UiPath Automation

Walkthrough

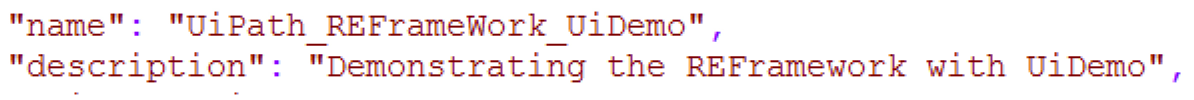
Walkthrough – UiDemo

Walkthrough – UiDemo

Strategy: As we need multiple robots for parallel processing, we need to split the workload in a reliable way. The best approach is to use Orchestrator Queues. Let us create two automation projects. Let's name them Dispatcher and Performer. The former reads the Excel file and uploads transactions to the queue, and the latter processes the uploaded queue items.

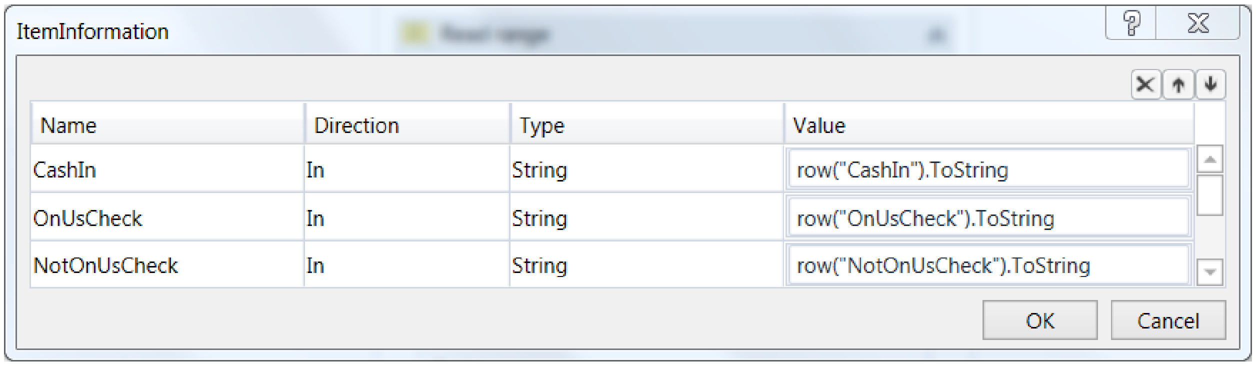
Workflow	Input	Output
Dispatcher	Excel file	Queue Items
Performer	Queue Items	Processed Transactions

- We start with the REFramework template, as it suits the purpose of the Performer very well. The very first things we need to do are firstly to rename the folder "ReFramework_UiDemo", and secondly, to edit the Project.json file in the root folder. Let's change the name to "UiPath_REFrameWork_UiDemo", and the Description to "Demonstrating the REFramework with UiDemo". It should look like this:



```
"name": "UiPath_REFrameWork_UiDemo",
"description": "Demonstrating the REFramework with UiDemo",
```

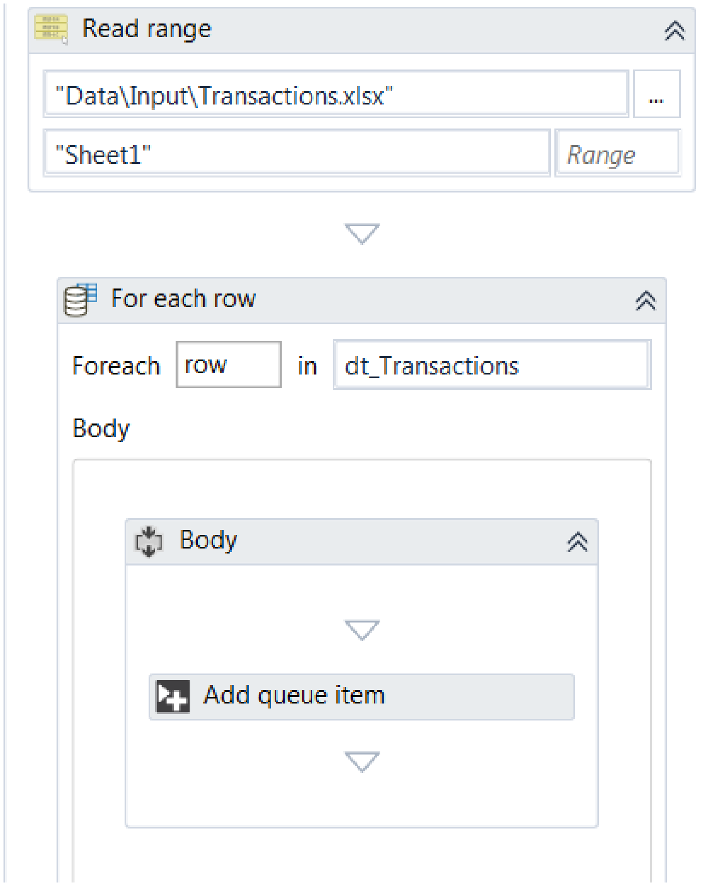
- The Dispatcher is very simple in this case – the robot should read the Excel file and add a Queue Item for each row. That would normally be a separate process that is scheduled to run before the Performer, but for simplicity and ease of testing, we will include it in the Performer.
    - Let's create a new sequence and name it **Dispatcher – UploadQueue**.
    - We need a **Read Range** activity to extract the information in the Excel file and store the output in a DataTable variable.
    - Each row is a transaction item, so next we need a **For Each Row** activity. Use an **Add Queue Item** activity in the **Body** section.
    - Edit the ItemInformation property by adding three arguments: CashIn, OnUsCheck and NotOnUsCheck. It should look like this:



ItemInformation

Name	Direction	Type	Value
CashIn	In	String	row("CashIn").ToString
OnUsCheck	In	String	row("OnUsCheck").ToString
NotOnUsCheck	In	String	row("NotOnUsCheck").ToString

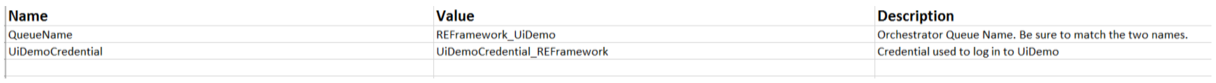
- Fill in the **QueueName** field, and then create a queue with same name in Orchestrator. To retry failed transactions, simply set the **RetryNumber** property to 2 when creating the queue.
- The Dispatcher sequence should look like this:



- Let's run this sequence and then check the queue for new items.

All right! The Dispatcher process has been built. Now let's move on to the **Performer**!

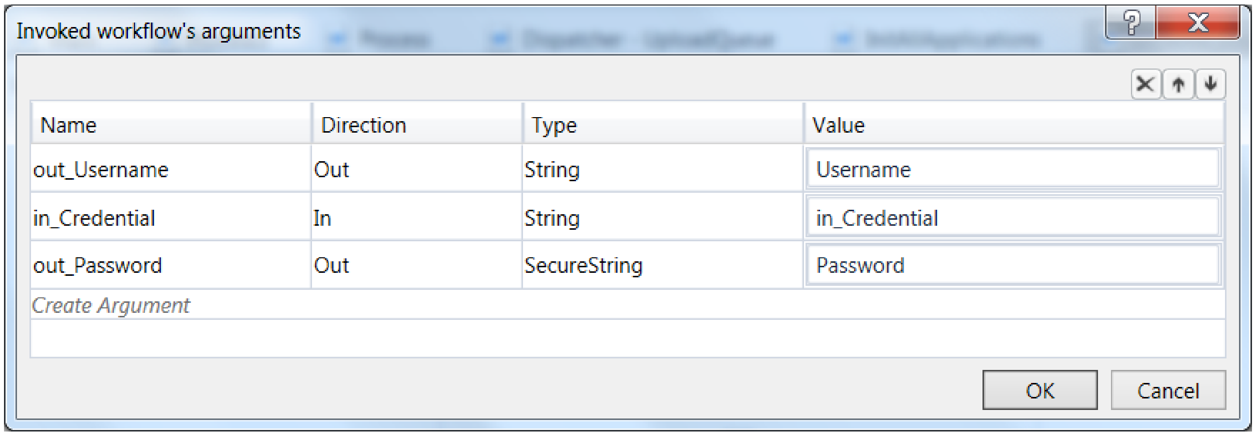
- The input for the Performer is an Orchestrator queue, which means there are very small changes we need to make in the Framework. We start with the Config file and match the **QueueName** to the one we used earlier. We need one more setting – a credential name for UiDemo. The value of **MaxRetryNumber** in the Constants sheet should be 0, as we are using the retry feature of the queue in Orchestrator. The Settings sheet should look like this:



Name	Value	Description
QueueName	REFramework_UiDemo	Orchestrator Queue Name. Be sure to match the two names.
UiDemoCredential	UiDemoCredential_REFramework	Credential used to log in to UiDemo

- Save and close the Config file.
- The next step is to initialize the process.
    - First, we need to log in to UiDemo before being able to perform repetitive transactions.
    - We should always think in terms of reusable components - the process of logging in definitely falls under that category. A good practice is to group all the xaml files in folders according to the application names.
    - Let's create a **UiDemo subfolder** in the root folder.
    - Next, create a new sequence in Studio and name it UiDemo_Login.xaml.
    - Move the file in the UiDemo folder.
- The UiDemo_Login.xaml file should start with a description. We can also use an annotation in the very first activity, whether it's a sequence or a flowchart, to mention the function of each component and the use of the arguments. We typically add a Precondition too, in case a particular screen needs to be active beforehand, for instance. A Post action is included as well. Usually, the Login sequence should also include activities that initialize the application, but we will do that in the next tutorials. In this case, let's assume that the application has already been opened when we run the login process. "The application started" is a Precondition of our component.
- The next step is to define the arguments. We only need an input argument named Credential. Note that the Credential argument acts as an identifier used to retrieve usernames and passwords from various credential stores. The Credential and the Username arguments are Strings, and the Password is a SecureString - a special .NET class used to protect sensitive information. When naming the arguments, the best strategy is to use a prefix through which the type of the argument is specified. That also helps us to distinguish arguments from variables. Create an "in_Credential" input argument of the String type.
- There are many options when it comes to storing credentials. For instance, we can use the **GetAppCredentials.xaml** file we talked about in the previous video.
    - The first attempt in the sequence is to retrieve the Credential Asset from Orchestrator. However, Windows Credential Manager can also be used. It's up to you to decide between the two options, but we recommend creating an Orchestrator Credential Asset to enable global access to it.
    - So, use an **Invoke Workflow** activity and indicate the GetAppCredentials.xaml file.
    - Import the arguments.

- The in_Credential argument should have the in_Credential value.
- Create two variables to store the username and the password.
- The arguments panel should look like this:

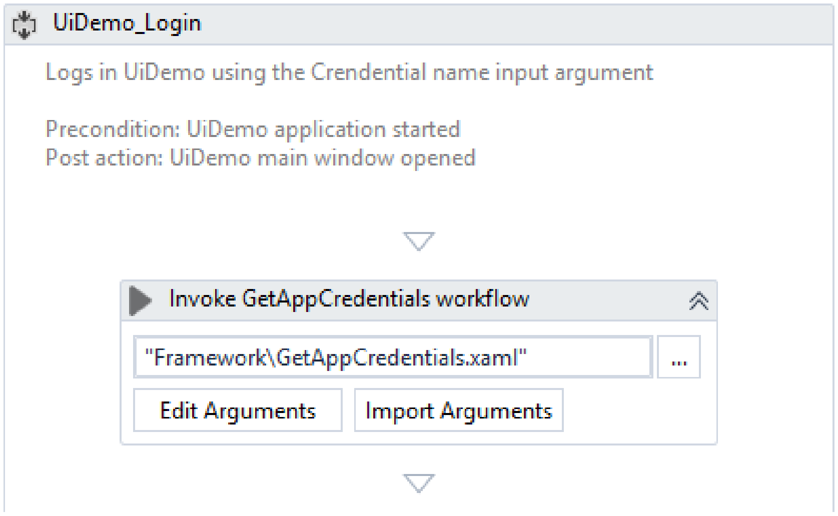


Invoked workflow's arguments

Name	Direction	Type	Value
out_Username	Out	String	Username
in_Credential	In	String	in_Credential
out_Password	Out	SecureString	Password
Create Argument			

Note: In Production, you might want to avoid using the GetAppCredentials.xaml file, for a more controlled behavior. You typically know the exact location of a set of credentials, which enables you to retrieve them easily. If the credentials are stored in Orchestrator as Assets, simply use the **Get Credential** activity in the **Orchestrator** category, under **Assets**. .

- Next, we need to type the username and the password, and then use a **Click activity** on the Log In button.
  - Use an Attach Windows activity and do the actions inside its scope.
  - Make sure the **SimulateClick** property is enabled.
  - To type the password, we need to use the **Type Secure Text** activity with the Password in the **SecureText** property.
  - Use the **SimulateType** property.
  - The final workflow should look like this:



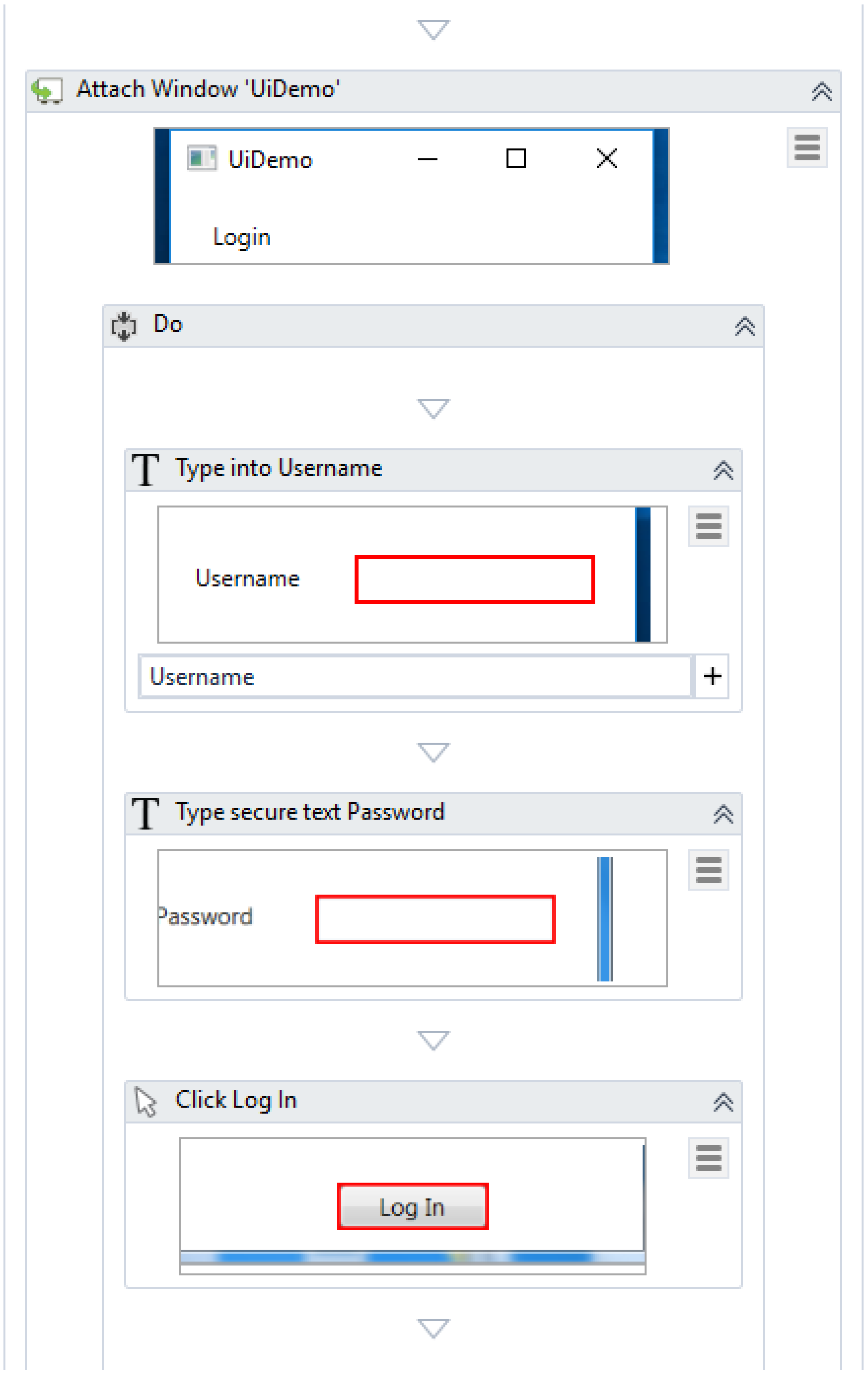
UiDemo_Login

Logs in UiDemo using the Crendential name input argument

Precondition: UiDemo application started
Post action: UiDemo main window opened

Invoke GetAppCredentials workflow

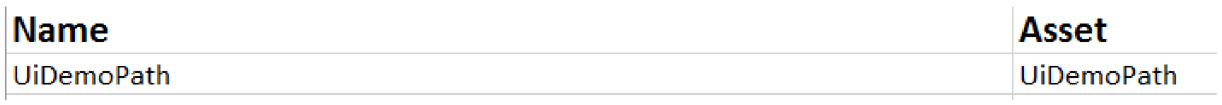
"Framework\GetAppCredentials.xaml"



After finishing any component, we need to test it thoroughly to reduce the debugging time later on. Default argument values can help, but sometimes you need to pass more complex objects as arguments. In that case, you can build Test Sequences to create the objects and pass them as arguments.

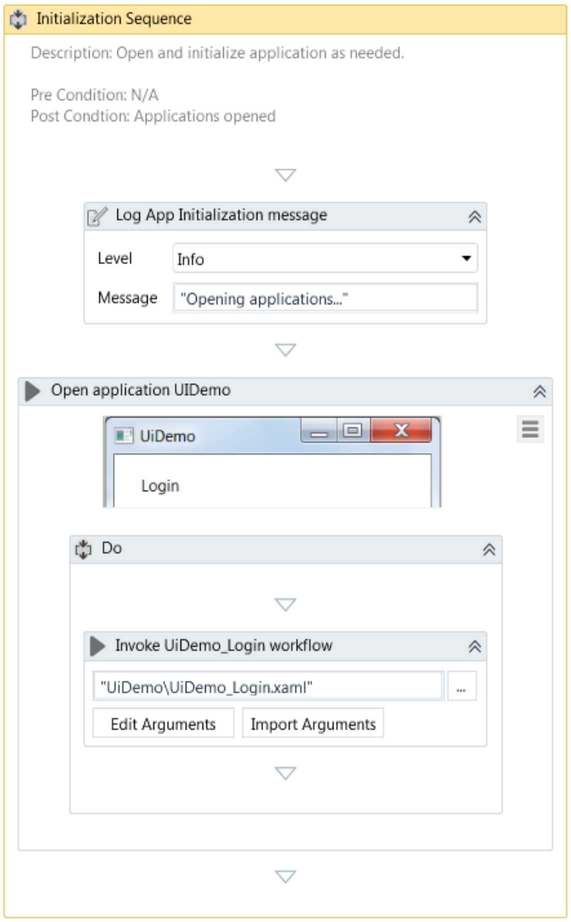
- Next up is the InitAllApplications.xaml file. We only use one application: UiDemo.
  - We need an **Open Application** activity.

    - Indicate the Log In Window in UiDemo. In the **FileName** property, there is the local path of the application, but we should avoid such hard-coded values in our workflows. Moreover, if we plan on using multiple robots, we might consider that different robots have different paths for the same application.
    - Luckily, we can use an Orchestrator Text Asset to store **Per Robot** values. We shouldn't hard-code Asset names either, as they might change from an environment to another, so let's use the Config file again and write the Asset name in the **Assets** sheet. It should look like this:



Name	Asset
UiDemoPath	UiDemoPath

- The Config dictionary is already used as input argument in the InitAllApplications.xaml workflow, so let's change the filename path to in_Config("UiDemoPath").ToString. Next, we need to invoke the UiDemo_Login workflow, import the arguments, and pass the Credential from the Config dictionary, as follows: in_Config("UiDemoCredential").ToString.
- The final workflow should look like this:


Initialization Sequence

Description: Open and initialize application as needed.

Pre Condition: N/A
Post Condtion: Applications opened

Log App Initialization message

Level: Info

Message: "Opening applications..."

Open application UIDemo

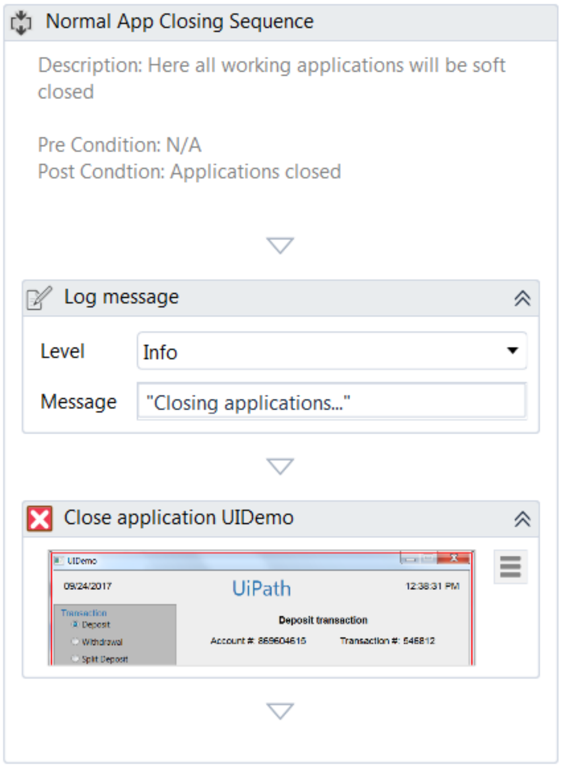
UiDemo
Login

Do

Invoke UiDemo_Login workflow

"UiDemo\UiDemo_Login.xaml"

- To test InitAllApplications, we need to run the _Test.xaml file, which first reads the Config file, and then invokes our file. All right! It works.
- Next, we need to configure CloseAllApplications and KillAllProcesses. These two are straightforward - they should look like this:



Normal App Closing Sequence

Description: Here all working applications will be soft closed

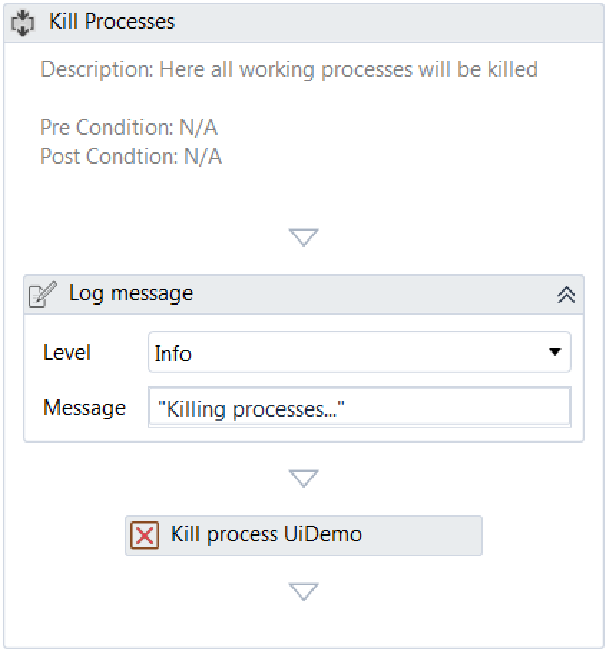
Pre Condition: N/A
Post Condtion: Applications closed

Log message

Level: Info

Message: "Closing applications..."

Close application UIDemo



Kill Processes

Description: Here all working processes will be killed

Pre Condition: N/A
Post Condtion: N/A

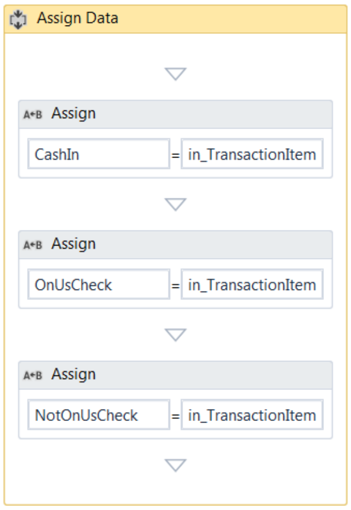
Log message

Level: Info

Message: "Killing processes..."

Kill process UiDemo

- We now need to complete the Process.xaml file. Since there are two decisions to take, and the second one stems from the first, the best layout for our Process is a Flowchart. After creating one, copy the annotation and the arguments from the existing Process sequence. Typically, we need to use the dynamic data from the queue item multiple times, so it's a good idea to create a set of local variables for it.
- Create a sequence and use the **Assign** activity to set the value of the variables, as follows:



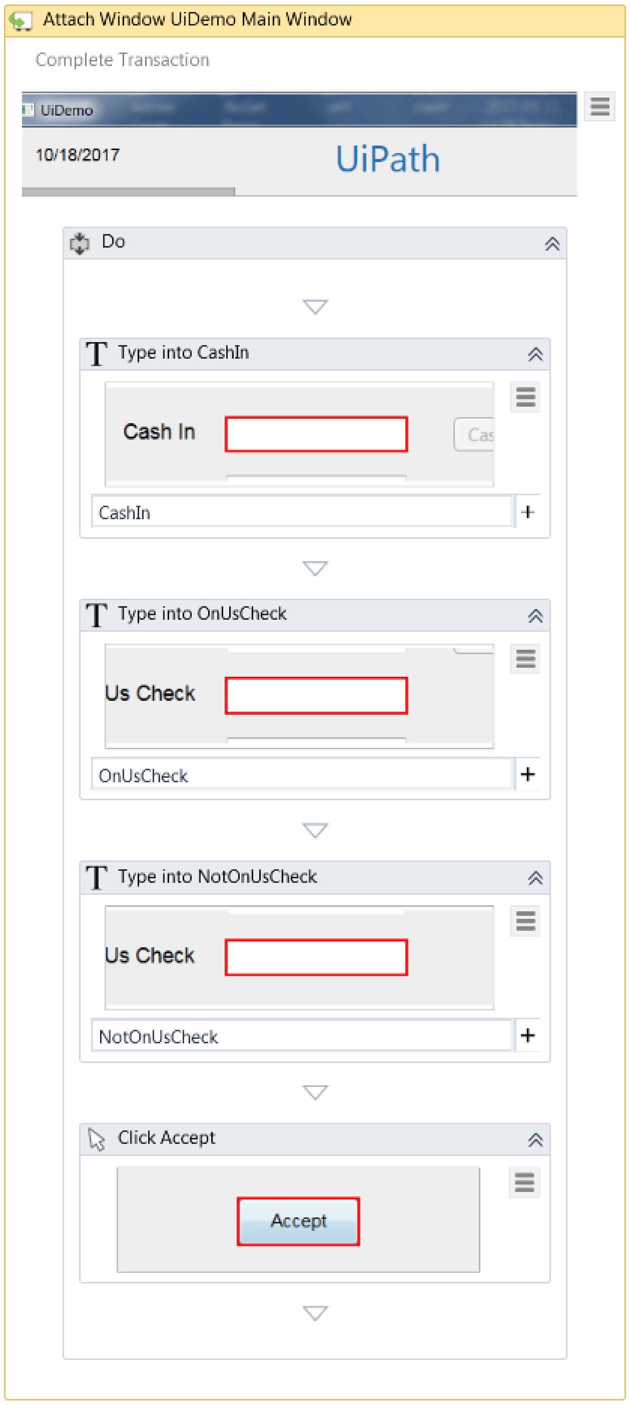
Assign Data

Assign: CashIn = in_TransactionItem

Assign: OnUsCheck = in_TransactionItem

Assign: NotOnUsCheck = in_TransactionItem

- We need to check if the input data is valid, and if every value is a number.
  - To that end, we can use the **Double.TryParse** method, and then store the results as Double variables.
  - Use a Flow Decision activity and set the condition to **Double.TryParse(CashIn, dbl_CashIn) AND Double.TryParse(OnUsCheck, dbl_OnUsCheck) AND Double.TryParse(NotOnUsCheck,dbl_NotOnUsCheck)**.
  - If it's invalid, let's use a **Throw** activity with a new BusinessRuleException object as the **Exception** property, as follows: **new BusinessRuleException("Input Data Invalid")**.
- If the Input Data is valid, add another Flow Decision to check if the value of **dbl_CashIn** is greater than 1000.
  - If it is, we use another **Throw** activity with a different Exception message.
  - If the value is less than 1000, let us use an **Attach Window** activity on the UiDemo main screen, three **Type Into** activities, and Click for the Accept button.
  - The **Attach Window** container should look like this:


Attach Window UiDemo Main Window

Complete Transaction

Do

Type into CashIn

CashIn

Type into OnUsCheck

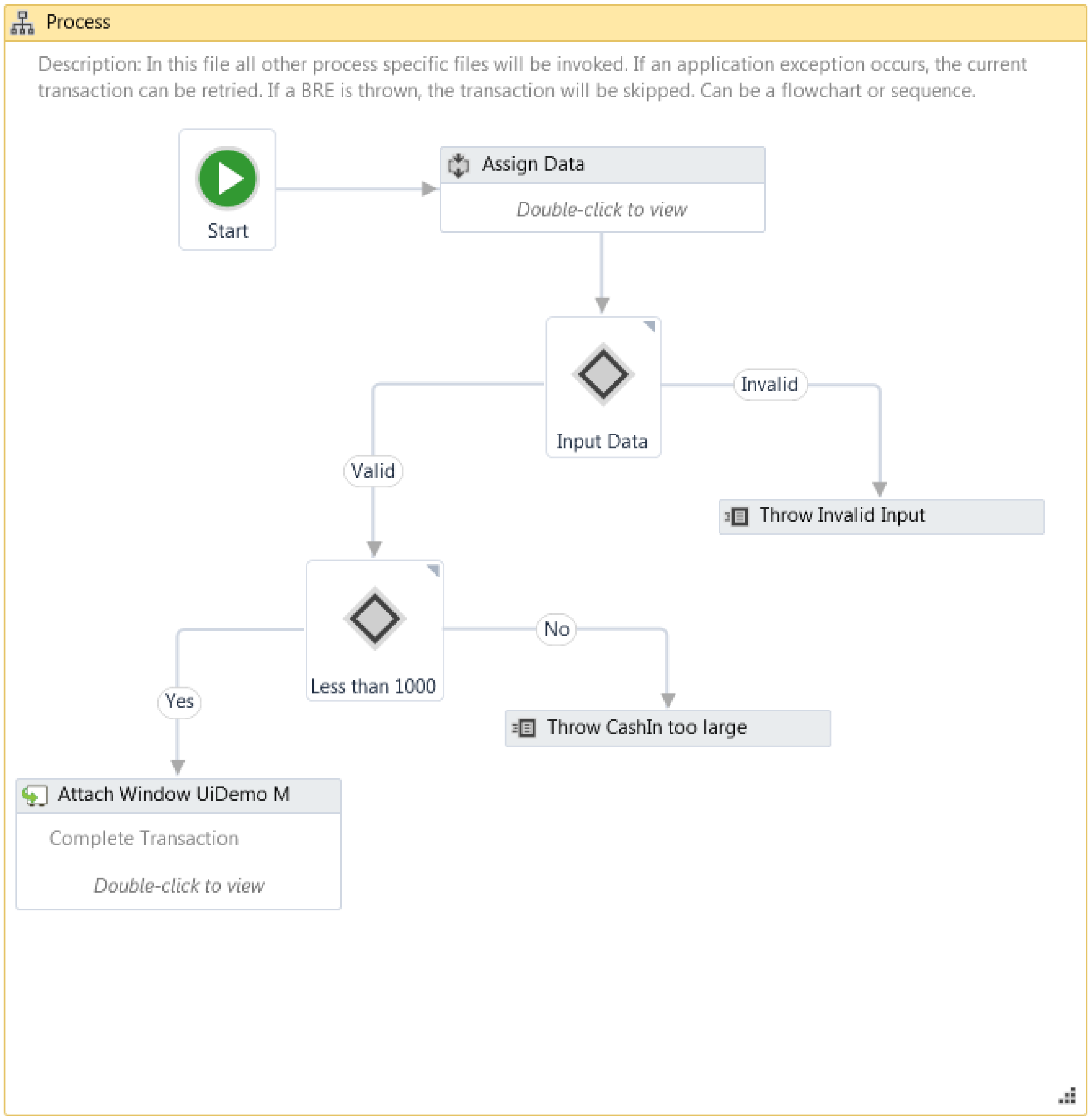
OnUsCheck

Type into NotOnUsCheck

NotOnUsCheck

Click Accept

- The Process flowchart should look like this:


Process

Description: In this file all other process specific files will be invoked. If an application exception occurs, the current transaction can be retried. If a BRE is thrown, the transaction will be skipped. Can be a flowchart or sequence.