Bot Development Best Practices

Version 1.0.1

Document Information

Version:	1.0.1
Created by:	Automation Anywhere Professional Services Team
Last Modified on:	04/19/2020
Product Compatibility	Applicable to A2019 and above

Table of Contents

Purpose

The purpose of this document is to provide guidelines, standards, and best practices for developing Bots/Automations using Automation Anywhere. Not every person developing an automation:

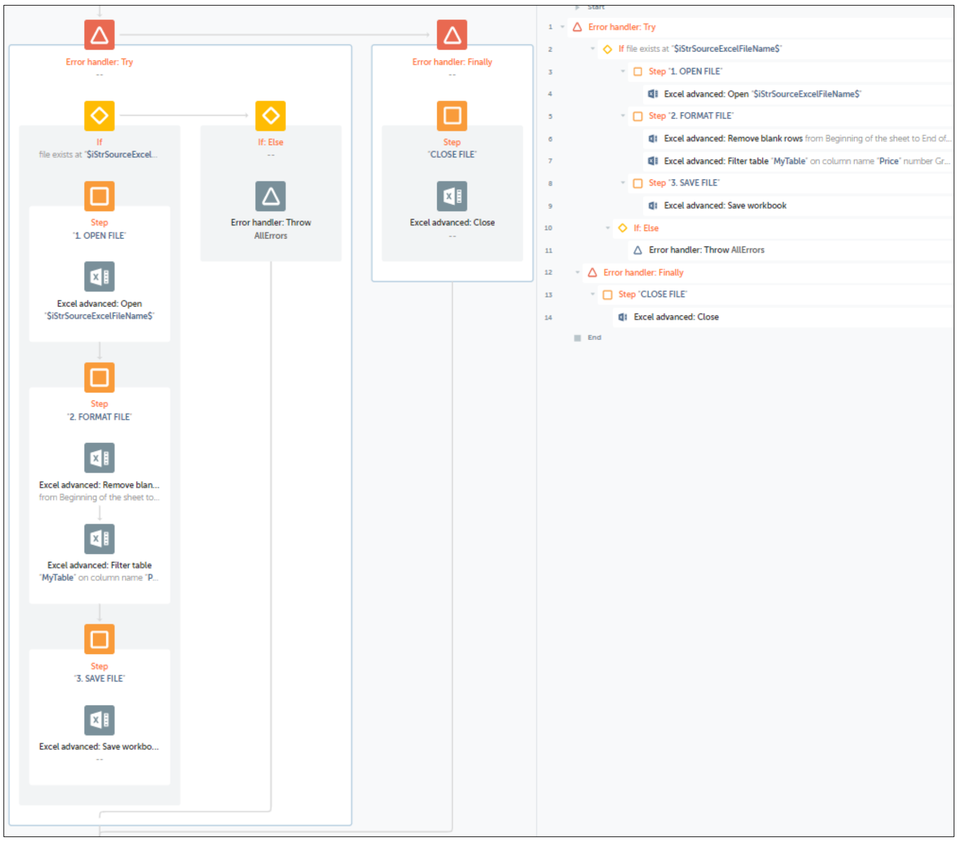
- is aware that a Bot (or any code set) is generally read several times more than it is changed
- is aware of some of the pitfalls of certain constructions in automation development
- is aware of the impact of using (or neglecting to use) certain approach on aspects like maintainability and performance
- knows that not every person developing an automation can understand an automation as well as the original developer

The following standards and best practices will help you create clean, easier to read, easier to maintain, stable automations.

Don’t Repeat Yourself Principle (DRY)

The Don’t Repeat Yourself Principle is a principle commonly found in software engineering. The aim of this principle is to reduce the repetition of information. In the context of automation, it refers to creating a set of actions or logic that is repeated throughout a single Bot.

For example, suppose an automation has the need to apply formatting to an Excel spreadsheet. The Bot might look like the following:



```
Start
Error handler: Try
    If file exists at '$iStrSourceExcelFileName$'
        Step '1. OPEN FILE'
            Excel advanced: Open '$iStrSourceExcelFileName$'
        Step '2. FORMAT FILE'
            Excel advanced: Remove blank rows from Beginning of the sheet to End of...
            Excel advanced: Filter table 'MyTable' on column name 'Price' number Gr...
        Step '3. SAVE FILE'
            Excel advanced: Save workbook
    If: Else
        Error handler: Throw AllErrors
Error handler: Finally
    Step 'CLOSE FILE'
        Excel advanced: Close
End
```

FIGURE 1

Now, let's consider the case where the Excel worksheet needs to be formatted not once, but 3 times. The temptation would be to simply copy and paste these lines into the three places where they are needed. It's fast and simple and helps resolve the immediate goal – getting this automation finished quickly. However, copying and pasting these actions 3 times is short-sighted.

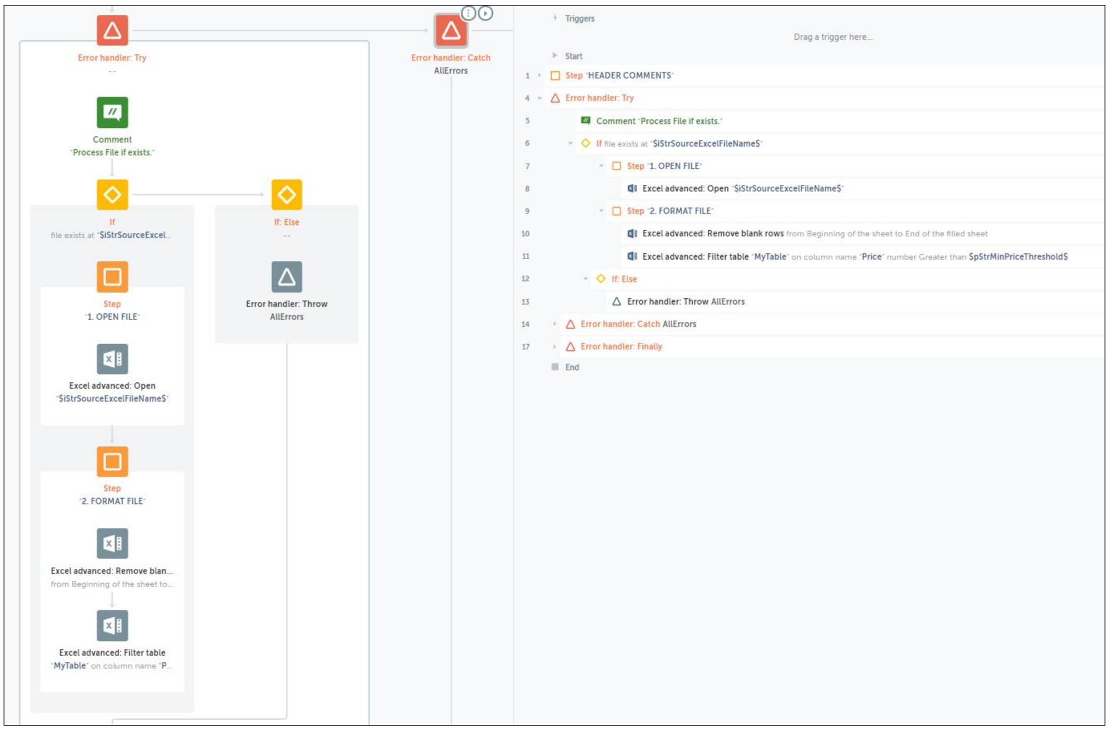
Consider for example a new request for additional formatting to the spreadsheets. Now the entire Bot must be opened and edited. In addition, there are 3 places to find this code, and because this is copied and pasted the automation is now 24 lines longer than it would have been. Long Bots are more difficult to edit and take more time. Additionally, a Bot that was tested and verified to be bug-free and production ready has now been edited and should now be re-tested.

By saving an extra minute of time copying and pasting these actions in 3 different places, 30 minutes to an hour of maintenance time have now been added because the Bot must be edited and is longer than it normally would have been . If a new developer has been assigned to making changes and they are unfamiliar with the automation, they must now do more analysis to determine all the places that need the change. If the Bot is short, that's not so bad, but if it's long, it's much more time consuming.

So, what's the solution? A sub Bot, that is called by the Bot needing this service performed. By placing these actions into a TaskBot that is called by a parent Bot, we realize several benefits:

1) The main Bot is now significantly shorter
2) When the enhancements to Excel formatting are required, only 8 lines must be located, analyzed and edited and they live in a single Bot, making the automation much easier to maintain
3) These actions can be called by any number of Bots, even in other automations

Sub-tasks can be referred to as a "helper Bot" or "utility Bot", since their only purpose is to support other Bots that call them. The following figure is an example of what the helper Bot for the above example might look like:


Error handler: Try
--
Comment
"Process File if exists."
If
file exists at "$iStrSourceExcel...
If: Else
--
Step
"1. OPEN FILE"
Excel advanced: Open
"$iStrSourceExcelFileName$"
Error handler: Throw
AllErrors
Step
"2. FORMAT FILE"
Excel advanced: Remove blan...
from Beginning of the sheet to...
Excel advanced: Filter table
"MyTable" on column name "P...
Error handler: Catch
AllErrors
Triggers
Drag a trigger here...

```
Start
Step "HEADER COMMENTS"
Error handler: Try
    Comment "Process File if exists."
    If file exists at "$iStrSourceExcelFileName$"
        Step "1. OPEN FILE"
            Excel advanced: Open "$iStrSourceExcelFileName$"
        Step "2. FORMAT FILE"
            Excel advanced: Remove blank rows from Beginning of the sheet to End of the filled sheet
            Excel advanced: Filter table "MyTable" on column name "Price" number Greater than $pStrMinPriceThreshold$
    If: Else
        Error handler: Throw AllErrors
Error handler: Catch AllErrors
Error handler: Finally
End
```

FIGURE 2

If any changes are required to this specific set of actions, only this helper Bot will need to be edited, and only this helper Bot will need to be retested.

Tip: Remember that CSV/text file sessions and browser sessions (web recorder) cannot be shared across Bots; Excel sessions could not be shared in versions prior to v11.3.3. So sub-tasks may have to be designed in such a way as to avoid breaking these sessions.

Rule of Three

Keeping in mind the Don't Repeat Yourself Principle, The Rule of Three is a great way to prevent introducing repeating code or actions. This rule is typically only applied to a small number of actions, or even single actions. For example, if a Bot calls a sub-TaskBot, and then calls it again when it fails, it is acceptable to have two call sites; however, if it is to be called five times before giving up, all of the calls should be replaced with loop containing a single call.

Tip: Sub-TaskBots should be small and focused (have only a single responsibility or only a few responsibilities).

Single Responsibility Principle

In object-oriented programming, the single responsibility principle states that every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely covered by the class. This principle directly applies to automation development as well.

Now imagine a single Bot that is 2000 lines long. This Bot is far too long and should be split into several Sub-TaskBots. The automation developer decides to put several of the repeated actions into another Sub-TaskBot. Consider for example he or she picks three repeated sections and puts them in a sub-task. This new helper Bot now handles printing a PDF, but also handles saving a file to a specific folder, and in addition to that it also handles moving a file from one folder to another. The developer manages which part is called by passing an action variable.

The above example would break the Single Responsibility Principle. The developer has reduced the number of lines in the master Bot, which is good, but has now a sub-task that is far too big. Additionally, if any one of those responsibilities (printing a PDF, moving a file, or saving a file) need to be modified, the entire helper Bot must be modified. This creates the possibility for introducing a bug in the Bot that would not have otherwise been affected.

The proper approach would be to create three sub-tasks, each having their own responsibility –

1. Print to a PDF
2. Move the file
3. Save the file

Decoupling and Loose Coupling

When possible, sub-tasks should not have a dependency on the calling Bot. In automation this is often unavoidable. However, developing with this in mind can improve the overall architecture and maintainability of an automation.

Using the login sub-task as an example, if the login Bot can only be called by one single master Bot, then it is tightly coupled to that master Bot. If the login sub-task is designed in such a way that the URL of the page it uses must be set by the calling Bot, then it cannot run by itself. It cannot be unit-tested alone, and other Bots cannot call it without knowing the URL of the login page before calling it. If the calling Bot must provide the login page URL to the login sub-task, then all tasks that use the login sub-task are more tightly coupled to that login sub-task. And if the URL changes, more than one Bot must be changed.

However, if the login sub-task contains all the information it needs to login to the web application, including the URL, then it is a truly stand-alone sub-task. It can be unit-tested, and it can be called by any other Bot without the need to be provided the URL. It is then "decoupled" from other Bots and is much more maintainable. This needs to be evaluated on case by case basis, since it may not always be possible to decouple.

Bot Design

Most well-designed automations will involve more than one Bot. It is useful to define standards for the types of Bots to support a Use Case, with their responsibilities and interactions.

Naming Convention

Use **PascalCasing** for Bot names. **Pascal Case** is the practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter (e.g., PrintUtility)

Keep Bot names short and concise. Consider limiting use of unnecessary characters such as underscores and blank spaces in Bot Names.

Name Bots as "MASTER" or "MAIN" if directly executed (e.g., by Control Room, API, or trigger – not called by any other Task). Identification of these Bots is important, as they may have unique responsibilities for activities such as reading Global Values/Configuration, Pre/Post-Processing, etc. (See Design Patterns)

For Bots which are always called by other Tasks, use a suffix such as "Utility" or "Helper", for example FileSaveHelper.bot.

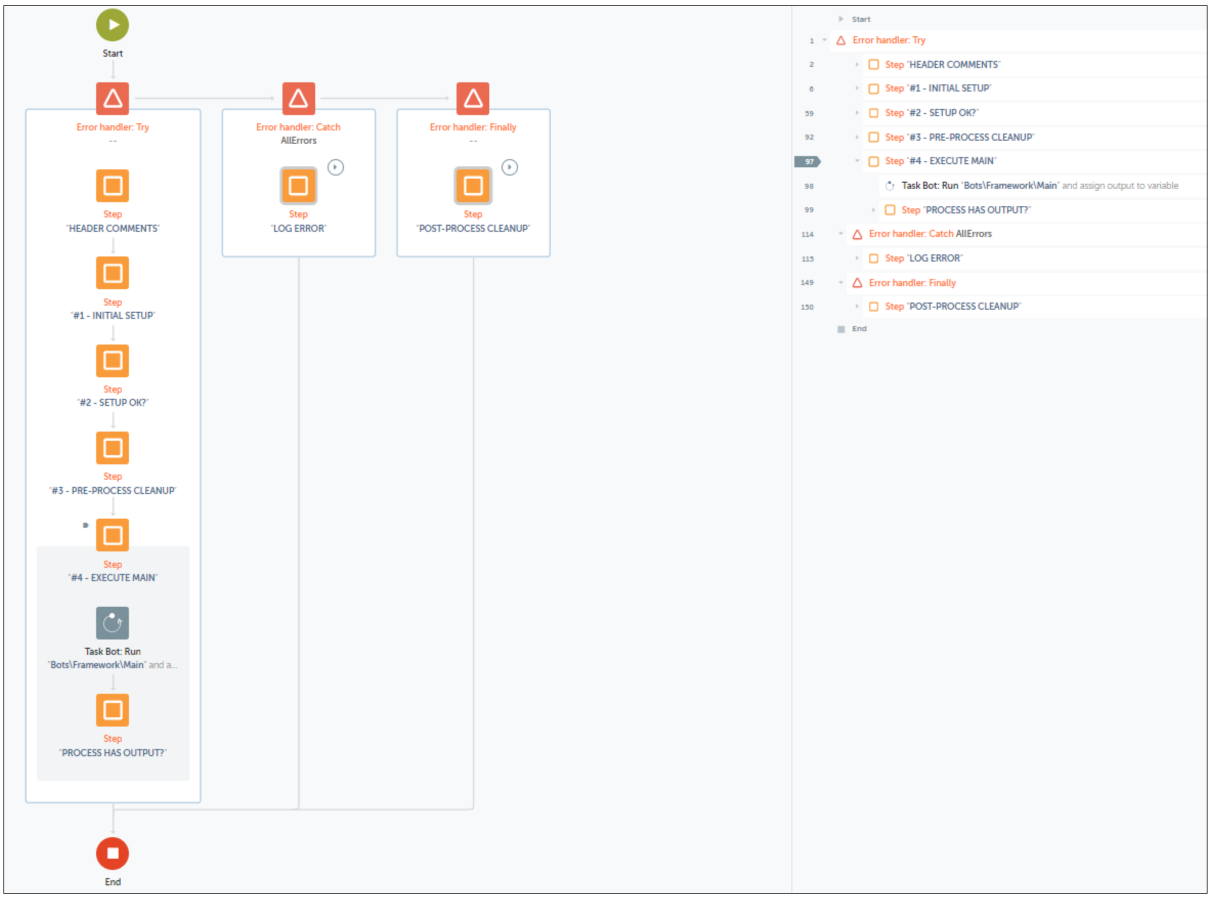
Design Patterns

The following are patterns for consideration, upon which "templates" could be developed for use when starting development on a new Use Case.

Design Pattern #1 – Master, Main, Subs

MASTER Bot

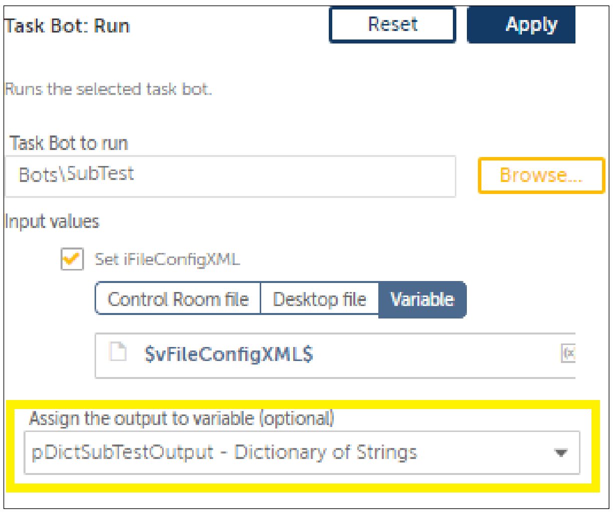
- The Bot directly called to initiate the process, via mechanisms including scheduling via Control Room, or API call.
- Major process steps in the TRY section could include the following:
  - Initial setup for the process
  - Validation that setup was successful (e.g., All required folders exist? Files? Initial variable values populated as required?)
  - Execute Pre-Processing "Desktop Clean-up"
  - Call the MAIN Bot to execute the business logic of the process
- In the FINALLY section for the outer TRY/CATCH block, Post-Processing "Desktop Clean-up" could be executed



```
Start
Error handler: Try
    Step "HEADER COMMENTS"
    Step "#1 - INITIAL SETUP"
    Step "#2 - SETUP OK?"
    Step "#3 - PRE-PROCESS CLEANUP"
    Step "#4 - EXECUTE MAIN"
        Task Bot: Run "Bots\Framework\Main" and assign output to variable
        Step "PROCESS HAS OUTPUT?"
Error handler: Catch AllErrors
    Step "LOG ERROR"
Error handler: Finally
    Step "POST-PROCESS CLEANUP"
End
```

FIGURE 3

– For any “Run TaskBot” action calling a “Sub” Bot, always specify a Dictionary to hold any values the “Sub” may return. At a minimum, it is recommended that any “Sub” return a message (e.g., oStrResult) to its calling Bot to indicate if any error occurred, and an error line number if applicable.


Task Bot: Run

Runs the selected task bot.

Task Bot to run: Bots\SubTest

Input values

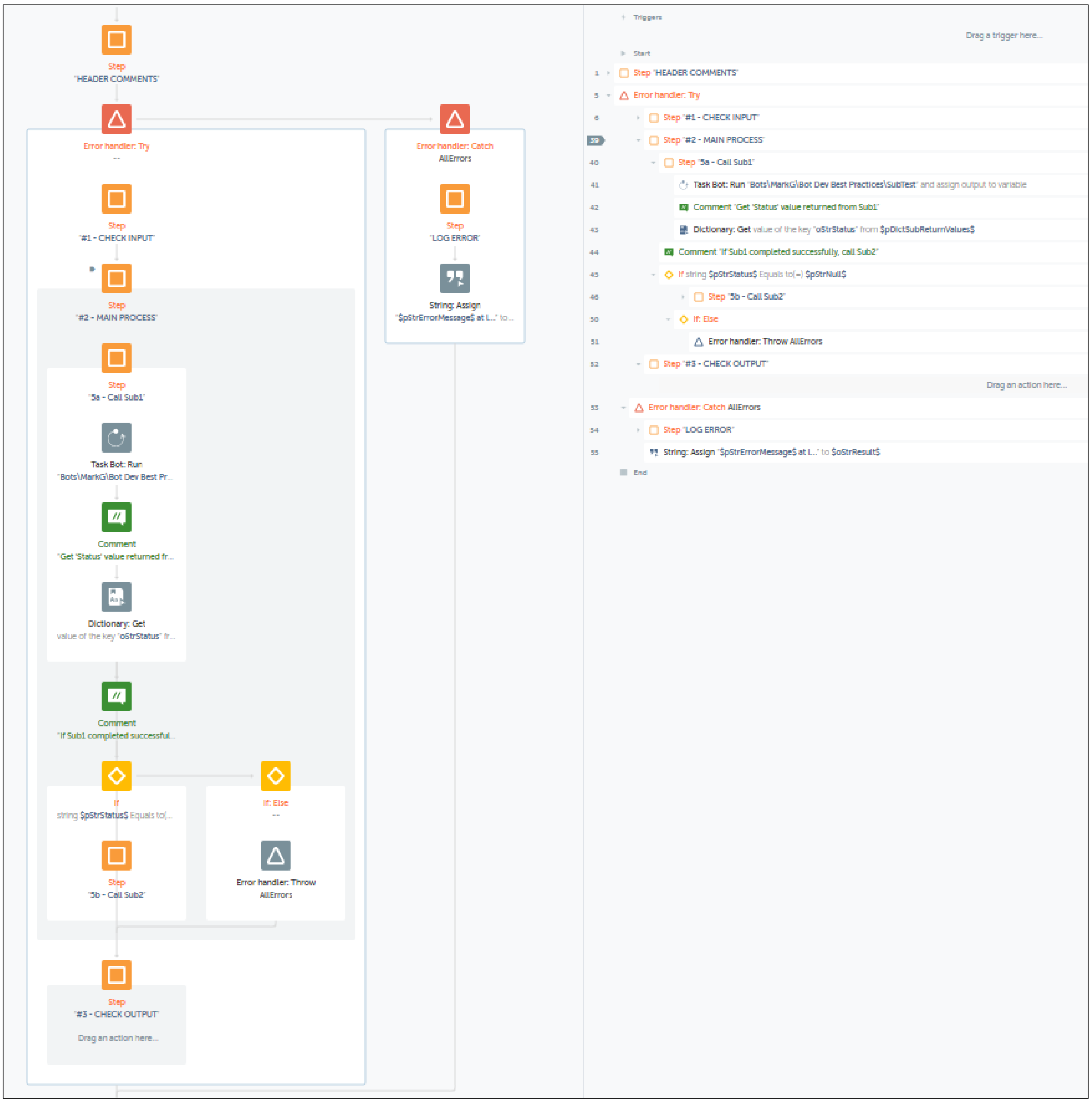
Set iFileConfigXML: Variable — $vFileConfigXML$

Assign the output to variable (optional): pDictSubTestOutput - Dictionary of Strings

FIGURE 4

MAIN Bot

– The Bot calls “Sub” Bots as necessary to execute the business logic of the process.
– Major process steps in the TRY section could include the following:
  - Validation of any input (e.g., Input variable values from MASTER, etc.)
  - Execution of “Sub” Bots. After execution of a “Sub” Bot, validate the result (e.g., oStrResult) in case it impacts the over-all process (e.g., Does it make sense to run “Sub” #2 if #1 failed?)
  - Validation of any output
  - Ensure population of any ‘output’ variable values based on the execution of MAIN, to return to MASTER (e.g., Assign value to oStrResult)
– In the CATCH section, log the error, and ensure population of any ‘output’ variable values (e.g., oStrResult) to pass back to MASTER
– Below is a sample outline for a “Main” Bot using this Design Pattern.

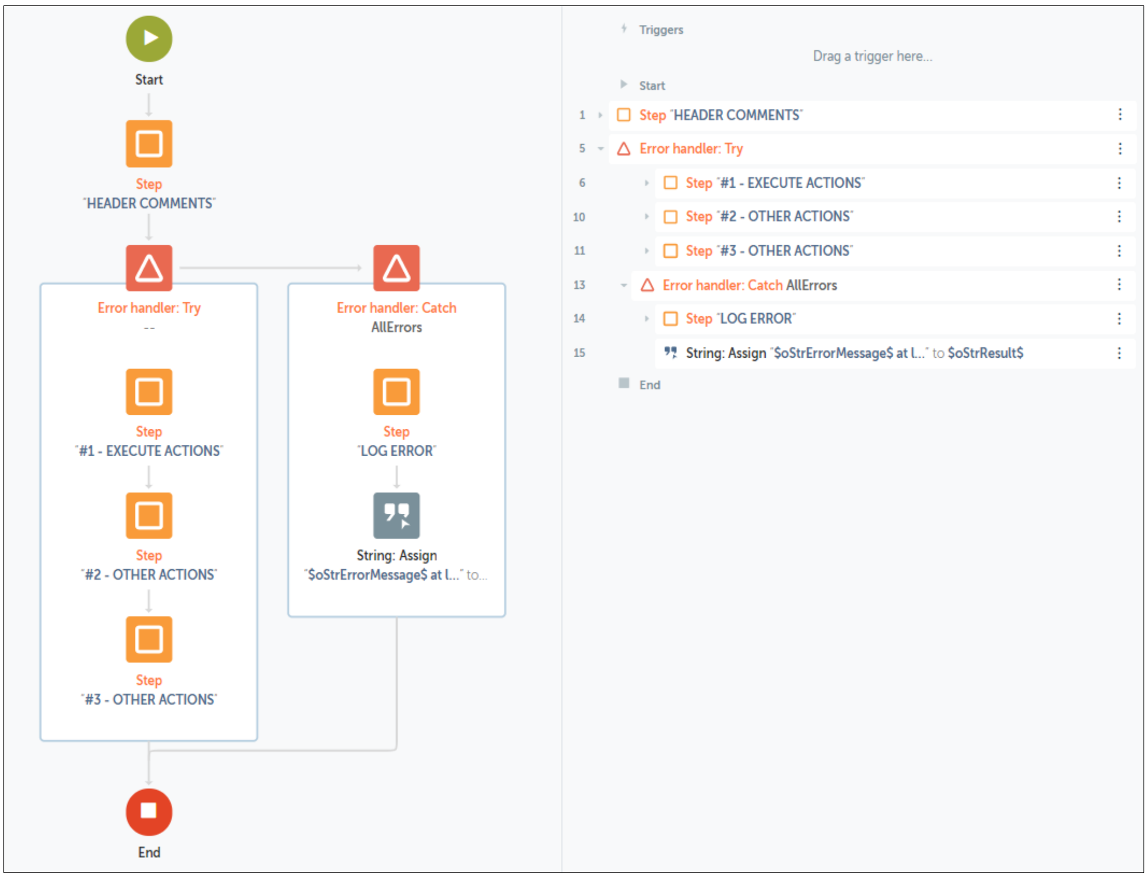

Step
'HEADER COMMENTS'
Error handler: Try
Error handler: Catch AllErrors
Step
'#1 - CHECK INPUT'
Step
'LOG ERROR'
String: Assign
'$pStrErrorMessage$ at I...' to...
Step
'#2 - MAIN PROCESS'
Step
'5a - Call Sub1'
Task Bot: Run
'Bots\MarkG\Bot Dev Best Pr...
Comment
'Get 'Status' value returned fr...
Dictionary: Get
value of the key 'oStrStatus' fr...
Comment
'If Sub1 completed successful...
If
string $pStrStatus$ Equals to(...
If: Else
Step
'5b - Call Sub2'
Error handler: Throw AllErrors
Step
'#3 - CHECK OUTPUT'
Drag an action here...

```
Triggers
                                                    Drag a trigger here...
Start
Step 'HEADER COMMENTS'
Error handler: Try
    Step '#1 - CHECK INPUT'
    Step '#2 - MAIN PROCESS'
        Step '5a - Call Sub1'
            Task Bot: Run 'Bots\MarkG\Bot Dev Best Practices\SubTest' and assign output to variable
            Comment 'Get 'Status' value returned from Sub1'
            Dictionary: Get value of the key 'oStrStatus' from $pDictSubReturnValues$
        Comment 'If Sub1 completed successfully, call Sub2'
        If string $pStrStatus$ Equals to(=) $pStrNull$
            Step '5b - Call Sub2'
        If: Else
            Error handler: Throw AllErrors
    Step '#3 - CHECK OUTPUT'
                                                    Drag an action here...
Error handler: Catch AllErrors
    Step 'LOG ERROR'
    String: Assign '$pStrErrorMessage$ at l...' to $oStrResult$
End
```

FIGURE 5

“Sub” Bots

- These Bots can be designed as necessary to execute the actual business logic required for the automation.
- Use ‘output’ variables to return a result indicator to the calling Bot (either ‘Main’, or another ‘Sub’ Bot) – e.g., oStrResult. The value could contain an error message if the processing was unsuccessful (error or exception occurred).
- Below is a sample outline for a “Sub” Bot



```
Triggers
        Drag a trigger here...
Start
Step "HEADER COMMENTS"
Error handler: Try
    Step "#1 - EXECUTE ACTIONS"
    Step "#2 - OTHER ACTIONS"
    Step "#3 - OTHER ACTIONS"
Error handler: Catch AllErrors
    Step "LOG ERROR"
    String: Assign "$oStrErrorMessage$ at l..." to $oStrResult$
End
```

FIGURE 6

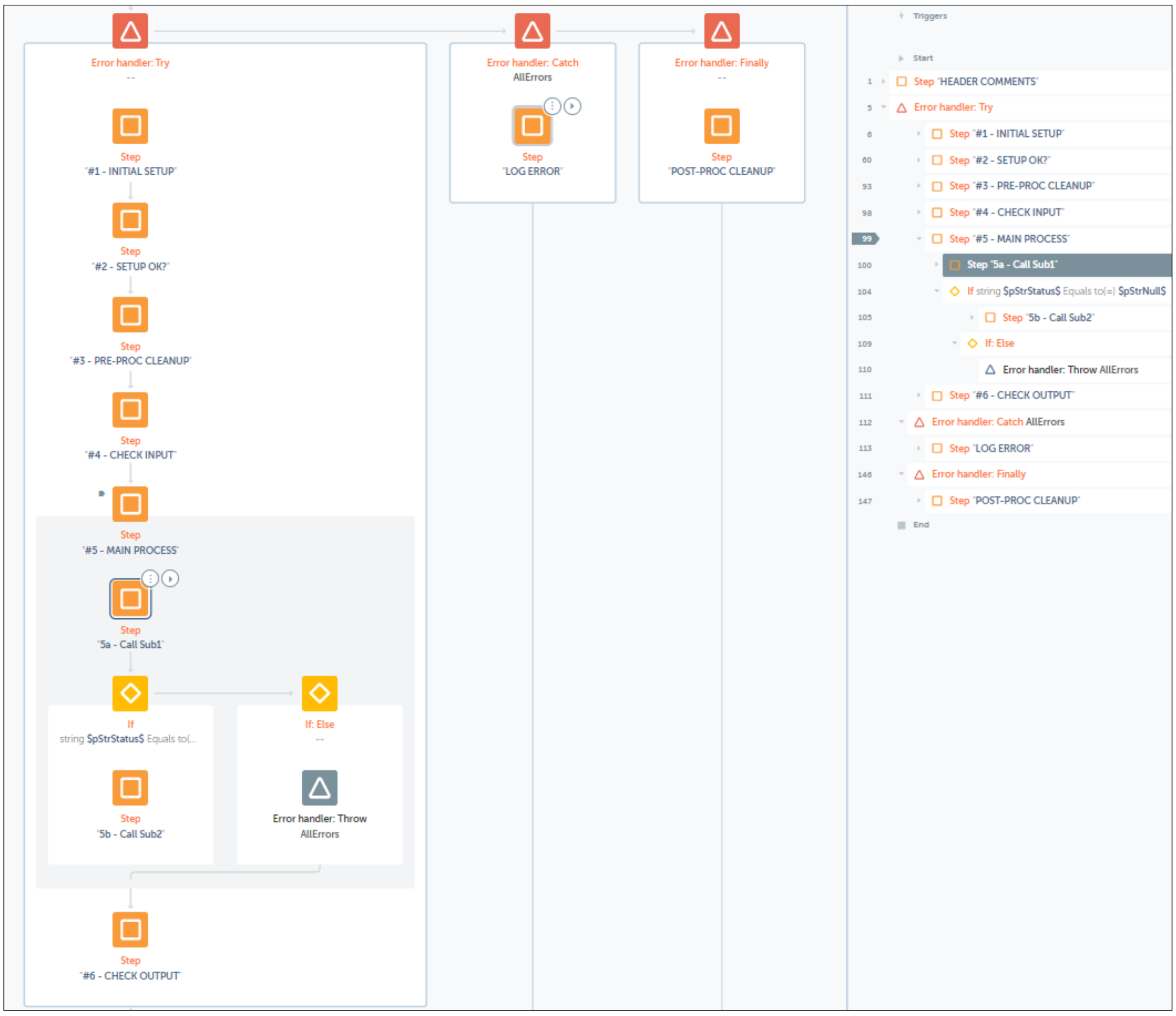
Note important actions executed in the “Catch” section – log the error (see Logging), then assign a value to variable oStrResult to be available to the “calling” Bot.

Design Pattern #2 – Main, Subs

This design pattern is a variation on #1, except that the functions included in ‘Master’ and ‘Main’ are all included in a single ‘Main’ Bot.

MAIN Bot

The following is a sample process flow for a ‘Main’ Bot using this pattern:



```
Triggers

Start
Step "HEADER COMMENTS"
Error handler: Try
    Step "#1 - INITIAL SETUP"
    Step "#2 - SETUP OK?"
    Step "#3 - PRE-PROC CLEANUP"
    Step "#4 - CHECK INPUT"
    Step "#5 - MAIN PROCESS"
        Step "5a - Call Sub1"
        If string $pStrStatus$ Equals to(=) $pStrNull$
            Step "5b - Call Sub2"
        If: Else
            Error handler: Throw AllErrors
    Step "#6 - CHECK OUTPUT"
Error handler: Catch AllErrors
    Step "LOG ERROR"
Error handler: Finally
    Step "POST-PROC CLEANUP"
End
```

FIGURE 7

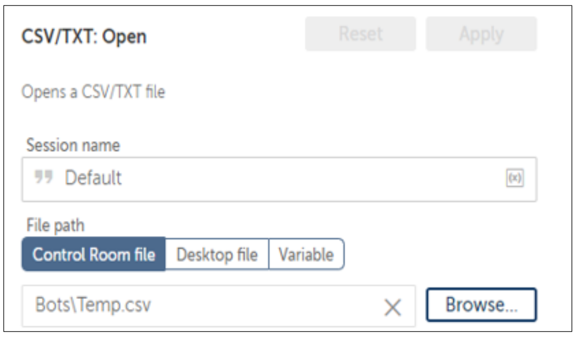
“Sub” Bots

– The general description of a “Sub” Bot from Design Pattern #1 applies to this Design Pattern.

Dependencies

All the files that are uploaded to the A2019 Control Room and are referenced in a Bot's Actions are considered dependencies for that Bot. Checking-In the Bot will automatically check-In the dependency files too. If the bot is Imported/Exported, the bot dependencies will be imported/exported along with the main bot.

It is recommended to upload the dependency files to the respective use case folders.


CSV/TXT: Open

Opens a CSV/TXT file

Session name: Default

File path: Control Room file | Desktop file | Variable

Bots\Temp.csv

FIGURE 8

Design for Reuse / Testability

If automations are developed using the above principles, the result will be many smaller Bots, many being reusable. These smaller Bots can then be tested alone, in a unit-test fashion.

For example, take an automation that always starts with a login to a web-based application. The login step should be in a separate sub-task. Using this approach, if anything changes with the login UI, only the login Bot must be modified. If several other Bots are using the login step, none of those Bots will need to be modified or re-tested if the login UI changes. If the Bot is created using the Single Responsibility Principle, it only performs one function – logging in to the application. Additionally, if the Bot is designed to be loosely coupled, it doesn't need to know anything about the master Bot calling it. The URL, user name and password can be put into the automation for testing purposes, and then that task can be tested independently. This is far more efficient than modifying a single Bot or set of Bots if something changes in a smaller unit.

This approach may not be possible in all cases, but when possible, this can make an automation much easier to maintain and deploy.

Steps

The "Step" action introduced in the A2019 release introduces a powerful way to organize actions into logical groups. When used effectively, use of the action greatly improves the readability of the code.

General rules:

- Use the Step action liberally in code, grouping related actions.
- Always include a title for the action and try to limit to no more than a few words. If more detail is required, a Comment action can be added as the first action in the Step.
- Make titles meaningful. Avoid non-descriptive titles like "Step 1".
- In each Bot, the top-level process steps can use CAPITAL letters, with sub-steps using mixed case. Additionally, the title can include a reference number to the process step in the Solution/Technical Design Document. These steps not only improve readability, but help ensure alignment between the Bots developed and the solution design.

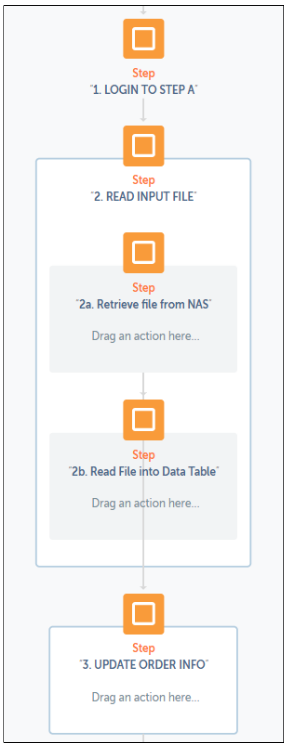


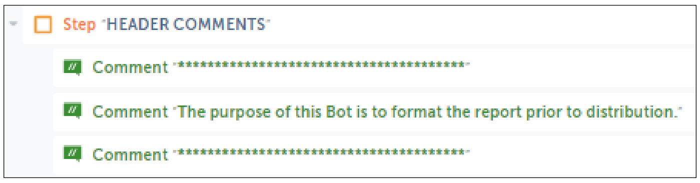
Figure 9

Commenting

Most automations require changes after they are placed into production. Sometimes those changes can be frequent, depending on the type and scope of the automation. The difference between a change being a relatively straight-forward Bot and a complete nightmare is determined by two things: how cleanly the automation was architected, and how well it is documented and commented. Good commenting can mean a difference of hours during a maintenance cycle.

General rules:

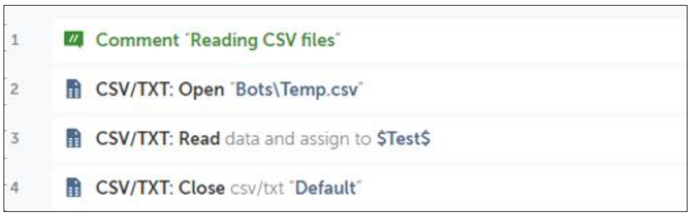
- Enclose multi-line comments within a "Step" action, to improve readability, particularly when in "Flow" view



```
Step "HEADER COMMENTS"
    Comment "****************************************"
    Comment "The purpose of this Bot is to format the report prior to distribution."
    Comment "****************************************"
```

Figure 10

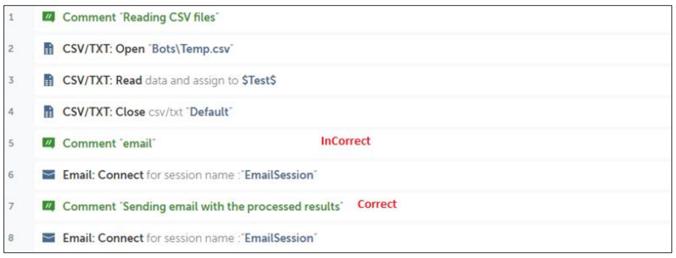
- Use one-line comments to explain assumptions, known issues and logic insights, or mark automation segments



```
Comment "Reading CSV files"
CSV/TXT: Open "Bots\Temp.csv"
CSV/TXT: Read data and assign to $Test$
CSV/TXT: Close csv/txt "Default"
```

Figure 11

- Make comments meaningful.



```
Comment `Reading CSV files`
CSV/TXT: Open `Bots\Temp.csv`
CSV/TXT: Read data and assign to $Test$
CSV/TXT: Close csv/txt `Default`
Comment `email`                                   InCorrect
Email: Connect for session name :`EmailSession`
Comment `Sending email with the processed results`   Correct
Email: Connect for session name :`EmailSession`
```

FIGURE 12

- When automating User Interfaces (UI) use Comments to denote where significant activities are taking place, so that Developer/Reader doesn't have to guess what UI element is getting affected.
- Include comments using Task-List keyword flags to allow comment-filtering.

Example:

// TODO: Place database action here
// UNDONE: Removed keystroke action due to errors here

- Always delete disabled commands before promoting your Bots to Production.
- Try to focus comments on the why and what of an action block and not the how. Try to help the reader understand why you chose a certain solution or approach and what you are trying to achieve. If applicable, also mention that you chose an alternative solution because you ran into a problem with the obvious solution.

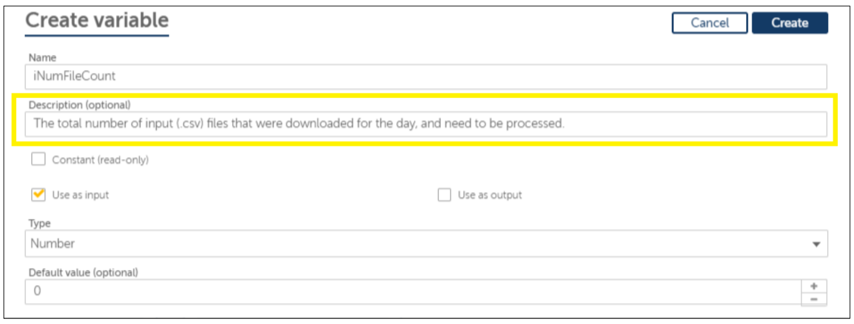
Variables

Use **camelCasing** for Variables. **Camel Case** is the same as **Pascal Case**, except that it always begins with a lowercase letter (e.g., backgroundColor).

Avoid single character variable names. Never use "i" or "x" for example. A reader should always be able to look at a variable name and gain some clue about what it is for, e.g., pStrEmployeeFirstName or pNumSocialSecurityNumber.

Avoid initializing your variables within the "Create variable" dialog, except where you are defining constant values. Instead, initialize variables using Assign Actions under the appropriate package (ie. String, Number, etc.).

Always include a meaningful description for variables you define.



Name: iNumFileCount

Description (optional): The total number of input (.csv) files that were downloaded for the day, and need to be processed.

☐ Constant (read-only)

☑ Use as input ☐ Use as output

Type: Number

Default value (optional): 0

FIGURE 13

NOTE: The following is a recommended naming standard for variables. While there are many options for naming conventions, ultimately it is most important that *some* standard be adopted and used consistently within the organization.

<type/scope indicator><data type><variable name>

<type/scope indicator> is a single character as follows:

- p = local variable (neither input nor output)
- i = input variable
- o = output variable
- io = input and output variable
- c = constant

<data type> indicates one of the data types introduced in A2019, as follows:

- Str = String
- Num = Number
- Table = Table
- Dict = Dictionary
- List = List
- Date = Date Time
- Bool = Boolean
- File = File
- Rec = Record
- Win = Window
- Any = Any

The following are some sample variable names using this standard:

- iStrAuditLogPath – a string received from calling Task
- oNumReturnValue - a number returned to calling Task
- pRecExcelRow – a record used locally (not "received from" nor "returned to" the calling task)
- pBoolFileExists – a Boolean used locally (not "received from" nor "returned to" the calling task)
- ioStrStatus – a string that is both "received from" and "returned to" the calling Task
- cStrNull – a string that holds no value – e.g., useful for String comparisons to see if a value is present.

This standard is helpful to facilitate searches for the ‘type’ of variable used in a Bot. For example, “iStr” can be used to find “input String values”. Dictionaries returned a calling Bot can be identified using “oDict”.

Name Flags with Is, Has, Can, Allows or Supports, like pBoolIsAvailable, pBoolIsNotAvailable, pBoolHasBeenUpdated. Name Tasks with a noun, noun phrase, or adjective.

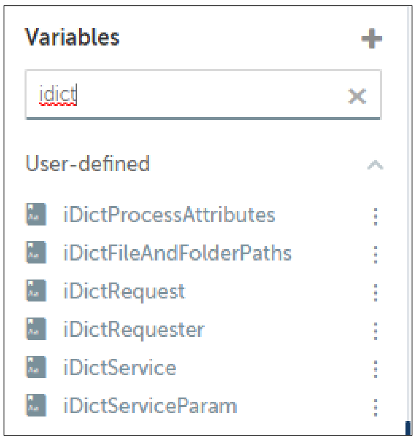


FIGURE 14

Sessions

A “session” can be thought of as a Reference Point to external files/data, and is a part of Actions such as Excel Basic/Advanced: Open, XML: Start session, etc. Once a Session Name is defined in such an Action, then the same file/data set can be accessed within subsequent actions, for example Get Cells, just by specifying the same Session name. Therefore, having unique and meaningful Session name is essential.

DO NOT use “Default” as Session name. You may also use a Variable in place of Session Name, which will further reduce possibility of making typos while typing Session names in actions.

Make sure you close all the Sessions that you open, to release lock/hold on external data source.

Folder Structure

Files that the Bot/Task uses, e.g., config, input, output, logs, etc., should be grouped in meaningful and segregated Folders. This should be further separated for different Environments. You don’t want to mix up DEV environment Output files and Logs with PROD environment. Access to specific UAT and PROD folders, such as Config, can be restricted only to Admins and respective Environment’s Bot/Service Accounts, to prevent sensitive data/values from getting exposed to Developers or unintended users. An example of such folder structure is shown.

It is recommended to host this folder structure on a NAS Network Shared Folder, instead of the local drive of Bot Runner, because it is very unlikely that the Bot Developers/Support Team will have access to PROD Bot Runner machines, for security reasons.

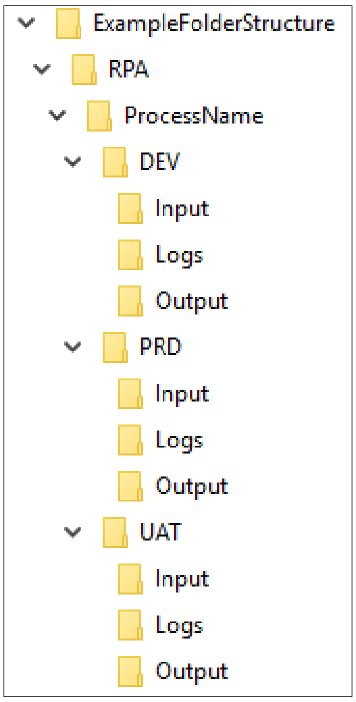


FIGURE 15

You could have a variable called iStrEnvironment in your Tasks, which will at runtime receive value of the Environment in which the Task is currently running, e.g., “DEV”, “UAT”, “PRD”. This value could be returned by a Library Task, e.g., GlobalSettings.atmx, which will look at the $AAControlRoom$ System Variable, which has unique Control Room URL for every environment, and then return appropriate value accordingly to the calling Task, e.g., “DEV”, “UAT”, “PRD”. Subsequently, if variable iStrEnvironment is used to build Path of your Folder Structure, your Task will automatically do the switching between Environment appropriate folders at runtime.

Image Recognition

When using the **Find Image in Window** action, we can search for a User Interface control/element in an application window using a target image. It is recommended that the target image file be stored on the Control Room, along with Bots related to the associated Use Case/Process.

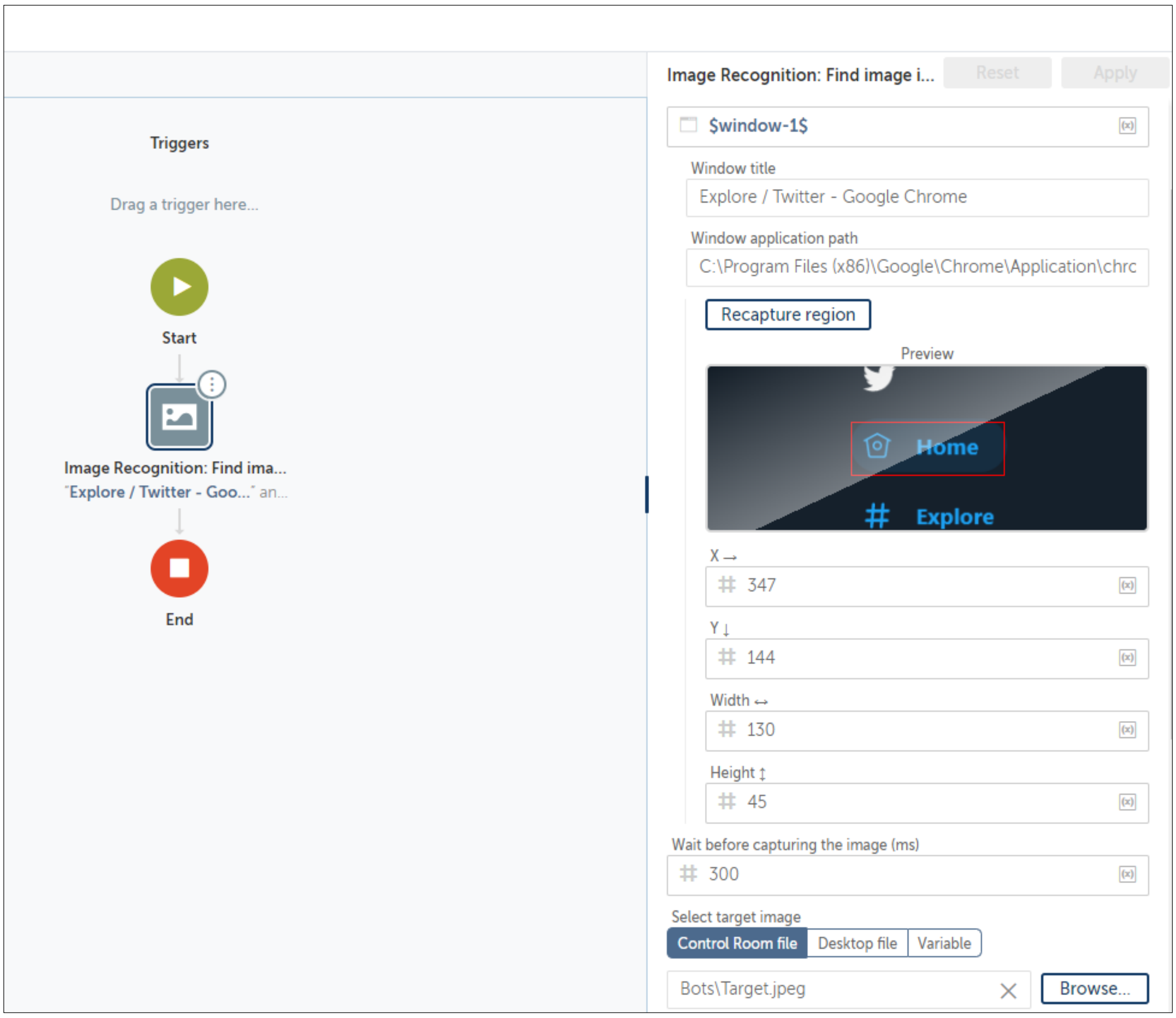


Image Recognition: Find image i...

$window-1$

Window title: Explore / Twitter - Google Chrome

Window application path: C:\Program Files (x86)\Google\Chrome\Application\chrc

X →: 347

Y ↓: 144

Width ↔: 130

Height ↕: 45

Wait before capturing the image (ms): 300

Select target image: Control Room file

Bots\Target.jpeg

FIGURE 16

Action Precedence

The following figure shows preferable actions for low to high portability. The high portability actions mostly do not depend on Bot Runner machine screen's resolution or an Object's coordinates. Hence, are less susceptible to break once ported to an unidentical Bot Runner machine configuration.

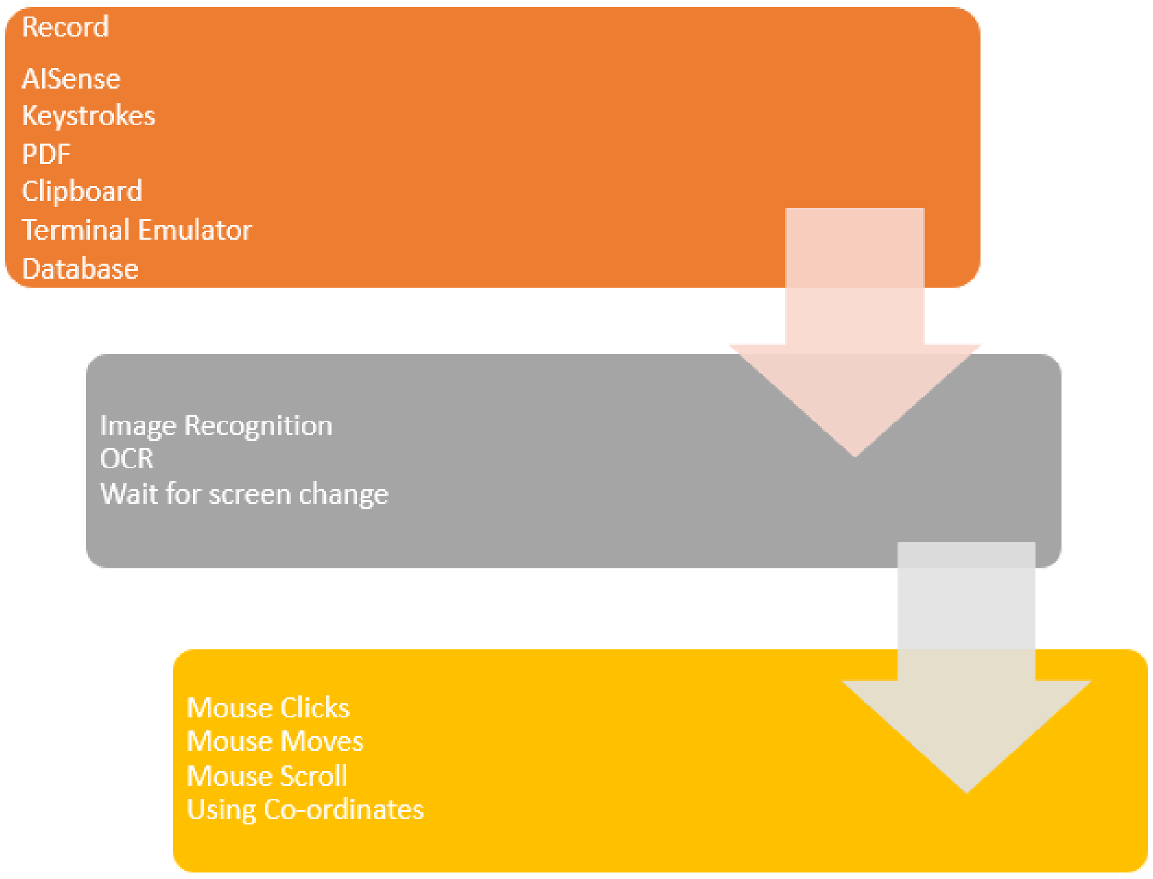


FIGURE 17

Go wild with Wildcard (*)

Asterisk (*) is the Wildcard character in Automation Anywhere world. To give you an example, for "Select Window" property of various actions such as Window -> Close, If Window Exists, Simulate Keystrokes, **Capture** action, to name a few, you can specify one or multiple wildcards (*) in the beginning, middle or end of the Window Title value.

For example, if you want to perform an operation if any window has the word 'Microsoft' in its title, then you can use wildcards to indicate any string before or after 'Microsoft' by updating that value to '***Microsoft***'. The Bot will first search for the exact window title ('Microsoft'), and if it does not find it, it will look for windows with the word Microsoft anywhere in the title.

You can use wildcards as well as variables within **Capture** action Property Values too. This way you can make the same **Capture** action work for different objects in different scenarios. For example, iterating over Rows in a HTML Table, one at a time, with the same **Capture** action.

Recorder - Capture Action

For this action, always choose Object properties with static values. For example – **Name, Class, Type, HTML ID, DOMXPath**. Make sure to exclude properties with dynamic/changing values from Search Criteria, such as **HTML Href, Value**, etc. Do not take this as a thumb rule. At times, you may see that the **Value** property contains static value all the time. In such case you may include that property to Search Criteria.

All Objects will provide a property called **Path**, which could be non-static in some situations. You will need to observe Bot executions and make a judgement call on whether to include or exclude it. General recommendation is to exclude **Path** property, pretty much all the time. Include it only if no other property has static value to uniquely identify that Object.

Web Application Objects have a property called **DOMXPath**, in addition to **Path**. It is highly recommended to include **DOMXPath** property in the Search Criteria since its value is always unique and static. In situations where you need to interact with one object in a collection, for example a Row in a HTML Table, you will need to identify the part of **DOMXPath** value that is varying from one object to another (typically a number starting from 1) and replace it with a variable. Now, you can iterate over each object in the collection by simply incrementing value of that variable by 1 within a Loop.

When using “Set Text”, typically available for Textbox objects, by default the Delay is 0 milliseconds (ms). The **Capture** action will set the “Value” property of the captured object to user provided value. This should work in most cases. However, if you experience that this is not working as expected, then introduce some Delay, such as 250 milliseconds (ms). Then, the **Capture** action will send Keystrokes to the cloned object. This delay will be used between each Keystroke.

You can use wildcards (*) as well as variables within Capture Property Values. This way you can make the same action work for different objects in different scenarios.

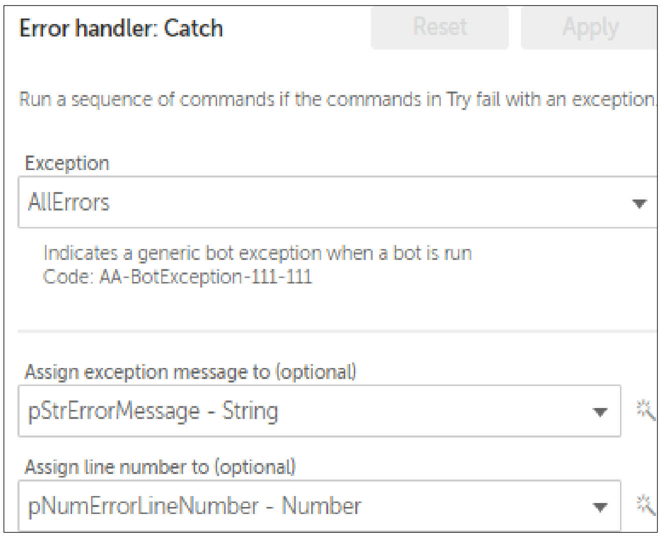
Error Handler (Try/Catch/Throw)

Automating applications, especially browser-based applications, can be a moving target at times. If a web page changes for example, it will often break the automation. Or most commonly, if a Page or Object doesn’t appear when the automation expects, it can cause an error. The key to a successful automation is predicting and handling expected events (a “Save As” dialog not appearing within a specific time frame for example), and unexpected events (a file not found message for example). You are not limited to one Try/Catch block in a Bot. You can and should have multiple and nested Try/Catch sections, as appropriate.

Never assume conditions will always be as you expect them to be. It is essential to make use of the Try/Catch construct in all your Bots. It is VERY IMPORTANT to make sure that your Bot quits gracefully, even in case of an exception, by closing all the application windows that it opened during execution. So that the subsequent Bot that runs on the same machine will not be interrupted by the windows left open by previous Bot’s run

General rules:

- Ensure all actions in a Bot are enclosed within a Try/Catch block.
- It is not recommended to issue a “Throw” action from within a “Catch” block; think of each Try/Catch block as “local” to the Bot.
- Always populate the error message and error line number in the “Catch” action – every Bot can include variables like pStrErrorMessage and pNumErrorLineNumber for this purpose.



Error handler: Catch

Run a sequence of commands if the commands in Try fail with an exception

Exception: AllErrors

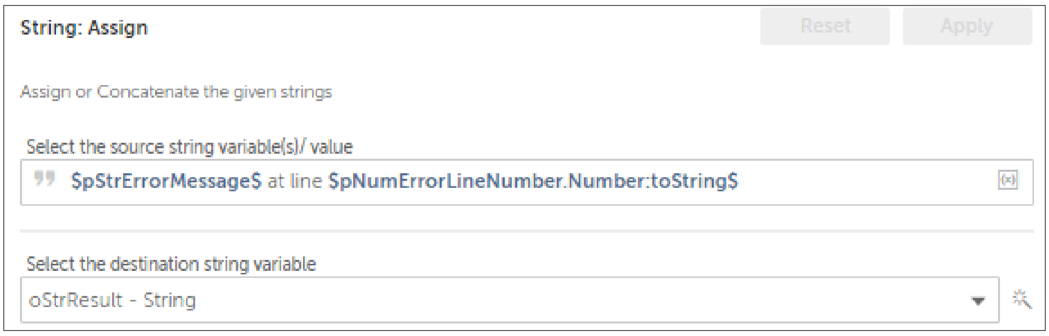
Indicates a generic bot exception when a bot is run
Code: AA-BotException-111-111

Assign exception message to (optional): pStrErrorMessage - String

Assign line number to (optional): pNumErrorLineNumber - Number

FIGURE 18

- If an error / exception is thrown by a “Sub” Bot, it is often necessary to indicate this result to the “calling” Bot. Include variables in any “Sub” Bots like pStrErrorMessage and NumErrorLineNumber, that are populated in the “Catch” action and assigned to an Output variable like $oStrResult$ to make it visible to the Calling Bot:



String: Assign

Assign or Concatenate the given strings

Select the source string variable(s)/ value: $pStrErrorMessage$ at line $pNumErrorLineNumber.Number:toString$

Select the destination string variable: oStrResult - String

FIGURE 19

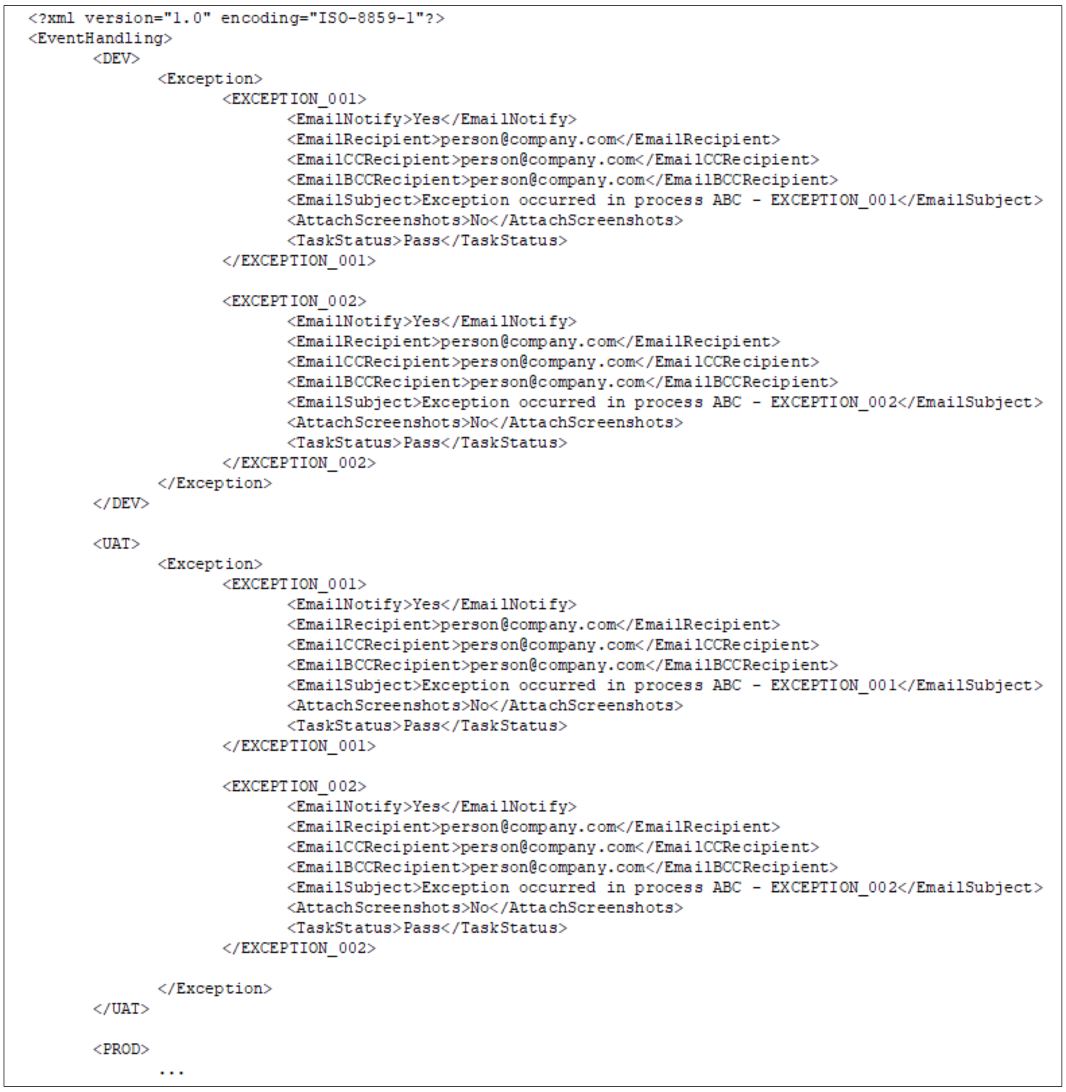
In the Bot that calls the “Sub”, the oStrResult value can assessed to see if an error occurred.

Note: If the Bot has Variables which will be assigned PII (Personally identifiable information) Data, such as Social Security Number, Date of Birth, etc. or the Bot interacts with User Interfaces showing such Data, then DO NOT select “Attach Snapshot” and/or “Attach Variable” options to adhere to Data Privacy policies.

Event/Exception Handling

Beyond the action errors that are caught by Try/Catch, there might be other process events/exceptions for which the code will explicitly check, and subsequently need to be handled. For example, if a certain process condition occurs, we may want to notify someone, and/or write information to a log for additional analysis.

A configurable “Event Handler” Taskbot could be developed, which could achieve this goal, while minimizing the need for code changes if the actions change. For example, an XML file could be maintained, containing a definition of the possible events/exceptions, and any notification requirements when those events/exceptions occur:



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<EventHandling>
        <DEV>
                <Exception>
                        <EXCEPTION_001>
                                <EmailNotify>Yes</EmailNotify>
                                <EmailRecipient>person@company.com</EmailRecipient>
                                <EmailCCRecipient>person@company.com</EmailCCRecipient>
                                <EmailBCCRecipient>person@company.com</EmailBCCRecipient>
                                <EmailSubject>Exception occurred in process ABC - EXCEPTION_001</EmailSubject>
                                <AttachScreenshots>No</AttachScreenshots>
                                <TaskStatus>Pass</TaskStatus>
                        </EXCEPTION_001>

                        <EXCEPTION_002>
                                <EmailNotify>Yes</EmailNotify>
                                <EmailRecipient>person@company.com</EmailRecipient>
                                <EmailCCRecipient>person@company.com</EmailCCRecipient>
                                <EmailBCCRecipient>person@company.com</EmailBCCRecipient>
                                <EmailSubject>Exception occurred in process ABC - EXCEPTION_002</EmailSubject>
                                <AttachScreenshots>No</AttachScreenshots>
                                <TaskStatus>Pass</TaskStatus>
                        </EXCEPTION_002>
                </Exception>
        </DEV>

        <UAT>
                <Exception>
                        <EXCEPTION_001>
                                <EmailNotify>Yes</EmailNotify>
                                <EmailRecipient>person@company.com</EmailRecipient>
                                <EmailCCRecipient>person@company.com</EmailCCRecipient>
                                <EmailBCCRecipient>person@company.com</EmailBCCRecipient>
                                <EmailSubject>Exception occurred in process ABC - EXCEPTION_001</EmailSubject>
                                <AttachScreenshots>No</AttachScreenshots>
                                <TaskStatus>Pass</TaskStatus>
                        </EXCEPTION_001>

                        <EXCEPTION_002>
                                <EmailNotify>Yes</EmailNotify>
                                <EmailRecipient>person@company.com</EmailRecipient>
                                <EmailCCRecipient>person@company.com</EmailCCRecipient>
                                <EmailBCCRecipient>person@company.com</EmailBCCRecipient>
                                <EmailSubject>Exception occurred in process ABC - EXCEPTION_002</EmailSubject>
                                <AttachScreenshots>No</AttachScreenshots>
                                <TaskStatus>Pass</TaskStatus>
                        </EXCEPTION_002>

                </Exception>
        </UAT>

        <PROD>
                ...
```

FIGURE 20

In the code, when such an event/exception occurs, the information could be written to an event log, and the EventHandler Bot called to process the event/exception – e.g., issue a notification using parameters from the XML (e.g., email recipient, CC recipient, subject, etc.). If the notification requirements vary for each environment, or change over time, the configuration file can be updated without a need to change the code.

Logging

Logs should be easy to read and easy to parse. There are two recipients of log files: humans and machines. When humans are the receiver, they may be a developer looking for information in order to debug, analyze performance, or look for errors. They may also be an analyst, looking for audit information or performance information. In either case, logs should be easy to look at and understand, and they should be easy to parse with a tool or import into Excel. What follows is a set of standards to ensure logging is properly executed.

There are about five different types of messages that could be logged. Depending on the type, they may go into separate log files. For example, informational messages should go into the primary process log, where as an error message would into both the process log and an error log. Debugging information should go into a debug log. It is recommended to use CSV as a format for Log files, with Date-Time Stamp value for Column 1. These files can be opened in Excel Application and you can take benefit of features such as Sorting, Filtering, Find, etc. Also, logs aggregation tools such as Slunk (if integrated) will also benefit from receiving log data in structured format.

Use **Log To File** action built into Automation Anywhere. A good format to use is Execution Start Time stamp, Date-Time stamp, machine name, user name, name of the task, and the message. Optionally, you could have a column for logging level (Debug/Error/Info/Warning) if you are PERMITTED to have only one log file. All these values should be comma (,) delimited, to make a valid CSV format log file for easy importing or parsing. Any variable/text which may contain a comma (,), for example $Error Details$ must be qualified with a pair of double-quotes (" ") to avoid breaking that value into two separate columns in your CSV file.

Types of Logs

Note: In all the following logs, the following system variables are useful:

- $System:Date$ (convert to String)
- $System:Machine$
- $System:TaskName$

Audit/Informational Log (Recommended) – This log is meant to be an informational log; ideally, an administrator or business user should be able to understand INFO messages and quickly find out what the application is doing. For example, if an application is all about booking airplane tickets, there should be a single log entry for each ticket saying “[Who] booked ticket from [Where] to [Where]”. Other log entries could include the following: each action that changes the state of the application significantly (database update, external system request). It can be used for monitoring normal operation of a task, but more importantly, it can be used for auditing – as such, the content should be non-technical, business-friendly language.

Some of the events and milestones that can be captured include:

- Main Process start time & end time
- Sub-process start time & end time
- Target Application launch and login success/failure
- Data/File/Folder/etc. validation failure
- Number of Transactions received in Input File
- Number of Transactions Processed without issues
- Number of Transactions failed validation or caused issue

Sample Format of Audit Log CSV File:

Date Time	Machine	Bot Username	TaskName	Description

Error Log (Recommended) – The error log is for detailed error messages. When an error occurs in a task, notification that an error occurred should go into the process log. Detailed information about the error should go into the error log. Logging errors is one of the most important roles of logging. If an error is logged from within a Try/Catch block, the Error Line number and Error Description coming from AA should be included in the message/description.

Sample Format of Error Log CSV File:

Date Time	Machine	Bot Username	TaskName	Error Line Number	Error Description

Debug Log – This log is for technical information, helpful to developers or others who troubleshoot issues. This information should go into its own log file and should be turned off when in Production mode. An pBoolIsProductionMode variable could be used to turn these statements off when the automation is moved to Production.

Performance Log - Performance logging can either go into the process/informational log. In some cases, it may be desirable to have it in its own log file. Performance should track how long it takes to perform specific steps, but too much granularity should be avoided. In most cases, performance logging should be limited to an overall business process. For example, how long it took to complete an order, or how long it took to process an invoice.

Know What You Are logging

Every time you issue a logging statement, take a moment and have a look at what exactly will land in your log file. Read your logs afterwards and spot malformed sentences.

Know Who is Reading Your Log

Logs should be easy to read and easy to parse. Users, customers, partners, and automation developers will be looking at logs to:

- Determine if a process completed successfully
- If a process didn't complete, they'll be looking for information as to why
- Determine if the automation is performing as expected
- Interactively "tailing" the logs
- Parsing the logs with a tool or importing the logs into Excel to gather and analyze metrics
- Importing the logs into a database if necessary

Avoid the "Magic Log"

Another anti-pattern is the "magic-log". If a log file is long, or has grown to a large size, it may be tempting to insert characters or strings that make things easier to find certain lines. They're meaningful to the developer, but completely meaningless to anyone else. This is why they're called "magic". An example would be "Message with XYZ id received". If this approach is used, you'll end up with a process log file that looks like a random sequence of characters.

Instead, a log file should be readable, clean, and descriptive. Don't use magic numbers, log values, numbers, or ids. Be sure and include their context. Show the data being processed and show its meaning. Show what the automation is doing. Good logs can serve as a great documentation of the automation itself.

Avoid Side Effects

Performance is the primary concern with logging. Normal logging actions themselves are not costly in terms of performance. However, consideration should be made for excessive logging (inside of a small loop with many iterations for example).

Log Sub-Task Variables (Log Level - DEBUG)

When passing variables back and forth to sub-tasks, adding debugging log statements is recommended. These should be used to see the values of variables when they enter the sub-task, and what they have been set to when being passed back (where relevant). An pIsProductionMode variable should be used to turn these statements off when the automation is moved to production.

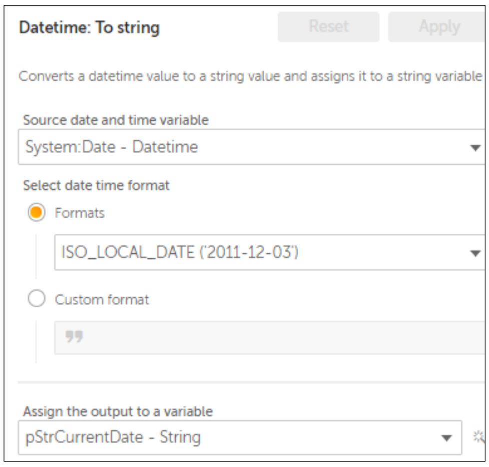
Consider Rotation and Dozing

Always consider how large a log file will become. By nature, log files generally become quite large in a short period of time. Implementing “log rotation” can be helpful, where log files can be moved, compressed, deleted, or renamed when they grow too large.

A recommendation is to include the current date in the name of the log files. Ex.

- ErrorLog_<DATE>.csv
- AuditLog_<DATE>.csv

where <DATE> represents a string. Older log files can then be compressed and archived (dozing). In this way, no one log file will grow to an unmanageable size. You will have max one file for any day and the Date in File Name will help you identify and review the intended block of log data.



Datetime: To string

Converts a datetime value to a string value and assigns it to a string variable

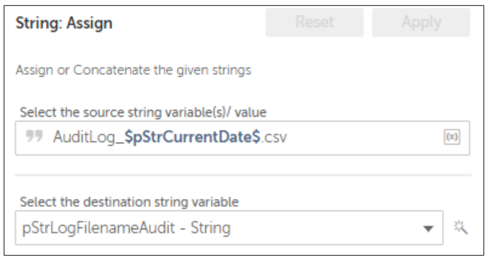
Source date and time variable: System:Date - Datetime

Select date time format

(●) Formats: ISO_LOCAL_DATE ('2011-12-03')

( ) Custom format

Assign the output to a variable: pStrCurrentDate - String



String: Assign

Assign or Concatenate the given strings

Select the source string variable(s)/ value: AuditLog_$pStrCurrentDate$.csv

Select the destination string variable: pStrLogFilenameAudit - String

Interfacing with External Systems

If an automation will be interfacing with other systems, e.g., APIs, REST or SOAP calls, be sure to log those calls, and if appropriate their responses.

Note: NEVER log Passwords or any PII (Personally identifiable information) Data, such as Social Security Number, Date of Birth, etc. in any type of Log File.

Configuration

Wherever possible, do not set/hardcode Variable values in the TaskBot directly. Instead, make these values configurable. The objective is to allow as many aspects of the process to be configured as can be reasonably achieved.

The recommended approach to manage configurable values depends on the nature of the data. The following are (2) types of configurable values with the recommended approach to manage them.

Global Values

For values that are applicable across Use Cases, implement them using the Global Values functionality. Some examples include the following:

- Delay Times (e.g., DelayInSecondsShort)
- Application URLs / Server Names
- Root Folder Path on Bot Runner for files related to Bots – e.g., c:\Bots\
- Root Folder Path on Shared Drive - e.g., g:\Bots\
- “Sender” Email Address for sending notifications
- Email Server/Port for sending notifications

Since “Global Values” are managed as part of a Control Room, they are by nature “environment specific”. As a Bot is promoted through different environments, the Bots can access different values for these parameters as necessary.

Use-Case-Specific Values

For values that are specific to an individual Use Case, they can be incorporated into a “config” file stored on the Control Room. Best practice is to use XML format for Config files, it is also the industry standard. Built-in AA XML Actions can be used to easily read values from such files and assign to variables.

Examples of parameters that could be stored in a configuration file would include the following:

- Functional Area / Department
- Process / Use Case Name
- Static values being used in calculation or formula

By making these values configurable, a Bot can locate files (e.g., Input) that might be stored on a Shared Drive in a sub-folder specific to that Use Case – e.g., \Department\Process Name

It is recommended to store your Config File on the Control Room, along with Bots related to the associated Use Case/Process.

Following is an example Config File format for this option:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration>
        <FunctionalArea>Finance</FunctionalArea>
        <SubFunction>AccountsPayable</SubFunction>
        <ProcessName>Invoice Processing</ProcessName>
        <MyParameter>15</MyParameter>
</configuration>
```

Note: Only non-sensitive information should be stored in Config files, or as "Global Values" in the Control Room. For any sensitive information such as Application or Database Login Credentials you should use Control Room Credential Lockers.

Maintaining Windows

Always close all windows that your task opens before terminating the task. Best practice is to attempt to close all possible windows that are open at the beginning of your task, so that it starts off uninterrupted with a clean slate.

It is a good idea to maximize a window as soon as it is opened using **Maximize Window** action. This way you ensure that all generally visible objects are still visible and are not hiding behind scroll bars.

Window titles are used to identify specific windows – avoid situations where you may open multiple windows with the same name. Also, consider using variables to specify Window Title, so that you can quickly update multiple actions by making change at one place by updating variable with new value.

Handling Processes involving PII

- Secure recording option should be turned on in the Control Room, which makes Bots identify objects on-screen without screen shots/images.
- Audit Logs/ Error Logs maintained during Bot runs need to be minimized and make sure not to add any PII data while logging details.
- Do not use any **Capture** actions which will capture the desktop or a window.

Automation Friendly Input Formats

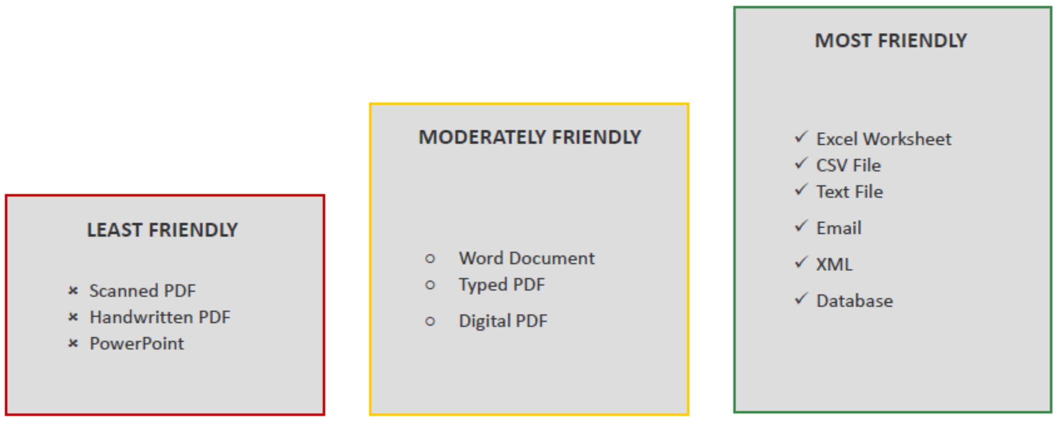


Figure: Categorization of document formats for ease of development

Given below are some automation unfriendly use cases:

- Multiple windows with the same title
- Varying screen resolutions
- Different OS/IE versions/MS Office versions
- Multiple monitors
- Interactive Flash or Silverlight forms
- Machine or data-driven slowness/variance
- Windows without titles
    - Usually these are embedded as 2nd or 3rd level sub windows, like popup windows
- Drag-and-Drop functionality
    - Must use mouse moves/clicks