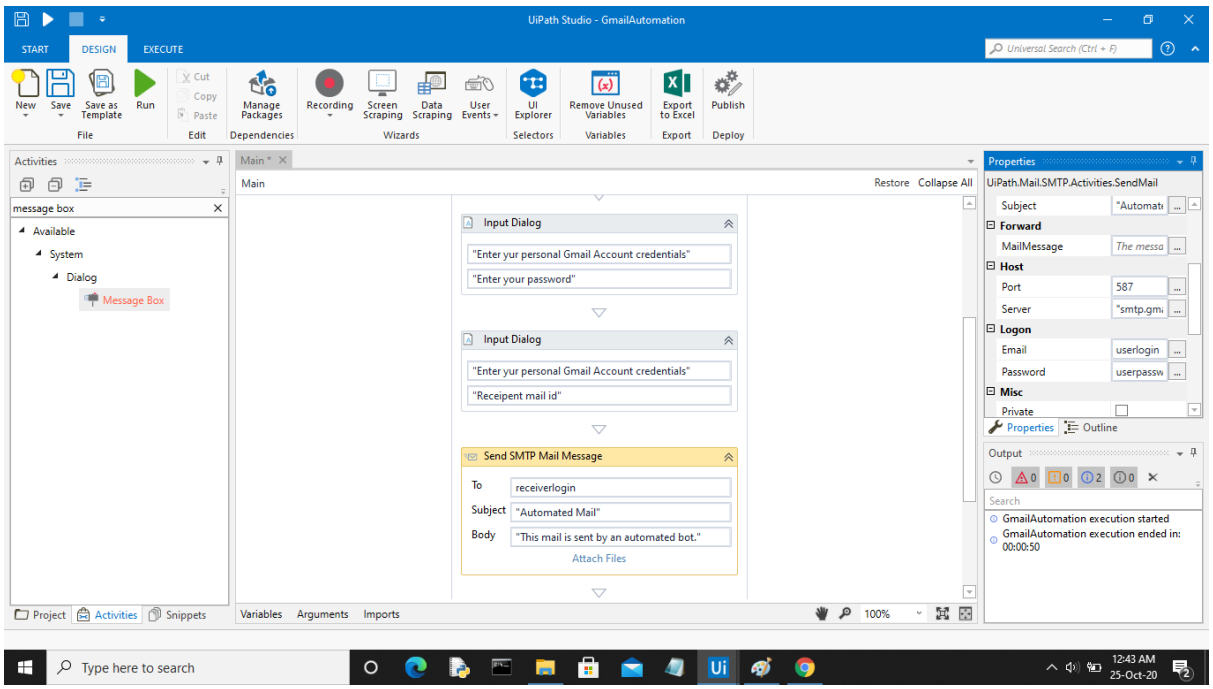
Sending Email:

Step-8 : Now in the ***Activity Panel*** search for ***Send SMTP Mail Message*** activity. Drag and drop it to the sequence.

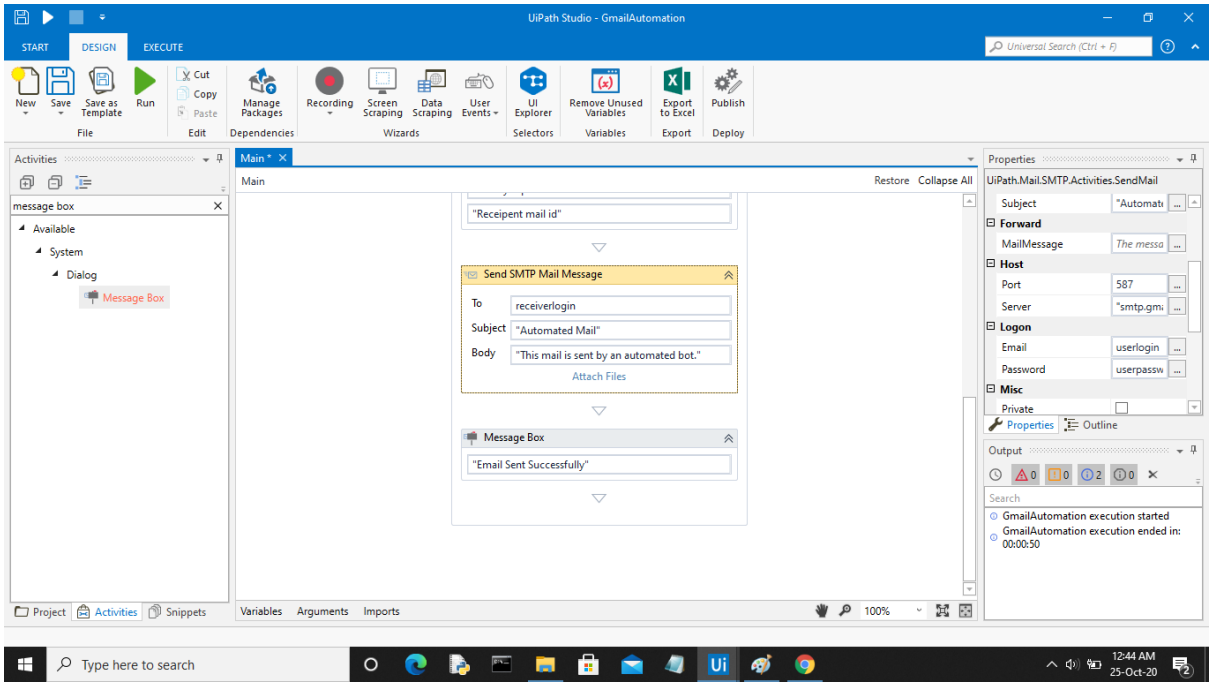
Step-9 : In the ***To*** field pass the variable that stores the recipient email id. Specify the ***subject*** and ***body*** of the mail.



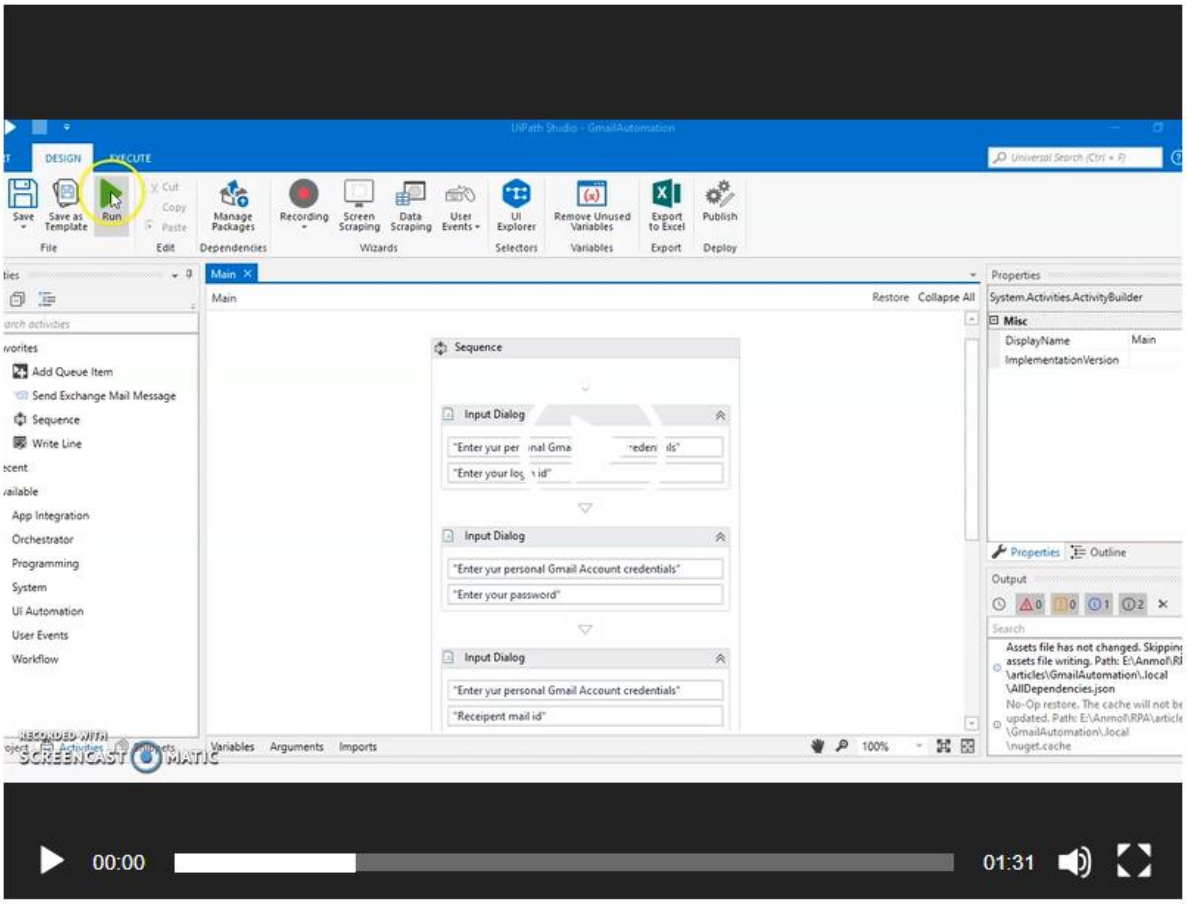
Step-10 : In the ***Properties panel,*** set *Port no.* to ***587*** and *Server* to ***smtp.gmail.com***. Pass the variables that store user credentials in the email and password field respectively.



Step-11 : Finally add a ***Message Box*** activity. Specify the message you want to display as a notification that the email was sent successfully.

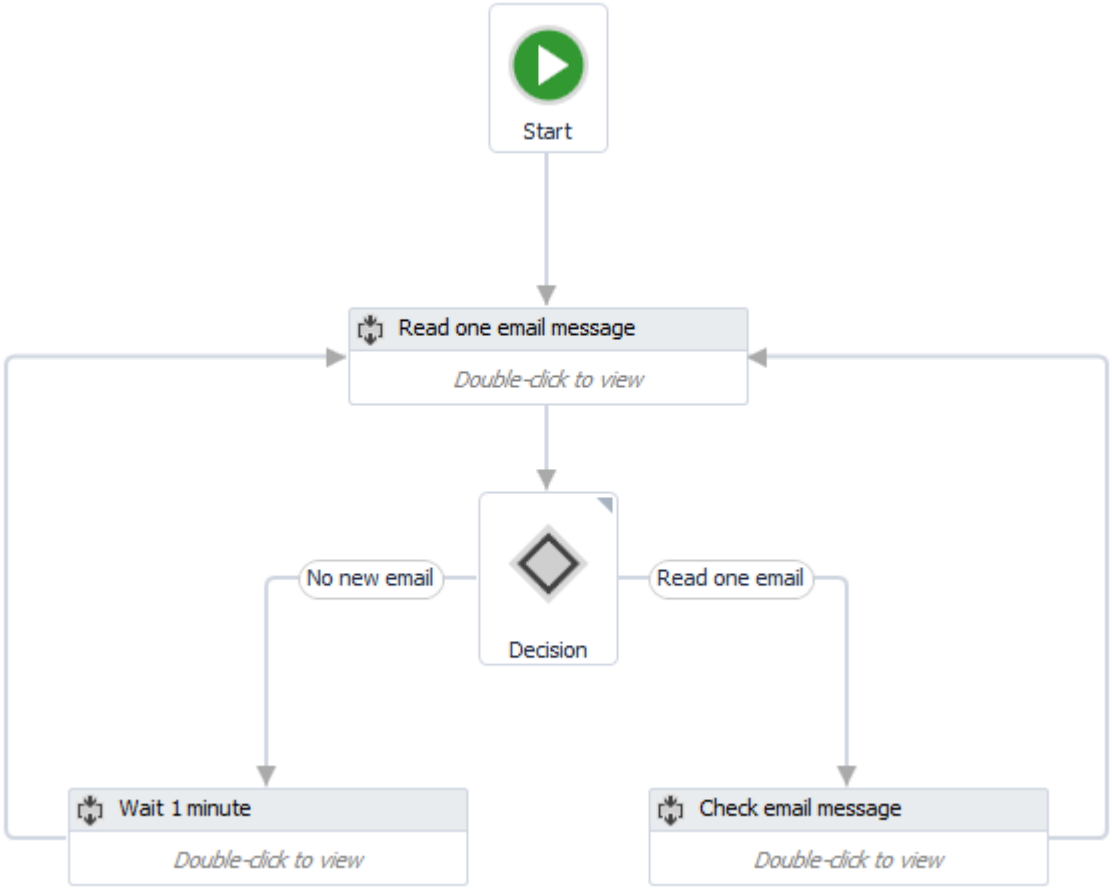


Step-12 : Save the process using the ***Save*** button in the design panel and then click on ***Run***. Your bot is ready for sending automated emails. The below video shows the working of the bot.

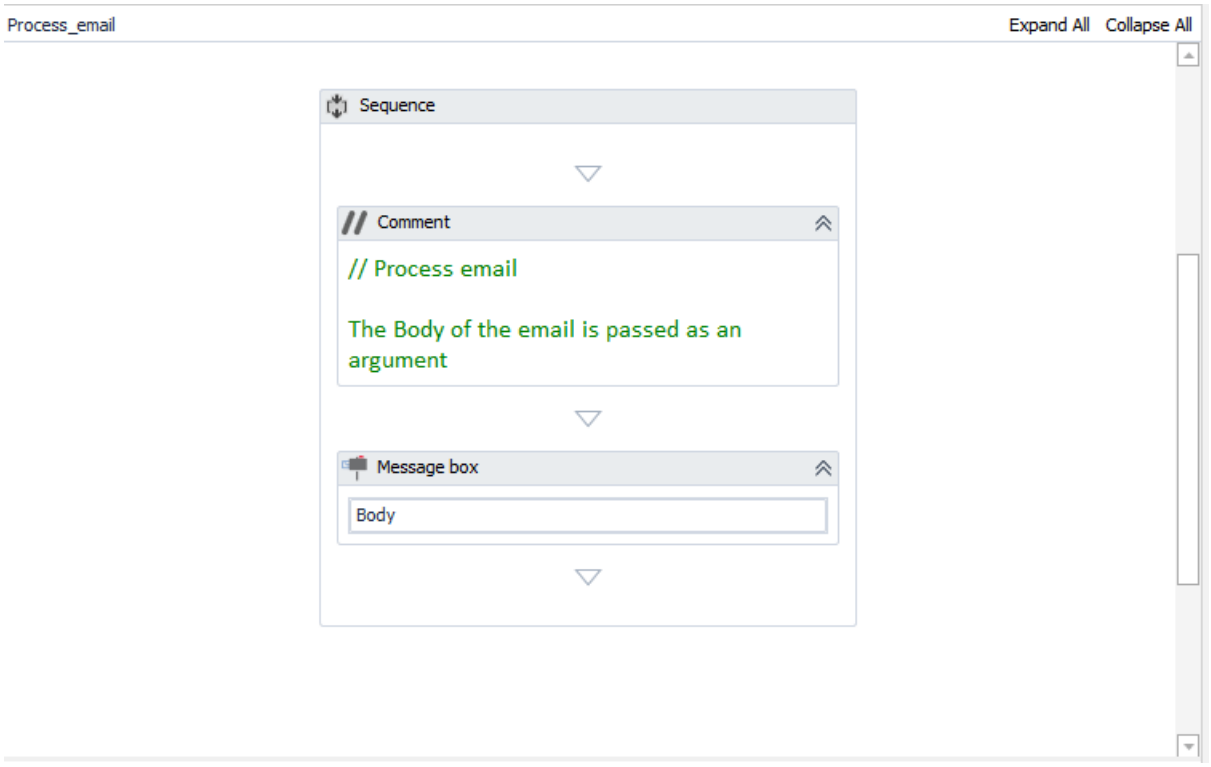


Incoming Email automation

Email triggers are the foundation of any email automation process. In the attached ***Mail_Trigger_Sample*** **Workflow**, the Inbox is checked every 60 seconds for fresh emails.



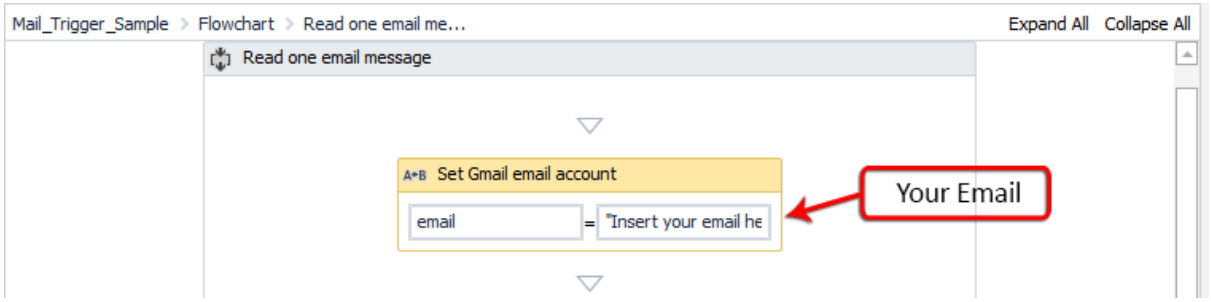
If a new email is detected and a certain condition is met (a specific Keyword is found in the Subject), then the message Body is passed as an argument to the ***Process_Email*** *W*orkflow.



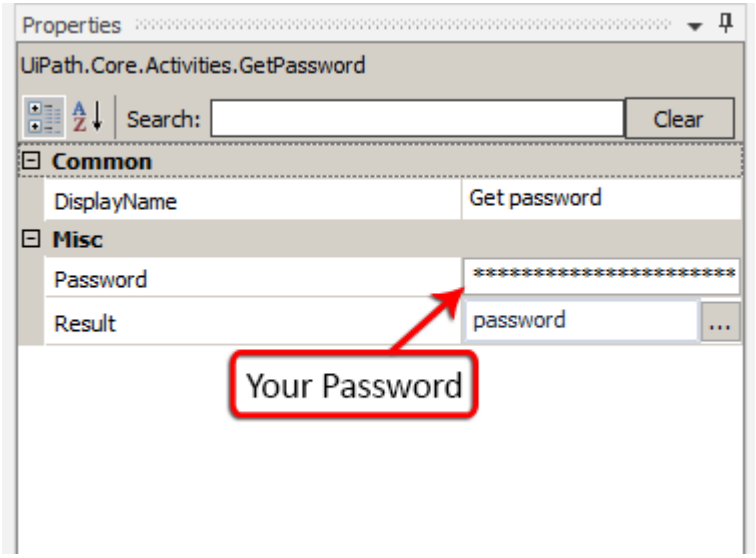
Workflow Steps:

1. Before running the ***Mail_Trigger_Sample*** workflow, you will need to set:

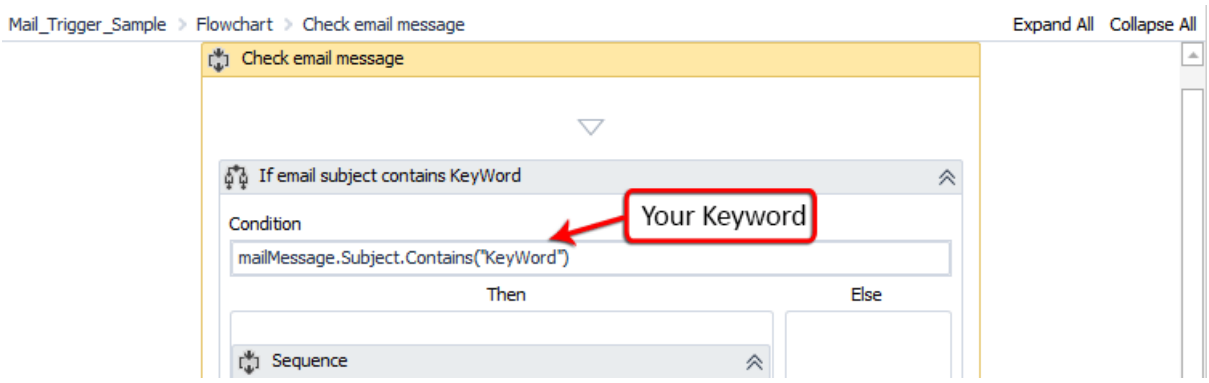
Email address:



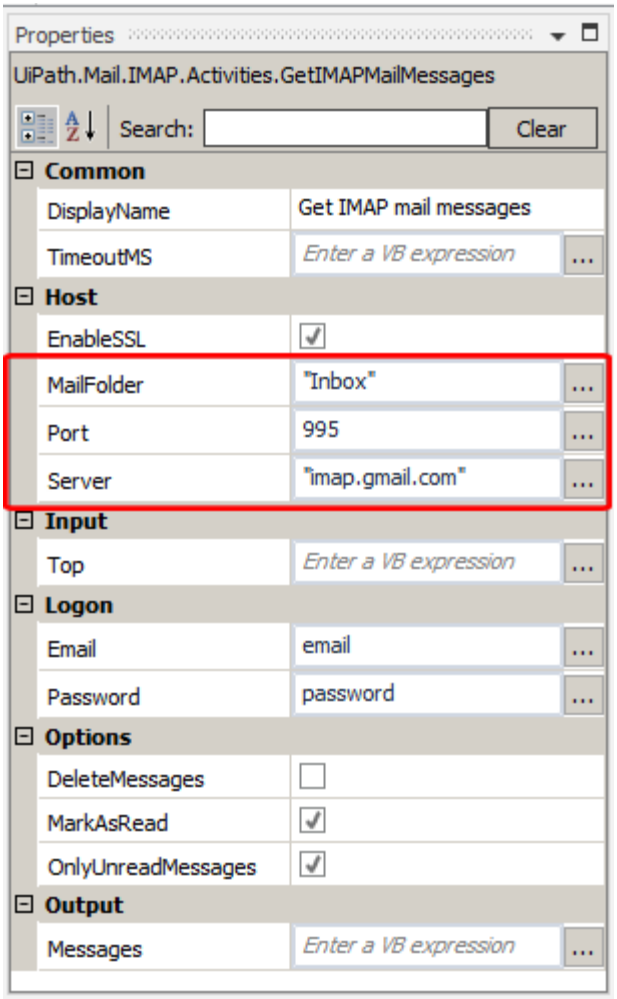
Password:



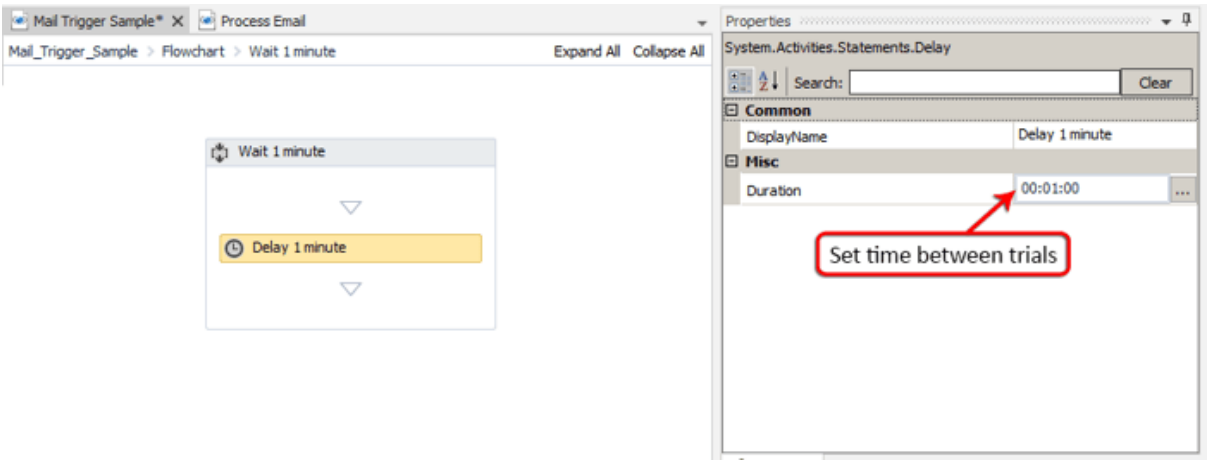
Keyword:



Also, please make sure you set your *MailFolder*, *Port*, and *Server* in the **GetIMAPMailMessages** activity properties. By default, the **Workflow** is configured to work with the Gmail settings.



2. When the **Workflow** is running, it checks your email for new messages (using the **GetIMAPMailMessages** activity). If no new email is found, it waits for 60 seconds and then it tries again. You can change the waiting time to other value, within the **Delay** activity.



3. If a new email is found, it is marked as read (you can change that by un-checking the *MarkAsRead* property in the **GetIMAPMailMessages** activity)

4. The **Workflow** then checks if the email's subject contains the keyword. If there's a match, the ***Process_Email*** **Workflow** is invoked and the email body is passed as an argument (Body)

5. The ***Process_Email*** **Workflow** will pop up a window with the message body.

Global Exception Handler

The **Global Exception Handler** is a type of workflow designed to determine the project's behavior when encountering an execution error. Only one **Global Exception Handler** can be set per automation project.

Note: The **Global Exception Handler** is not available for library projects, only processes.

The **Global Exception Handler** has two arguments, that should **not** be removed.

The first argument is errorInfo with the **In** direction and it stores information about the error that was thrown and the workflow that failed. The level of the error to be logged can be set in the Log Message activity.

Note: Use the ActivityInfo property for errorInfo to get the name of the activity which threw the exception and view it in the **Output** panel.

The second argument, result has the **Out** direction and it is used for determining the next behavior of the process when it encounters an error. The following values can be assigned to the result argument:

- **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. Use the RetryCount method for errorInfo to count the number of times the activity is retried.
- **Abort** - The execution stops after running the current **Global Exception Handler**.

Note: Any workflow may be flagged as a **Global Exception Handler** in Studio, except for library projects and Main.xaml.

To control the workflow's behavior in case of an error, the **Global Exception Handler** retries the activity three times and then aborts with an error message.

Handling Errors During Debugging

When an exception is detected during debugging, the activity which faulted is highlighted, the execution is paused, and the exception's type and details are mentioned in the **Locals** and **Call Stack** panels.

Debugging actions like **Continue**, **Stop**, **Retry**, **Ignore**, **Restart** and **Slow Step** are available in the ribbon. **Ignore** is used for continuing the execution from the next activity.

The **Retry** button retries to execute the current activity, without the Global Exception Handler stepping in. The **Continue** action runs the Global Exception Handler, taking into consideration the previously chosen values for the result argument, either **Continue**, **Ignore**, **Retry** or **Abort**.

When using the **Global Exception Handler** with a project that includes a Try Catch, make sure to group activities into a **Sequence** inside the **Try** container. Otherwise, the **Global Exception Handler** does not execute.

In the case of nested activities, the **Global Exception Handler** executes for each activity in the call stack. However, it does not execute for activities directly encapsulated in a **Try Catch**, unless they're contained in an activity.

Example of Using the Global Exception Handler

The following example showcases the project's behavior when an exception is thrown during execution.

The automation project is set to type some text in a TXT file and then close the application, but not before saving the file.

Creating the Workflow

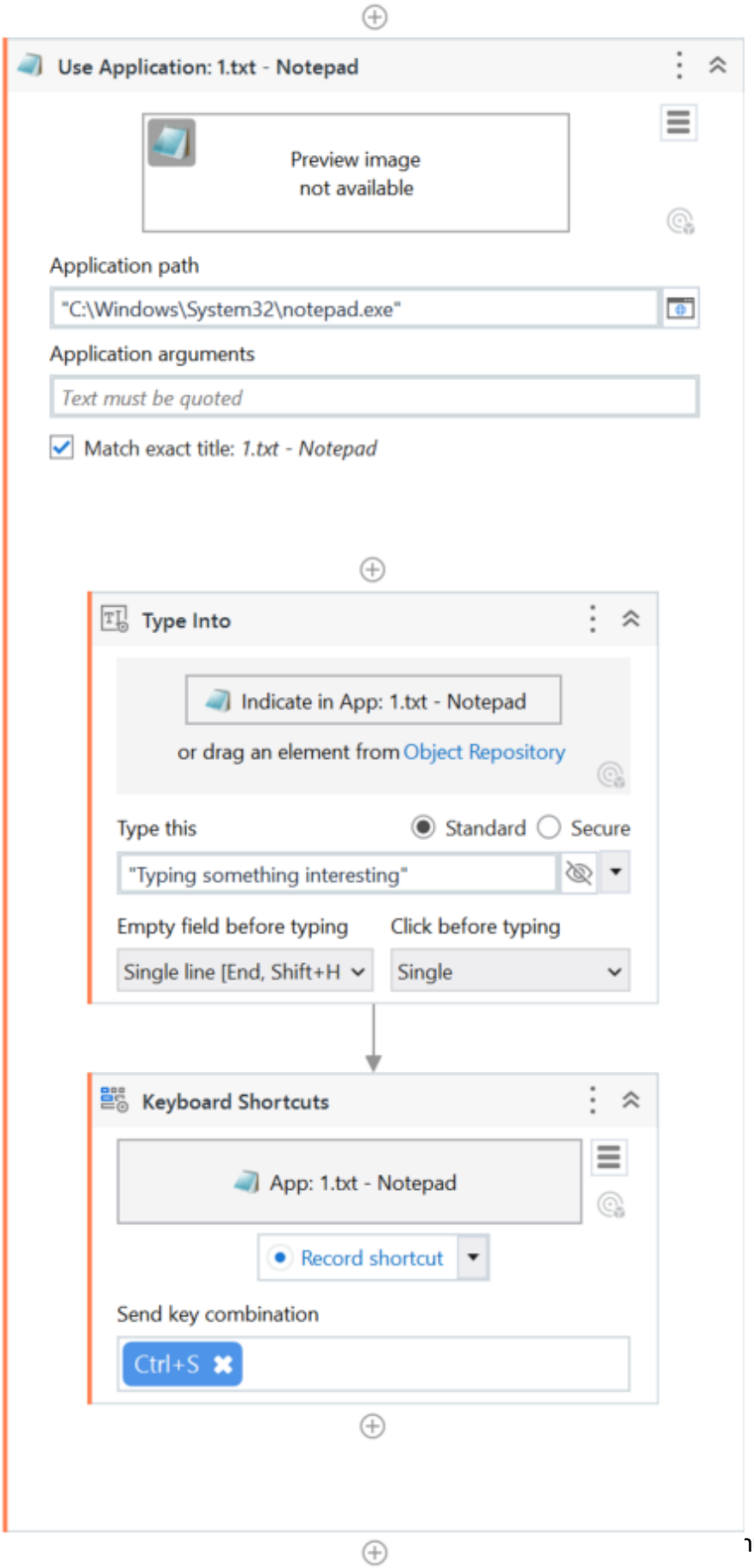
1. Create a **Blank Process** by following the steps in the Creating a Basic Process page.
2. Open Notepad and save a document on your machine. You can name the file 1.txt.
3. In the Activities panel, search for Use Application/Browser and drag it to the Designer panel.
4. In Use Application/Browser:
    - Click **Indicate application to automate**, and then move the mouse pointer to the Notepad window. When the window is highlighted, click anywhere inside it.

The Use Application/Browser activity is updated, the path is added to the **Application path** field, and a screenshot of the window appears inside the activity.

- In the **Properties** panel, select the **Always** option for the **Close** property. This ensures Notepad is closed after the automation runs.

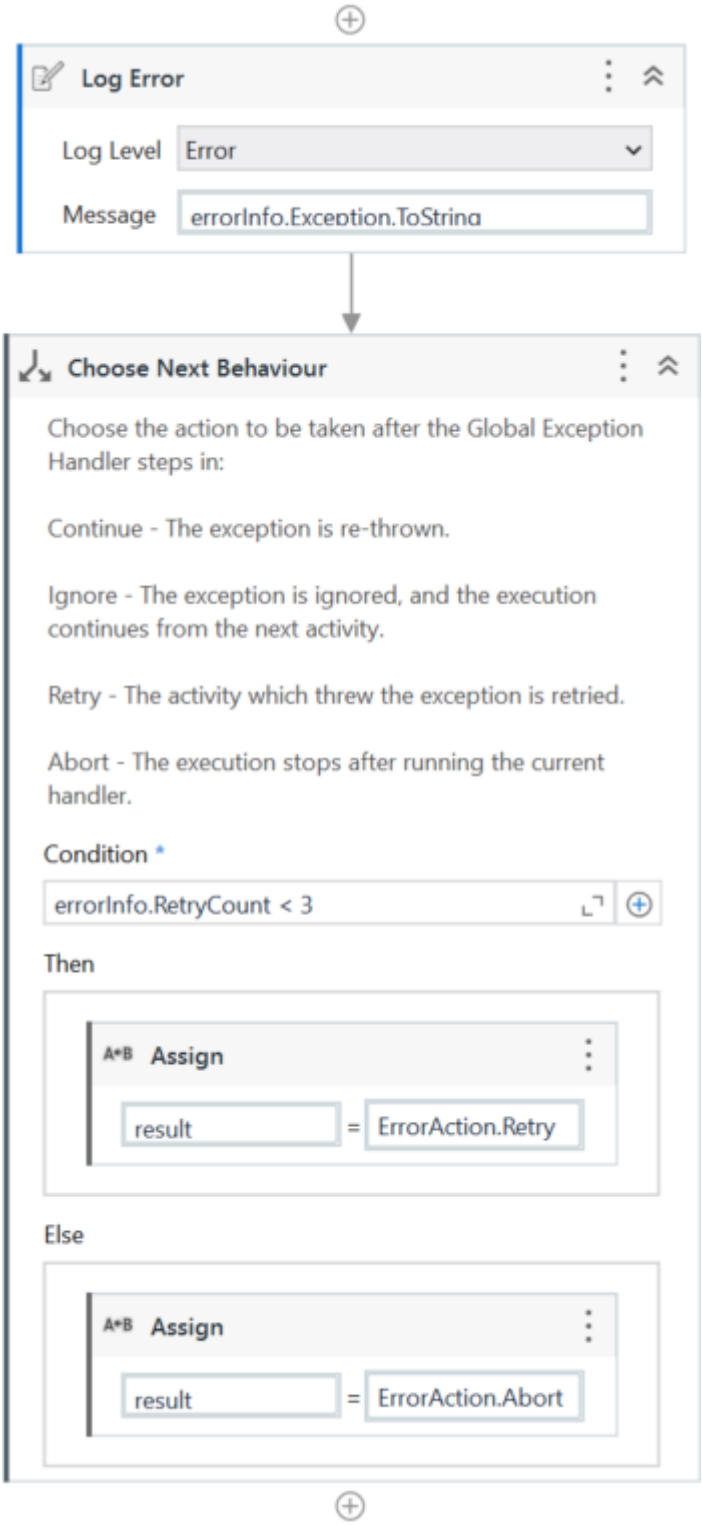
5. Add a Type Into activity in the **Use Application/Browser** activity's **Do** container. Click **Indicate in App** to select the Notepad window, and add enter a text between quotation marks in the **Type this** field. This activity writes the text into Notepad.
6. From the **Activities** panel, add a Keyboard Shortcuts activity to the workflow. Indicate the Notepad window, then select **Record shortcut** and press **Ctrl + S** to record the key combination that saves the file after the text was typed in.

The resulted workflow should look like this:


Use Application: 1.txt - Notepad
Preview image not available
Application path
"C:\Windows\System32\notepad.exe"
Application arguments
Text must be quoted
Match exact title: 1.txt - Notepad
Type Into
Indicate in App: 1.txt - Notepad
or drag an element from Object Repository
Type this
Standard
Secure
"Typing something interesting"
Empty field before typing
Click before typing
Single line [End, Shift+H
Single
Keyboard Shortcuts
App: 1.txt - Notepad
Record shortcut
Send key combination
Ctrl+S

Adding a Global Exception Handler

1. In the **Design** tab part of the **Ribbon**, select **New > Global Handler**. The **New Global Handler** window opens. Type in a **Name** for the handler and save it in the project path. Click **Create**, a **Global Exception Handler** is added to the automation project.



2. Go back to the workflow you created earlier and modify it so that an activity fails to execute. For example, in the Use Application/Browser activity, select the **Match exact title** option and make sure the file is closed before you click **Run File** in the ribbon.

When the **Global Exception Handler** encounters an exception, it logs the name of the activity which faulted and starts retrying the activity three times. If it encounters the same exception each time and the number of retries reaches 3, the execution is aborted at the level of the activity which threw the exception.

If during one of the retries an exception isn't encountered, the execution of the workflow continues and the **Global Exception Handler** doesn't step in.

Debugging and Exception Handling

UiPath is one of the most popular **RPA** tools used for Windows desktop automation. It is used to automate repetitive tasks without human intervention, the tool offers drag and drop functionality of activities that you must have learned in the **previous blogs**. In this blog on Error Handling in UiPath, I will cover all the basics of how you can handle errors in projects.

Error Handling in UiPath mainly consists of two topics that you need to understand:

- Debugging
- Exception Handling

Once you go through the above two topics, we will discuss few **tips & tricks** which will make you aware of some common errors, and how to avoid them.

Debugging

Debugging in simple terms is the process of identifying and removing errors from the project. Now, to debug errors, you need to go to the **Execute** tab. The Execute tab has 3 sections, **Launch** section, **Debug** section & the **Logs** section, as you can see in the below image:

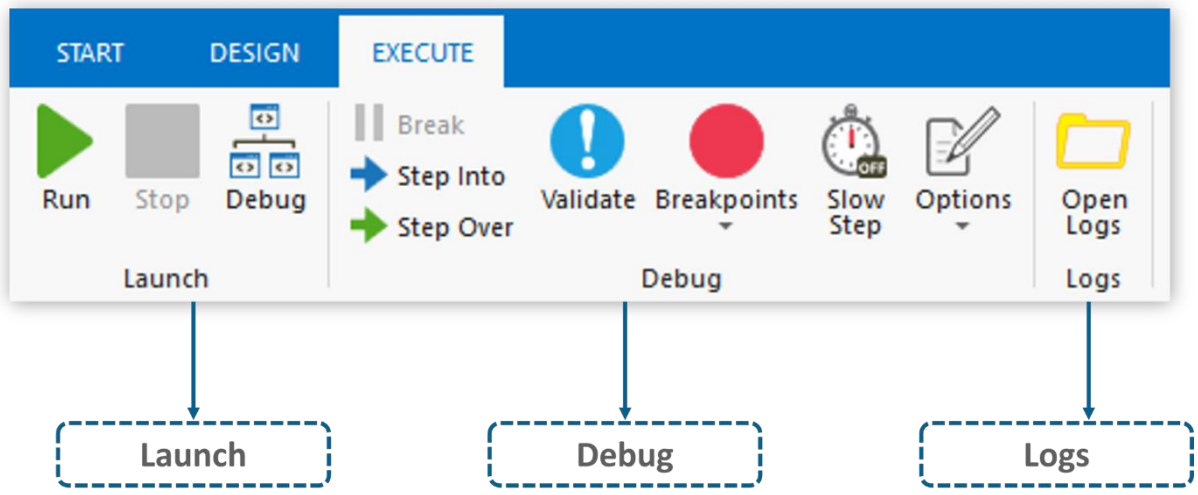


Fig 1: Execute Tab in UiPath – Error Handling in UiPath

Let me explain, the functionalities of each section one by one.

Launch Section:

The launch section has 3 options:

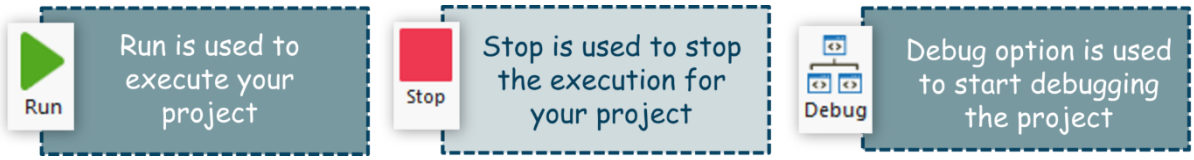


Fig 2: Launch Section Options – Error Handling in UiPath

As you can see in the above image, the **Run** option is used when you simply want to execute your project. So, with this option, you would not see the step by step execution, but would directly see the output, if it successfully executes. The **Stop** button is used to use stop the execution of your project in the middle and **Debug** is used to Debug the errors step by step.

Debug Section:

The Debug section has 5 options:

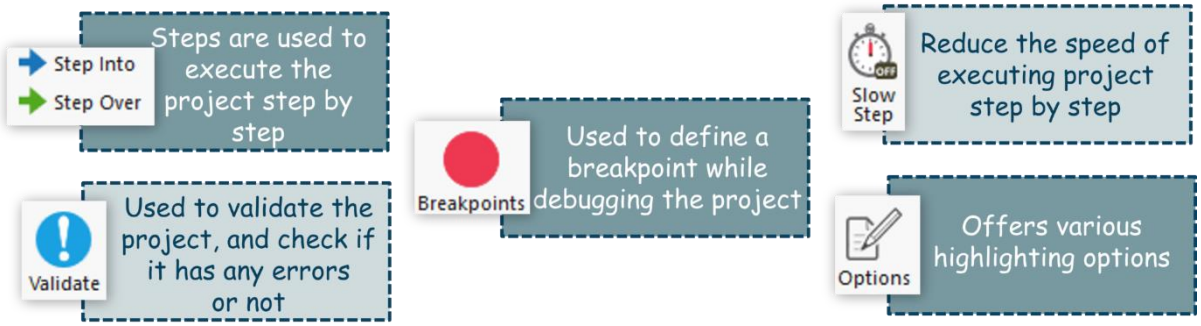


Fig 3: Debug Section Options – Error Handling in UiPath

- **Steps** are used to execute your project step by step. So, when you click on **Step Into**, it executes the next step and then it waits.
- **Validate** button is used to validate your project and check if you have any errors or not. So, when you choose this option, UiPath will check if your automation has any errors and if it has any errors it will return you the error.
- **Breakpoints** are the points at which you want to stop the execution and start debugging step by step. The breakpoints button offers two options:
  - Toggle Breakpoints
  - Remove All Breakpoints
- **Slow Step** slows down your execution so that, you get a track of what is happening.
- **Options** provide various highlighting options to highlight the activities. So, you can use this when you want to highlight any activity while you are debugging your project.

Logs Section:

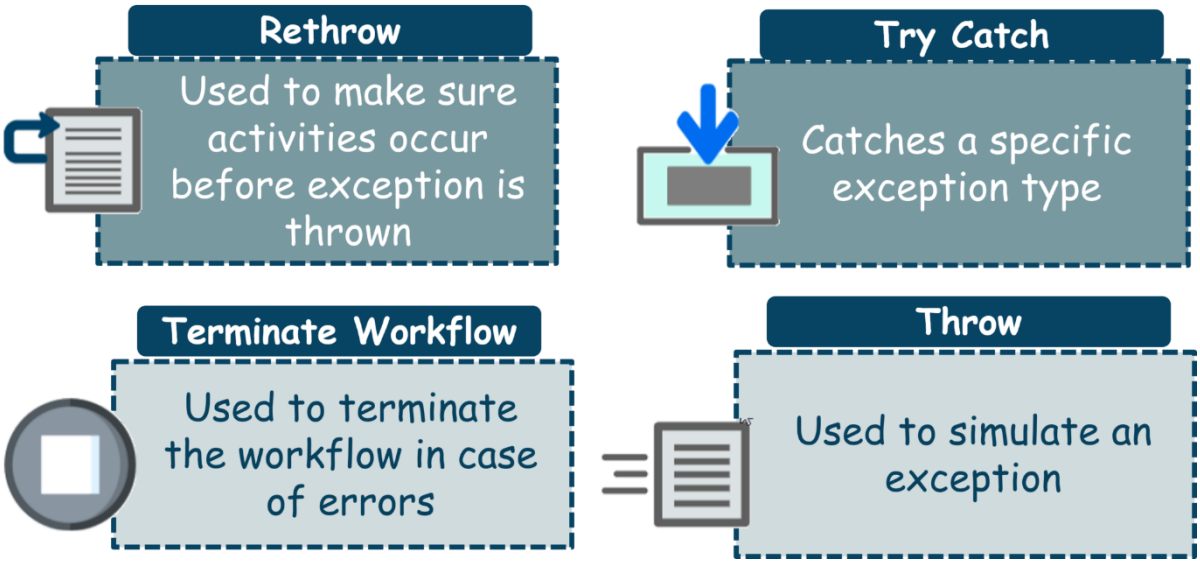
The log section has only one option, which is **Open Logs**.

The **Open Logs** button lets you debug the program with the help of the logs. You can check where your values went wrong from the logs.

So, that was about Debugging folks. Let us move to our next topic which is Exception Handling.

Exception Handling

Exception Handling mainly deals with handling errors with respect to various activities in UiPath. The Error Handling activity offers four options: Rethrow, Terminate Workflow, Throw, Try Catch.



- **Rethrow** is used when you want activities to occur before the exception is thrown.
- **Terminate** workflow is used to terminate the workflow the moment the task encounters an error.
- **Throw** activity is used when you want to throw error before the execution of the step.

- **Try Catch** activity is used when you want to test something and handle the exception accordingly. So, whatever you want to test you can put it under the **try** section, and then if any error occurs, then it can be handled using the **catch** section, based on your input to the catch section. Apart from the try-catch, we also have a **Finally** section which is used to mention those activities which have to be performed after the try and catch blocks are executed.

Now, that you folks know the various options that UiPath offers for handling errors. It is a good time that you know the common mistakes that people do and learn how to resolve them.

Debugging Techniques and Tools

Debugging is a crucial aspect of developing automation projects in UiPath. It helps identify and fix errors, ensuring smooth and efficient workflows. Here are some common debugging techniques and tools that can aid in this process:

1. **Breakpoints:** Set breakpoints in your workflow to pause execution at specific points. This allows you to inspect variables and understand the state of your project at different stages.
2. **Log Messages:** Use log messages to record information at various points in your workflow. This helps track the flow of execution and identify where issues may be occurring.
3. **Immediate Panel:** Utilize the Immediate Panel to execute expressions and view their results in real-time. This is useful for testing small pieces of code without running the entire workflow.
4. **Highlight Elements:** Enable the option to highlight elements during execution to visually confirm that the correct elements are being interacted with.
5. **SaveMyLeads Integration:** For projects involving data integration, use services like SaveMyLeads to streamline and monitor data flow between different platforms, ensuring that integrations are functioning correctly.

By leveraging these debugging techniques and tools, you can effectively troubleshoot and optimize your UiPath automation projects. Proper debugging not only saves time but also enhances the reliability and performance of your workflows.

Catching errors:

Try Catch

System.Activities.Statements.TryCatch

Description

Catches a specified exception type in a sequence or activity, and either displays an error notification or dismisses it and continues the execution.

There is no limit to how many **Catches** you can use in a **Try Catch** activity. This activity requires at least one catch to be added.

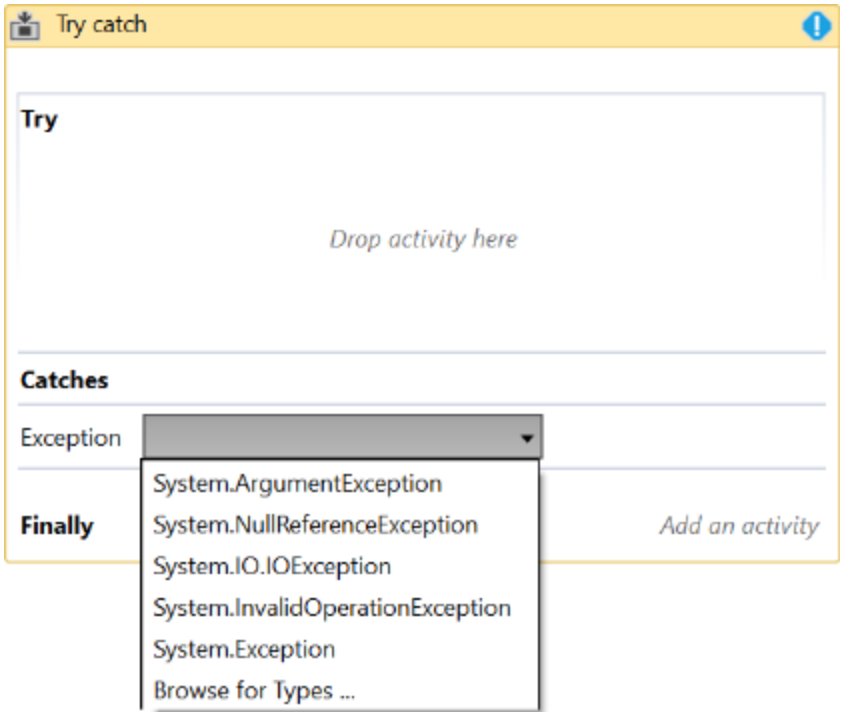
Project compatibility

Windows - Legacy | Windows | Cross-platform

Configuration

The activity body contains three fields:

- **Try** - The activity performed which has a chance of throwing an error.
- **Catches** - The activity or set of activities to be performed when an error occurs.
    - **Exception** - The exception type to look for. You can add multiple exceptions.
- **Finally** - The activity or set of activities to be performed after the **Try** and **Catches** blocks are executed. This section is executed only when no exceptions are thrown or when an error occurs and is caught in the **Catches** section.



Note:

- If an activity is included in the **Try** section, and the value of the **ContinueOnError** property is True, no error is caught when the project is executed.
- The **Try Catch** activity does not catch fatal exceptions such as:
    - FatalException
    - OutOfMemoryException
    - ThreadAbortException
    - FatalInternalException

Properties

Common

- **DisplayName** - The display name of the activity.

Misc

- **Private** - If selected, the values of variables and arguments are no longer logged at Verbose level.

Note: Pressing "Ctrl + T" places the selected activity inside the **Try** section of a **Try Catch** activity.

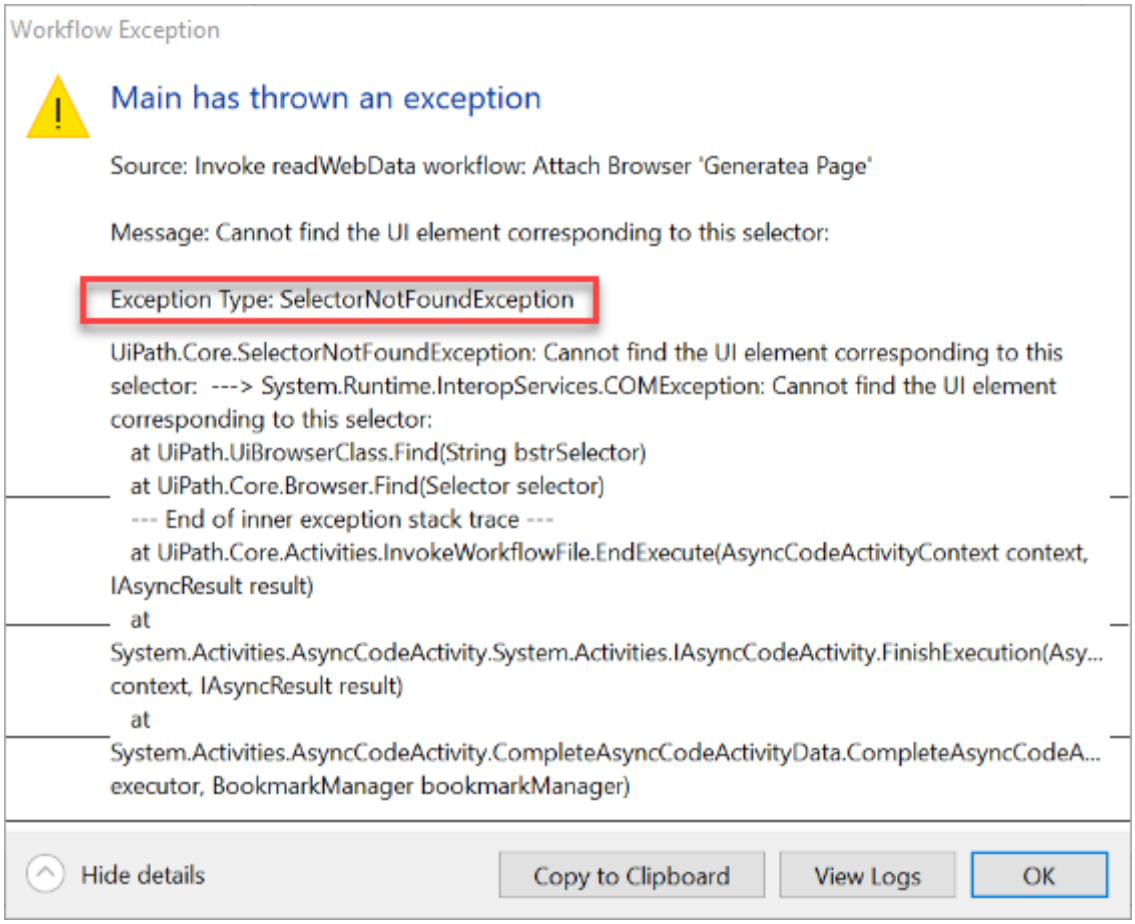
Example of using the Try Catch activity

To better understand the importance of the **Try Catch** activity, we created an automation that gathers multiple names from a random name generator website and writes them in an Excel spreadsheet.

Note: Due to a technical limitation, the reported exception Source in a Try-Catch scenario might differ if the exception occurs inside of an invoked workflow.

A **Build Data Table** activity is used to create a table in which to store the gathered names. Another workflow is invoked to read the web data. Finally, an **Excel application scope** activity is used to write the gathered information in the Excel file.

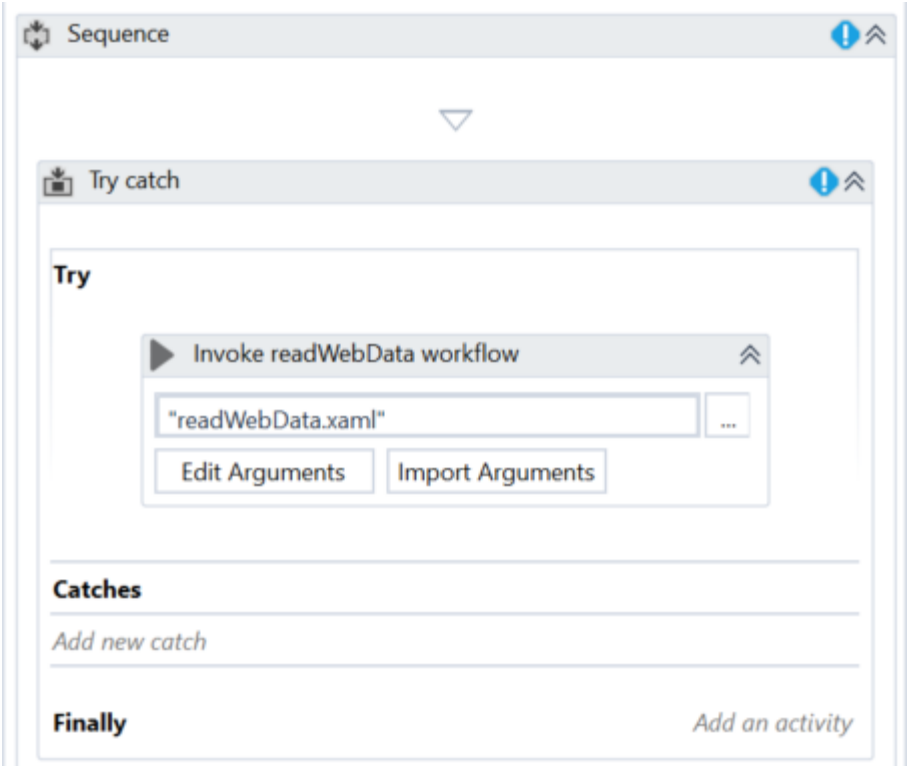
First of all, let's run the automation to check for any errors. Notice that a **Workflow Exception** window is displayed. The **Exception Type** field tells us what the problem is. This is used in the **Catches** section of a **Try Catch** as the exception type to look for during the workflow execution.



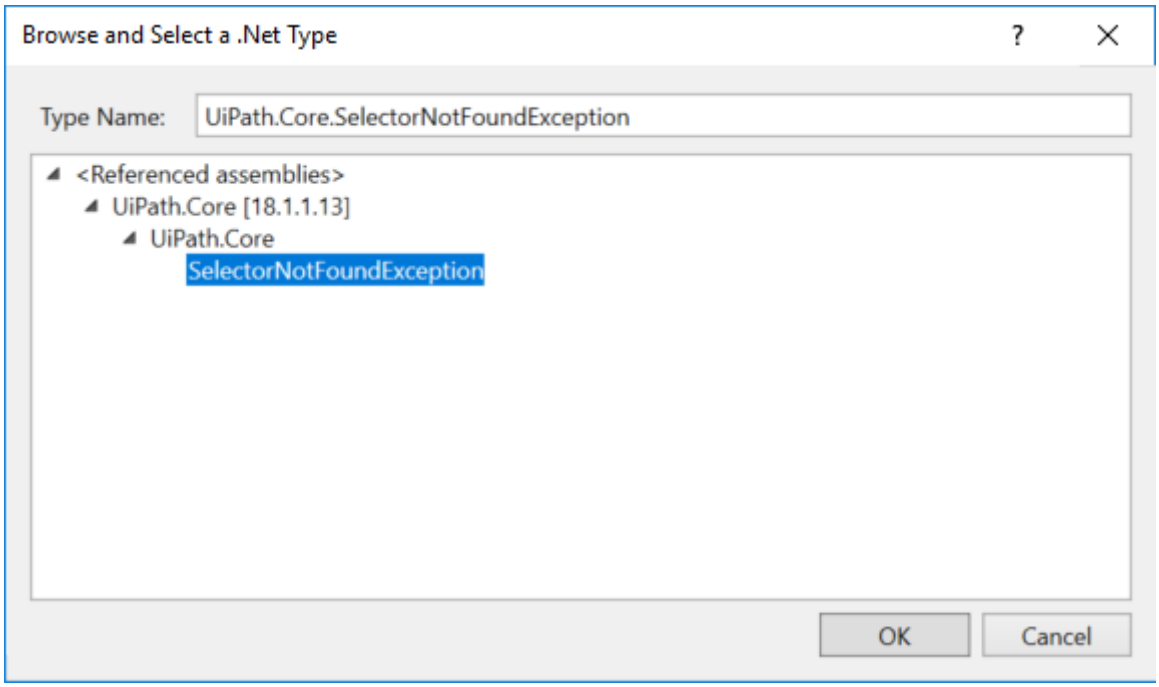
As you can see in the screenshot above, when running the example workflow, there seems to be a problem with the **Attach Browser** container selector. The issue is that the selector fails to identify the browser window with the “Generate a Random Name - Fake Name Generator” name.

To catch this exception, we need to perform the following:

1. Drag the **Try Catch** activity from the **Activities** panel above the **Invoke workflow** activity.
2. Place the **Invoke workflow** activity in the **Try** section of the **Try Catch** activity. This watches the **Invoke workflow** activity in case it throws an error.

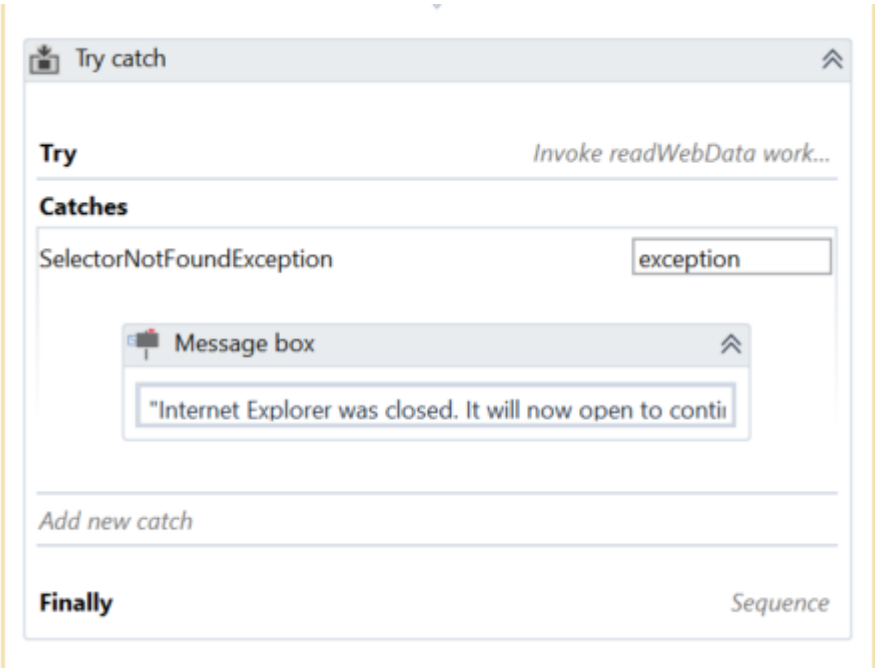


3. In the **Catches** section, select the UiPath.Core.SelectorNotFoundException exception from the drop-down. If it's not there, you can find it in the Browse and Select a .Net Type window.



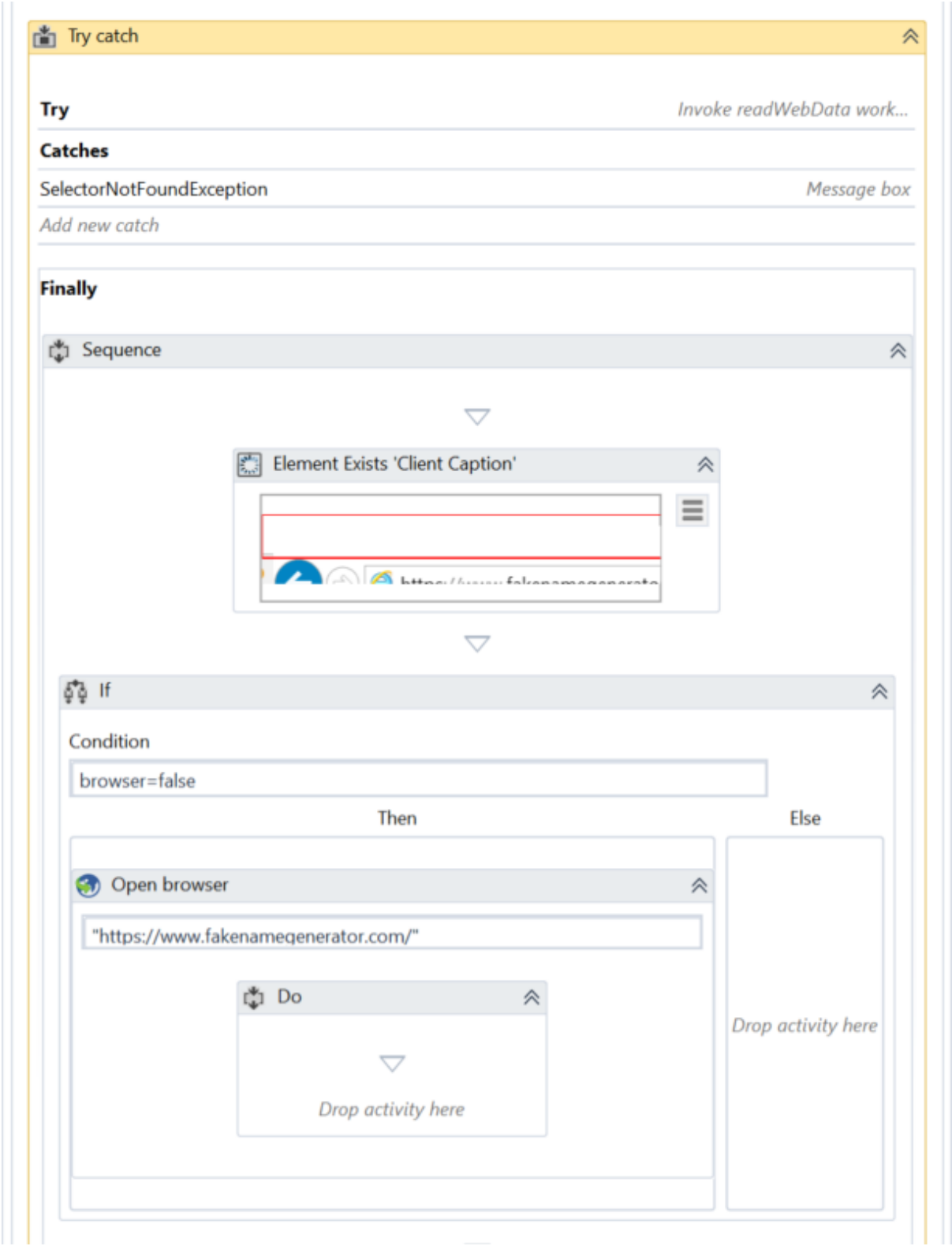
4. Optionally, you can add a **Message Box** activity in the **Catches** section. You can fill in the **Content** field with an informative message between quotes, in our case "Internet

Explorer was closed. It will now open to continue the workflow execution". This means that whenever the exception is caught, this message box is displayed, to inform the user that the browser is about to open so that the workflow is successfully executed.



5. Drag the **Element Exists** activity in the **Finally** section. This is used to check if Internet Explorer is open on the page of interest, https://www.fakenamegenerator.com.
6. Open Internet Explorer and access the previously mentioned page.
7. Use the **Indicate on screen** functionality to select the Internet Explorer window.
8. Select the **Element Exists** activity and edit its selector so that it looks like this <wnd app='iexplore.exe' title='Generate a Random Name - Fake Name Generator - Internet Explorer' />. This selector ensures that the **Element Exists** activity only looks for an active Internet Explorer window in which the aforementioned page is open.
9. In the **Output** property, create a variable with a relevant name, such as browser. This is a boolean variable which helps you determine whether or not Internet Explorer is active on the indicated page.
10. Add an **If** activity under the **Element Exists** activity. This is used to open Internet Explorer if it's closed, and continue the workflow otherwise.
11. In the **Condition** field, write browser=false. This condition is used to verify if the browser is opened or not, and perform other actions, based on its value.
12. Drag an **Open Browser** activity in the **Then** section. If the **Condition** is met (the browser is closed), then the **Open Browser** activity is used to open it, without affecting the workflow.
13. In the **Url** field type https://www.fakenamegenerator.com.

14. Leave the **Else** section empty so that the workflow continues as expected if Internet Explorer is already opened on the indicated website.



15. Run the workflow and notice one of the following:

- If Internet Explorer is closed - The user is informed that Internet Explorer is about to open so the workflow can continue. The browser opens, all expected data is gathered and written to the Excel file.
- If Internet Explorer is open - The workflow is executed as expected.