WITHYOU
WITHME

RPA Blue Prism Developer Course

Written Guide

Contents

Intent

This document is designed to augment your online learning experience and provide you with the opportunity to review and revise the content that has been discussed during the individual lessons of the course

Feedback

Please contact your WithYouWithMe RPA instructor if you have feedback on this document or any of your WYWM courseware.

Inputs and Outputs

Overview

Hi and welcome to the lesson where we will be going through Inputs and Outputs

In this lesson we'll introduce the concept of inputs and outputs to processes or objects and we'll use what we've learnt so far to incorporate input and outputs into some of the processes we've already built.

As usual we'll switch between the slides and Blue Prism. To get the most out of the video, make sure you follow along in Blue Prism and pause the lesson if you need to.

Inputs and Outputs

Just as a business object provides a process the ability to interact with an application, it is the inputs and outputs that move data back and forth between our processes and our objects.

If you recall back to our order system object, we had two data items within our object, the values of which we wrote to our application. The only issue with this is that at the moment we're storing these data items on the object layer or within our object.

This isn't ideal because it means that the object can only be used by one user with one set of login credentials. In the event that our password changes or a different user wants to use this object, we need to gig down into the object layer each time we want to make adjustments.

In the real world, we'll likely use this login object across many different processes and we should therefore design our objects so that they can be used by many different processes and call on different login credentials each time.

For this reason, we want to have variables or dynamic values stored on the process layer, And by this we mean values that are likely to change should be stored on the highest layer possible.

Let's jump across to Blue Prism and in this exercise we're going to make some adjustments to our order system business object and we'll then build a process that sends some data from our process to our objects before it's inputted to our application.

Inputs

First let's open our Order system business object and go to the Enter Details Action Tab. We can see here that our staff number and password values are stored within our object. As we said, this is all well and good if bp is the only staff member that's going to use this action, but that's not very scalable and we want several bots with a range of details to be able to call on this object, and that means that we're going to need to store the Staff Number and Password details within our process and we're going to want the values that are stored on our process layer to be sent to these data items each time the object in called upon.

To set this up we first need to go into the properties of our start stage. We can leave the name start but we're going to need to add a couple of inputs. Let's add a row and we'll call the first input staff number, it's a text data type, and we drag in the data item that we want our staff number to be sent to from our process. It's also good to add a description for extra clarity so we'll put a brief description of what this input is.

Now let's do the same thing for password by adding another row, giving it the name password, you'll recall it's a password data type and we want our password value from our process to be stored in the password data item.

Great, this means that when this action is called within a process, inputs will be required at the process layer and we'll see shortly that Blue Prism will remind us of this when we set up our process.

Now click ok and we can remove the initial values from both of our data items because this data is now being sent from the process layer rather than being stored with our object on the object layer.

Ok, we're done here for now so let's make sure this object is published so it's available to our processes, it is, so we save our object and exit. It's really important to make sure that when we're manipulating processes and objects, we save and close both the process and the object that will be effected by our changes. This avoids any caching issues and allows Blue Prism to work from a clean slate.

Now, let's make a process that sends some values to our order system object. So we create a new process called Inputs Process.

Let's try and login to order system, how do we do this? Well, we get three action stages that launch, enter our details and then click sign in. So let's do that now. We'll drop them in, make them all a bit bigger and set them up.

The first one's easy, we'll call it launch Order System and we'll select our order system object and launch action. Next is our Enter details action. We'll call this one enter details, select our object and our action. This time you can see that as soon as we selected the Enter Details action two inputs appeared, do you recognise them from before? This is Blue Prism telling us that if we're going to use this action we need to give the object some more information.

Now we could just type our staff number and password in here manually but it's best practice to call on values using data items so we'll set that up now.

We click ok to get out of here. We then drag two data items across, make the first one a bit bigger, call it Staff Number, it'll be a text data type, and our staff number is bp so we put that in the initial value field. Now click ok.

Next we make our second data item a bit bigger, call it Password, make it a password data type and set our initial value as password. Now we click ok.

Now that we've set our data items up we can give our Enter Details action stage some inputs. So let's go back in, and drag our staff number data item into the staff number input and our password data item into our password input. Now we validate them both. Now we haven't set any output requirements up for this action but we could check these by selecting the outputs tab.

If we did have an output from this action it would be visible here and we would need to have a place to store the value being sent from our object to our process in the same way we just described. But we're happy with what we've setup so far so lets click ok.

Now all we need to do is click Sign In, so we call our last action stage sign in, select our object and select click signin button. No inputs or outputs this time so we click ok.

Now let's link it up, reset, and step through our process so we can see what's happening.

First we launch the order system application the same as before. Now we go to enter our details but as you can see, the data items are blank, but as soon as move off our start stage the values are sent from our process and are populated in our data items. Our write stage then writes them to our application and we click the sign in button as we did before.

Fantastic, now let's run it in real time and put our process and the inputs to our object through their paces.

Outputs

Great, that worked really well. It's important to remember that outputs are exactly the same as inputs and if there was some reading of an application done on the object layer, we could store whatever was read in a data item on the object layer and then send it back to our process via the properties of our end stage. We can see this if we go into the properties of our end stage and have a quick look. If we added an output here this would then generate the requirement for an output when we select the corresponding action, and we'd then set up a data item on the process layer to accommodate this value when it was produced in the same way we did for inputs.

Exception Fundamentals

Overview

Hi and welcome to the lesson where we will go through the fundamentals of Exceptions

In this lesson we'll introduce the concept of exceptions within our process logic and the Blue Prism functionality we can employ to effectively deal with these exceptions.

As usual we'll switch between the slides and Blue Prism. To get the most out of the video, make sure you follow along in Blue Prism and pause the lesson if you need to.

Exceptions

In the examples and exercises we've used in the course so far, we've only looked at ideal scenarios and in solution design we refer to process flow that takes the preferred path as following the happy path. We have not yet considered what to do when things don't go as we've planned or follow what we refer to as the unhappy path. Given the dynamic nature of business processes and the applications they interact with, the ability to be able to identify and manage issues on the unhappy path through robust automation design is a critical skill that we need to demonstrate as RPA Developers in industry.

Thorough testing of a solution prior to deployment will reveal many of these issues and based on what's identified, adjustments to the design of our solution can then be made to increase our solution's robustness. That said, Exception Handling is not about making sure the Developer's code is correct. While we must also test the solution that's been developed, exception handling is principally focused on successfully managing the range of variables that our bot will need to deal with when exposed to a real work environment.

These kinds of issues or errors are known as exceptions and the logic we incorporate within our process to deal with these issues is known as exception handling.

If exceptions occur while a process is running and they are not handled properly, the process will generate an error and will not be able to progress. This is a big problem if we consider that we might be processing 1000 lines of data and say an error occurred on the 5th line. In this case we would prefer that Blue Prism identify this exception, let us know, and continue to process the remaining data. This concept is the essence of why we employ exception handling.

Rather than failing in the event of an exception, processes and objects should be designed so that they can manage exceptions and move on. Although not all exceptions are recoverable and in some cases it won't make sense to even attempt a recovery, the ability to handle exceptions at least provides us with the opportunity to decide what to do when something goes wrong.

Exception handling is done with a set of stages that we haven't used before so let’s have a quick look at them now.

Recover and Resume Stages

Recover and resume stages are used to salvage and move on from an exception. The use of these stages also introduces a new concept to the design of our process logic where an alternate path is created.

Now let’s create some exception logic across in Blue Prism, we’ll create an object in the object studio called Exception Object.

What we want to do is have a calculation stage divide two values and provide us with an answer. So we’ll set that up now.

Now we’ll reset our object and run it. Great, that’s working as we hoped.

Now we’ll set a trap for blue Prism by having it divide by 0, which I’m expecting will induce an exception.

So if we change the initial value of our second number data item to zero, click okay, reset and run our process we’ll see if this induced an exception.

There you go, we got an error message because Blue Prism noticed that we were trying to divide by 0, which we obviously can’t do. So we’ll click ok on this error and we’ll build in some basic exception handling logic.

We’ll use a recover and a resume stage to deal with this exception.

So we’ll drag across a recover stage, a resume stage and an end stage. We’ll make them all a bit bigger and we’ll link them up.

Now if we reset and run this again, we can see the error still occurred, but this time it didn’t fail and give us the error message in a dialogue window, our processes went straight to the recover stage, it was then resumed and because we didn’t have anything else in our process logic after resume it proceeded to end.

Our process was sent to our recover stage because a recover stage will “catch” any errors that occur and send them on an alternative path. For this reason our recover stage isn’t physically connected to anything upstream for it to be used by our process.

This gives us the opportunity to create some recovery logic in the event we have an error within our process. In this basic sequence we simply took our process to a different end stage.

Recovery Mode

When an exception is “caught”, our process or object is said to be in recovery mode, which means than an exception is live and any error in our process while an exception is still live will cause our process to fail to a stage that is unrecoverable. Passing through our resume stage diffuses the live exception and enables us to come out of recovery mode and continue with a normal process flow. For this reason we limit the processing that’s done in recovery mode to only extremely reliable functions to reduce the likelihood of an unrecoverable failure before we reach our resume stage.

It’s important to note that the resume stage doesn’t fix anything. This is the responsibility of the designer and developer. You can think of the recover and resume stages as getting our process back on its feet, after we’ve recovered from an exception, we then need to deal with it after the resume stage.

Generating Exceptions

In some cases, it may be advantageous to generate exceptions deliberately. In the example here, if diving by zero was the reason that a real life case couldn’t be worked, we’d prefer to mark this with a message that we can look at later, or when we come in to work in the event our bot is running over night.

This message might help us better understand what went wrong with our process and may assist in the identification of rework that may be required as a result of the error.

So in this case our error has occurred because our second number is 0. What I want to do now is capture this error and produce or “throw” an exception before it causes an error.

So above our calculation stage let’s uses a decision stage to assess whether our second number is 0, and if it is, let’s have our process do what we call “throw an exception”.

So first let’s delete the link, give ourselves a bit more room and grab a decision stage, we’ll make it a bit bigger and we’ll call it “Is Second Number 0?”. In our expression field we’ll drag across our second number data item and we’ll ask whether it is equal to zero. Now we validate and press ok.

Now if our second number isn’t zero or the response to our decision is false we’re fine to proceed with the rest of our process, but if our second number is zero giving us a true result we want our process to throw an exception.

So we drag an exception stage across, make it a bit bigger and link it up. In simple terms, Exception stages are used to label errors. As we mentioned before, Recover/Resume are used to initiate logic to deal with errors.

Now let’s run the action and see what happens, so we reset and click go. You can see in this case we didn’t wait for an error to occur, we caught the error with our logic before it occurred, sent it to our exception stage, which then sent us to our recover and resume stages.

Now this is great, but imagine if this bot was running over night and we came in the next morning to find our process at the end stage of an exception path. If this were a more complex process we wouldn't have any idea what had happened to cause this error.

For this reason we can add messages to our exception stages that let us know what induced the exception.

We can double click the exception stage and because this exception was thrown based on a value being zero we'll make a selection or type some text to reflect this. I'll type "Trying to divide by zero". Note that exception type doesn't require quotation marks but if I go to exception detail it will. Note the calculator icon to the right of this field. This is a good indicator that we need to denote our text with inverted commas. In this case our exception detail is "Second Number = 0". Then we click ok.

Now finally, let's extract the message from our exception by adding a calculation stage between our recover and resume stages. We'll call on an exception function in the function section of the window, open exceptions and select exception details. We can either drag it in or click paste.

Now we'll create a data item using our store in field and call it Exception Details. Let's click the blue button, validate and click ok.

We'll make that all a bit bigger and connect everything up. Then we'll be ready to test our exception handling logic. I'm expecting that our decision stage will identify that our second number is equal to 0 so it will throw an exception by proceeding to our exception stage. This will then move to our recover stage and in recovery mode our calculation stage will extract our exception message and send it to our data item, before resuming and processing to the end stage.

Lets see how close we got by resetting and pressing go.

Great. Thats exactly what we wanted to see. We now know exactly what threw our exception.

So lets publish and save this object and we'll use it again in our next lesson.

Managing Exceptions

Overview

Hi and welcome to the lesson where we will go through the management of Exceptions

In this lesson we'll build on what we've already learnt about exceptions and employ some exception handling techniques in a process that will call on the exception object that we used in the previous lesson. In this lesson it's also important to reference the Blue Prism Exception Handling training guide which can be found in the materials tab of this lesson.

As usual we'll switch between the slides and Blue Prism. To get the most out of the video, make sure you follow along in Blue Prism and pause the lesson if you need to.

Let's jump straight into Blue Prism this time and we'll our Exception Object that we created in our last lesson.

Managing Exceptions

Now in this object, you'll recall that we recognised that an exception existed if we tried to divide by zero and so we created some logic to deal with this exception. But what if we get an error that we didn't foresee or what we call an unhandled exception?

If not dealt with, exceptions can escape out of business objects and we ideally want all our exception management to be done on the process layer, so lets set that up now.

Here you can see the recovery logic that is part of this business object. Let's cut all this out leaving only the exception stage behind. We now save this making sure that our page is published and go back to the process studio where we create a new process called exception process.

We'll drag across an action stage, we'll call it Exception and set it up to call on the object and action we just created. Now we click ok and link everything up.

Now once we run this process, the action one stage will go into our object, find that our second number is still zero and produce an error message because we just cut out our recovery logic on the object layer. Let's reset and give it a go.

As you can see we have the exception type and details that we added to our exception stage in the previous lesson. So we'll click ok.

Now in this case we've deliberately chosen not to deal with the exception within the business object and in doing so, we have allowed the exception to escape upwards to the process layer. So let's paste the recovery logic we just cut from our object and place it to the right our our current process.

Now let's reset and run this process again. This time we can see that the exception still occurs on the object layer, but it is sent up to be managed on our process layer and instead of getting the error message, our recover stage caught the exception, we extract the exception message using our calculation stage and we then resume our process taking it out of recovery mode.

Escaping of exceptions works the same way in processes and objects, however we want to do Exception HANDLING at the Process layer because exception logic (ie. what you do with an error) is driven by business requirements, hence it belongs in the Process layer.
However, Exception LABELLING (ie. using Exception stages) is done on either layer.

Exception Bubbling

The way in which an exception moves upwards through the layers of a solution is known as exception bubbelling. As exception will bubble upwards until it is handled and if it is not handled it will eventually bubble up to the main page on our highest process layer and cause it to fail.

We can use bubbelling to our advantage as we don't need to catch an exception straight away, rather, we can let it bubble up and handle it at a higher level, maintaining good process visibility.

Deciding when and where to handle exceptions is a key part of solution design that we will return to in later lessons.

Exception Blocks

We can also add a greater deal of control to our exception handling through the use of exception blocks. A block is used to isolate an area of a diagram that a recovery stage is responsible for. Without a block, a recover stage will handle or catch any exception on that page. When a recovery stage sits inside a block, it will only catch exceptions in that block, and ignore all others.

Blocks therefore allow us to use more than one recovery stage on the same page.

Let's jump back across to Blue Prism and start using some exception blocks. We'll start by using the exceptions object, so let's open that backup now.

Now we're back here, you'll recall that we cut out our recovery logic and put it up on our process layer. So if we run this process you can see that we get an error message and our object won't recover on its own.

So let's add some basic recovery logic back in using a recovery stage, a resume stage and an end stage. Now we'll make them a bit bigger and link them up.

Now let's reset and run this process again to remind ourselves what happens.

Great, now let's use the block by selecting it and clicking and dragging. In this case we'll only include our exception one stage.

Now, if we reset and press go let's see what happens. Well in this case there were no changes because there wasn't any recovery logic within the block, and without a recover stage a block has now effect, so the exception found our recover stage as normal which caught it and directed us toward the recovery path.

Now let's duplicate our recover resume and end stage by copying and pasting them. At the moment, if we were to run this object, there would be a conflict between which recover stage our exception would through to. I'll demonstrate this by running our process again now.

Yep, As we suspected there was a conflict and while our exception was still captured by a recover stage, we want to avoid this situation by using our blocks correctly. We can do this by is clicking on the writing of the block to highlight it and extend it to encompass our first set of recovery logic.

Now let's reset and run again to make sure what I've told you about blocks is correct.

Great, you would have seen that the recovery stage within the block was used and our exception was sent on this path. As we've said, this is because this exception stage is inside the same block as our recovery stage and so the recovery stage assumes responsibility for recovering from this exception.

Now let's try something different. Let's change the shape of our block again so that only our recover logic is in the block and our exception stage and our second set of recovery logic is outside.

If we reset and press go what do you think will happen?

We'll in this circumstance, the block has effectively isolated our first set of recovery logic and so our exception will ignore this recovery stage and it will be caught by our second set of logic that is also outside our block.

Let's give it a test.

Great, there we go. Now continue to experiment around on this page to get a really good understanding of exceptions and blocks as they are a critical aspect of RPA design and development in inductsy.

Now, we've seen how Blue Prism manages exceptions, but how is this used in the real world. Let's have a quick look at an example and understand how what we've just learnt can be applied to real processes.

We can see in this process that there are two main types of exception handling, one where we allow the process to resume, and the other where a recovery stage aborts the process.

If we look at block 1, we can see that in the event that our start up page produces an error it will be caught by our recover 1 stage but that will immediately abort our process. This is because we don't want our process to continue if we can't start our application.

However if we look at block 2, we can see that if an exception occurs when we run our account status or apply credit pages it is caught by or recover 2 recovery stage and it is marked as an exception before it resumes and continues on with the rest of the process. You can also see from the notes used in each path that the recovery logic is what we refer to as the unhappy path. This use of notes helps when multiple people are using objects and processes, and is therefore considered best practice.

Retrys

Overview

Hi and welcome to the lesson where we will discuss the concept of retrys.

In this lesson we'll build on what we've already learnt about exceptions and circular paths to incorporate some retry logic that, in the event of an exception will, retry an action or task.

As usual we'll switch between the slides and Blue Prism. To get the most out of the video, make sure you follow along in Blue Prism and pause the lesson if you need to.

Retrying an Action

So far when we've considered exception handling, an exception has been thrown and we've then recovered that exception in recovery mode and resumed our process before continuing on with the remaining logic within our process.

In most cases we will want to retry an action that's induced an error within our process before we continue on down the unhappy path. Put simply, the concept of retrying simply means to recover an exception and then steer the flow back into the main part of our process flow in the hope that the problem will be alleviated by another attempt.

In this case the decision whether to retry is governed by the exception details and the number of retries that have been attempted. To avoid retrying too many times we can employ data items in the same we we did during our circular paths exercise to count the number of retrys that we attempt, as well as the maximum number of retries we'll try before rethrowing the original exception in the event the error still exists.

This logic ensures that an action within our process will only be retried if the retry count has not been reached.

In the event that our process does retry the action that initially threw the exception, we will need to utilise a resume state as we've done previously, but we may also need to incorporate some restart logic that will get our process or object back to the same configuration that we saw when the original exception occurred.

Let's jump across to Blue Prism and incorporate this logic into our Exception process

Retrys and Circular Paths

So let’s give ourselves a bit more room and build the same logic that we saw in the slides using two data items, the first that counts our number of retrys and the second that dictates our retry limit.

We’ll also need a calculation stage that adds to the value of retry count each time it cycles through and we’ll need a decisions stage that assesses whether retry count is less then or equal to retry limit and based on the outcome of that assessment our process will either loop around again, or rethrow the exception. We’ll also need to link up our recover and resume stages and we’ll need another exception stage in the event that we reach our retry limit.

You know how to to do this so we’ll run through this in quick time to speed things up.

Now we’ve set that up and it’s linked back up we reset and run our process and have a look at what happens.

You can see that we’ve still identified that we’re dividing by 0 which produces an exception, but this time we’ve retried our process three times before we’ve rethrown the exception with this exception stage.

The decision to retry will typically be determined by the automation design team but as a general rule we would look to retry System or Internal exceptions but we wouldn’t retry either business exceptions or try once exceptions.

Common Retry Errors

Before we go, there are two key types of retrys to be aware of, infinite retry loops and nested rertys.

An infinite retry loop is when exception retry logic does not contain any mechanism to break out of the loop. Normally a counter is used to limit the number of retries that will be performed as we saw, but without this, retrying could go on for ever and we call this an infinite retry loop.

We also have nested retrys. This is when one sequence of retry logic sits within another and the numbers of retries in each multiply together. Retry logic is often limited to 3 loops but if multiple sequences have been inadvertently nested together there could be 9 (3x3) or even 27 (3x3x3) loops.

This is usually because the sequences exist on different pages or in different diagrams. This physical separation often makes the whole arrangement harder to visualise, particularly if you are not familiar with all of the diagrams.