Enterprise RPA Toolkit for Microsoft Power Platform

Wednesday, January 12, 2022 10:06 AM

What is this toolkit for?

This toolkit helps implement Microsoft Power Platform, or more specifically, Power Automate Desktop, in a typical enterprise environment.

What are the typical asks in such situations? Enterprise users typically have a multitude of automation processes. Often there's a dedicated team in IT tasked with maintaining the bots that have been built, as well as building new bots. The responsibility of the IT team is typically defined as follows:

- Ensuring the individual work items are handled within the time set in a Service Level Agreement, and
- Fixing the bots whenever they stop working.

Often there is a clear boundary between fixing a bot (when an item can be handled according to the process design, but the bot still fails to do it) and fixing a problem in data or in an app (when an unhandled exception occurs). Fixing a bot falls into IT responsibility, fixing the data or handling business exceptions isn't.

Now when the IT team is tasked with keeping their bots 24/7, they will most likely want to be able to test their processes frequently to detect problems before they impact the business processes. **Unit and process testing** is one area where this toolkit can help.

Another common ask is **separation of business and IT duties**. The IT team often wants to give business people live insight into how their work items are being handled. This toolkit helps provide such a live view to business users, and provide them with the ability to run some basic actions, like restarting a flow or modifying the input data to work around a data issue. All of these are possible without giving someone any kind of admin permissions, nor allowing them to see work items that do not belong to their process.

Next is **reusability**, or more broadly - efficiency of bot design. In a large enough organization it becomes quite common that the same underlying business app is used in multiple processes, often in similar roles. A person asked to build a new bot should be able to reuse the work done before, and not have to start from scratch every single time.

With reusability in place, the next thing a team needs is the ability to package these reusable components and easily bring them in to any project. Such reusable modules should be upgradable without the need to rebuild processes that use them - so ideally, an administrator should be able to replace a module with it s newer version when the underlying application changes. And yes, **modularity** is one of the patterns enabled by this toolkit.

One more useful capability is **work item queueing**. With Power Automate, you get flow level queueing - that is, if you scheduled a flow and it failed to complete, you can see that in a report. But if you restart the process, it will create a new flow queue item - which will not be connected to the previous one. When it's about one process that failed, you can keep track of failures and restarts manually. But in a large environment, you can quickly get hundreds of failed processes and lose track of which one has been completed and which hasn't. Work item queueing helps address this by creating a master record for your work item, and link all attempts to execute the process to this work item.

Finally, **exception handling**. Certain exceptions are temporary and technology driven, such as an app that is temporarily unavailable. In many cases, such a process can be safely restarted later. But other exceptions can be business in nature, where a user needs to manually fix some data before the process can be restarted. This toolkit helps configure the RPA platform to respond to such

exceptions accordingly - e.g., by notifying a business owner or scheduling another run.

What are these modules anyway? Do I need 1 or more?

The use of solutions helps separate the job of app module builder, and that of SME (*subject matter expert* in the field being automated). The app module builder takes care of integration, while the SME is building the actual business flow.

By keeping these two roles separate, we facilitate reuse of the content created, and reduce maintenance costs. If the underlying app is ever modified in a way that breaks an existing process, it becomes the app module builder responsibility to fix it. Ideally, the SMEs should never have to update their processes again - the interfacing layer should be the place where updates are applied.

Who is the intended audience?

It's pretty technical. This toolkit is meant for RPA developers and administrators, who look for software and recommendations for building a robust automation platform.

Toolkit or feature?

This is a toolkit - it's entirely built in "user space", or in other words - it's built with ordinary programming tools available in the Power Platform, like flows and apps. You can inspect how the features are built, and you can add your own automations and tools on top of the ones provided.

Some RPA practitioners believe a vendor should provide these capabilities as standard features rather than toolkits. We believe the toolkit approach can benefit from Power Platform's ease of self-automation, that is building bots and flows that automate the platform itself.

How to begin working with this toolkit

We recommend that you experiment with these tools before you start building your real business solution with it. Take advantage of the environment separation in Power Platform, create a trial database and play there. To leverage the toolkit, you will need to least to:

- Install the Orchestration Center,
- Create an automation module for you specific app (or multiple modules for multiple apps),
- Build the top-level flow that leverages your automation module(s),
- And finally, create a way for your work items to arrive in this environment, and a way to notify the rest of the world that they have been handled.

Conventions

Whenever *a piece of text* is written in italic, it is a name of an object, a flow, a menu item or something similar.

Bold is used to emphasize important pieces of content or places where a mistake is likely, so please pay attention.

Any occurrence of [PREFIX] should be replaced with your actual chosen module prefix in square brackets (i.e., [MYAPP]). See Using the test case template for more information on module prefixes. This is **not** the same as your usual Dataverse publisher prefix, although it plays a similar role.

Permissions and resources you will need

Tuesday, March 15, 2022 9:46 AM

In order to set up this toolkit, you will need the following:

- A **Power Platform environment** with Premium account capabilities (Office 365 is not enough)
- A **Power Automate Desktop attended RPA** license (trial is fine)
- A **Dataverse** database created in your environment
- In that environment, you will need **System Administrator** permissions at least **temporarily**, to set up the needed security roles)
- You will need access to Azure Portal, at least temporarily, to configure an **App Registration within Azure AD**,
- You will need an **Azure Key Vault** resource created, the free plan is most likely sufficient.
- A machine running Windows with **Power Automate Desktop installed**. For evaluation, this could be the PC you're normally using, but eventually this would be a virtual machine for unattended flow runs.

Quick start

Tuesday, March 15, 2022 9:49 AM

The easiest way to see this toolkit at work is to watch an overview video, but that's not ready yet :)

With the following tutorial, you will:

- Install the core module for orchestration center
- Install a sample module
- Use the module to create a simple desktop flow

Install the core OrchestrationCenter module

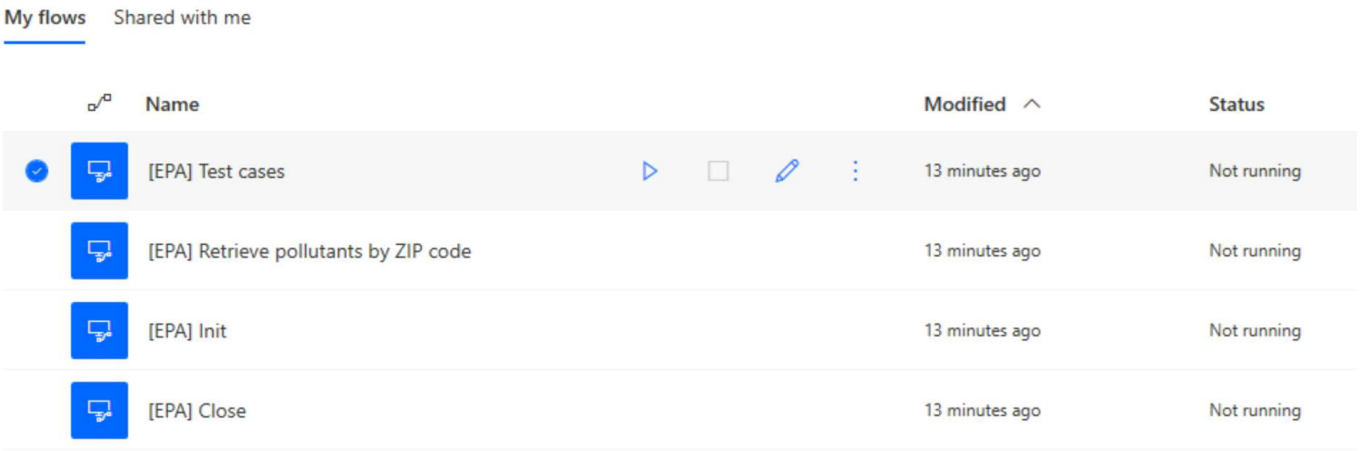
Here are the setup steps for first time hands-on exploration:

1. Get your Power Platform environment and System Administrator permissions in that environment (trial environment is perfectly fine)
2. Import the **OrchestrationCenter** solution (managed)
    a. While importing the OrchestrationCenter solution, you will be asked to create a connection - which basically means storing credentials to your System Administrator account. These permissions will be used to start your case desktop flows.

Install the EPAExample module

1. Create a Dataverse access object, using your Azure Key Vault and Azure AD (follow these instructions: Dataverse Access Object). Store it in the Key Vault and write down the name of the vault for later
2. Open the *Orchestration Center* app, choose *RPA Modules* from the left hand side menu, and click *New* in the toolbar. Enter *EPA Website Integration* for the record name and save it. When the record is saved, extract the record ID for your newly created record:
   https://org780c4e2c.crm4.dynamics.com/main.aspx?appid=0eb13763-3fa4-ec11-b400-00224899ee9e&pagetype=entityrecord&etn=rpakit_rpamodule&id=**1e24b726-54a4-ec11-b3fe-0022489aa375**
3. Import the **EPAExample** solution (managed). While importing, you will be asked to create:
    a. Connection to Key Vault - use the key vault resource name you obtained earlier
    b. Connection to Desktop Flows - here you'll connect to your PC with Power Automate Desktop:
        i. Run Power Automate Desktop
        ii. Login with your account
        iii. Start trial if you haven't got a license for Power Automate Attended RPA
        iv. Select the environment where the toolkit is being set up
        v. Open settings, then click on *Open machine settings*
        vi. Register your machine
        vii. Go back to your web browser, create a connection to Desktop Flows. For method of connection, pick *Directly to machine*, select your machine from the list, then enter domain and username (example: EUROPE\johnc) and your domain password. These would be the credentials you use to log in to your computer.
    c. Connection to Dataverse
    d. Value for *Module ID - EPA Example* variable. Use the ID extracted in step #2 above.

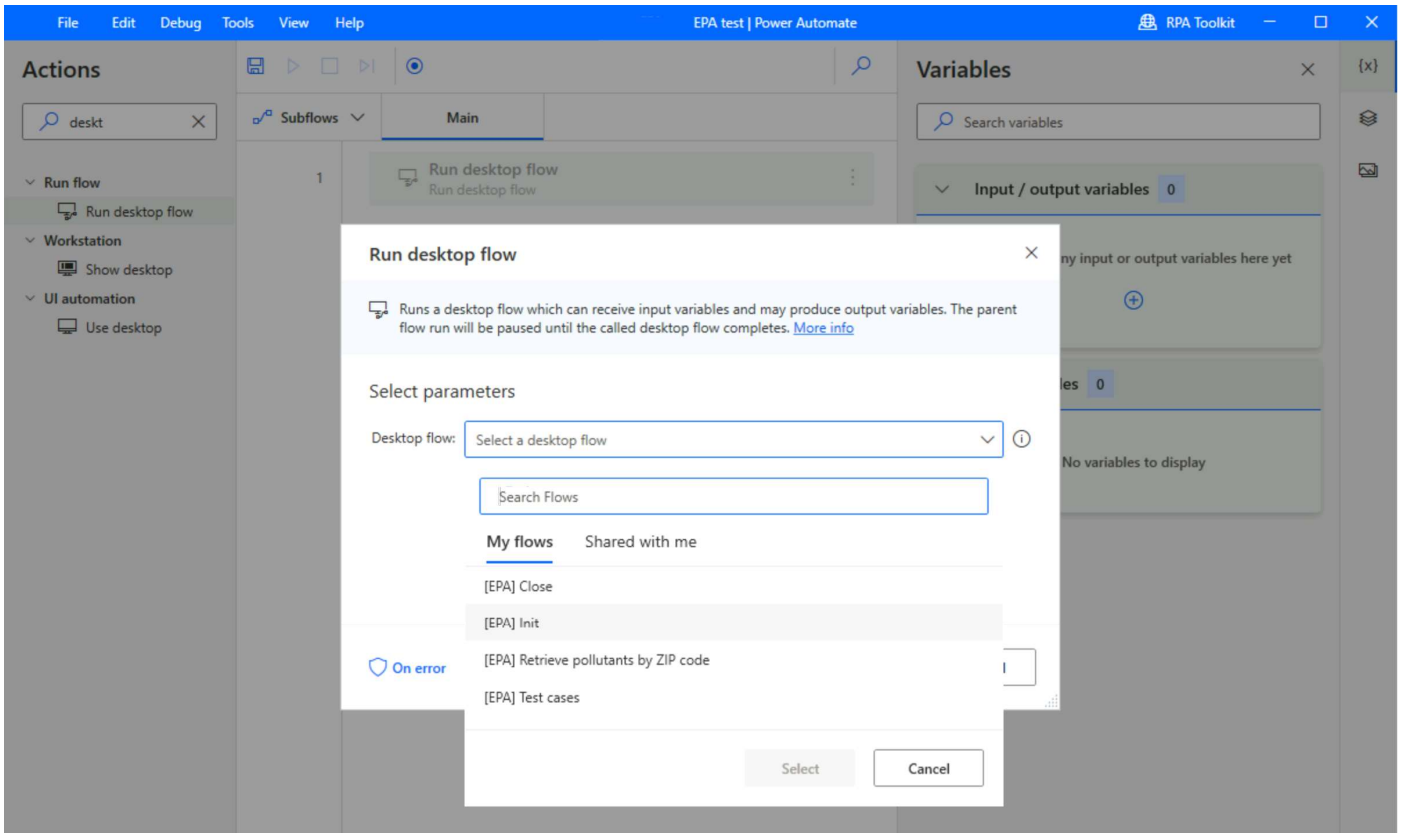
Once the above steps are done, you should see the following EPA module example functions in your Power Automate Desktop:



Creating your first flow using the EPA module

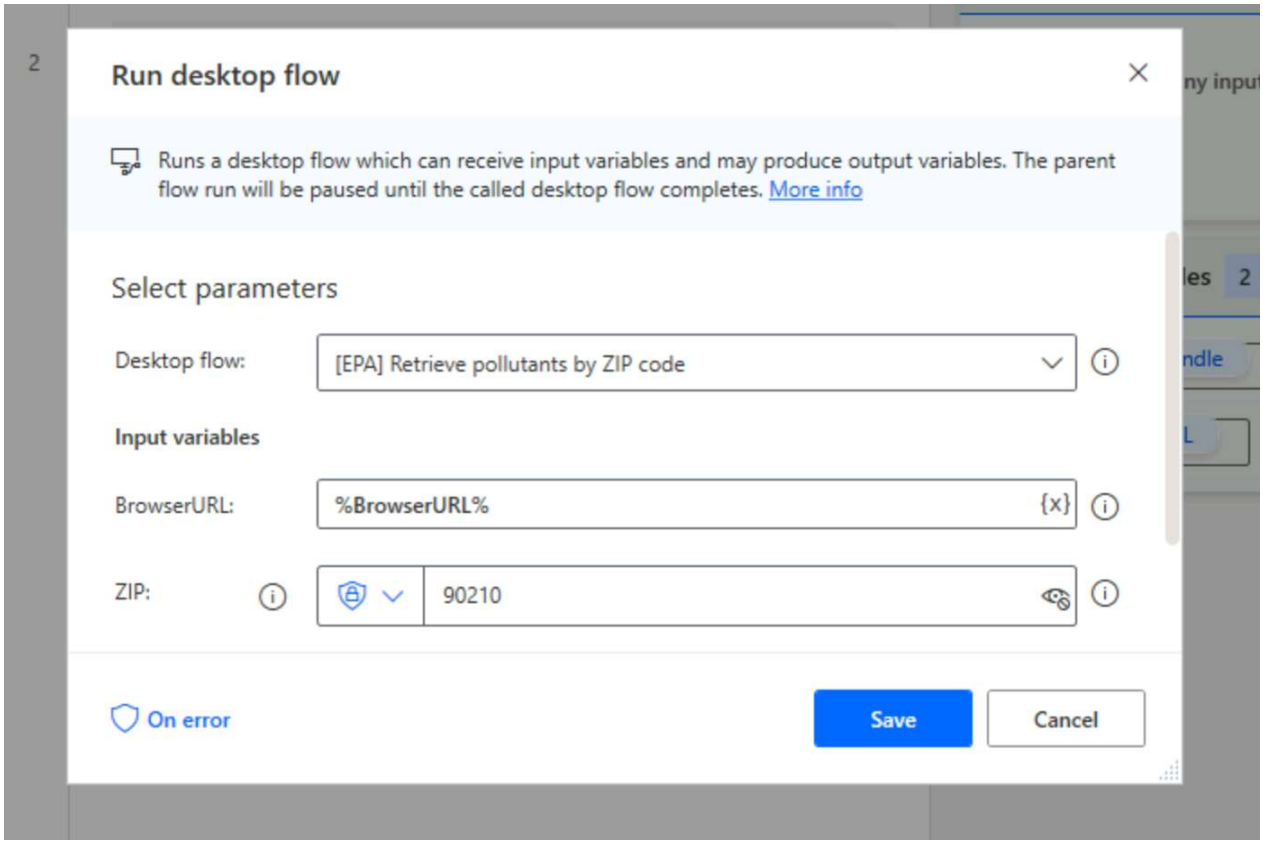
The EPA module sample provided implements connection with the United States' Environmental Protection Agency. It's only actual function is retrieving the air pollution information for a given US ZIP code.

Let's build a sample process to retrieve and display the primary pollutant information for Beverly Hills, CA.
Create your own flow with a Run Desktop Flow step:

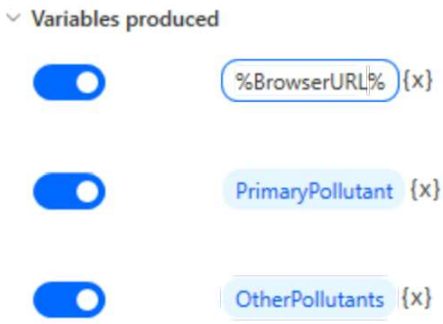


Pick *[EPA] Init* from the list.

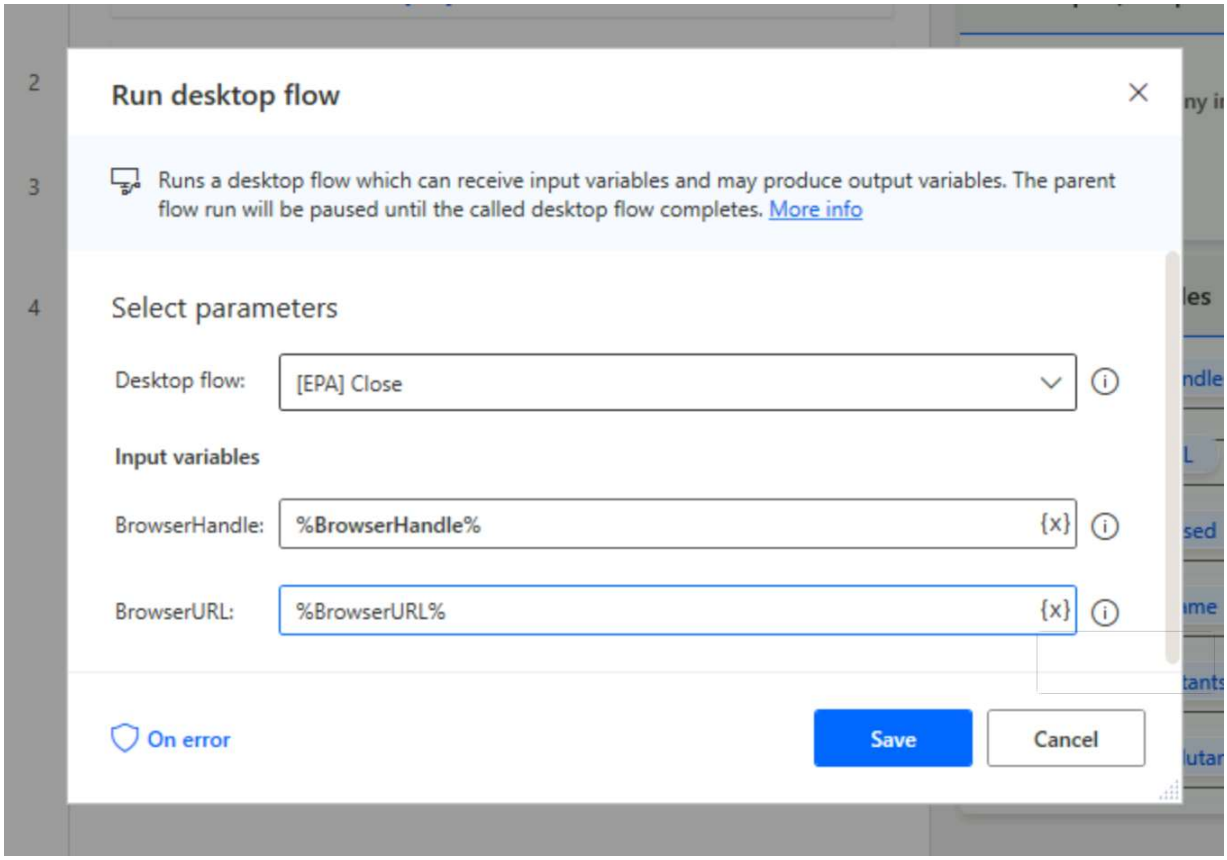
Add one more *Run Desktop flow* action - this time invoke *[EPA] Retrieve pollutants by ZIP code*. Enter 90210 for ZIP code, and %BrowserURL% for the browser URL parameter.



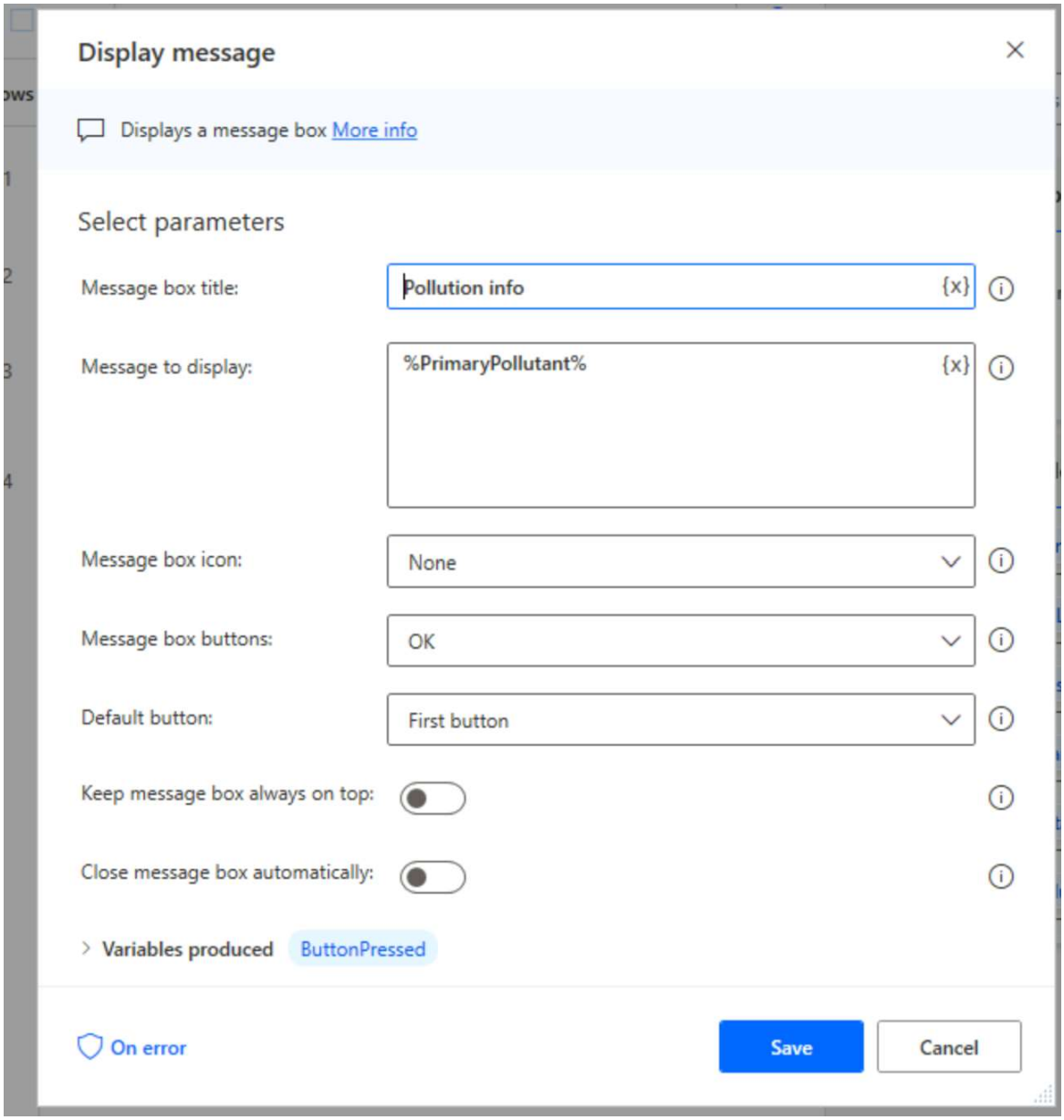
Now, scroll down to *Variables produced* and edit the **BrowserURLOutput**, change this to **BrowserURL**



Next, add the *[EPA] Close* function call:



Finally, use *Display message* to show the information retrieved:



This is what your process will look like:

When it starts, you should see Chrome browser open up, retrieve data from the webpage, then close, and show a message box:


Pollution info
PM2.5, 61, Moderate
OK

Best practices

Wednesday, March 30, 2022 7:50 AM

Architecture of your solution

Wednesday, March 30, 2022 11:48 AM

Design with failure in mind

Don't just follow the happy path. Proper solution architecture will keep the operations cost low.

If your process architecture is created without planning for failures, then any issue with external systems might:

- Break your process
- Cause robot/solution downtime
- Force manual effort in cleaning external systems involved

Issues caused by robots failing

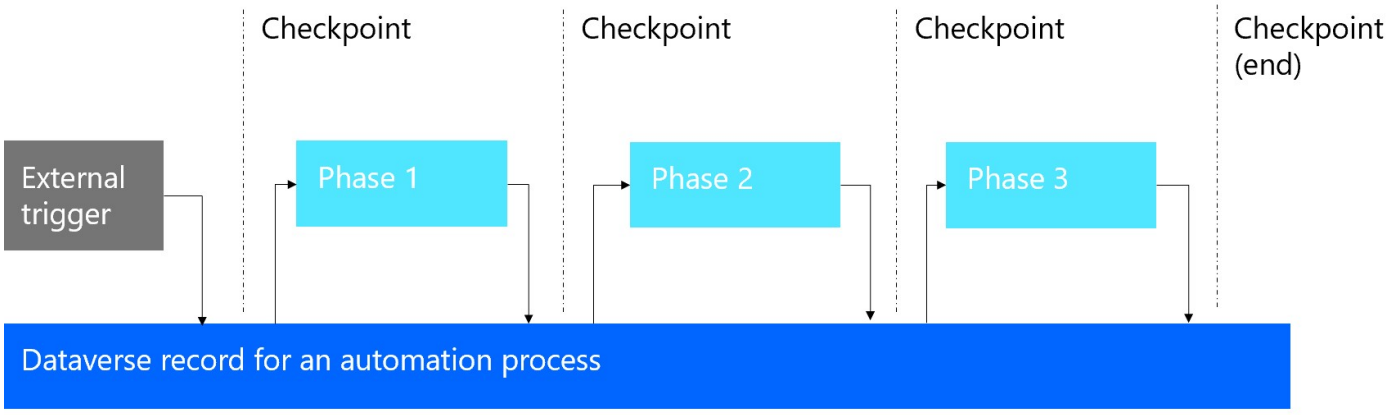
When a robot fails in the middle of a process, the result is similar to a situation when an employee would walk away from their desk leaving unfinished work. This could result in:

- Applications not closed
- Transactions open but not committed (draft)
- Partially updated records

In such a case, it might be impossible to carry out the steps from the beginning as if nothing happened. Sometimes, a human operator might have to log in to clean up, for example by reverting certain actions or cancelling the transactions that were started.

Avoiding manual cleanup

To reduce the risk of manual cleanup, an automation solution should always be conscious of what has been done already. If any step has been carried out that should not (or cannot) be repeated, a record must be made in robot's database that such **checkpoint** has been reached. This way, any process restart will only resume from that known-good checkpoint.



Checkpoints with Dataverse

Use Status Reason field to configure different states of your database record (most recently reached checkpoints). Edit data model to define additional status reasons (other than "Active" or "Inactive". Optionally, you can define a flow diagram - to define permitted moves from one status reason to another.

Going from one checkpoint to another

There should always be a cloud flow that takes you from one checkpoint to the next. That cloud flow should start when *Status Reason* **changes** to a specific value, and carry out all the work in a given phase. The last step of your flow will save the next checkpoint name into *Status Reason*, triggering the next flow in sequence.

Recommendation: Your cloud flows names' should include the starting and ending checkpoints.

Example: "[Created -> Saved] Save a record in the app" (where *Created* and *Saved* are checkpoint names)

Interim results

If your flow needs interim results that are produced in one phase and used in one of the next ones, save them to Dataverse so that the next flow in sequence can pick them up.

New and *Parking* status reasons

Create two special checkpoints for your tables' *Status Reason* field:

- *New* means that a record has been created but it's not been moved to the first checkpoint where automated processing would start,
- *Parking* will indicate that processing has been intentionally stopped.

This will allow you to:

- For *New* records, create any associated records before automation starts
- For *Parking* records, have a neutral value that can be used to restart current phase. For example: *Created* -> *Parking* -> *Created* will start the cloud flow that begins at *Created* checkpoint.

Optional: Consider numbering your checkpoints and using these numbers as part of the name, e.g. *01 Created*

Optional: Consider numbering your checkpoints in multiples of 10, to allow adding interim checkpoints when process is redesigned later. Example: *10 Created*, *20 Saved*

Checkpoints and error handling

Do not create additional steps for handling exceptions. The *Status Reason* should contain last successfully reached checkpoint. Store information about encountered errors elsewhere.

Optional: clear your error information fields at the beginning of each checkpoint.

Recommendation: use Auditing in Dataverse to track changes (history) of each recordd.

Error handling in cloud flows

Wednesday, March 30, 2022 7:50 AM

Why you need to handle errors in cloud flows?

Cloud flows are executed on best effort basis. The platform will do its best to execute your flows, but if it fails, the only difference will be a line in the logs. Your cloud flows will not be automatically restarted or otherwise flagged to the user. Designing your automation solution to gracefully handle errors is your responsibility.

Types of errors

In automation world, different things can break your solution:

- Entire apps or certain features may be temporarily unavailable,
- The input data may be incomplete or in some other way invalid,
- If you need a human in the loop, that human might not cooperate the way you expected,
- Software bugs may prevent your work from being carried out.

Some of these errors may rectify themselves or someone will take care of them anyway. Others will require intervention of a human operator. Sometimes input data will have to be edited, and bugs will require workarounds.

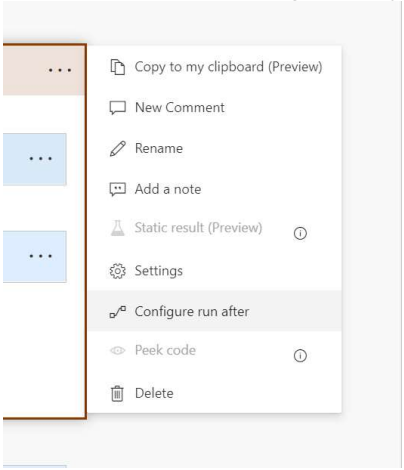
You need to design your solution to minimize disruptions and manual effort arising from these errors.

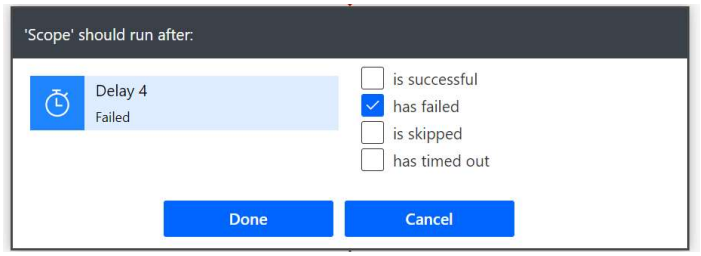
Patterns

Wednesday, March 30, 2022 11:41 AM

The *Configure Run After* option

Your basic tool for error handling is the *Configure Run After* option:





The options are:

- Is successful - if previous operation is completed
- Has failed - if an error occurred
- Is skipped - if due to flow logic the action was not attempted (for example, it was in a decision tree branch that was not used)
- Has timed out - if an external system has been called and did not respond in due time

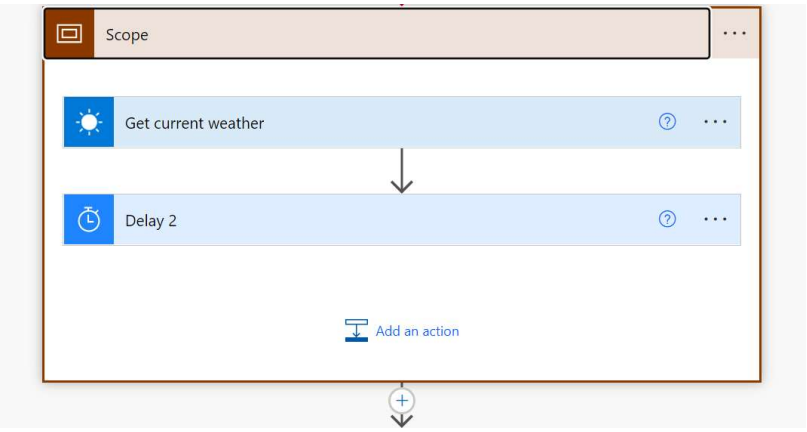
How different cloud flow components behave in case of errors

Now let's look at how error status is aggregated for multiple actions, i.e. in loops.

Component	**Default exception handling behavior**
Simple chains of actions	If one fails, all subsequent actions are cancelled.
Scope	Within the scope, if an action fails, the remaining ones will be skipped. The scope is considered to be failed if the last action carried out has failed. Usually this means success if all actions were successful, and failed if any action failed - but you can tailor that with *Configure Run After* on each individual action.
Parallel paths - split	The path that had an error stops, the other one continues independently. Since after splitting there is no option to join back, the flows continue. The entire flow run is considered a success if BOTH parallel paths are successful.
Parallel paths - join	The join will happen only if BOTH joined paths ended with a success (unless configured otherwise, see below)
Loops	If an action or block within a loop fails, execution resumes for next loop item. The entire loop is considered failed if ANY item fails.
Nested error handling constructs	For example, a scope within a scope. The inner construct will be executed as described in this table. Then, the outer construct will behave as if the inner was a single action.

Wrapping multiple steps with a single on error rule

If you want to intercept errors in a common way for many sequential actions, use Scope to connect them into one block:

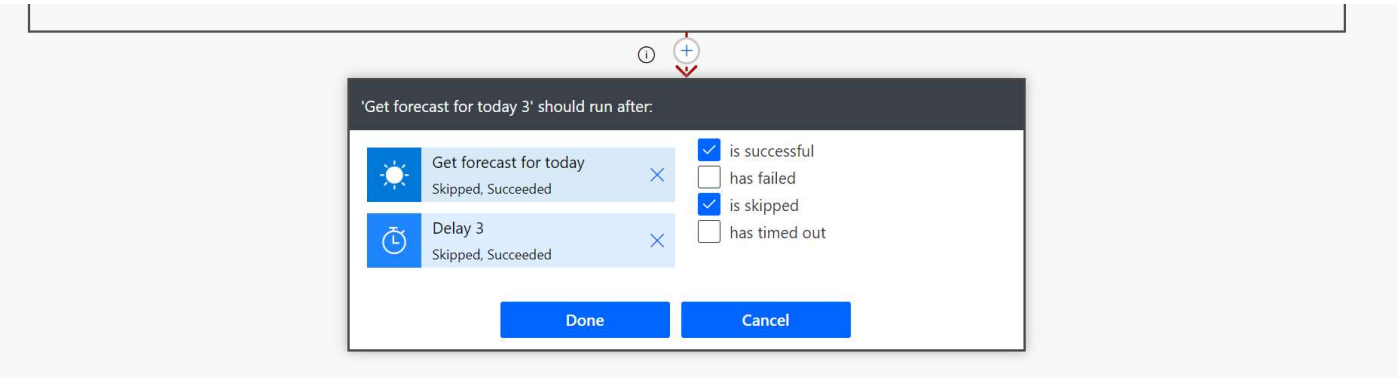


An action with *Configure Run After* that is executed after a scope will look at the scope outcome (see table above for information how scope is evaluated in case of error).

Splitting flow into two paths - on error and on success

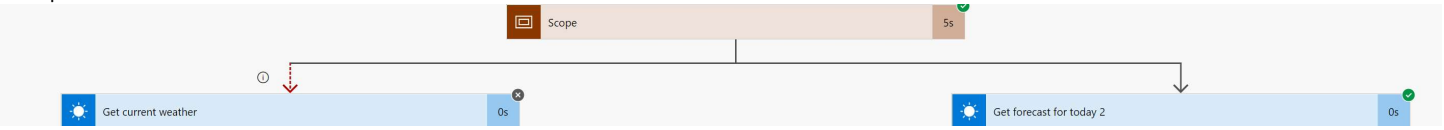
Normally, if two paths join, they must be both successful for the flow to continue.

If parallel paths are used for exception handling, then obviously they can never be simultaneously successful. To join paths in such a situation, use *Configure Run After* on the immediate next action to configure that **both** preceding paths **may be either successful or skipped**:



Using two paths for exception handling

Example:



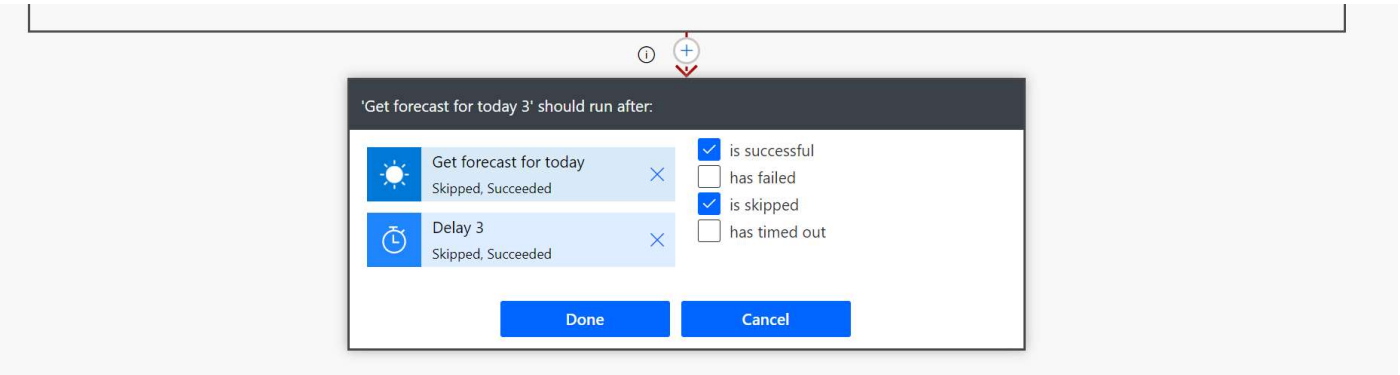
When one of two parallel branches is set to run when previous step is successful, and the other branch if it has failed, only one will execute.

The selected branch continues as the only active process. Subsequent actions do not need the *Configure Run After* option unless you want to handle the next potential exception.

Recommendation: use left hand side for the happy path (on success), and the right hand side for error handling. This will mimic the standard layout for decisions.

Example:

Recommendation: if using two paths for exception handling, join the paths once exception is handled and use *Configure Run After* to make sure process will continue (see previous topic for more info)



Recommendation: Always join paths at the end of exception handling split.

Use dummy action of Delay with 0 (zero) value if you have no further work in your flow.

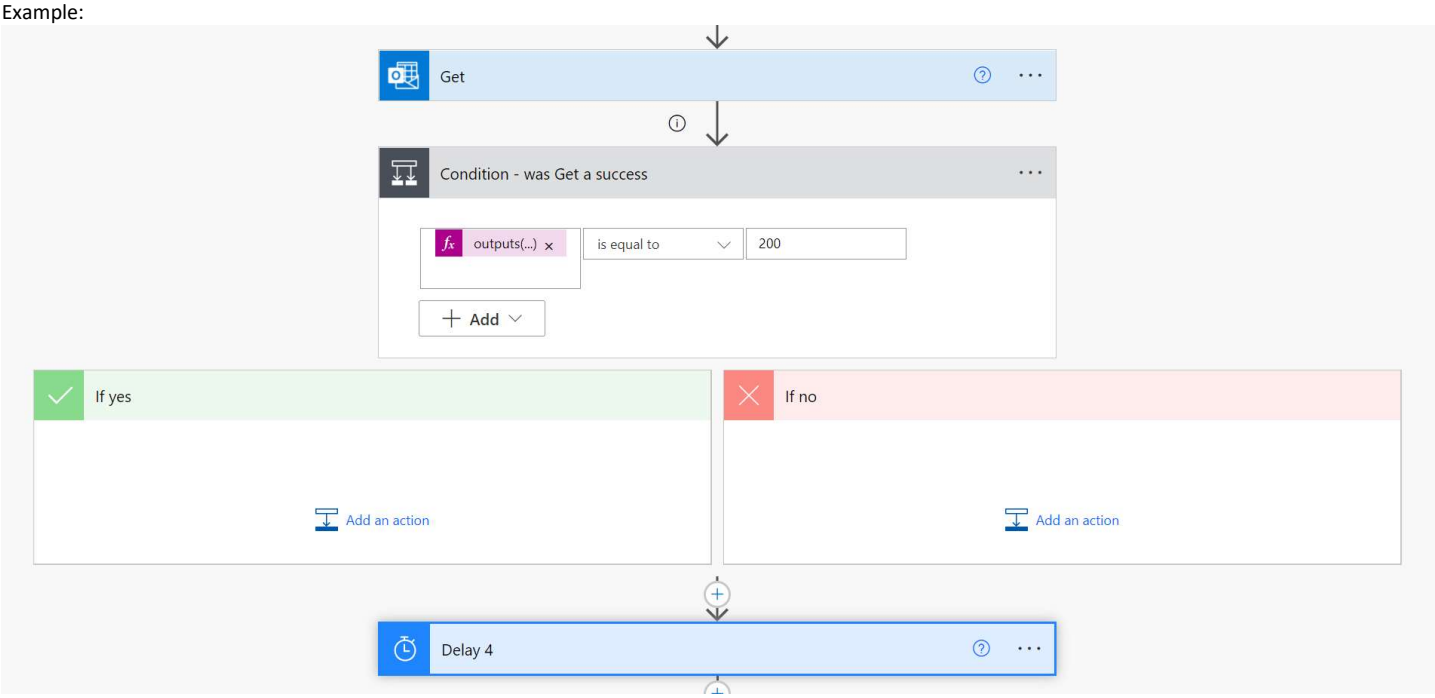
Programmatically check if action was successful

If you need to programmatically find out if an action was successful (for example, later in the process when *Configure Run After* is no longer an option), use the following formula to extract the status:

```
outputs('Get')?['statusCode']
```

Status code of 200 means successfully completed.

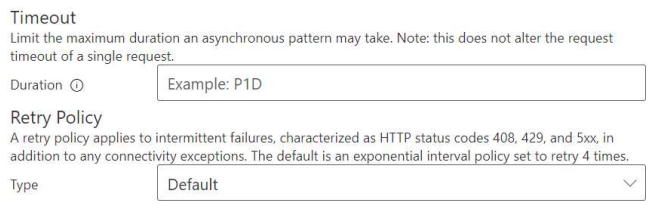
Example:



Repeating an action

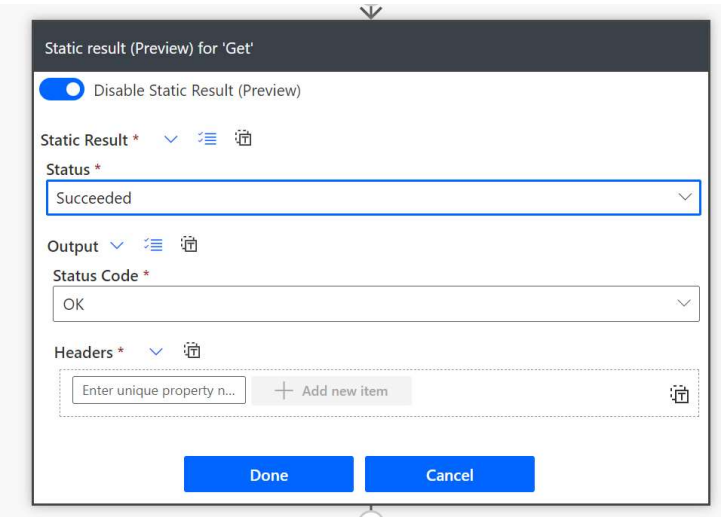
Power Automate will automatically retry actions on external systems through connectors. There's no need to build retry logic in your process, unless you want something more complex than just one step being retried.

You can tweak your action's retry and timeout behavior using *Settings*:

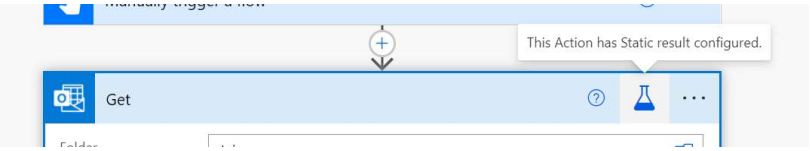


Testing your error handling rules

Use *Static result* feature to temporarily make a step fail, to test how the rest of the flow will behave.



An icon will appear to remind you that static result has been configured:



Recommendation: always test your exception handling rules.

Catching an error from a child desktop flow

Wednesday, March 30, 2022 9:58 AM

Getting the specific error message that occurred

To extract the outcome of a desktop flow, first use the *Configure Run After* feature to make sure your subsequent steps will **always** run - and not only in case of success.

Then, use the Status property of Run Desktop Flow action to find out what the status was. Code 200 is success, all other codes indicate failure:

```
outputs('Run_flow')['statusCode']"
```

For error message, check headers from the output of Run Flow action:

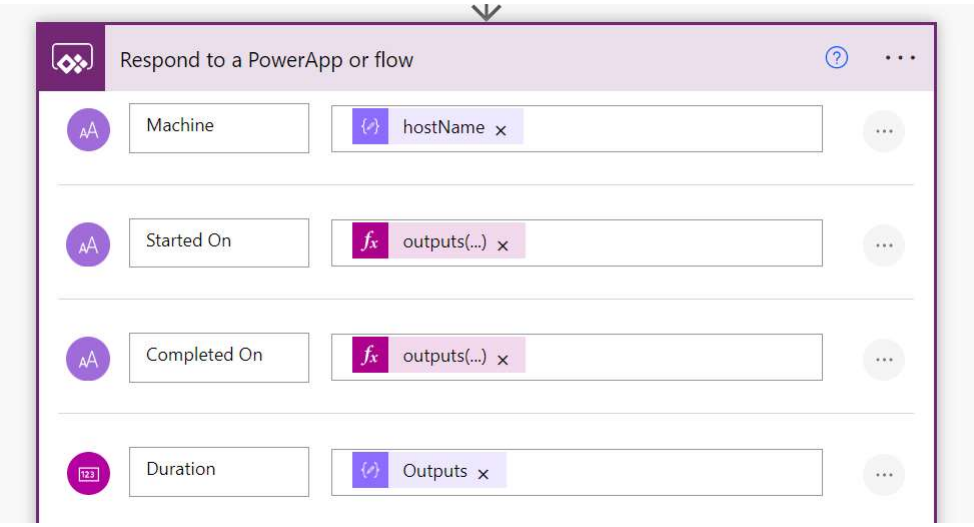
```
outputs('Run_flow')?['body/error/message']
```

Finding out which machine did the flow run on (and other execution details)

To get more details about a specific desktop flow run, you will need the flow run ID first. To obtain it, check headers of the "Run desktop flow" action output:

```
outputs('Run_flow_-_ExtPOC_Accept')['headers']['x-ms-run-id']
```

Then, use the *Get machine run data by Run ID* child flow to retrieve the data. The flow is provided as part of the Orchestration Center package. The following information is provided in response:



Error handling in desktop flows

Wednesday, March 30, 2022 7:51 AM

Desktop flows are called from cloud flows

Desktop flows are typically started from cloud flows - or only from cloud flows if we're talking about unattended automation. If any error occurs within a desktop flow that started from a cloud flow, the exception will be passed to the parent. It is then up to the parent flow to handle the exception.

Why handle errors in desktop flows?

For two reasons. First, if the error is something you can react to and remediate. If an exception is **temporary** in nature, and **trying again** might help - it's a good practice to detect it and try again. Allowing the flow to fail completely might be costly in terms of time and resources, as your robot will have to start over to get it's work done.

The other reason is **helping the parent flow** deal with your error by replacing the detailed fault information with a concise problem statement.

Consider these two outputs from a child flow - would you prefer to resolve this one:

```
"When" block failed (Subflow: StartBrowserIfNotAvailable, Action: 1,
Action name: Launch new Chrome
Error Message: Failed to assume control of Chrome (Communication with
browser failed. Try reloading extension).
Exception of type
'Microsoft.Flow.RPA.Desktop.UIAutomation.WebAutomation.Core.Browser.We
bBrowserException' was thrown. :
Microsoft.Flow.RPA.Desktop.Robin.SDK.ActionException: Failed to assume
control of Chrome (Communication with browser failed. Try reloading
extension).
(... and this goes on for 10 more rows)
```

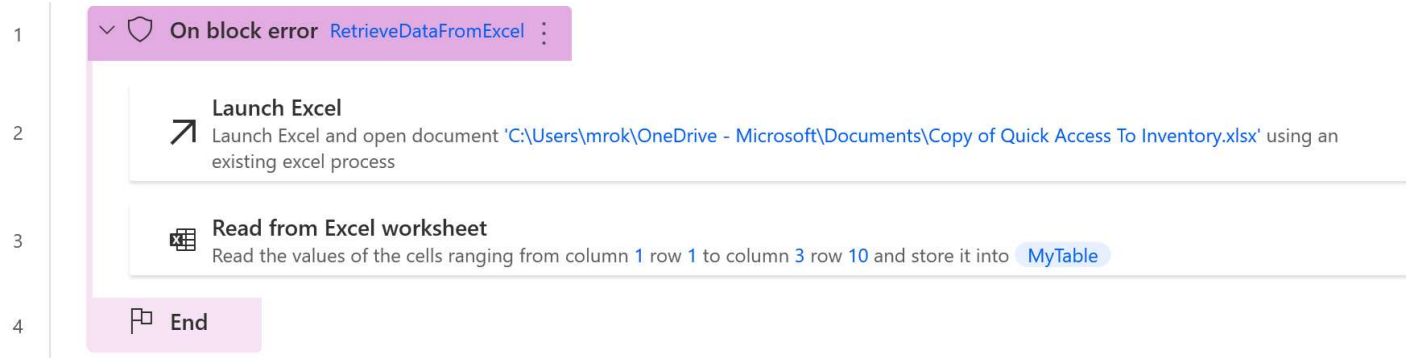
Or this one?

```
"When" block failed (Subflow: ThrowException, Action: 2, Action name:
Stop flow
Error Message: Could not connect to web browser.
)
```

Helping the parent flow (or the human tasked with debugging) here is done by catching the original issue and framing it with user-friendly message.

On Block Error

The *On Block Error* action allows you to wrap one or more actions with a common exception handling rule.



If either action 2 or 3 fails, the common rules for exception handling specified in action 1 will apply.

Recommendation: Use the on block error *name* field to give a meaningful name to your block

Exception remediation strategy

The *On Block Error* action should apply one of the following strategies, starting from most preferred:

1. Handle the specific issue that you expect to happen
2. Retry the specific action
3. Retry the block of actions

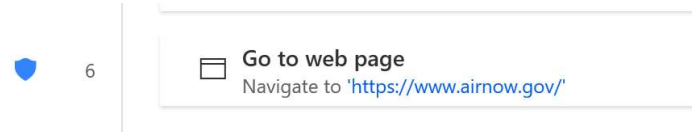
4. Fail with a user-friendly message

Recommendation: apply more specific remediation strategy first, and more generic strategies later, or in a broader context.

Recommendation: for specific remediation strategy that applies to a single action only, use the On Error option in that action, rather than wrapping it with an on error block.

This will allow you to use the built-in retry action feature.

Actions that have a specific error handling rule configured are indicated on flow screen with a shield icon:



Recommendation: use the On Block Error name to indicate the selected strategy it is more than just a more friendly error message

Example:

Retrying a single action

Wednesday, March 30, 2022 9:43 AM

For a single action error remediation, you can configure the number of times this action will be retried before further rules will be processed:

However, if handling a specific error requires retrying more than one step (for example, you need to close an app and run it again), you will need to build your own retry protection mechanism.

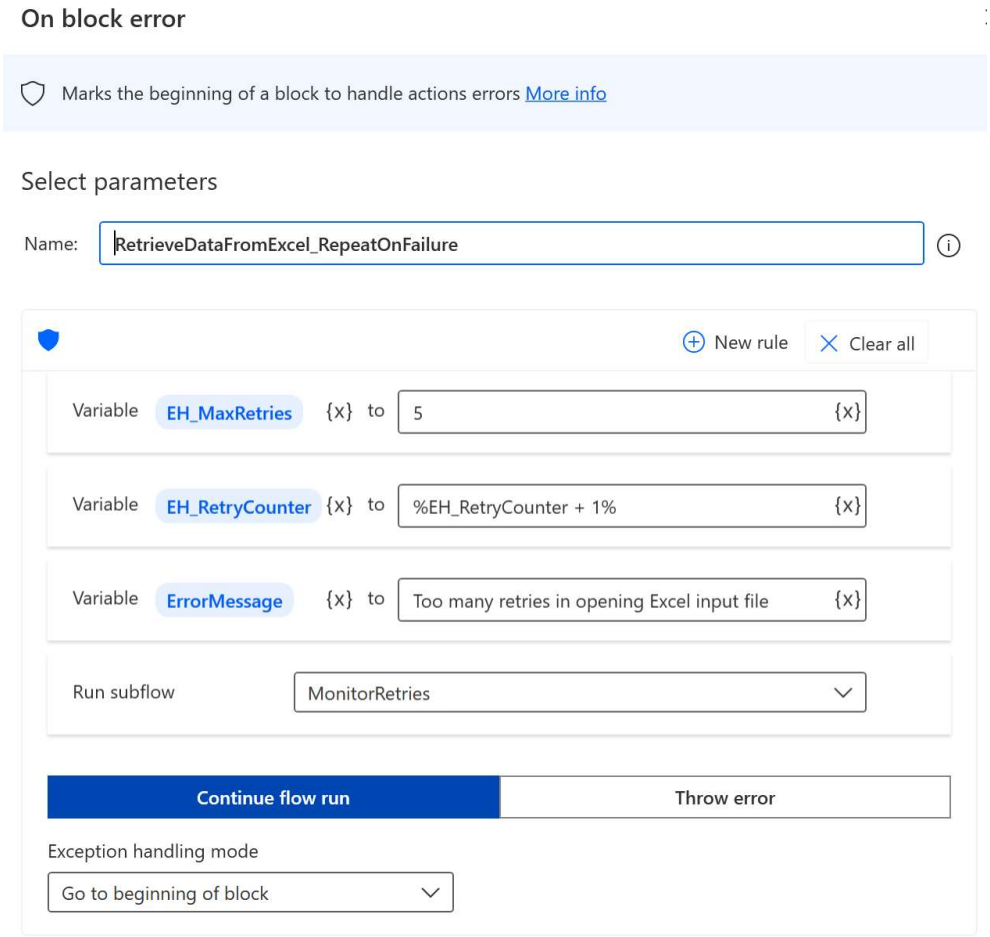
Important: There's no timeout or built-in counter for retry actions in On Error Blocks.

Retrying an entire block

Wednesday, March 30, 2022 9:42 AM

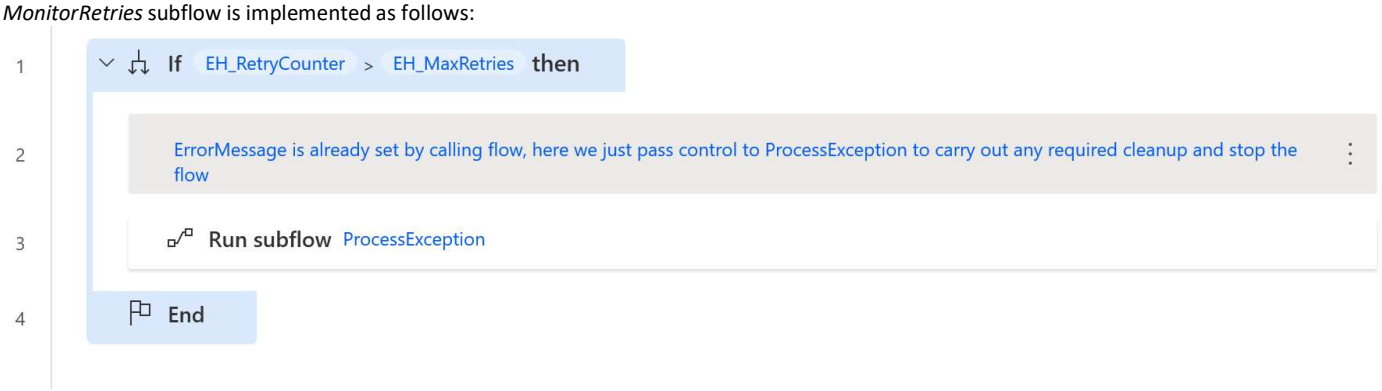
There's no timeout or built-in counter for retry actions in On Error Blocks. If handling a specific error requires retrying more than one step (for example, you need to close an app and run it again), you will need to build your own retry protection mechanism.

Sample implementation of retry counter:



With the above configured, **you can now safely turn on** the *Go to beginning of block* mode.

The *MonitorRetries* subflow is implemented as follows:



Reusable *ProcessException* flow

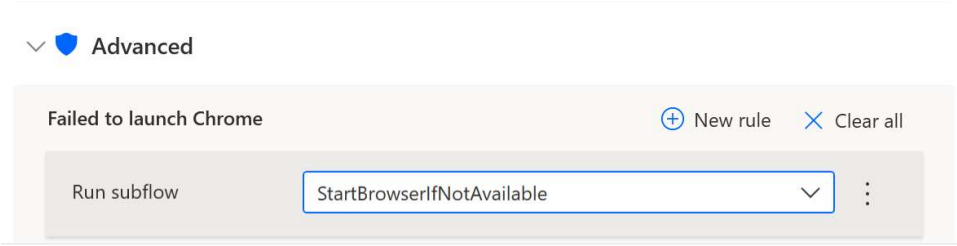
See ProcessException flow

Nesting exception handling strategies

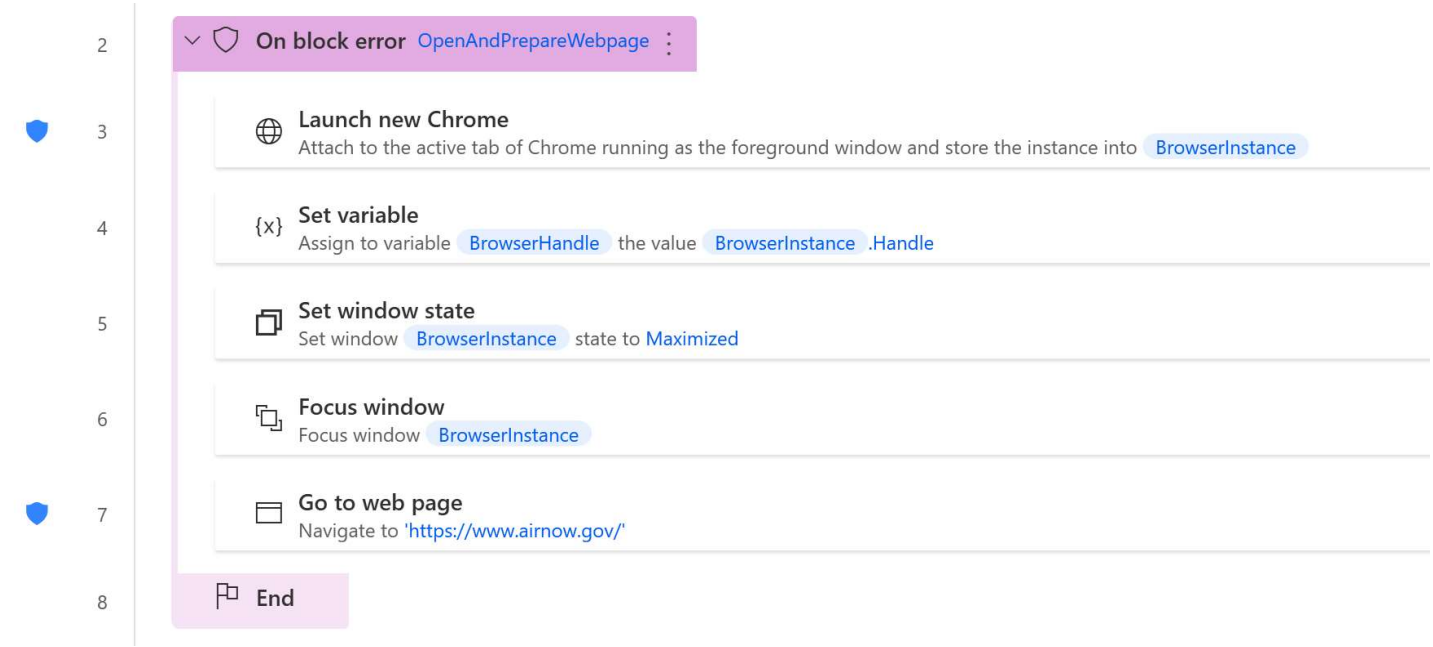
Wednesday, March 30, 2022 9:41 AM

Sometimes a single action in your block could have a specific way to handle errors, while a broader block - which this action is part of - could have a more generic strategy.

For example, for the action of connecting to a browser you might configure a specific strategy for not finding a web browser window running:

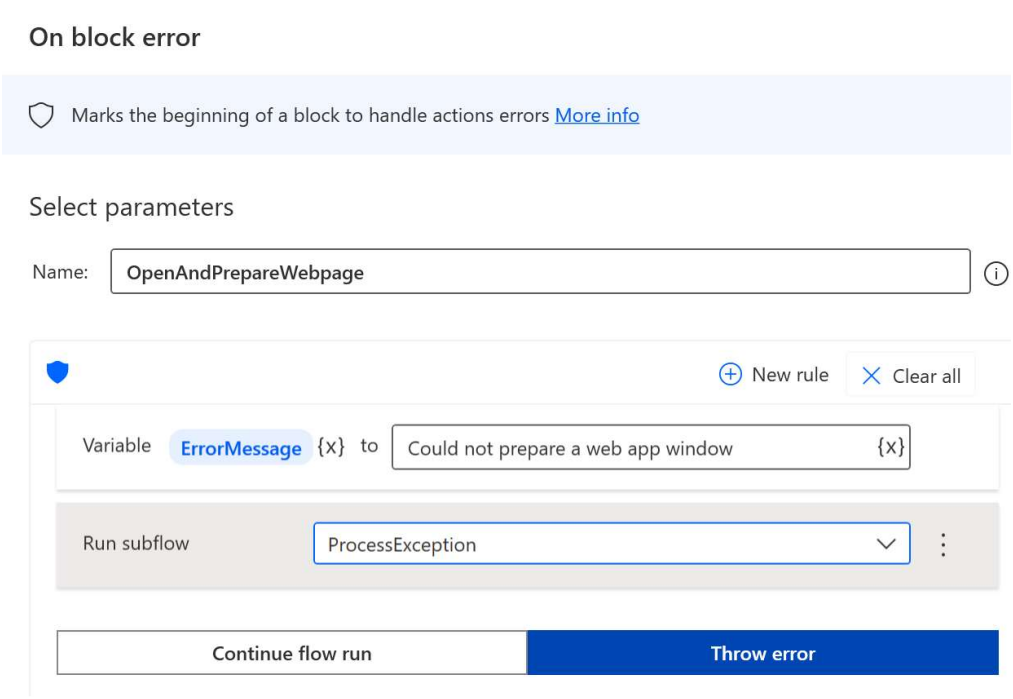


That action could be part of a larger block that intercepts all previously uncaught issues and replaces then with a generic issue description:



In the above example, actions 3 and 7 have their own, specific error handling rules. If anything happens that is not caught with these rules, the more generic exception handling from action 2 will apply. You can also nest *On Block Error* actions if needed in more complex situations.

In the example shown above, the On Block Error is configured to simply throw a more informative error message:



Note: your *ProcessException* flow can potentially do more than just throwing an error - for example, you can use it to close the app windows you opened.

Your Automation Solution

Wednesday, January 12, 2022 10:57 AM

Your process automation solution will use the following solutions that have to be installed first:

- Orchestration Center - provides supporting Dataverse tables for test cases and work items
- XXX Integration - one or more of your integration modules, built to interface between the Business Process solution and the actual app being integrated. The XXX would be replaced with the app name that you integrate with.

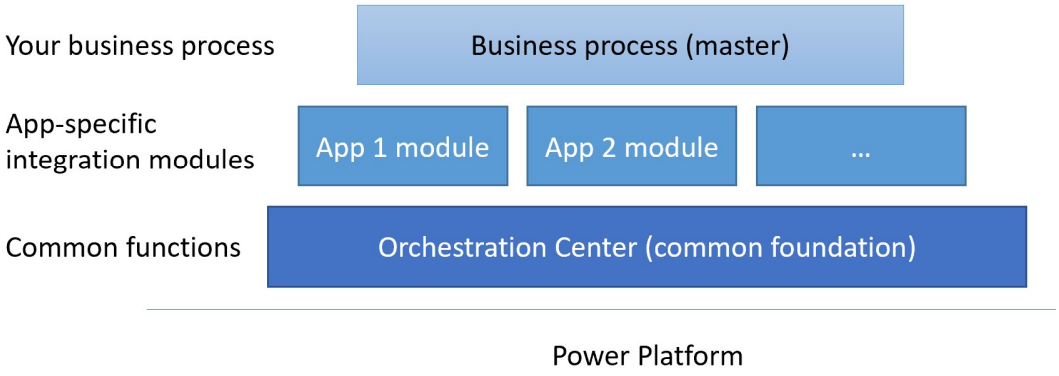
As a fundamental rule, you should avoid manipulating apps directly from the business process, and use integration module(s) instead.

Then, you will build and install the final RPA solution, where you will use these integration modules to achieve your business goal. Typically, this "final" solution will be named after the process that you're building, i.e., "Invoice reconciliation".

Often in a large organization, you might have as many as 100 processes handled by your automation platform. With Power Platform, you can create independent environments for your development, test and production needs. Furthermore, the production role can be also split into multiple separate environments - typically when they are handled by different automation teams, or for data sovereignty reasons they need to store data in different geographies.

Solution structure

Wednesday, January 12, 2022 12:49 PM

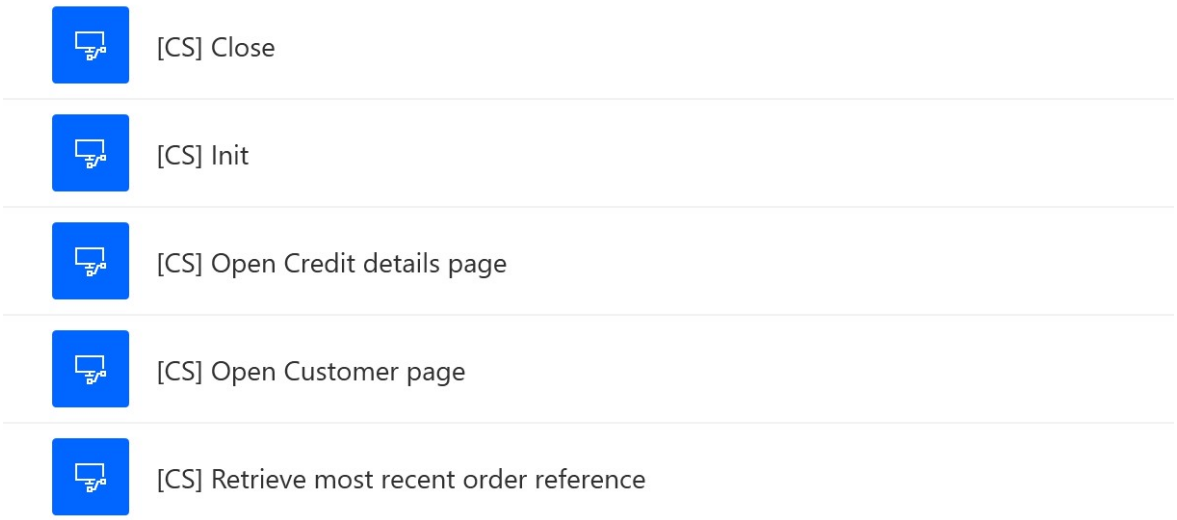


Your business process (master) solution should use *modules* built for connection to any other apps.

Modules are solutions (usually imported as managed) that provide actions on the target application. Typically, there's an **Init** action that opens the target app, performs login, and does any preparation steps needed by all the other flows. There's also a **Close** module that performs cleanup - logs out, closes the web browser.

For any other action that might be needed by the RPA flow makers, there should be a separate flow in your module.

Example:



Naming

All flows in one module should follow the naming convention, where [CS] is the short name for the app being integrated (here: a "Credit System"). The CS should be the name of the solution, or the beginning of the name.

Usage Sequence

The RPA maker should always call **Init** first. Then, they should call any business actions as required, and finally **Close** to clean up.

Any flow other than **Init** may have one or more requirements. For example, in the above solution **[CS] Open Customer page** requires **Init** only. But **[CS] Retrieve most recent order reference** requires that a customer page is opened first.

Using environments

It is strongly recommended that *modules* would be developed in a separate environment ("dev"), where they are built,

tested and packaged for usage. Then, as *managed* solutions they are exported from "dev" and imported to the build location for the complete solution.

The use of solutions helps separate the job of app module builder, and that of SME (*subject matter expert* in the field being automated). The app module builder takes care of integration, while the SME is building the actual business flow.

By keeping these two roles separate, we facilitate reuse of the content created, and reduce maintenance costs. If the underlying app is ever modified in a way that breaks an existing process, it becomes the app module builder responsibility to fix it. Ideally, the SMEs should never have to update their processes again - the interfacing layer should be the place where updates are applied.

Once the update is completed, tested and ready to move to production, it would be exported as a new version of a managed solution. That solution would be then imported to the "production" environments, and the business would start leveraging the updated capacity.

It's worth noting that this approach significantly reduces the maintenance cost - if the same module is used in 5 different business flows, a single fix can resolve issues in all 5 of them. Building strong test cases in the module helps pinpoint any problems as they appear. Also, test cases help prove to the business flow maker that the app connection is in proper working order.

DEV/TEST/PROD separation

Wednesday, January 12, 2022 1:15 PM

Development, testing and production should be separated environments. Any environment-specific parameters, such as logins to a test app, or URLs to non-production instances can be then stored in that environment as environment variables.

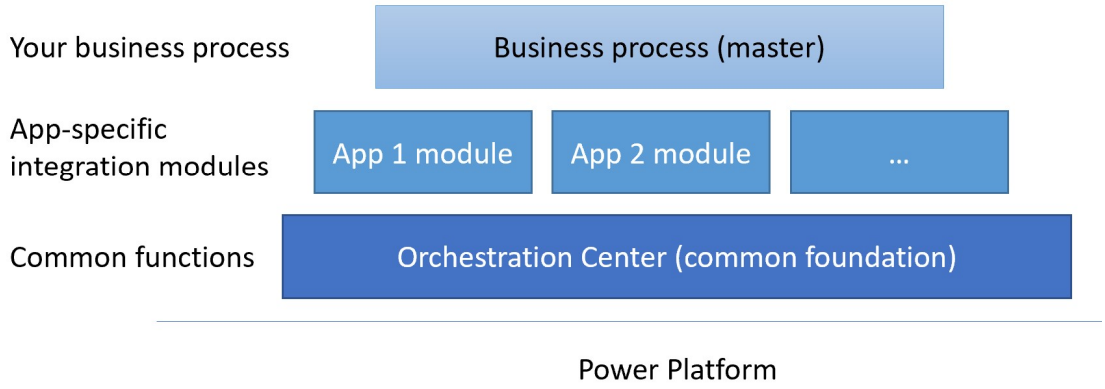
Since environment variable values are not transferred across environments, this approach helps make sure that any testing or development effort will not impact the production data in any way.

If desired, it is completely possible that app module builders and business flow makers would have no access to the production system. They don't need such access to develop RPA solutions. When the works is done and products are uploaded to production environment, the RPA operations team will provide credentials/URLs to the actual apps being integrated.

Solutions import sequence

Wednesday, January 19, 2022 5:15 PM

This chapter describes the structure and dependencies of managed solutions that would be imported (installed) in the target production environment. See subpages for more information about the contents of these solutions.



Orchestration Center

Provides critical dependencies for the integration modules, as well as UI to access the data collected throughout the automated processes.

App integration module(s)

Your custom app integration modules fall under this category. They will all depend on the orchestration center being loaded first, as it provides critical components of both testing and work item capabilities.

Final automation solution ("business" or "master")

This solution serves one business process, and contains cloud flows, desktop flows and data structures that are required to handle a given automated job. The final automation solution should not work with the underlying apps directly. Instead, it should only connect to the apps or systems through dedicated integration modules.

Orchestration Center components

Wednesday, January 19, 2022 5:15 PM

Main components:

- Apps
  - Orchestration Center - the main UI
- Tables
  - RPA Modules - one record will be created and used by each installed module
  - Tag Rules - here are the default and custom rules for tagging *Work Items* based on the exceptions found during the desktop flow execution
  - Test Case - one record will be created automatically for any test case ran (as long as it is using the template provided)
  - Test Case Runs - documents every single run of any test case
  - Test Run History - a "test run" occurs when test cases for a particular module are ran, either scheduled or on demand
  - Work Items - common table for all events that should trigger a desktop flow run. Work item will contain both the input data (Payload) as well as - if the process is ran successfully - the output (Payload Output). Every attempt to run that process, whether successful or failed, will be linked to this common record as a child
  - Work Item History - these are individual runs of desktop flows. A work item history will be first created when the process is scheduled for execution, and then depending on outcome - updated with either a Failure or Success status. Work Item History records are linked to parent Work Item, which contains the input and latest output data
- Cloud flows
  - Apply Tag Rules - applies rules stored in Tag Rules upon failure of a Work Item job. The tags are then stored in the Tags field in respective Work Item record.
  - Fill SLA deadline if not provided (default 24 hrs) - sets the SLA Deadline field on new Work Items, if a deadline was not provided during record creation. The default is 24 hours.
  - Retry policy implementation - implements the basic retry policy of restarting a job once, after 10 minutes. Upon restart, the job is assigned the RETRIED flag.
- Dashboards
  - Automation overview - shows the status of work items queue, test results and recent work statistics.
  - RPA Details - links up a Power BI dashboard that analyzes the performance of bots
- Choices (option sets)
  - Automatic test cycle - hourly or daily. The implementation of a particular test cycle is provided by individual RPA modules.
  - Execution Status - used by work items:
    - Added - a draft record, not yet scheduled to run
    - Queued for execution - setting this value will cause the job to be sent to appropriate bot
    - Test result - either a success or a failure
    - Work Item type - one value for every job type (one desktop flow) that will be scheduled through the Work Items mechanism.

App integration modules

Wednesday, January 19, 2022 5:16 PM

To create an app integration module, please use the supplied template.

Mandatory components of an app integration module:

Type	Name	Purpose	Adaptations required from toolkit template
Cloud Flow	Ad-hoc test case run	Run it manually to start the test cases.	None
Cloud Flow	Run *xxx*	Xxx represents one type of work item (see *Work Item Type* choice).	Create one for each of your *Work Item Types*. In the flow's trigger, replace 768280000 with the choice id for your work item type. rpakit_type eq **768280000** and rpakit_executionstatus eq 343970001 This is **the place** for building your cloud+desktop flow combinations to deal with your work items.
Cloud Flow	Run test cases (child)	To collect all the credentials needed from Key Vault and run desktop flow with test cases, passing the credentials.	Use the **Retrieve credentials** scope element to put your Azure Key Vault actions for any user id or passwords your test case desktop flow requires. Make sure these values are actually supplied to the desktop flow.
Cloud Flow	Scheduled test case run	To start "Run test cases (child)" in regular intervals.	None.
Desktop Flow	[*AppName*] Test cases	This flow contains subflows for each individual test case you have built.	Add your test cases as subflows. Call each and every subflow from the Main subflow. See Tests
Environment Variables	As needed	To hold configuration parameters for the apps you integrate with.	Create any environment variables needed to access your target systems (only non-confidential values may go here, for everything else - use Azure Key Vault).
Security Role	Power Automate Desktop user for test cases	To allow PowerShell scripts in PAD (used in test case template) to manipulate Dataverse entities like Test Case, and to read environment variables.	Only if you need your test cases to do additional operations on Dataverse.

As your integration module connects to orchestration tables (i.e. to Work Items and RPA Modules), it will require the Orchestration solution to be installed first.

The "[Sample] Integration" module provided is a template for building integration modules. Do not use the solution itself - create your own solution and add the same components to it. Otherwise you will not be able to import more than one such module into the target environment.

Process Automation Solution

Wednesday, March 2, 2022 5:57 PM

Also known as the "final" or top-level RPA solution, as it sits on the top and uses capabilities of all other components.

This one is up to you (TODO - describe in more detail), but typically it would contain:

- Some cloud flows to fill work items based on your specific process triggers,
- Some cloud flows to take completed work items and inform any other systems or teams that should pick up the work from there,
- And finally some dashboards, cloud flows or exception handling rules to deal with known and unknown exceptions.

The top-level solution can also create some additional data structures for holding your data, typically linked to or extending the Work Item table.

Testing Solution

Wednesday, March 2, 2022 6:02 PM

An optional component (TODO - describe in more detail) that will help in end-to-end testing of your process. This one could be installed in test environments but omitted from production.

Building desktop flows

Wednesday, January 12, 2022 10:06 AM

General advice on creating Desktop Flows

Tuesday, March 22, 2022 11:40 AM

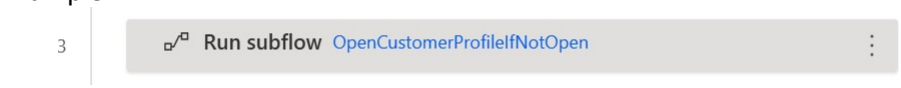
TODO

Satisfying the requirements

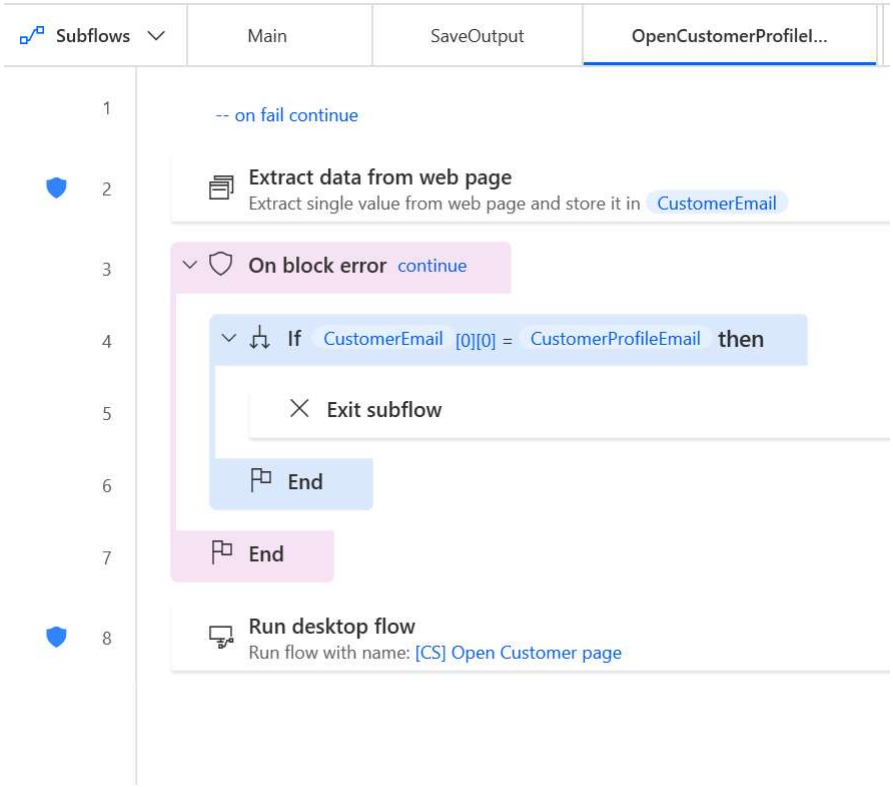
Wednesday, January 12, 2022 1:02 PM

~~If an assumption can be easily satisfied by calling another flow (other than **Init**), it is better to remove it from prerequisites and call the required flow instead.~~

Example:



Code:



Here, the process checks if the user profile being open (#2) is actually the one we want to be seeing (#3). If it is the right user, subflow exits back to main. If it is not the right user profile, is calls **[CS] Open Customer page** to satisfy the requirement.

Note that an error in data extraction means we have completely wrong page open, so this flow quietly moves on to running a desktop flow to open the profile. Any error in actions #3..#7 is also silently ignored and leads to calling **[CS] Open Customer page** to satisfy the dependency.

Browser handle

Wednesday, January 12, 2022 12:02 PM

Since you cannot (currently) pass a browser instance variable between desktop flows, the following alternative solution is used.

Init flow

This flow is expected to attach to existing browser or open a new instance. Usually, it takes the following input (your use case may vary):

- *Username* and *Password* as sensitive values
- *Application URL* of the app to open - making it a parameter helps build test environments that work with test instance of your app

The *Init* flow returns two values:

- *BrowserURL* is taken from %Browser.URL% and is the most recently open address (it can change throughout your process, so you will need to keep it up to date throughout your work with the browser)
- *BrowserHandle* is taken from %Browser.Handle% and does not change throughout execution. Unfortunately, a flow cannot connect to the browser by handle, the only thing you can do with the handle is close window.

Example:

Your "actual work" flows

They should receive the following input:

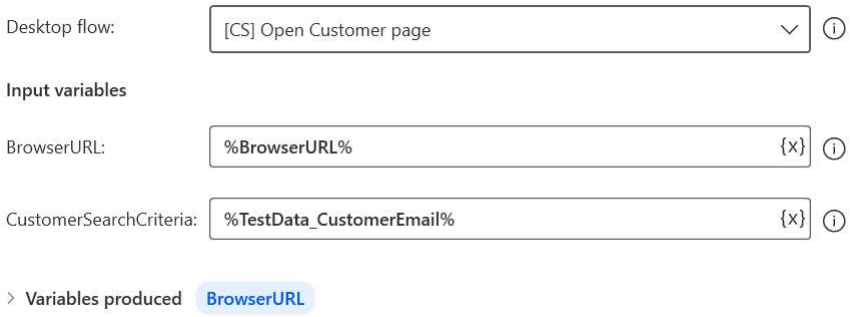
- *BrowserURL* - the most recent URL in the web browser
- Plus any business data required to do the job

Assigning a meaningful default value to *BrowserURL* will help you in debugging.

They should return the following output:

- *BrowserURLOutput* - in your calling flow, you should redirect this to *BrowserURL* variable like this:

Example:

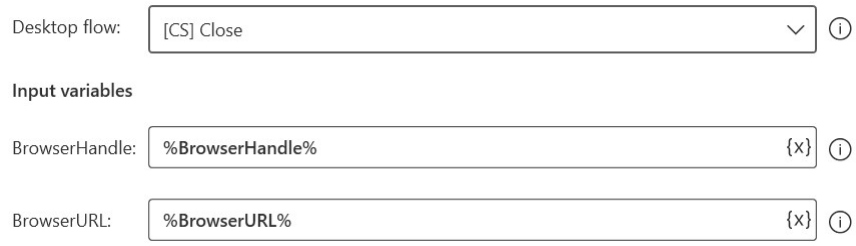


See Subflow structure for more recommendations.

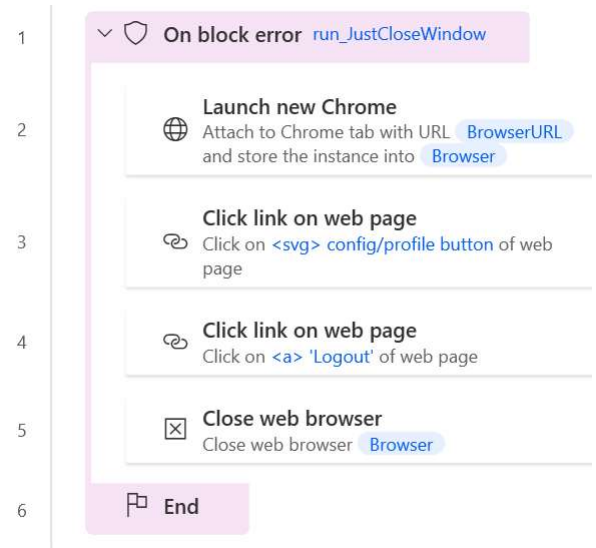
Close flow

This flow should use BrowserHandle and/or BrowserURL to close the browser.

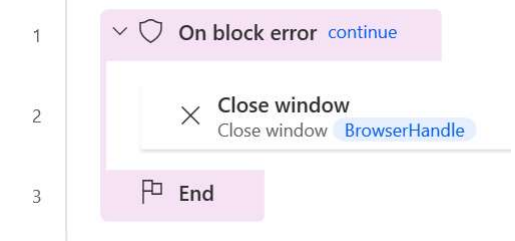
Example:



It should use reasonable means to attach to browser and log out of the app, and if that fails, just use the handle to close browser window. Example code:



JustCloseWindow subflow is started on any errors:

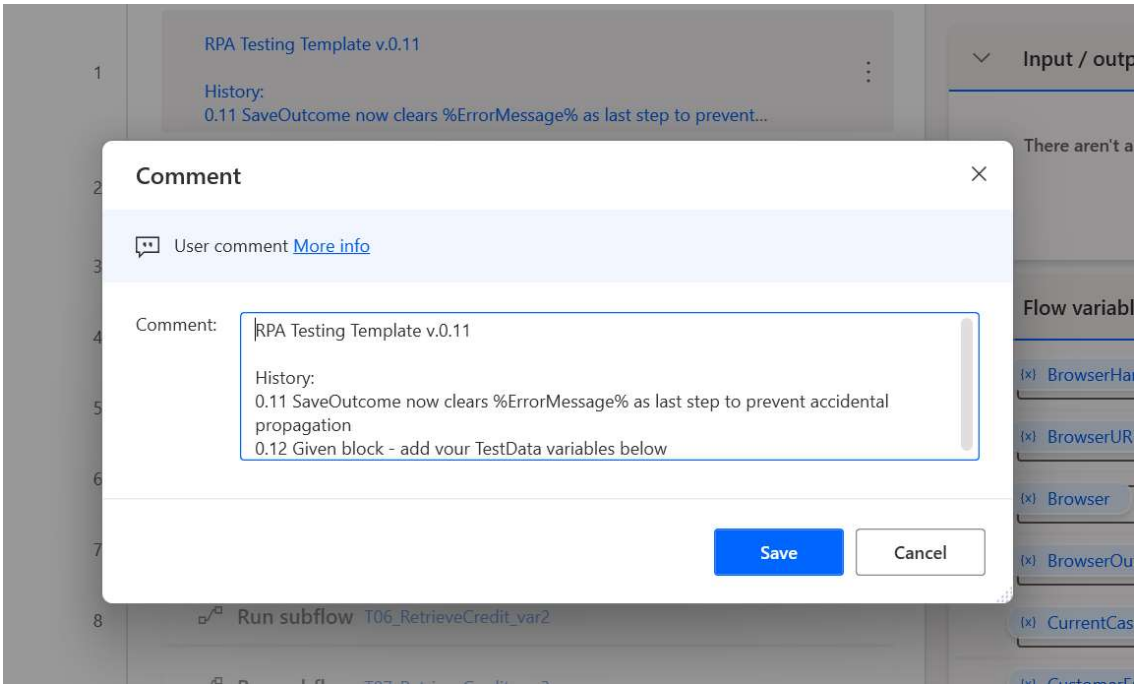


The on block error section here just ignores any issues found.

First comment block

Wednesday, January 12, 2022 12:22 PM

Every flow should start with a comment block listing flow name, version, update history, prerequisites that have to be completed prior to starting (i.e. web browser should be open on the customer profile).



RPA Testing Template v.0.11

This flow carries out all test cases and shows output in a message box.

Prerequisites: none.

History:
0.11 SaveOutcome now clears %ErrorMessage% as last step to prevent accidental propagation
0.12 Given block - add your TestData variables below

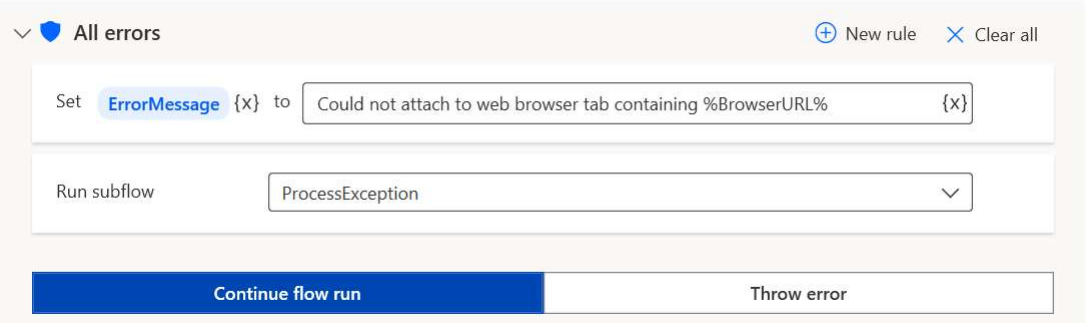
Standard variable names

Wednesday, January 12, 2022 10:09 AM

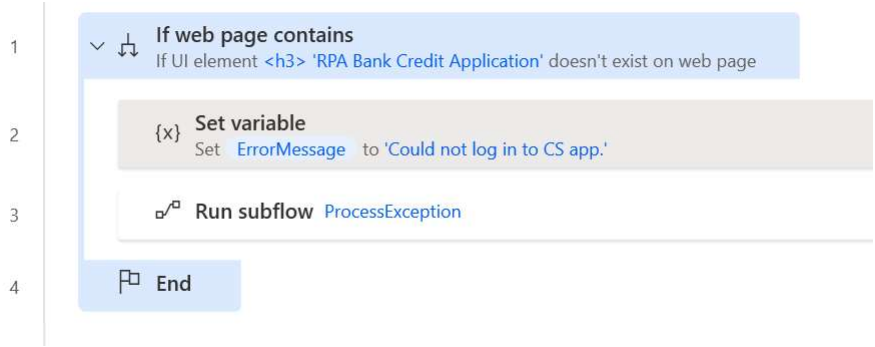
ErrorMessage

used to pass custom exception text to **ProcessException** subflow

Example 1:



Example 2:



EH_MaxRetries

Used in exception handling. Stores the maximum number of permitted retries. If %EH_RetryCounter% is greater than %EH_MaxRetries%, an error will be thrown (error message needs to be first stored in %ErrorMessage%).

EH_RetryCounter

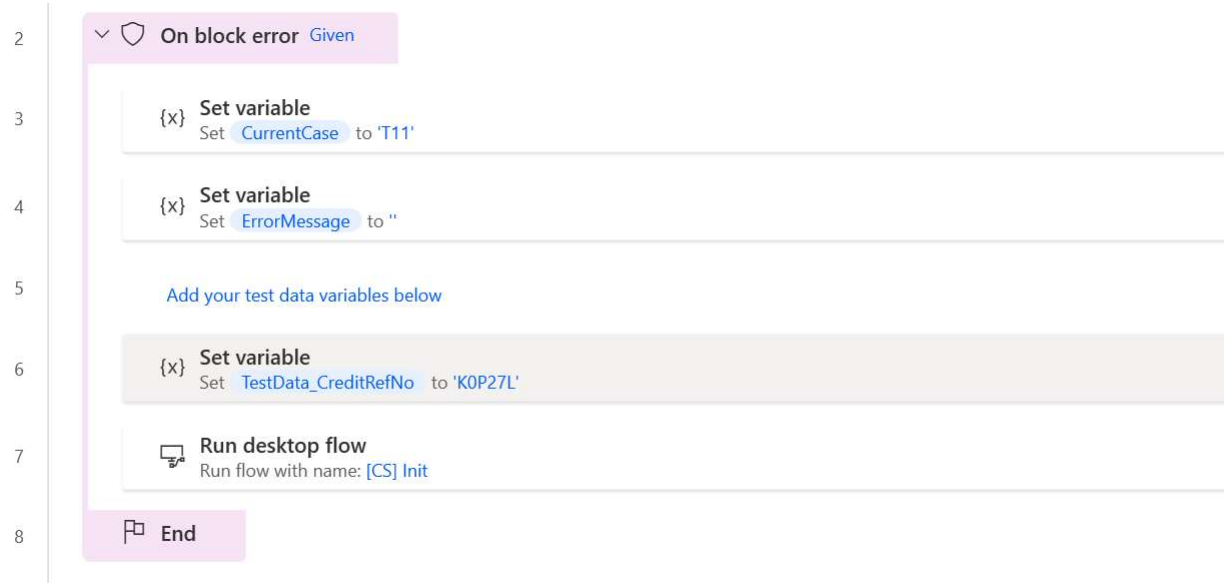
Stores the current number of retries.

Standard variable prefixes

Wednesday, January 12, 2022 10:17 AM

TestData_

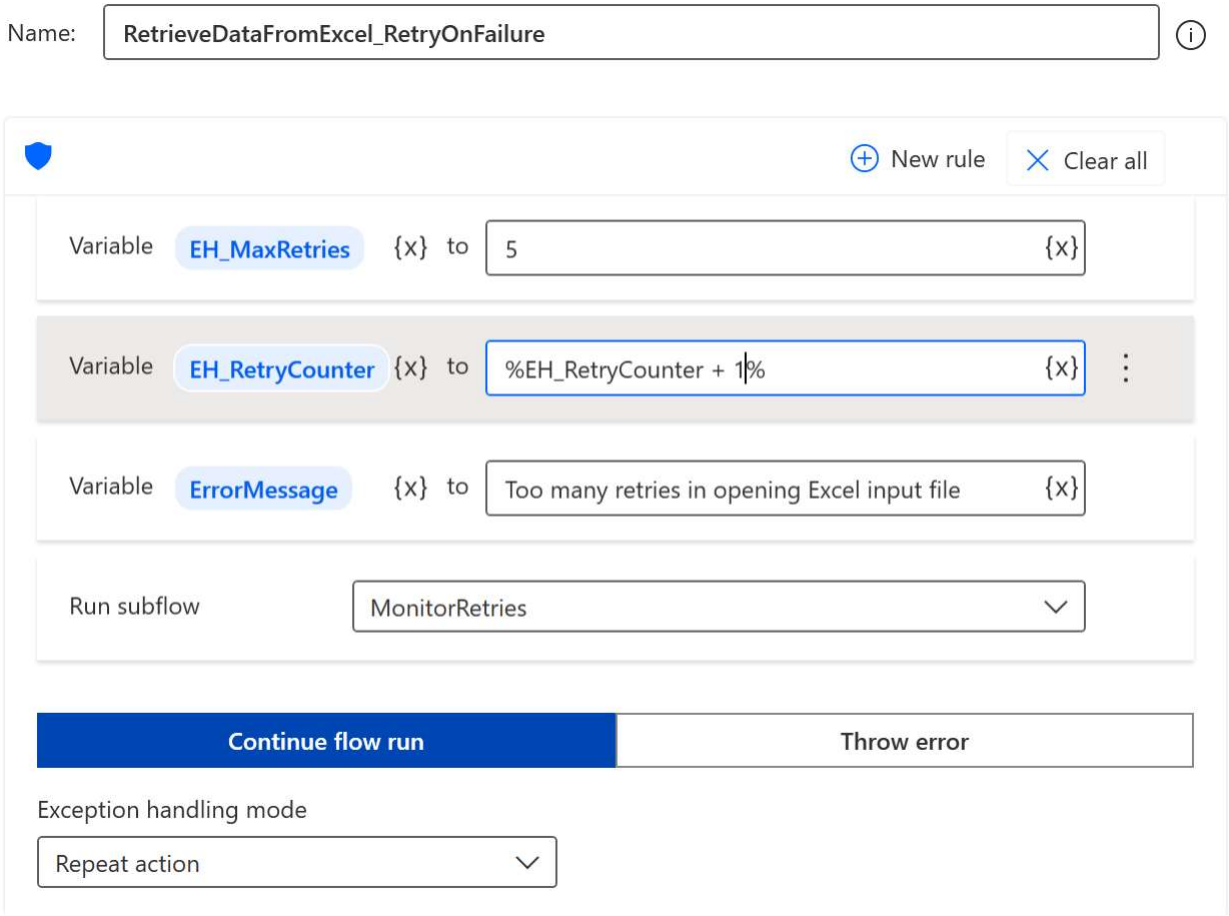
If your test case requires a piece of data, like a customer ID, put it in a TestData_* variable and set its content in the **Given** section.



EH_

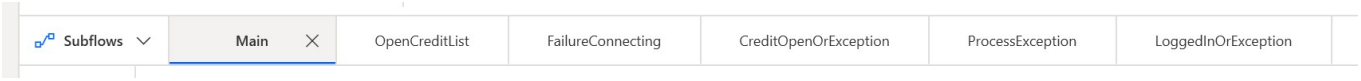
Variables used in your **exception handling** processes.

Sample:


Name:
RetrieveDataFromExcel_RetryOnFailure
New rule
Clear all
Variable
EH_MaxRetries
{x}
to
5
{x}
Variable
EH_RetryCounter
{x}
to
%EH_RetryCounter + 1%
{x}
Variable
ErrorMessage
{x}
to
Too many retries in opening Excel input file
{x}
Run subflow
MonitorRetries
Continue flow run
Throw error
Exception handling mode
Repeat action

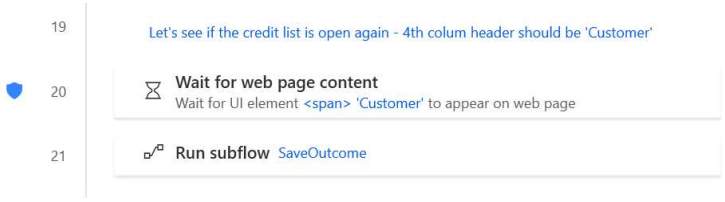
Subflow structure

Wednesday, January 12, 2022 10:06 AM



Main flow

It should end calling the **SaveOutcome** subflow. Example:

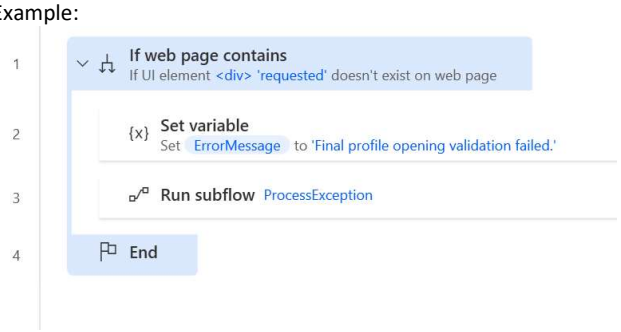


If there's any place where your flow could end successfully using the Stop flow action, you should call **SaveOutcome** there as well.

...OrException flows

These flows check your assertions. They are not taking any parameters, they are meant to silently finish if the assertion is satisfied, and fail otherwise.

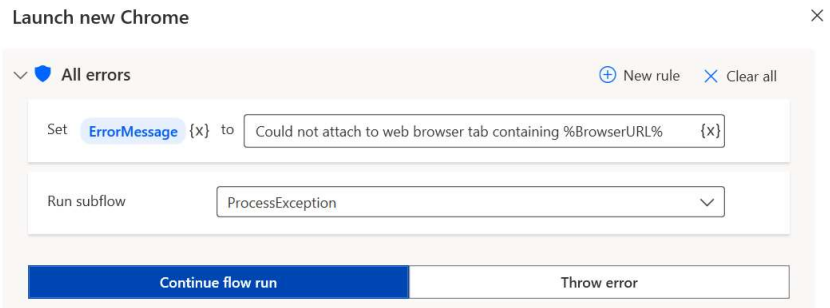
Example:



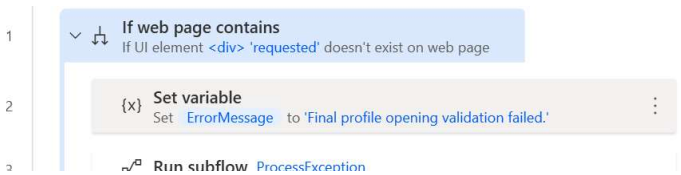
Always check every single assumption that your flow has, and fail gently if they are not satisfied. Failing gently (through ProcessException) will help user understand what their fault was.

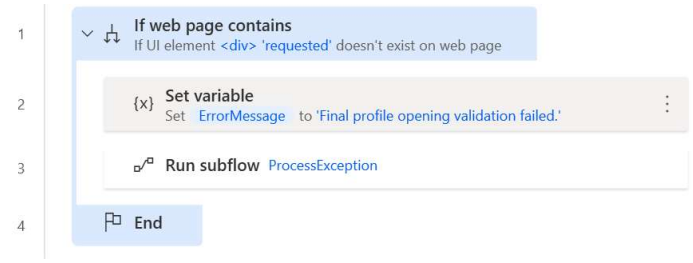
Example assumptions:

The web browser is open - your attach to browser action will fail otherwise, so make sure you override that error using **ErrorMessage** and **ProcessException**. Example:

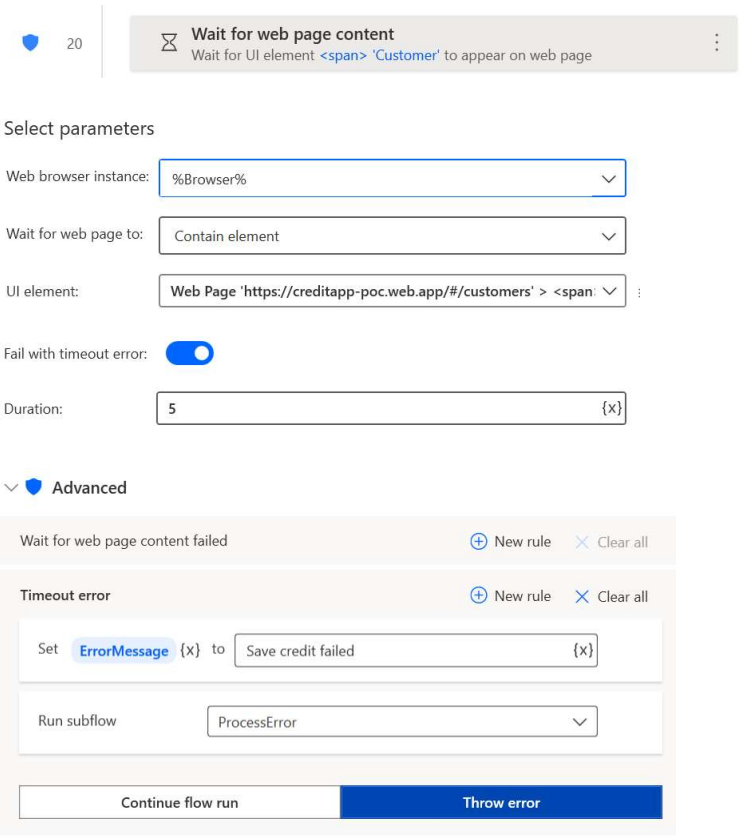


Logged in to a specific app - check if some common UI element is available on the page, and override the exception:





If the application is in progress of doing something, and it's not the start condition for your flow but a result of actions in that flow, it is better to use **Wait for web page content** and handle the **Timeout** error, than to just check if web page contains some element:



ProcessException

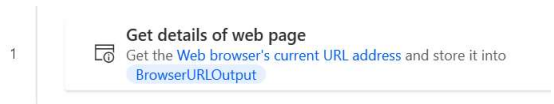
See ProcessException flow. Only launched on unsuccessful close.

SaveOutcome

This flow saves the most recent URL the browser is pointing to in **BrowserURLOutput** variable.
SaveOutcome can also be used for any cleanup actions - but you should make sure it never throws any error on its own. If any of the cleanup actions is at risk of an error, you should catch the exception and handle it.

SaveOurcome will be launched on both successful and unsuccessful close.

Example code:



All other subflows

Use as required.
However, if your subflow is requiring any parameters, and these are not among the Standard variable names, it is a food practice to prefix the parameter flow variables with the prefix of the flow name.

Example: in a **SaveCustomerName** flow, the name to be saved is provided through

SaveCustomerName_Name variable.

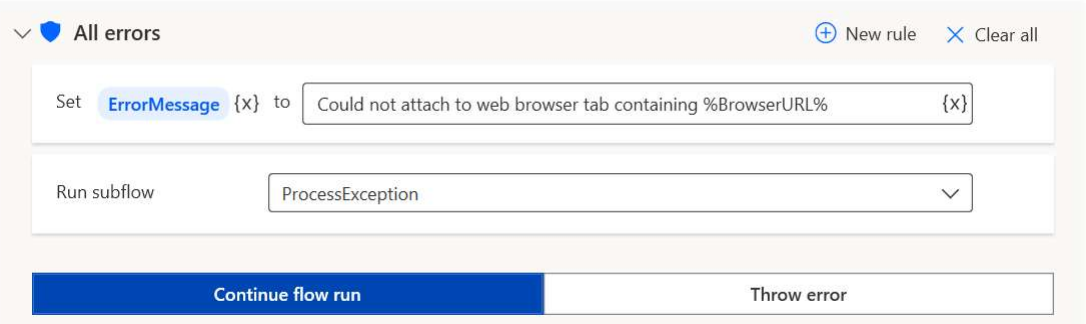
ProcessException flow

Wednesday, January 12, 2022 11:58 AM

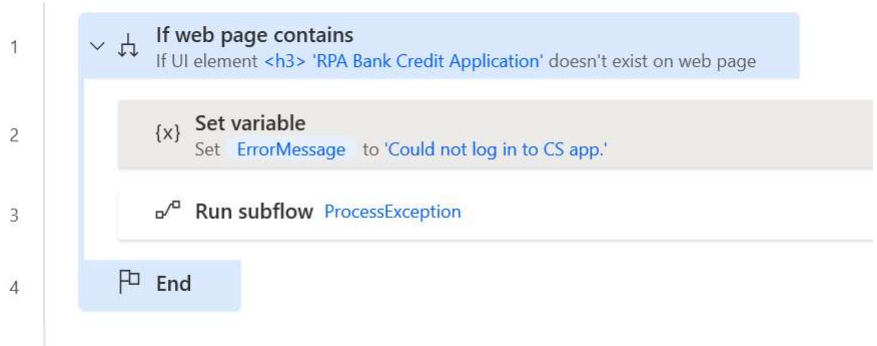
If you want to override an error and provide your own leading message, or you have a business error (no issue in PAD, but still your flow cannot continue), you should call **ProcessException**.

Before calling, set **ErrorMessage** to your custom error message.

Example 1:

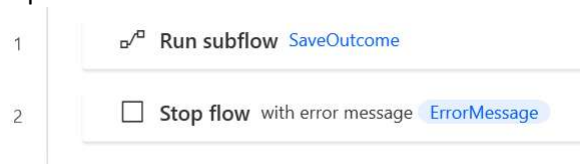


Example 2:



The actual code for **ProcessException** updates the BrowserURLOutput variable with the latest the browser is pointing to, and throws an exception. Then, the calling process should look at exception details and catch the exception if desired.

Example:



Integrating desktop flows into your business process

Tuesday, March 22, 2022 5:23 PM

Business people want bots to be dependable. The team within an organization that is responsible for ongoing bot maintenance should be able to automate not only the process itself, but also the supporting subprocesses that handle the unwanted - application errors, data quality issues, and individual application or feature unavailability.

Key word here is resilience. What we do not want is a flow that delivers it's best, but then fails without leaving any trace if anything goes wrong. Even worse is if the administrator needs to spend considerable amount of time to understand which bot run failed and how to restart it.

What we do want is bots that take responsibility for their work. A bot that is resilient should satisfy the following non-functional requirements:

1) Any external trigger that is sent to the bot should be immediately stored in a database. It should remain there until it completes or until an operator decides to drop it. While it is stored in the database, it should be visible in reports, and allow for:
    - SLA evaluation,
    - Retries (when the user believes the error is potentially temporary in nature,
    - Changes to the triggering event data, in case a business change needs to be implemented,
    - Manual restarts,
    - Resuming from one or more specific interim checkpoints when the process does not need to start from scratch.
2) Any attempt to run the bot process, whether successful or failed, should be logged with the identification of the machine where a particular run was executed.
3) Any output of the process should remain in the database with proper status until it is successfully transmitted out of the environment to a relevant external system.

With the above requirements satisfied, one can be sure that any record successfully stored in bot's database will be dealt with, and either it will be completed or the calling party will be properly notified about the final outcome.

This chapters shows how to build such bots.

Teams involved

Tuesday, March 22, 2022 5:48 PM

Name	Function
Business owner (outside of IT)	Orders the creation of a bot. Provides detailed specs that show in detail how the bot shall behave, and how to deal with any exceptional situations. Resolves business issues within their bots (when the problem can only be resolved with a manual intervention or change to input data, usually beyond the scope of the specs)
Automation Specialist (business or IT)	Creates all needed automation components, leveraging existing tools from CoE to maximize impact. Whenever a specific piece of automation is beyond the skill level of the specialist, or has significant reuse potential, the specialist involves developers from CoE to build that piece.
RPA team (often within IT)	Ensures smooth bot execution, from the moment when a Work Item is stored in the environment until the final status is communicated to outside systems as required. Configures and maintains additional automations that handle the exceptions that are found.
Automation Center of Excellence (CoE)	Creates and maintains integration modules for commonly used apps. Updates the module content as required when new functions are required by business owners.
Governance	This team configures the entire Power Platform, setting the governance rules such as Data Lifecycle Policies.

Exception handling basics

Thursday, March 24, 2022 10:49 AM

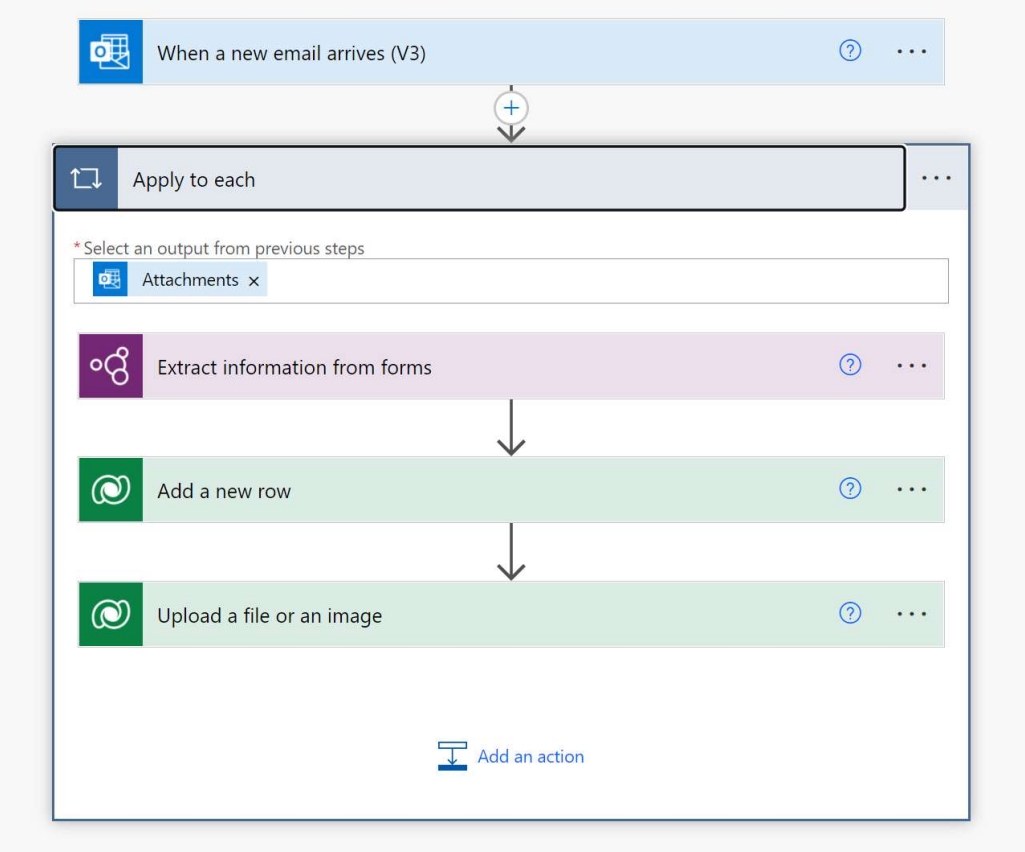
If you look only on the out-of-the-box functions of Power Automate, any kind of flow - whether cloud or desktop - is executed on best effort basis, and any failure will be logged. But it doesn't go any farther than that - if no measures were taken to catch and handle these exceptions, then nothing will happen.

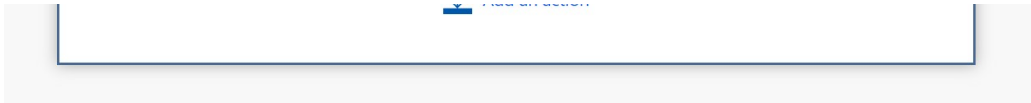
Example 1:
These flows have failed, but since there was no exception handling in place - the result is an error in logs.

	Mar 1, 10:00 AM (3 wk ago)	00:12:51	Succeeded
	Mar 1, 08:01 AM (3 wk ago)	00:28:52	Succeeded
	Mar 1, 05:01 AM (3 wk ago)	03:00:39	Failed
○	Mar 1, 02:00 AM (3 wk ago)	03:00:56	Failed
	Feb 28, 06:12 PM (3 wk ago)	00:25:17	Succeeded
	Feb 28, 06:12 PM (3 wk ago)	00:12:44	Succeeded
	Feb 28, 06:00 PM (3 wk ago)	00:12:22	Failed
	Feb 28, 05:51 PM (3 wk ago)	00:12:34	Succeeded
	Feb 28, 10:00 AM (3 wk ago)	00:15:46	Succeeded
	Feb 28, 08:01 AM (3 wk ago)	01:40:13	Succeeded
	Feb 28, 07:17 AM (3 wk ago)	02:08:40	Succeeded
	Feb 28, 06:04 AM (3 wk ago)	03:05:45	Succeeded

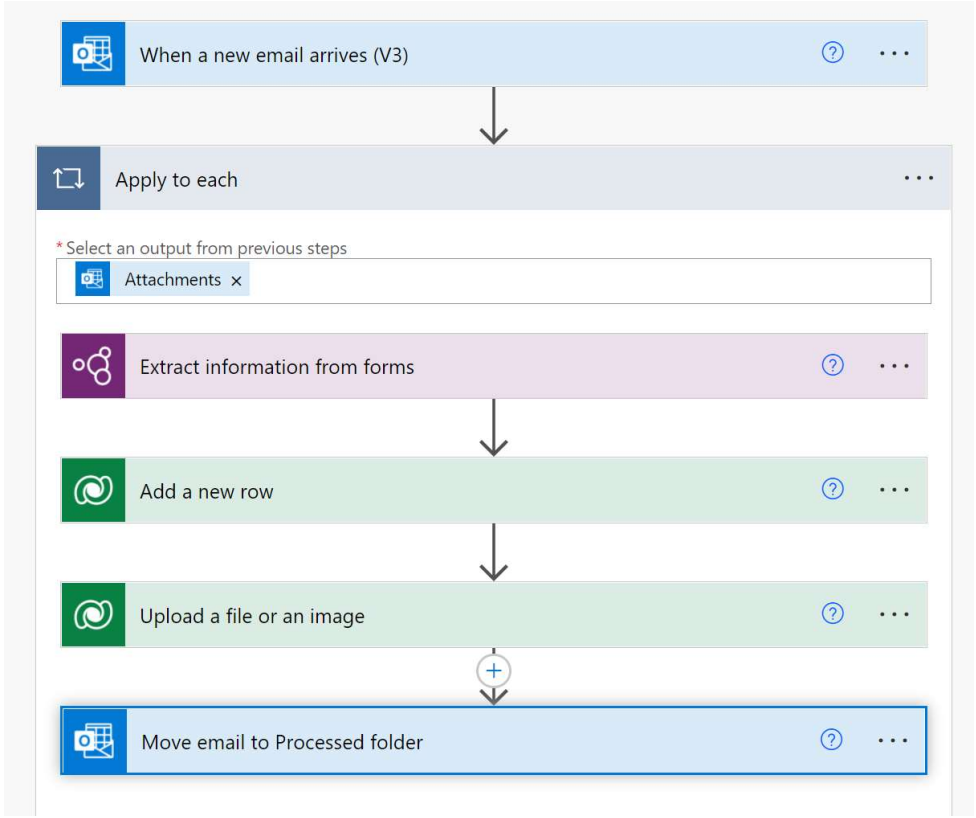
Your Rule #1 of automation: Handle all the exceptions you can foresee. For the remaining ones, always build a process in a way that incomplete process runs would leave incomplete entries somewhere, so that the incomplete entries could be spotted with a report.

Example 2:
This process works on emails in a mailbox. If the processing fails for any reason, you will have to look at the log file to spot a problem.



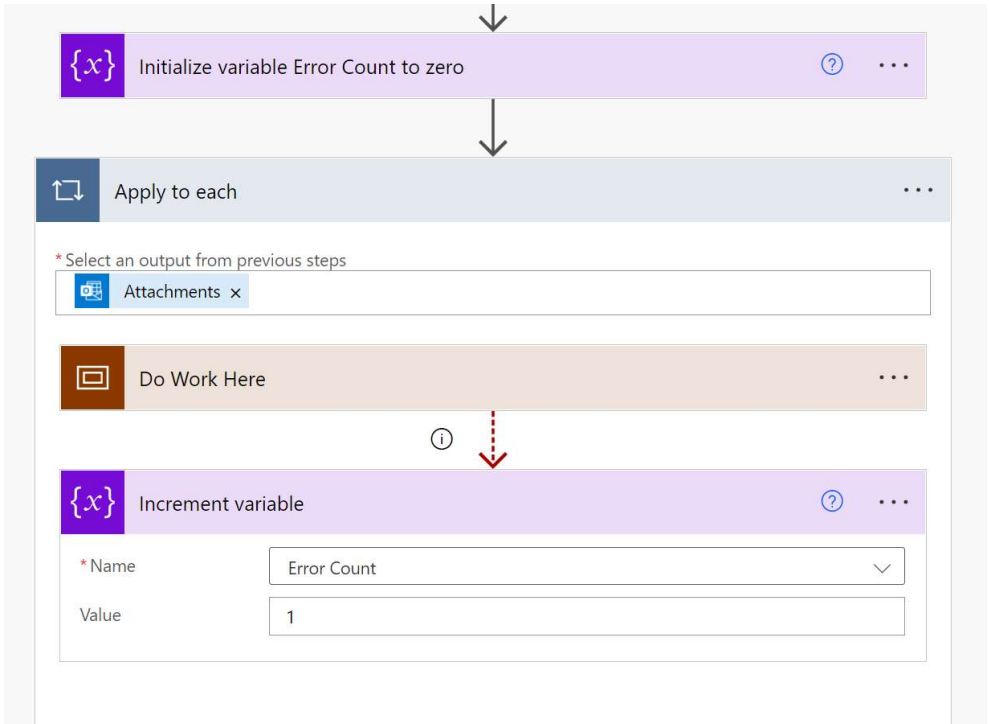


However (and assuming it's a personal mailbox so somebody would read these emails normally), if your last step of the process would be moving that email into another folder, then any unhandled exception would leave the email in the inbox, allowing the mailbox owner to spot it and intervene.

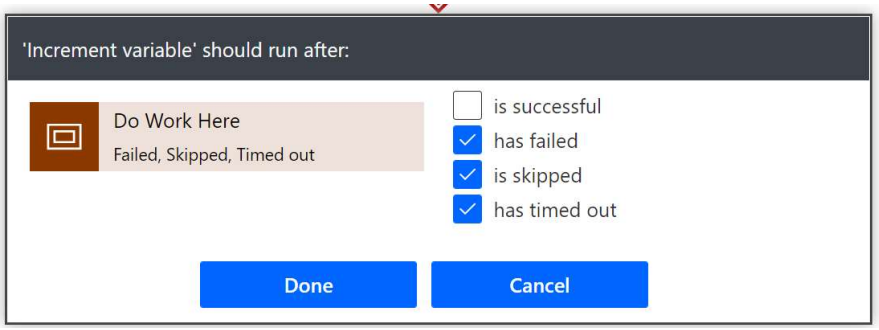


This is **indirect** error handling - an error would be still thrown, but you are likely to spot it.

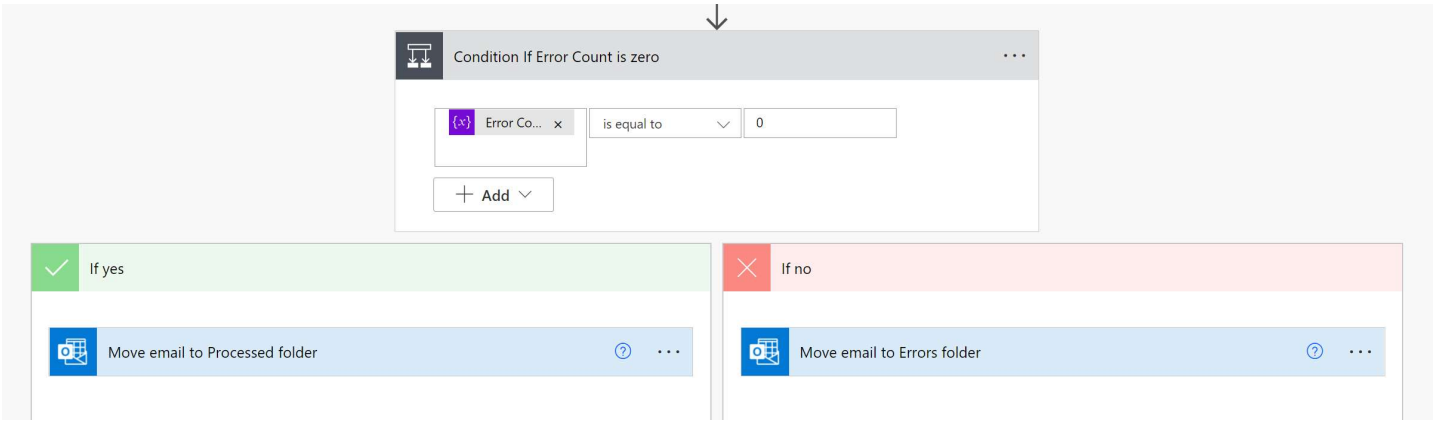
A better way is to plan for issues to happen. Here the actions within the loop were moved to a *Scope* action (called *Do Work Here*). A variable for error count is initialized first and set to zero. Then, in case of any errors within the scope action, error count will be incremented by 1.



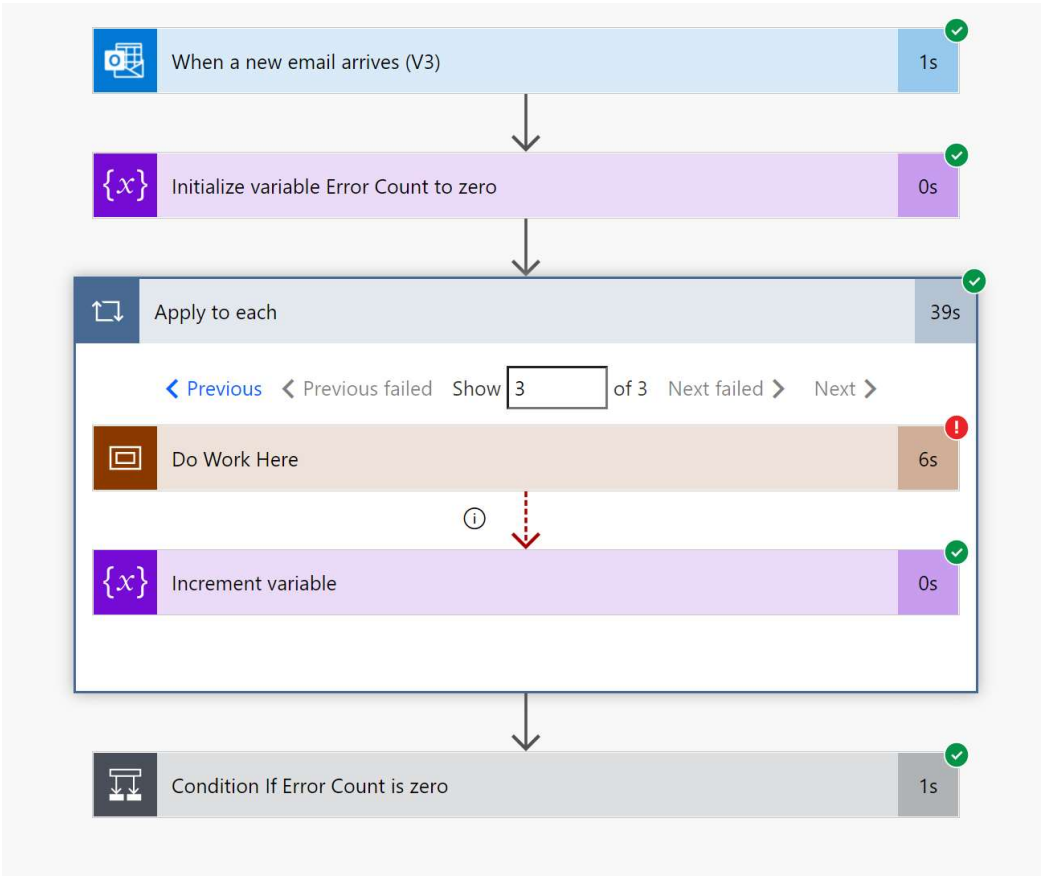
Note that the increment variable action has *Configure run after* set to execute only in case of issues:



So finally after the loop is done, we look at the error count, and if it is non-zero, we move the email into Errors folder for human operator to review.



Since we used the *Configure run after* option, our error is considered to be handled, and the cloud flows continues its work:



Also, despite the issue with handling email attachment #3 in the above picture, the whole run is still deemed a success:

Mar 24, 11:35 AM (6 min ago)	00:00:40	Succeeded

Final note:
If the moving to another folder fails, for example someone removed the Errors folder, the email will remain in inbox and is likely to be spotted.

Work Item queueing

Thursday, March 24, 2022 11:01 AM

As shown in the previous chapter, your goal with exception handling is that actions that may fail should be wrapped with rules to ensure these issues are handled.

Sometimes an issue can be caused by an external application being not available. Often, the process maybe safely retried a few minutes or hours later, hoping that the target system is working again.

Other times there might be some issues with the data, requiring a human intervention. These situations will involve an operator, who might be either changing the automation input data, or manually making changes to the records in the application.

To deal with these issues, we will need reports. With potentially multiple different workloads being automated in a single environment, we cannot afford to have a separate report for each workload - there should be a single view covering all the different workloads.

So we need a more flexible data model for storing all workloads, all items being handled by robots, in a single data structure (ideally in Dataverse). Such data model could then be a foundation for reports, for user interventions, and for configuring which flows should be retried and when.

Work Items - why do we need them?

A *work item* is a single record of data that your automation should handle. Depending on the purpose of your automation, a record could be:

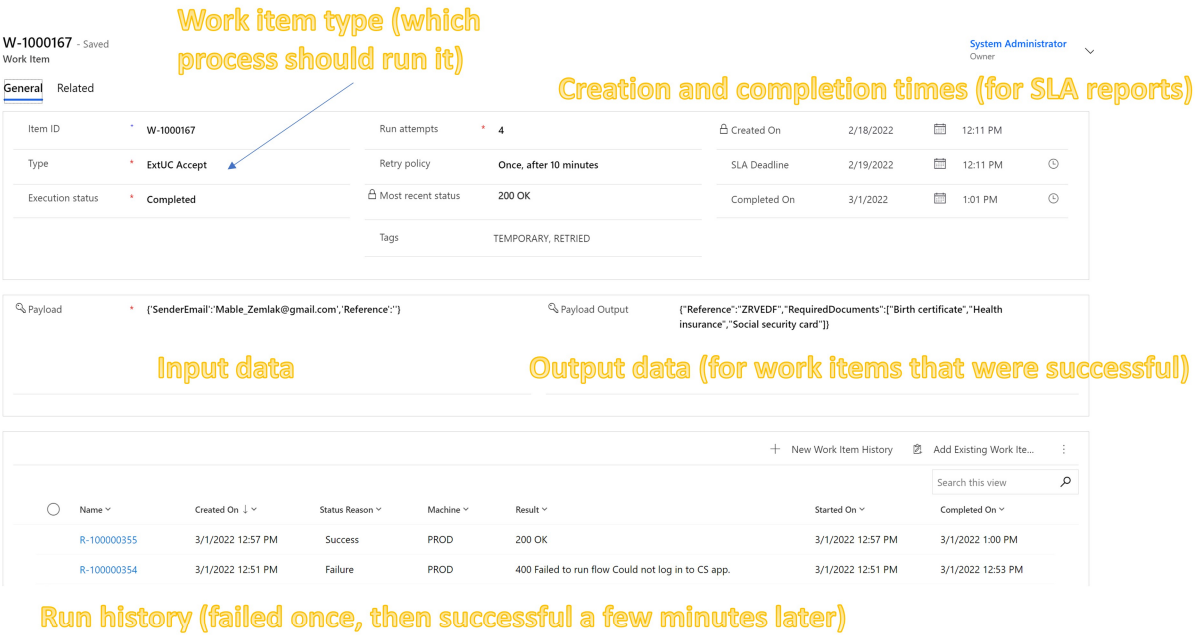
- An email received from a customer,
- A record created in a list,
- A document containing an invoice, etc.

Power Automate has a queue of flows - cloud flows that have to be executed, or desktop flows that were scheduled to run. But with potentially multiple attempts to execute automation for the same single record, there is no overarching frame linking all attempts for that single records.

Mar 24, 11:35 AM (6 min ago)	00:00:40	Succeeded
Mar 24, 11:23 AM (18 min ago)	00:00:40	Failed
Mar 24, 11:21 AM (20 min ago)	00:00:16	Succeeded
Mar 24, 11:18 AM (22 min ago)	00:00:43	Failed
Mar 24, 11:18 AM (23 min ago)	00:00:49	Failed
Mar 23, 11:00 AM (1 d ago)	00:00:21	Succeeded
Mar 15, 05:03 PM (1 wk ago)	00:00:15	Succeeded
Mar 15, 04:42 PM (1 wk ago)	00:01:30	Succeeded
Mar 15, 04:42 PM (1 wk ago)	00:01:14	Succeeded
Mar 15, 04:34 PM (1 wk ago)	00:00:18	Succeeded

Figure 1: OK, some flows have failed, but which ones have been eventually retried and were successful?

Work Item screen



In the above example, a work item with ID *W-1000167* has been submitted, and desktop flow *ExtUC Accept* should be started. The input data are in JSON format, and contain two variables:

- SenderEmail ("Mable_Zemlak@gmail.com")
- Reference (empty)

This work item has failed once due to login issues (see bottom of the page), and then retried a few minutes later, this time it was successful. The process ended with the following output:

- Reference ("ZRVEDF"),
- Required documents (a list containing "Birth certificate", "Health insurance" and "Social security card").

The payload input & output structures could vary for different processes.

For the purpose of reporting, notifications and SLA evaluation, the work item contains the creation time, the target SLA deadline, and the actual completion time.

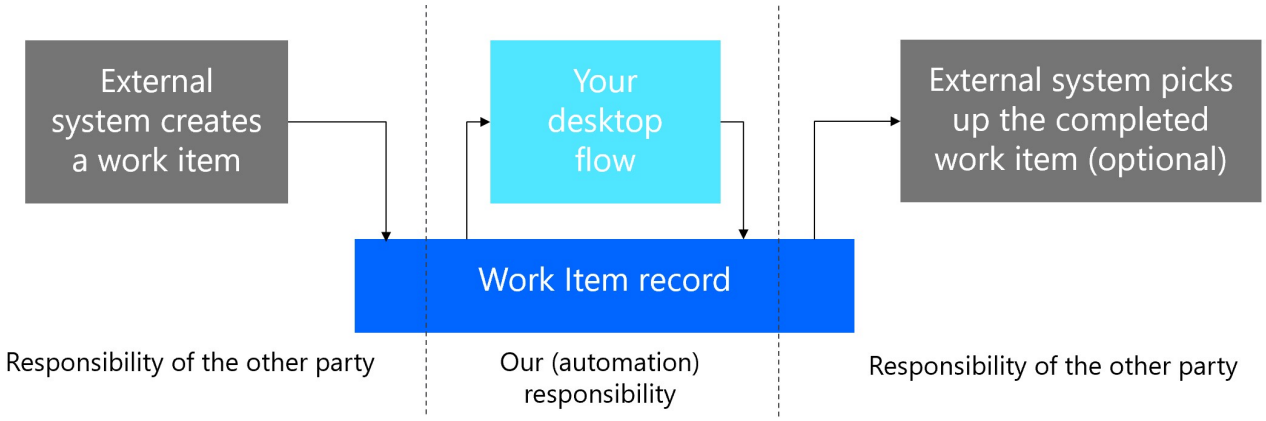
Finally, for the purpose of automatic retries, the screen contains also a retry policy (indicating if, and how soon, the process should automatically restart), the number of attempts so far, and a list of tags assigned to the work item. Tags are meant to help distinguish different error situations, and impact the retry policy or reports.

Work items in your top-level process

Thursday, March 24, 2022 12:42 PM

With a complex process and many systems playing their roles, it is important to define the separation boundaries. With boundaries set, it will be easier to understand which of the potential errors should be handled by which actor.

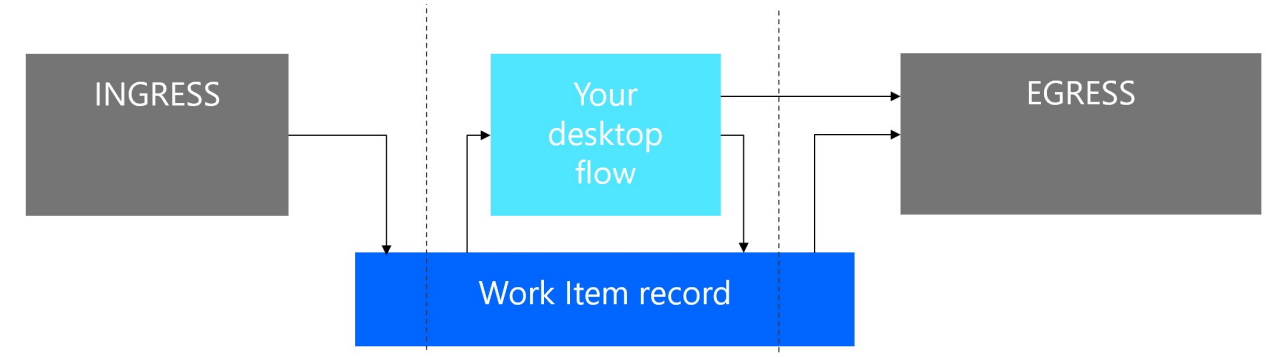
One proposed approach is shown here:



In the above diagram:

- It is the responsibility of the source to create a record in the table if our automation system (to queue their work item for execution),
- It is "our" responsibility to handle every record that was created in work items table and deal with it until it's either closed (potentially with retries), or the work item is dropped and will no longer be retried.
- Notifying the next actor in the process that some work has to be done can be done in two ways:
  - Either the passing of the baton (like in a relay race) is done in the desktop flow,
  - Or it is the target party responsibility to pick up completed work items, for example by means of a cloud trigger connected to work items table.

From the automation environment perspective, the inflow of new items into work item queue is called **ingress**, and the outflow of results is called **egress**.



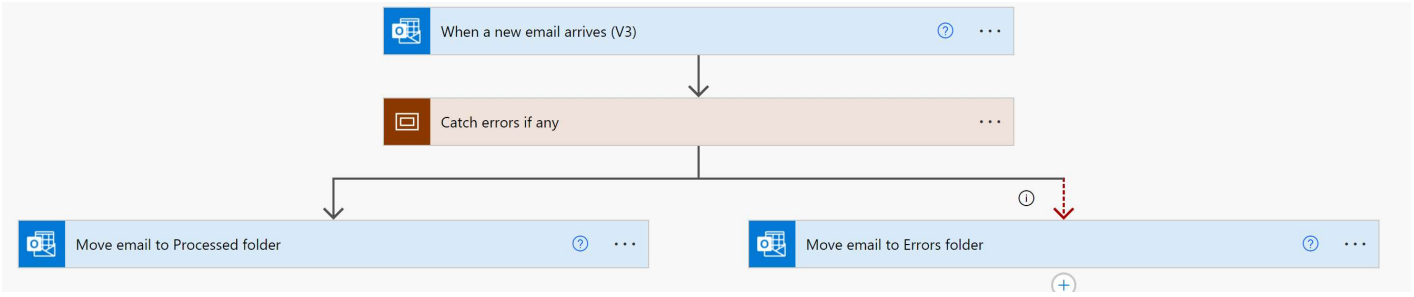
Ingress with Power Automate cloud flows

The following example shows how a cloud flow can start for every new email and create a Work Item queue record.
The mandatory values that must be set on work item creation are:

- Execution status = *Queued for execution*
- Payload = your JSON object with input data
- Run attempts = 0 (this field stores the number of attempts so far, not the desired or maximum)
- Type = the type of your input flow (you must have a cloud flow for each of the possible values

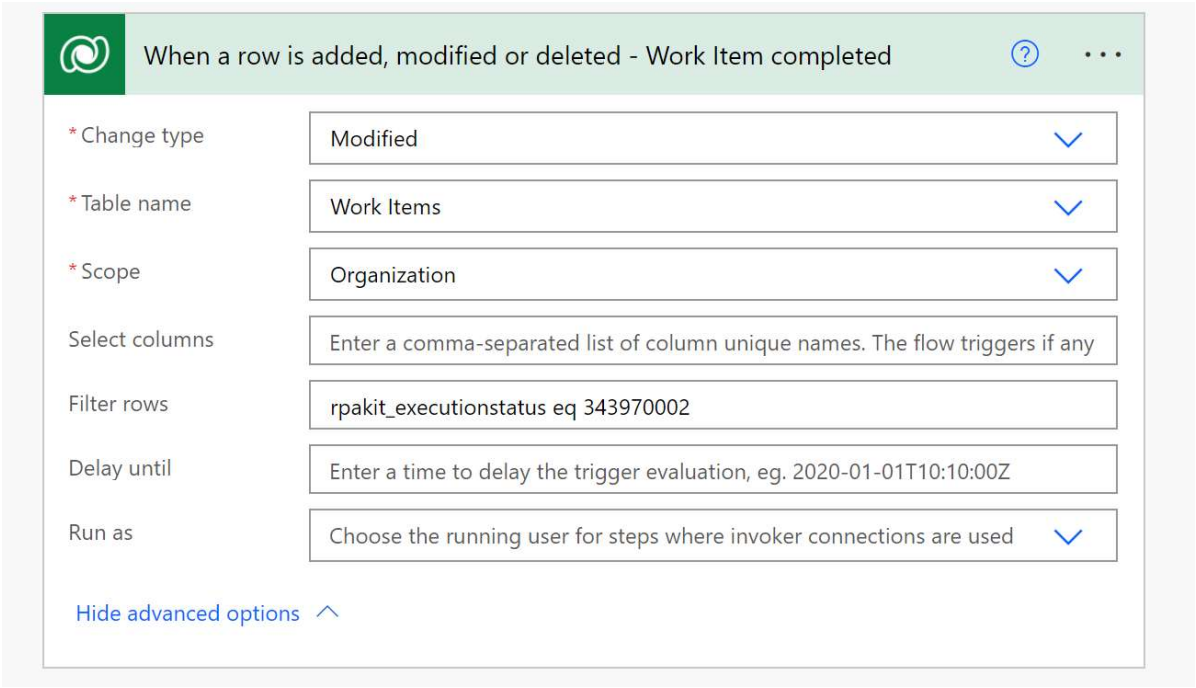
in your business process solution.

In line with recommendations on exception handling from previous chapter, this flow ends with moving the email either into *Processed* mail folder, or into *Errors* mail folder.



Egress with Power Automate cloud flows

For egress, we can build a cloud flow that reacts to change of Work Item *Execution status* to *Completed*. Here's a sample trigger:



The filter condition is:

rpakit_executionstatus eq 343970002

… which means *Completed.*

Creating & Using modules

Wednesday, January 12, 2022 10:06 AM

Modules provided in this toolkit

Tuesday, March 1, 2022 2:21 PM

Contents

Sample Integration - this is an almost clean template for creating your own integration modules. Use this sample to create your own modules.

EPA Website Integration - this is an assembled module for integration with the Environmental Protection Agency, offering a simple action. Use this sample to learn how modules are built.

EPA Website Integration scenario

This module implements the capability to connect to AirNow.gov and retrieve pollutant information for a ZIP code.

Available functions

- *[EPA] Init* opens the webpage and checks if search box is available
- *[EPA] Retrieve pollutants by ZIP code* - for a given U.S. ZIP code, provides primary pollutant, other pollutant(s) and location friendly name for verification purposes. Throws an error if primary pollutant is not available in a given ZIP.
- *[EPA] Close* - closes the web browser

Test cases implemented

- T01 Basic Flow - checks a simple sequence of init + retrieve + close for Beverly Hills, CA. Verifies if location name matches ZIP code to see if we actually got data for Beverly Hills.
- T02 Sequential Calls - checks if two calls done in sequence will actually provide a different location name on 2nd call. Created to verify if multiple calls without closing the browser actually work OK.
- T03 Bad ZIP code - checks for error thrown on invalid ZIP code.

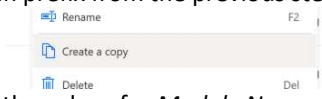
Using the test case template

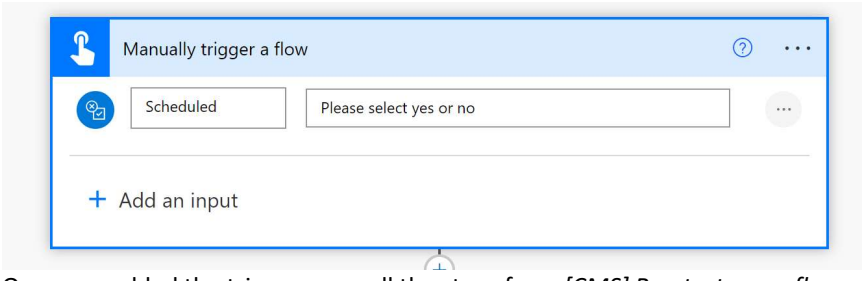
Monday, February 28, 2022 7:46 PM

The template for an integration module is provided as unmanaged solution. Since you can only load a particular solution once in any environment, you must take the following steps to make your own copy prior to any development.

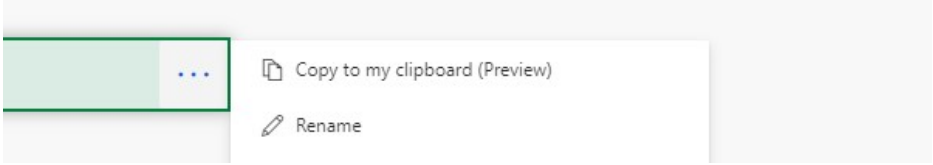
Creating your copy of the *Sample Integration* solution

Here are the steps:

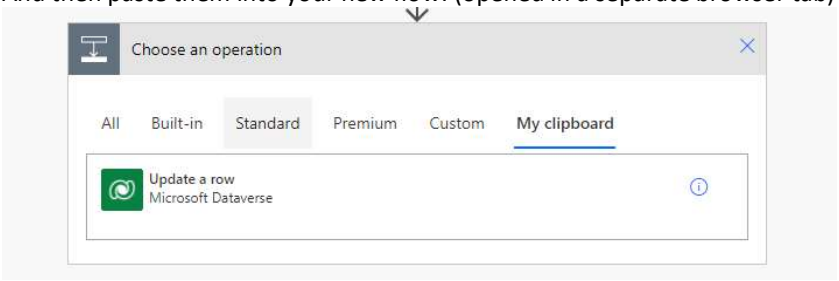
1. Import the *Orchestration Center* solution. Here we use a managed solution - we do not intend to make any changes to it.
2. Import the *SampleIntegration* unmanaged solution. You will use it later as a source of desktop flow templates.
3. Create your own, new solution (use your own publisher prefix, not kit's "rpakit_")
4. Decide on your module name and prefix.
    a. Module name should be alphanumeric (+ spaces)
    b. Module prefix should ideally be a short acronym, uppercase - the prefix will be added to all test case names in square brackets
    c. Module version - use just major and minor version number, i.e. 1.1 (not 1.1.0.7)
5. Create a stub of *[PREFIX] Test cases* **desktop** flow by making a copy of the *[SAMPLE] Test cases* template. Add the newly created flow to your new solution. Replace [PREFIX] with your actual chosen prefix from the previous step.
    a. 

6. Fill in the values for *Module Name*, *Module Version* and *Module Prefix* in steps 2-4 of the *Init* subflow. Save & close.
7. Create the Dataverse Access Object (as shown in Dataverse Access Object) and add to your Azure Key Vault.
8. Create the necessary Dataverse records:
    a. From the main Power Automate window (with the test cases flow closed), hit *play* to run the flow. Paste your Dataverse Access Object when asked for input parameters.
    b. The flow should complete and report issues with test case *T01_BasicFlow* - this is normal:
    c. From the *Orchestration Center* solution, run the *Orchestration Center* app. You might want to bookmark it for later.
    d. Navigate to *RPA Modules*. You should see a record created for your module. Click on the name to open its form.
    e. From the browser URL, note the last GUID - this is the ID of your module record. Write it down, we will need it in the next step.
    f. Example - you need the highlighted part: https://org45b2a0fb.crm4.dynamics.com/main.aspx?appid=6b2f536c-3e99-ec11-b400-000d3abe520e&pagetype=entityrecord&etn=rpakit_rpamodule&id=**99cc4196-4799-ec11-b400-000d3abf3b6c**
9. Now let's switch to Power Automate on the web. Create your own *[PREFIX] Run test cases* **instant cloud** flow as part of the new solution:
    a. Your flow's trigger will be *Manually trigger a flow*. Add one yes/no parameter named *Scheduled*:
    b. 

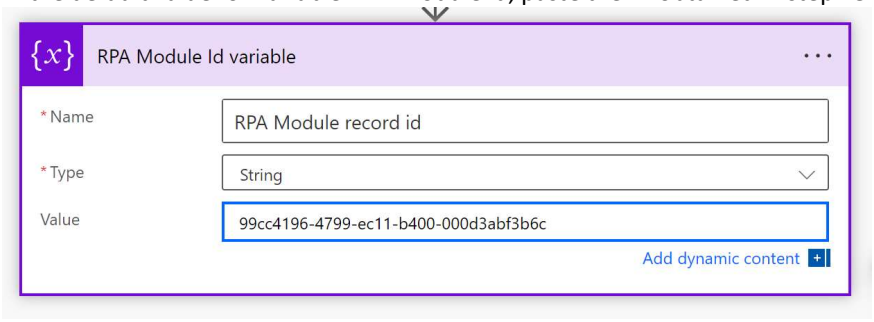
    c. Once you added the trigger, copy all the steps from *[CMS] Run test cases flow* using clipboard:

d. 

e. And then paste them into your new flow: (opened in a separate browser tab)

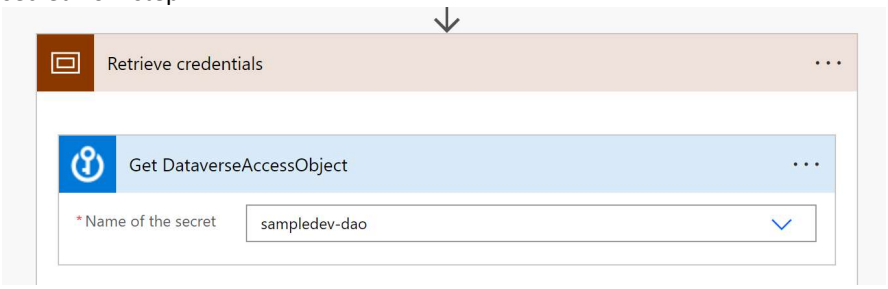
f. 

g. In the default value for variable RPA Module Id, paste the ID obtained in step #8:

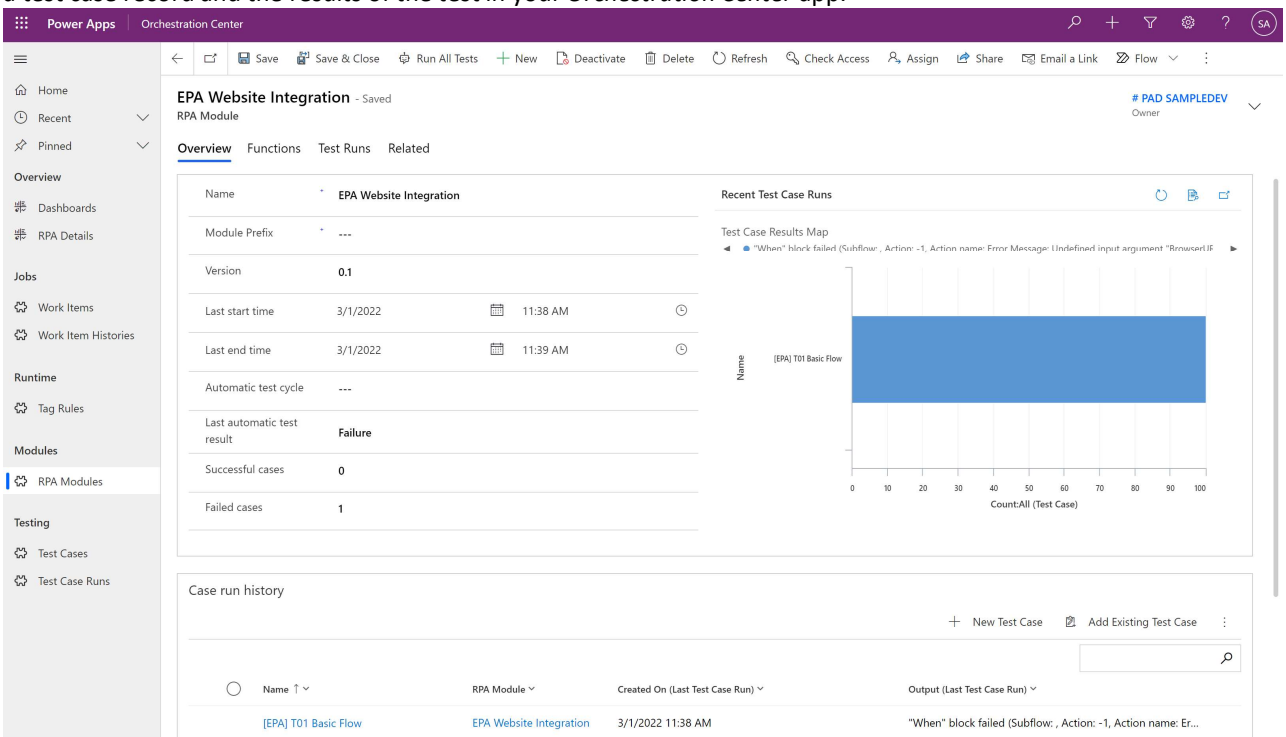
h. 

i. When copying the *Run a flow built with Power Automate for desktop* action, make sure you link it to your new desktop flow *([PREFIX] Test cases).*

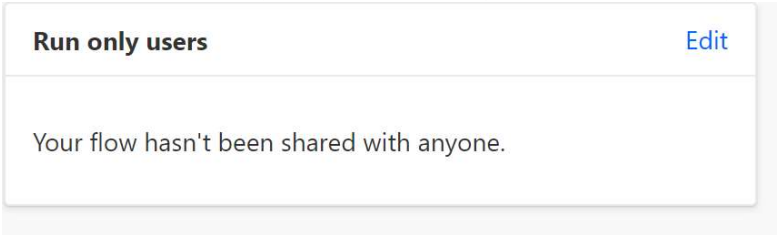
j. In the *Get DataverseAccessObject* action, adjust the parameter to match your actual secret from step #7:

k. 

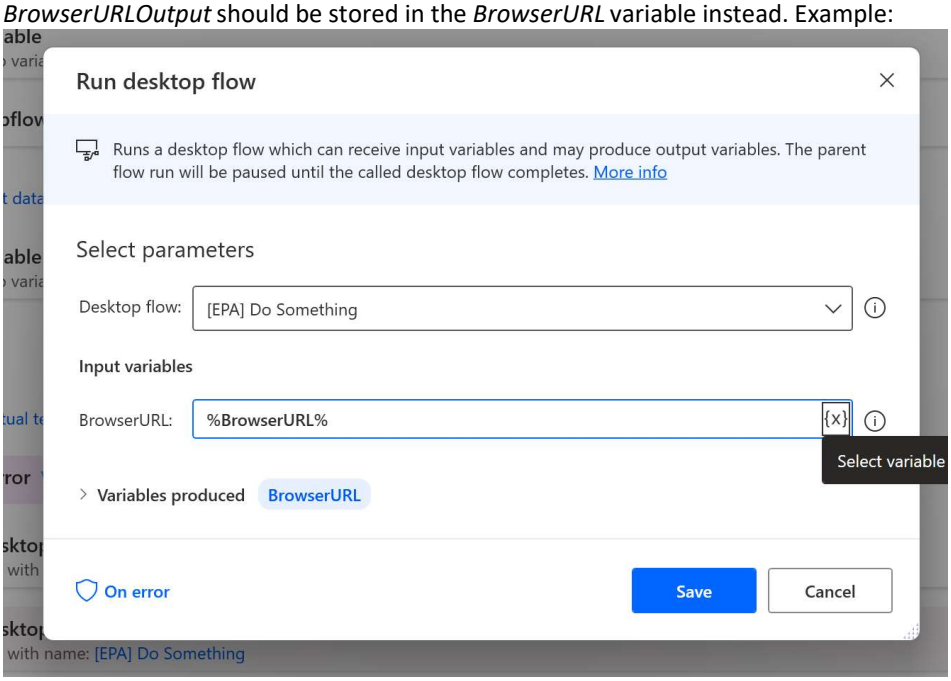
l. You might want to run this flow for testing. If everything runs smoothly, you should see a test case record and the results of the test in your *Orchestration Center* app:

m. 

10. In a similar way, copy *[SAMPLE] Ad-hoc test run* to *[PREFIX] Ad-hoc test run*:

a. You will need to create the trigger manually, as triggers cannot be copied through clipboard.
b. Make sure you open the advanced options in the trigger, and copy also the values for select columns and filter rows.
c. Update the search criteria in *Filter rows* to **match your module name** (see #4 above).
d. Once the flow is saved, go back from edit page to flow overview. Open *Run only users*:
e. 

f. For every connection that says *Provided by run-only user*, switch to the other option which should be a specific connection for your username. Accept the warning message.
g. Add the newly created flow to your solution.

11. Once more, copy a flow one step at a time - from *[SAMPLE] Scheduled test case run* to *[PREFIX] Scheduled test case run*:
    a. Make sure you link the child flow action to the proper child flow - it should be your new *[PREFIX] Run test cases (child)*.
    b. Add the newly created flow to your solution.
12. Create your module's Init, Close and all other desktop flows as required. Do not rename the [SAMPLE] ones, instead open Power Automate Desktop and make copies with new names (with your [PREFIX]). Make sure you add all your desktop flows to the new solution.
13. If you intend to use the T01 test case example, please adjust the function calls for Init, Do Something and Close to their respective versions for your [PREFIX].
    a. Whenever you change the *Run desktop flow* action in any flow (i.e., in the test case), make sure that you update the mapping of its output variable. The output parameter *BrowserURLOutput* should be stored in the *BrowserURL* variable instead. Example:
    b. 

    c. The BrowserURL variable should be always in both the input data as well as output mapping, to ensure that the parent flow is always aware of what URL is the web browser pointing at.
    d. See Browser handle for more information on browser URL handling.
14. When done with the module functions, export the module into a managed solution to bring it to your target assembly environment.

Dataverse Access Object

Monday, February 28, 2022 6:19 PM

The **Test cases** flow template uses Dataverse to record outcomes of the tests that have been carried out. The *Dataverse Access Object* is a connection string that you need to pass as a parameter to the **Test cases** desktop flow.

Azure AD configuration

The following steps need to be carried out to allow desktop flows to access Dataverse:

1) In *Azure Portal*, configure *Azure Active Directory* for your tenant (for small projects and trial tenants, this is a free service).
2) In *Azure Active Directory*, go to *App registrations* and create a registration for Power Automate Desktop.
3) Since ideally you should have separate users for each environment, you might want to include your environment name in the name of the app registration (i.e. "PAD user for DEV")
4) Once your app registration is saved, note the following values - you will need them later:
    a. **OAuth 2.0 token endpoint (v2)** - click on "Endpoints" to obtain
    b. **Application (client) ID**
    c. **Client secret** - click on "Add a certificate or secret" and then "New client secret" to obtain. You will need to record it immediately after creation, as it will not be accessible later.

The final Dataverse Access Object is a JSON-encoded array of strings, where index 0 is the token endpoint, 1 is client ID, 2 is client secret and 3 is your organization URL with https:// prefix (no trailing '/' needed).

Sample:
['https://login.microsoftonline.com/89ca8979-abcd-9cde-ab31-0e14fcc21aaa/oauth2/v2.0/token',
'0e286789-b506-1234-1234-dd2deed57809',
'oiuewf98p23oirewf98opijlk239r8pweoijkc',
'https://yourorganization.crm4.dynamics.com']

Creating an application user

Next, we need to create an application user and assign a security role:

1) Open Power Platform Admin Center (https://aka.ms/ppac)
2) Select your environment, click *Settings*, then *Users + Permissions* and finally *Application users*.
3) Click *New app user*, then select the application you created in previous step
4) Select business unit (top level) and security role (*Power Automate Desktop user for test cases*)

Storing

Recommendations for handling your Dataverse Access Object:

- The client secret you created will expire by default in 6 months. Make sure you have a process in place for timely refresh of your client secrets.
- Avoid reusing the same application/client IDs and secrets for multiple environments - you should have a separate string for each environment
- The JSON value is effectively a connection string that grants access to your environment. Store it in a proper credentials vault, i.e. Azure Key Vault.
- Never set any default values for the *DataverseAccessObject* input parameter
- Never assign admin permissions to the newly created app user - use the provided security role for this.

Module self-testing capability

Tuesday, March 22, 2022 5:23 PM

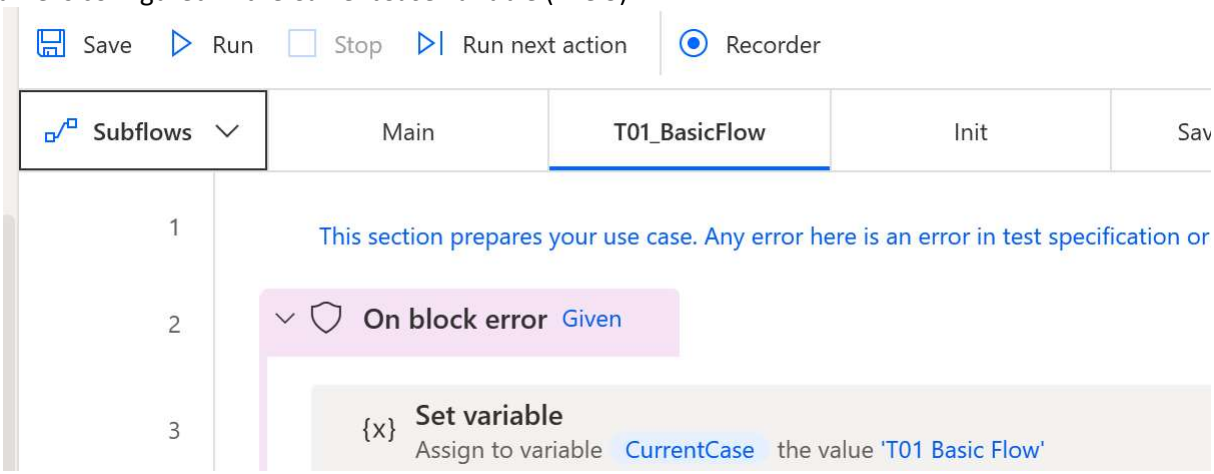
Customizing the template

Monday, February 28, 2022 5:52 PM

Required steps prior to using the template

You must do the following:

- TODO Take the Txx_Template subflow as a starting point

- Each test case must have a unique name
- The name is configured in the *CurrentCase* variable (line 3):
  - 

Test case naming

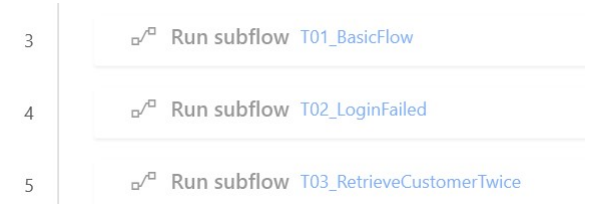
Wednesday, January 12, 2022 10:20 AM

Txx_CaseName

Txx - two digit case number. It is not required that there would be no gaps - feel free to use multiplies of 10 for signifying areas. Example:

T1x cases are for setting credit requested parameter.

CaseName - name of the action or behavior being tested.



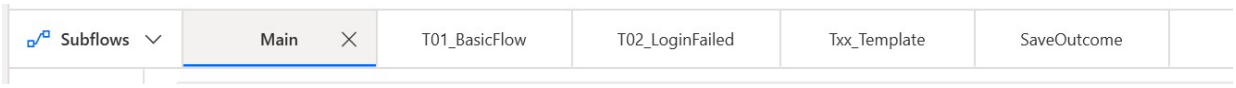
Txx_CaseName_variant

If there are multiple variants of the function behavior, test them in cases grouped with common case name. Example:

These are variants, not *versions*. All variants are meant to be used, they're just different things being tested.

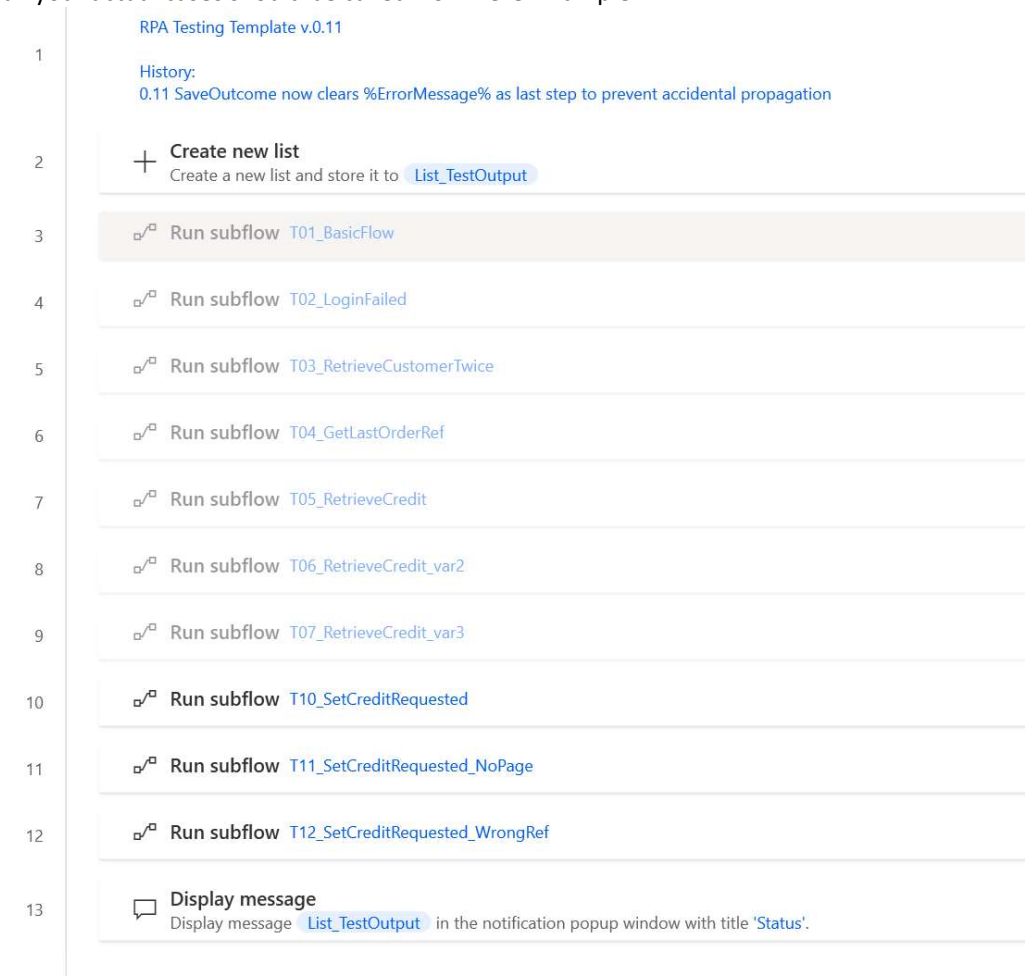
Subflow structure

Wednesday, January 12, 2022 10:06 AM



Main

all your actual cases should be called from here. Example:



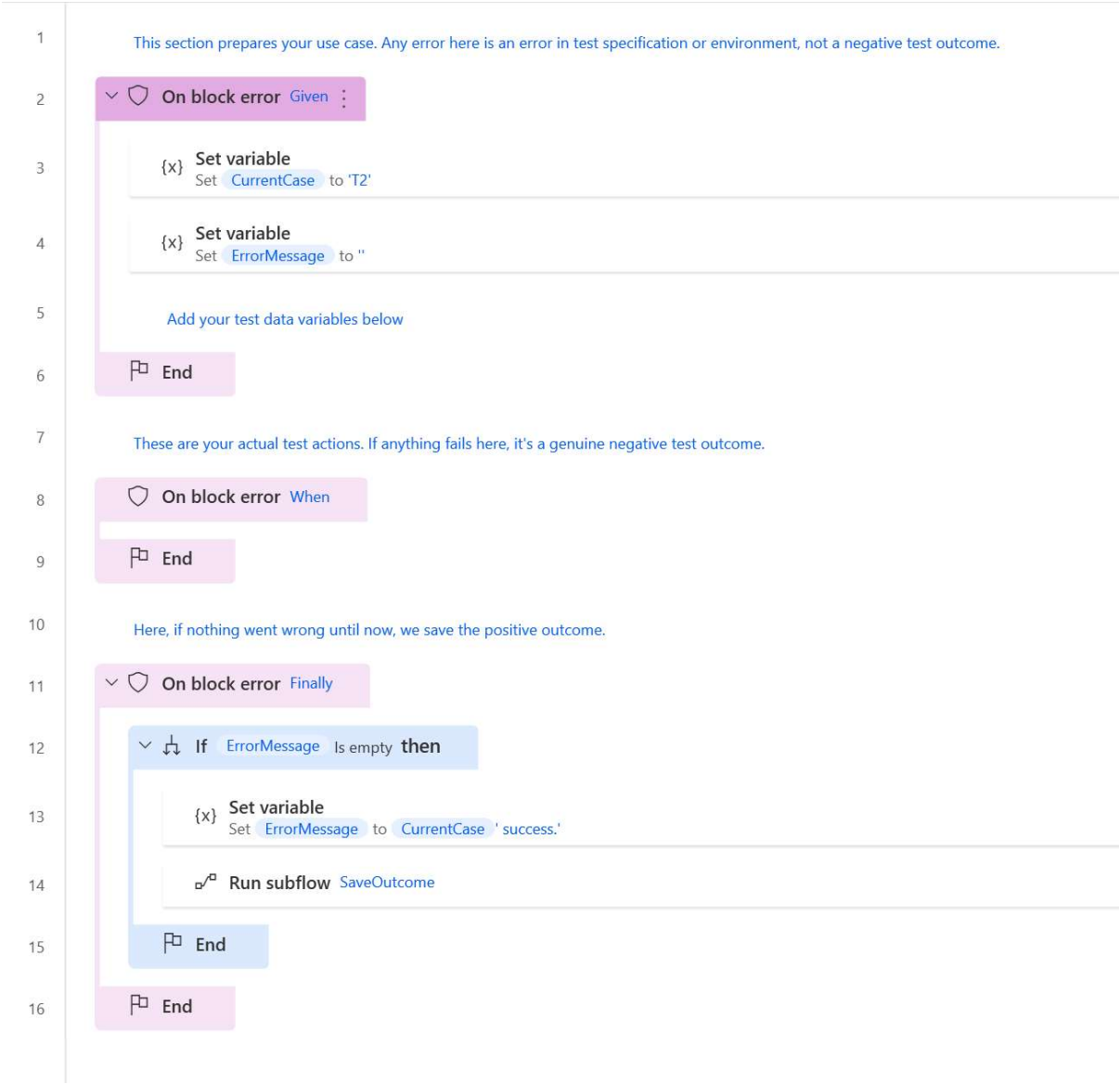
You may disable cases you don't need while you're working on the scenarios. Cases should not depend on each other - every case must contain its own starting conditions and must clean up (i.e. close browser).

T00_CaseName

These are your cases. They should append "Txx success" to List_TestOutput variable on success, or error description on failure. Use SaveOutcome for this.

Txx_Template

A template for building more test cases. Example:



SaveOutcome

Given-When-Finally

Wednesday, February 23, 2022 10:15 AM

Modification of standard pattern Given-When-Then, adapted to the Power Automate Desktop (*then* is a reserved keyword)

Reference:
What is "Given - When - Then"? | Agile Alliance

? *** describe finally / invert block

Data driven testing

Monday, February 28, 2022 6:20 PM

*** todo: you can create a dataflow that will load your test records into Work Items and compare outputs.

*** todo: data driven testing add-on module (addl. field in Work items, flow to compare, dashboard for display)

Delivery Project Flow

Monday, January 24, 2022 3:53 PM

Delivery Project Approach

The use of modules enables the following project delivery approach.

Step 1: Design

Decide on what modules you need, what functions need to be present, and what will be the input & output for each one of these functions. Assign team members to be responsible for each of the modules, and the role of module integrator - the person who creates the ultimate solution leveraging available modules.

Step 2: Build stubs

Team members responsible for the modules build each one of them and create the desired functions. At this stage, functions are built as stubs - they have the correct input & output parameters, and they return valid output data when called. At this stage, the function output is fixed and no RPA work is done.

Step 3: Build top level process and modules

The team member assigned the role of "RPA integrator" imports module stubs into target DEV environment and begins creating the top level process, while module developers begin creating their actual functions. Module developers create test cases in parallel to the functions, and the integrator creates relevant input data and reference output for testing the final flow.

Step 4: Integration

Individual modules, which are progressively more functional in their subsequent versions, are provided to the RPA Integrator for integration testing.

Step 5: Testing (continuous)

RPA Integrator schedules an automatic test that loads test data into the work item queue, waits for completion and compares output results. Module developers schedule automatic test process of their respective functions. Every time an error is discovered with any of the functions that was not detected earlier through test cases, a test case should be create to target that specific scenario.

Environment structure

Tuesday, March 1, 2022 9:27 AM

For a small project you will need the following environments:

- Module Dev - here you will be building individual modules. If you have many people in the module developer role, they can share this environment.
- Assembly Dev - here you will be importing your modules as managed solutions and building the integration flow(s) on top of them.
- Test, Production - as always.

For a larger project, you might consider splitting shared *Module Dev* into separate environments for your developers. This way you they will be able to backup & restore if needed, without disturbing others.