



Automatización UiPath

Itinerario

Itinerario – UiDemo

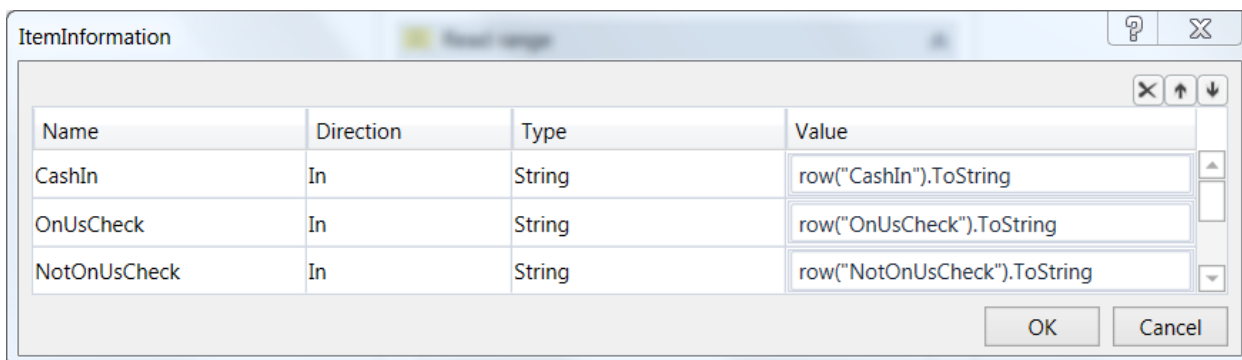
Estrategia: como necesitamos varios robots para el procesamiento en paralelo, tendremos que dividir la carga de trabajo de una manera fiable. El mejor planteamiento es utilizar Orchestrator Queues (colas). Crearemos dos proyectos de automatización. Recibirán el nombre Dispatcher (distribuidor) y Performer (ejecutor). El primero está encargado de leer el archivo Excel y cargar las transacciones en la cola, y el segundo procesa los elementos cargados en cola.

Flujo de trabajo	Entrada	Salida
Dispatcher	Archivo Excel	Elementos en cola
Performer	Elementos en cola	Transacciones procesadas

- Empezamos por la plantilla REFramework que se adapta muy bien al objetivo del Performer. Una de las cosas que debemos hacer al comienzo es en primer lugar cambiar el nombre de la carpeta «ReFramework_UiDemo» y en segundo lugar editar el archivo Project.json de la carpeta raíz. Cambiemos el nombre a «UiPath_REFramework_UiDemo» y la descripción a «Demostración de REFramework con UiDemo». Debería quedar así:

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

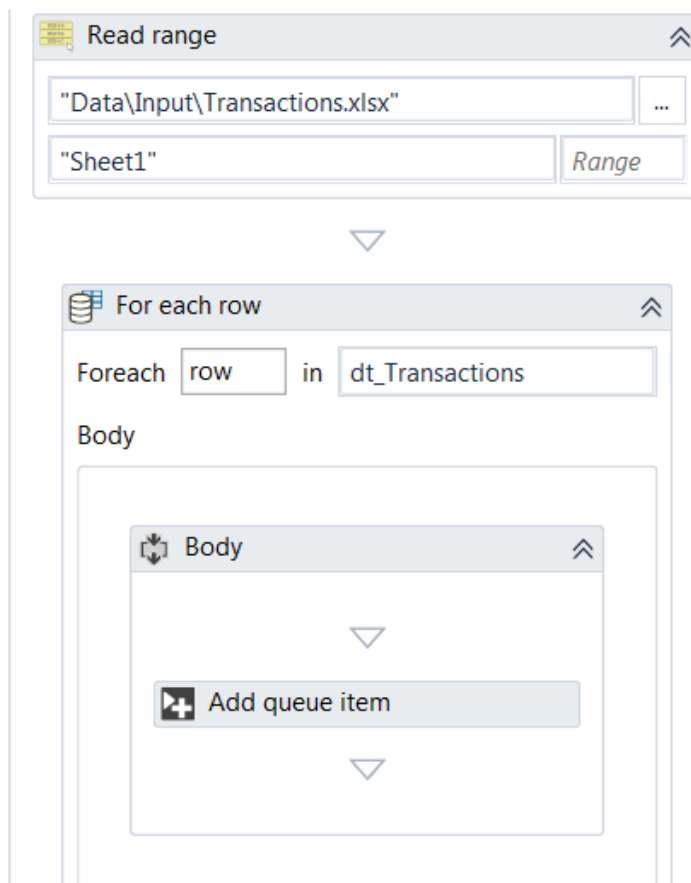
- El Dispatcher es muy simple en este caso: el robot debe leer el archivo Excel y añadir un Queue Item (elemento de cola) por cada fila. Lo normal es que esto sea un proceso independiente que se programe para su ejecución antes de la del Performer, pero para simplificar y facilitar las pruebas, lo incluiremos en el Performer.
 - Creemos una secuencia con el nombre **Dispatcher – UploadQueue**.
 - Necesitamos una actividad **Read Range** para extraer la información del archivo Excel y guardar la salida en una variable DataTable.
 - Cada fila es un elemento de transacción, así que a continuación necesitamos una actividad **For Each Row**. Utilice una actividad **Add Queue Item** en la sección **Body**.
 - Edite la propiedad ItemInformation añadiendo tres argumentos: CashIn, OnUsCheck y NotOnUsCheck. Debería quedar así:



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

OK Cancel

- Complete el campo **QueueName** y después cree una cola con el mismo nombre en Orchestrator. Para los reintentos de las transacciones fallidas, solo tiene que asignar a la propiedad **RetryNumber** el valor 2 cuando cree la cola.
- La secuencia del Dispatcher debería quedar así:



- Ejecutemos esta secuencia y después comprobemos la cola de elementos nuevos.

¡Ya está! Hemos creado el proceso Dispatcher. Pasemos ahora al **Performer**!

- La entrada para el Performer es una cola de Orchestrator, por tanto tendremos que hacer solo unos pequeños cambios en Framework. Empezamos por el archivo Config, donde **QueueName** debe coincidir con el valor que hemos utilizado antes. Es necesario

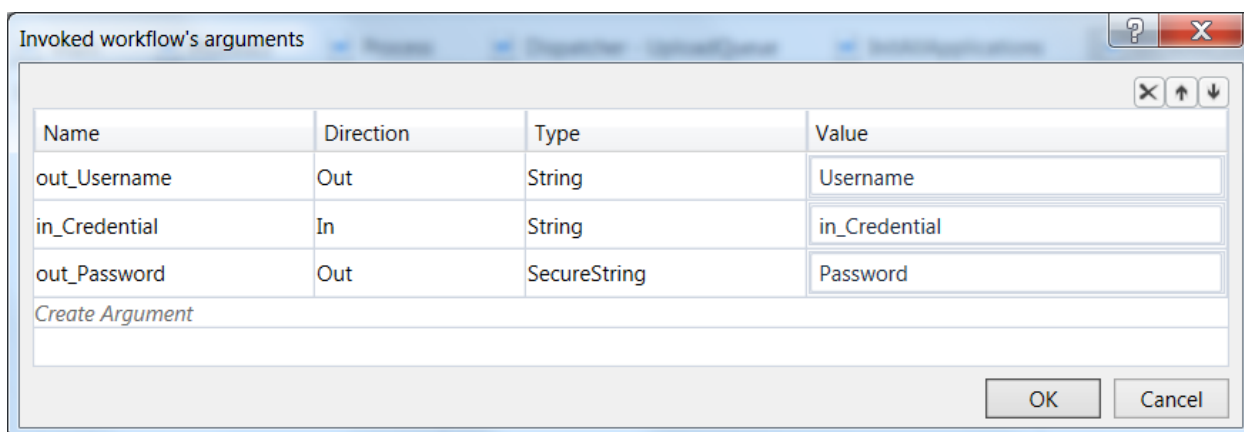
un ajuste más: un nombre de credencial para UiDemo. El valor de **MaxRetryNumber** en la hoja Constants debe ser 0, porque estamos utilizando la función de reintento de la cola en Orchestrator. La hoja Settings debería quedar así:

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

- Guarde y cierre el archivo Config.
- El paso siguiente es inicializar el proceso.
 - En primer lugar, necesitamos iniciar sesión en UiDemo antes de poder ejecutar transacciones repetitivas.
 - Siempre debemos pensar en términos de componentes reutilizables y el proceso de inicio de sesión definitivamente corresponde a esta categoría. Una buena práctica es agrupar todos los archivos xaml en carpetas según el nombre de aplicación.
 - Creemos una **subcarpeta UiDemo** en la carpeta raíz.
 - A continuación, cree una secuencia en Studio con el nombre UiDemo_Login.xaml.
 - Mueva el archivo a la carpeta UiDemo.
- El archivo UiDemo_Login.xaml debe comenzar por una descripción. También podemos utilizar una nota en la primera actividad, ya sea una secuencia o un diagrama de flujo, para indicar la función de cada componente y el uso de los argumentos. Normalmente también añadiremos una condición previa, en caso de que una pantalla concreta tenga que estar activa con anterioridad, por ejemplo. También se incluye una acción posterior. Normalmente, la secuencia de inicio de sesión también debe incluir actividades que inicialicen la aplicación, pero eso lo haremos en los siguientes tutoriales. En este caso, supongamos que la aplicación ya está abierta cuando ejecutamos el proceso de inicio de sesión. «Aplicación iniciada» es una condición previa de nuestro componente.
- El paso siguiente es definir los argumentos. Solo necesitamos un argumento de entrada con el nombre Credential y dos de salida para guardar el nombre de usuario y la contraseña. Tenga en cuenta que el argumento Credential actúa como identificador para recuperar nombres de usuarios y contraseñas desde distintos almacenes de credenciales. Los argumentos Credential y Username son de tipo String y la Password de tipo SecureString, una clase .NET especial que se utiliza para proteger información confidencial. Al poner nombre a los argumentos, le mejor estrategia es utilizar un prefijo en el que se indique el tipo de argumento. Esto también ayuda a distinguir los argumentos de las variables. Cree un argumento de entrada «in_Credential» de tipo String..
- Hay muchas opciones para el almacenamiento de credenciales. Por ejemplo, podemos utilizar el archivo **GetAppCredentials.xaml** del que hablamos en el vídeo anterior.
 - El primer intento de la secuencia es para recuperar el Credential Asset desde Orchestrator. No obstante, también se puede utilizar el Administrador de

credenciales de Windows. Usted puede decidir entre las dos opciones, pero le recomendamos que cree un Orchestrator Credential Asset para darle acceso global.

- Prosiga con una actividad **Invoke Workflow** e indique el archivo GetAppCredentials.xaml.
- Importe los argumentos.
- El argumento in_Credential debe tener el valor in_Credential.
- Cree dos variables para almacenar el nombre de usuario y la contraseña.
- El panel de argumentos debería quedar así:




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

Create Argument

OK Cancel

Aviso: En un entorno de producción, puede ser deseable evitar el uso del archivo GetAppCredentials.xaml, para tener más control sobre el comportamiento. Lo habitual es conocer la ubicación exacta de un conjunto de credenciales y esto permite recuperarlas con facilidad. Si las credenciales se almacenan en Orchestrator como Assets, solo tiene que utilizar la actividad **Get Credential** en la categoría **Orchestrator**, debajo de **Assets**.



- A continuación, tenemos que escribir el nombre de usuario y la contraseña, y después utilizar una **actividad Click** indicando el botón de inicio de sesión.
 - Verifique que la propiedad **SimulateClick** está activada.
 - Para escribir la contraseña, necesitamos utilizar la actividad **Type Secure Text** con el valor de Password en la propiedad **SecureText**.
 - Utilice la propiedad **SimulateType**.
 - El flujo de trabajo final debería quedar así:


UiDemo_Login

Logs in UiDemo using the Credential name input argument

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



▼


Invoke GetAppCredentials workflow


"Framework\GetAppCredentials.xml"

Edit Arguments Import Arguments



▼


Type into Username


Username



Username +

▼


Type secure text Password


Password

▼


Click Log In


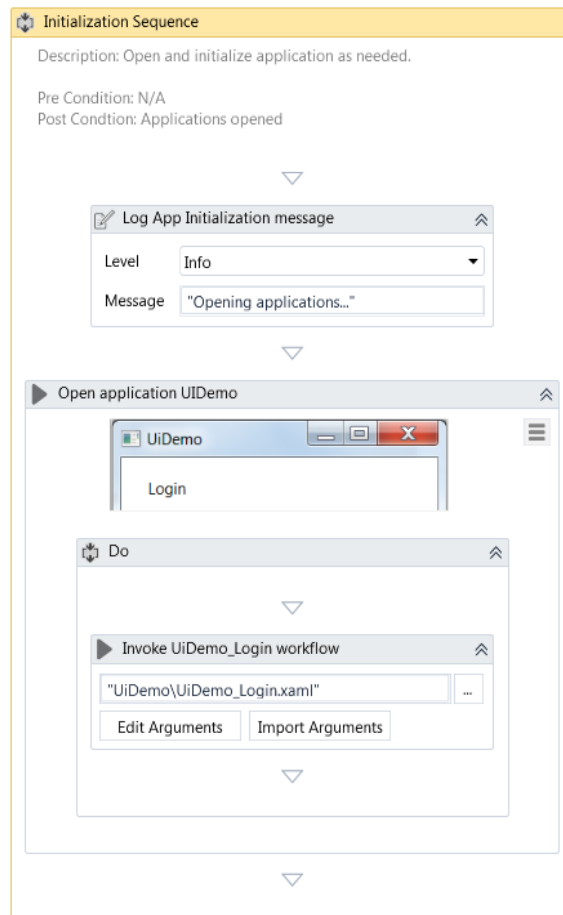
▼

Tras finalizar cualquier componente, tenemos que probarlo a fondo para reducir el tiempo dedicado más adelante a la depuración. Los valores predeterminados de los argumentos pueden servir, pero a veces es necesario pasar objetos más complejos como argumentos. En ese caso, puede construir secuencias de prueba para crear los objetos y pasarlas como argumentos.

- A continuación está el archivo InitAllApplications.xaml. Solo utilizaremos una aplicación: UiDemo.
 - Necesitamos una actividad **Open Application**.
 - Indique la ventana de inicio de sesión en UiDemo. En la propiedad **FileName**, está la ruta de acceso local de la aplicación, pero debemos evitar incluir estos valores al codificar nuestros flujos de trabajo. Es más, si pensamos utilizar varios robots, debemos tener en cuenta que los distintos robots tendrán distintas rutas de acceso para la misma aplicación.
 - Afortunadamente podemos utilizar un Orchestrator Text Asset para almacenar valores **por cada robot**. Tampoco deberíamos incluir en el código los nombres de los Asset, ya que pueden cambiar entre un entorno y otro, por tanto utilizemos el archivo Config una vez más y escribamos el nombre del Asset en la hoja **Assets**. Debería quedar así:

Name	Asset
UiDemoPath	UiDemoPath

- El diccionario Config ya se utiliza como argumento de entrada en el flujo de trabajo InitAllApplications.xaml, así que cambiemos la ruta de acceso al archivo a `in_Config("UiDemoPath").ToString`. A continuación, tenemos que hacer una llamada al flujo de trabajo UiDemo_Login, importar los argumentos y pasar el valor de Credential desde el diccionario Config del modo siguiente: `in_Config("UiDemoCredential").ToString`.
- El flujo de trabajo final debería quedar así:



- Para probar InitAllApplications, necesitamos ejecutar el archivo _Test.xaml que primero lee el archivo Config y después hace una llamada a nuestro archivo. ¡Ya está! Funciona.
- A continuación tenemos que configurar CloseAllApplications y KillAllProcesses. Son bastante sencillos, deberían quedar así:

Normal App Closing Sequence

Description: Here all working applications will be soft closed

Pre Condition: N/A
Post Condition: Applications closed

▽

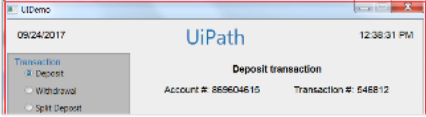
Log message

Level: Info

Message: "Closing applications..."

▽

Close application UiDemo



▽

Kill Processes

Description: Here all working processes will be killed

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

▽

Log message

Level: Info

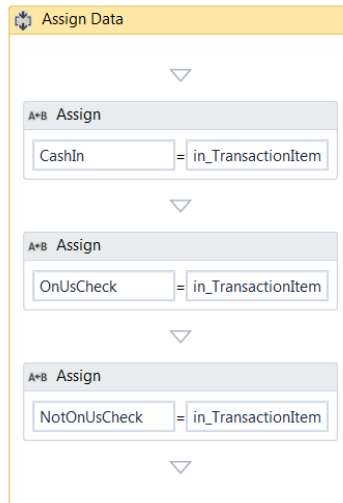
Message: "Killing processes..."

▽

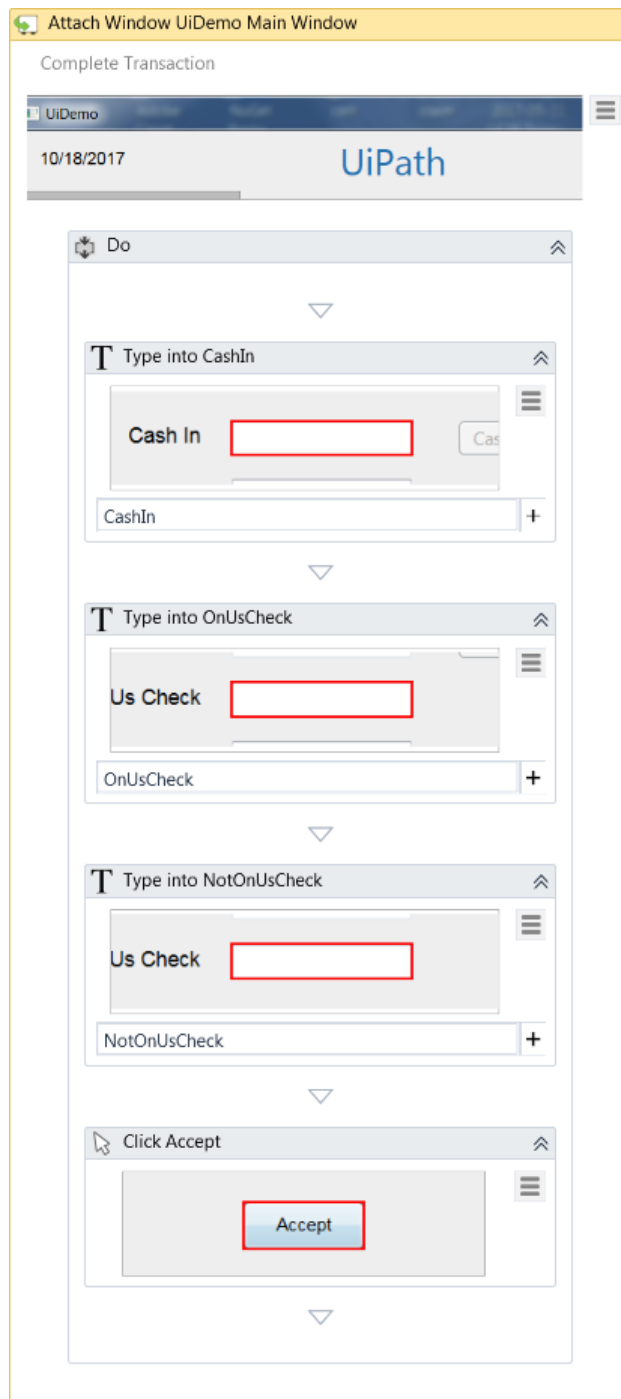
Kill process UiDemo

▽

- Ahora tenemos que completar el archivo Process.xaml. Como hay que tomar dos decisiones y la segunda deriva de la primera, el mejor diseño para nuestro proceso es un Flowchart. Después de crear uno, copie la nota y los argumentos desde la secuencia del proceso existente. Lo habitual es tener que utilizar los datos dinámicos procedentes del elemento en cola varias veces, por ello es una buena idea crear un conjunto de variables locales para esto.
- Cree una secuencia y utilice la actividad **Assign** para definir el valor de las variables, del modo siguiente:



- Tenemos que comprobar que los datos de entrada sean válidos y que todos los valores sean números.
 - Para ello, podemos utilizar el método **Double.TryParse** y después almacenar los resultados como variables Double.
 - Utilice una actividad Flow Decision y asigne la condición a **Double.TryParse(CashIn, dbl_CashIn) AND Double.TryParse(OnUsCheck, dbl_OnUsCheck) AND Double.TryParse(NotOnUsCheck,dbl_NotOnUsCheck)**.
 - En caso de que sea no válida, utilizaremos una actividad **Throw** con un objeto BusinessException nuevo como la propiedad **Exception**, del modo siguiente: **new BusinessException("Datos de entrada no válidos")**.
- Si los datos de entrada son válidos, se añade otra Flow Decision para verificar si el valor de **dbl_CashIn** es superior a 1000.
 - Si lo es, utilizamos otra actividad **Throw** con un mensaje distinto.
 - Si el valor es inferior a 1000, utilicemos una actividad **Attach Window** en la pantalla principal de UiDemo , tres actividades **Type Into** y una Click para el botón Accept.
 - El contenedor **Attach Window** debería quedar así:



- El diagrama de flujo del proceso debería quedar así:

