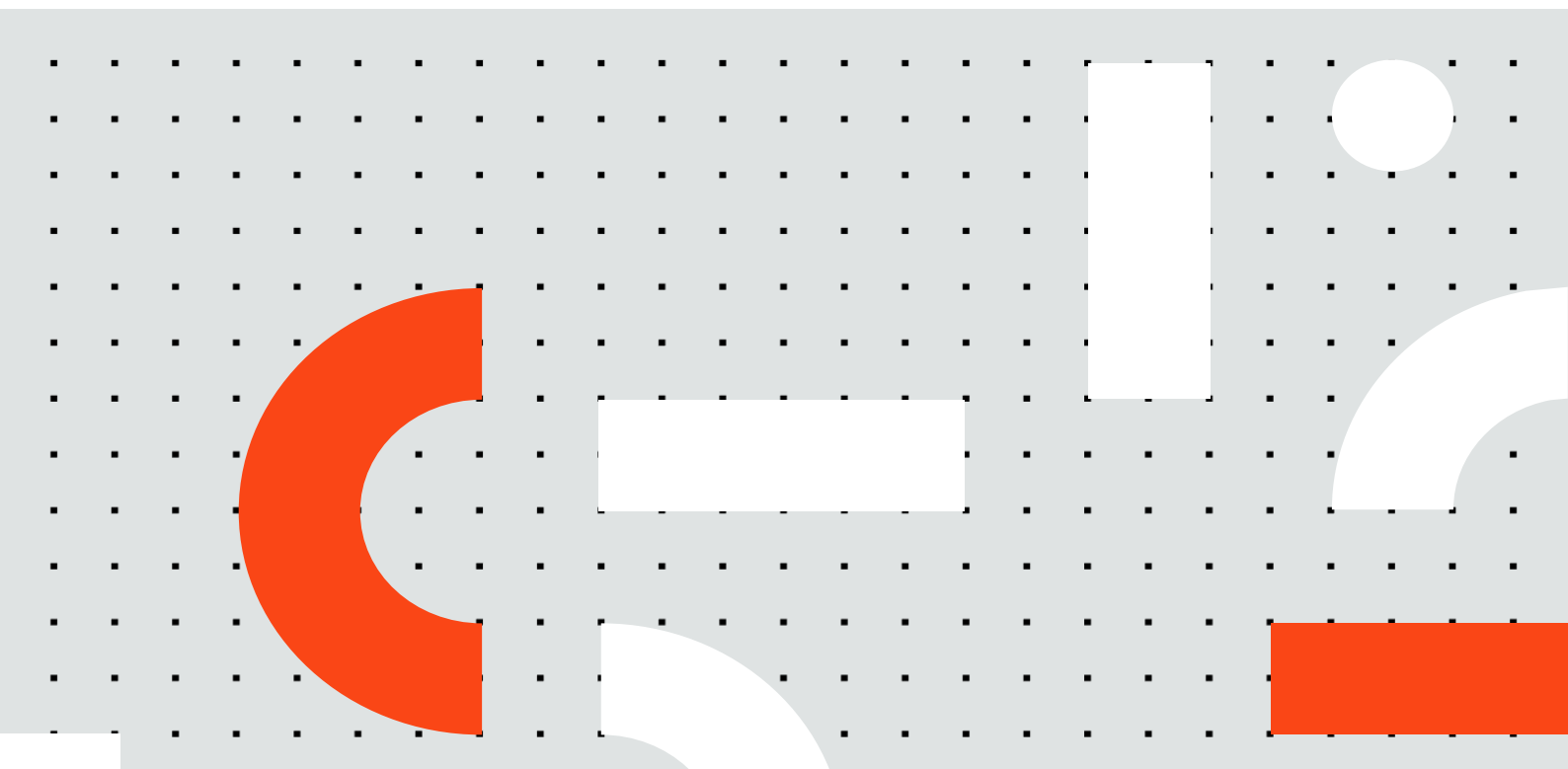




# Robotic Enterprise Framework



## 目次

概要 .....	3
トランザクションの処理 .....	3
主要な機能 .....	4
設定 .....	4
ログ .....	8
例外処理および例外からの回復 .....	10
アーキテクチャ .....	11
ステート .....	11
共有変数 .....	14
ワークフロー .....	16
Framework¥InitAllSettings.xaml .....	17
Framework¥KillAllProcesses.xaml .....	17
Framework¥InitAllApplications.xaml .....	18
Framework¥GetTransactionData.xaml .....	18
Process.xaml .....	20
Framework¥SetTransactionStatus.xaml .....	21
Framework¥RetryCurrentTransaction.xaml .....	23
Framework¥TakeScreenshot.xaml .....	24
Framework¥CloseAllApplications.xaml .....	24
フレームワークを使用する .....	25
フレームワーク ファイルへの変更 .....	25
Data¥Config.xlsx .....	25
Main.xaml .....	26
Framework¥GetTransactionData.xaml .....	27
Process.xaml .....	29
Framework¥SetTransactionStatus.xaml .....	30
実際の例 1: キューを使用する .....	31
実際の例 2: 表形式データを使用する .....	33
テスト フレームワーク .....	38
構成 .....	38
使用例 .....	38
拡張機能の配布とサポート .....	43

## 概要

RPA イニシアチブの成功は、プロジェクトが適切に構成されているか否かによって左右されます。プラスの結果を得るには、自動化に適したプロセス、つまり正しく定義されたステップから構成され、例外の発生率が低い、十分に練られたプロセスを選択し、明確なドキュメント（ソリューション設計書や業務プロセス定義書）を作成することに加えて、実装自体の品質も高める必要があります。

さまざまな RPA の実装にはそれぞれに特色がありますが、たいていの成功したプロジェクトには、一連の共通した実践事項が見られます。プロジェクトを支障なく実装し、理解し、保守するには、そのような共通の実践事項の中でも特に、柔軟な構成、堅牢な例外処理、意義のあるログが重要です。また、大規模な実装の場合は、処理されるデータ量のためスケーラビリティも重要な要素になります。

Robotic Enterprise Framework (REFramework) は、これらの重要な実践事項を網羅した UiPath Studio テンプレートであり、ほとんどの RPA プロジェクト、特にスケーラブルな処理が必要なプロジェクトの出発点として使用できます。REFramework はあらゆるプロセスに合わせて調整できますが、その利点はトランザクション プロセスを実装する場合に特に顕著になります。トランザクション アイテムは互いに独立しているため、トランザクション レベルで例外を処理し、ログを管理し、処理済みの各アイテムの詳細情報を提供できます。そして、失敗したトランザクションを簡単にリトライしたり、最終的にはスキップしたりできます。

このガイドでは、現実的なユース ケースと実際の例を使って、フレームワークを詳しく説明します。最初に「トランザクションの処理」でさまざまな種類のプロセスを紹介し、それらと REFramework の関連について説明します。その後に「主要な機能」で、フレームワークの主な特徴について概説します。次に、フレームワークを構成するワークフローについて「アーキテクチャ」で詳しく説明します。「フレームワークを使用する」では、フレームワークの実際の使用方法を明らかにし、2 つの例を順を追って説明します。次の「テスト フレームワーク」では、フレームワークの単体テスト機能の使用方法について説明します。最後の「拡張機能の配布とサポート」では、配布とサポートに関連するフレームワークのライセンスとポリシーについて規定します。

## トランザクションの処理

業務プロセスにはそれぞれの特性がありますが、通常は、データを処理する際の特定のステップ

を繰り返す方法に基づいて分類できます。

たとえば、ユーザーが指定した PDF ファイルから特定のデータを抽出し、そのデータを Web システムに入力する業務プロセスを考えてみましょう。この場合、別の PDF ファイルからデータを抽出するには、ユーザーがプロセスを再度実行し、新しいファイルを入力として渡す必要があります。

ただし、ユーザーが 1 つの PDF ファイルではなく複数の PDF ファイルを一括で指定すると、まとめて指定した各ファイルに対して同一の処理ステップが繰り返し実行されます。この場合、各 PDF ファイルを互いに独立した状態で処理できるのであれば、このプロセスにおいて各ファイルは 1 つのトランザクションであるといえます。つまりトランザクションは、単独で処理できる単一の作業単位を表します。

トランザクションの種類はプロセスによって異なりますが、重要なのは、自動化対象のプロセス内のトランザクションを明確に定義することです。REFramework ではトランザクション処理が元から考慮されており、**[トランザクションを処理]** ステートに定義されている同一のステップが各トランザクションに対して実行されます。「ステート」では、トランザクションのソースの指定方法と各トランザクションの処理方法について詳しく説明します。

## 主要な機能

REFramework は容易なトランザクション処理の手段を提供するだけでなく、安定したスケラブルな自動化プロジェクトの実装に役立つ、設定、ログ、例外処理などの機能を備えています。

## 設定

簡単にプロジェクトを保守し、すばやく構成値を変更できるように、構成値をワークフロー自体から分離しておくことをお勧めします。構成ファイルにプロジェクト全体で使用するパラメーターを定義すれば、ワークフローに値をハードコードせずに済みます。

REFramework には **Config.xlsx** という名前の構成ファイルが付随します。このファイルは、**Data** フォルダに格納されており、プロジェクトの構成パラメーターを定義するために使用できます。

これらのパラメーターは、**Main.xaml** ファイルの **Config** ディクショナリ変数に読み込まれます。このディクショナリは、フレームワークの他のファイルにも引数として渡されます。

簡単に操作できるように、この構成ファイルは 3 つのシートを持つ Excel ブックになっています。

ます。

- **Settings:** プロジェクト全体で使われる設定値で、通常は使用環境によって内容が変わります。たとえば、キューの名前、フォルダーパス、Web システムの URL などです。
- **Constants:** ワークフローのすべてのデプロイで同じであると想定される値です。たとえば、特定の画面に入力する部門名や銀行名です。
- **Assets:** Orchestrator のアセットとして定義される値です。

**Settings** シートと **Constants** シートの行は、フレームワークの初期化段階で **Config** ディクショナリに読み込まれるキーと値を表します。**Name** 列に構成のキーを指定し、**Value** 列にそのキーに関連する値を指定します。**Description** 列には行の説明が表示されますが、この説明はディクショナリには含まれません。表 1 に、**Constants** シートに定数を定義する例を示します。

表 1: 定数の例

Name	Value	Description
Department	経理	既定の部署名です。
Bank Code	ABC123	支払いに使用する銀行コードです。

たとえば、**Constants** シートに部門名の定数を追加します。名前は「Department」、値は「経理」、説明は「既定の部署名です。」とします。その後、ワークフローの実装中に、開発者は「Config("Department")」を使用して「経理」という値を取得できます。図 1 は、構成ファイル **Config.xlsx** と **Config** ディクショナリの関係を示しています。

### Config.xlsx 構成ファイル

Name	Value	Description
Department	経理	既定の部署名です。
BankName	ABC 銀行	既定の銀行名です。

### Config ディクショナリ

キー	値	使用法
Department	経理	<code>Config("Department").ToString</code>
BankName	ABC 銀行	<code>Config("BankName").ToString</code>

図 1: Config.xlsx と Config ディクショナリの対応

多くの定数が既定で定義されており、**Description** 列に定数の目的が詳述されています。その中で特に重要なのは **MaxRetryNumber** です。この定数には、システム例外で失敗したトランザクションの処理をロボットがリトライする回数を指定します。詳しくは、「例外処理および例外からの回復」をご覧ください。

トランザクションのソースとして Orchestrator キューを使用する場合、**MaxRetryNumber** の値は「0」である必要があります。「0」は、Orchestrator でリトライを管理することを示します。キューを使用しない場合、**MaxRetryNumber** の値を望ましいリトライ回数を表す整数に変更する必要があります。

**Assets** シートは他の 2 つのシートと異なり、**Name** 列に **Config** ディクショナリに含めるキーを指定し、**Value** 列に Orchestrator に定義されているアセットの名前を指定します。

図 2 は、Orchestrator で定義されているアセット、**Config.xlsx** ファイルの **Assets** シートでのアセットの定義、および **Config** ディクショナリを使ったワークフローでのアセットの参照方法の関係を示しています。

### Orchestrator のアセット

アセット名	型	値	説明
CountryName	Text	日本	既定の国名です。

### Config.xlsx 構成ファイルの Assets シート

Name	Asset	Description
CountryAsset	CountryName	既定の国名です。

### Config ディクショナリ

キー	値	使用法
CountryAsset	日本	<code>Config("CountryAsset").ToString</code>

図 2: Orchestrator のアセット、Config.xlsx、および Config ディクショナリの関係

たとえば、Orchestrator に CountryName というアセットがあり、**Assets** シートに **Name** 列が「CountryAsset」で **Value** 列が「CountryName」の行があるとします。初期化段階で、CountryName アセットの内容が取得され、**Config** ディクショナリの CountryAsset キーに対応する値として挿入されます。

上記の例では、Orchestrator のアセット名 (CountryName) と対応するディクショナリ キー (CountryAsset) に異なる名前が使用されていますが、双方に同じ名前を使用することが一般的です。同じ名前を使えば構成ファイルの保守が容易になり、開発中に名前の間違いが少なくなります。

**Assets** シートはほとんどの型のアセットに対応していますが、Credential 型のアセットは指定できません。資格情報にはユーザー名とパスワードの 2 つの値があるためです。Orchestrator に定義されている資格情報アセットを使用するには、**Settings** シートにアセットを含めます (図 3)。**Name** 列に **Config** ディクショナリのキーを、**Value** 列に資格情報アセットの名前を、**Description** 列に資格情報の説明を指定します。ワークフローの実装で **[資格情報を取得]** アクティビティを使用して、Orchestrator から資格情報を取得します。

### Orchestrator の資格情報アセット

アセット名	型	ユーザー名	パスワード	説明
System1Credential	Credential	UserABC	Pass123	System1 にアクセスするための資格情報です。

### Config.xlsx 構成ファイルの Settings シート

Name	Value	Description
System1Credential	System1Credential	ACME System 1 の資格情報です。

### Config ディクショナリ

キー	値	使用法
System1Credential	System1Credential	<code>Config("System1Credential").ToString</code>

図 3: Orchestrator の資格情報アセット、Config.xlsx、および Config ディクショナリの関係

**Config.xlsx** の最後の注意点ですが、構成ファイルは暗号化されていないため、資格情報を直接格納しないでください。代わりに、Orchestrator のアセットまたは Windows の資格情報マネージャーを使用して機密データを保存する方が安全です。

## ログ

自動化プロジェクトでログを適切に使用すると、アクションやイベントの可視性、デバッグの容易性、監査の意義が向上するなど、さまざまな利点があります。

REFramework には包括的なログ構造が組み込まれており、さまざまなレベルで **[メッセージをログ]** アクティビティを使用することで、トランザクションの状態、例外、およびステート間の遷移を出力できます。使用されるログ メッセージのほとんどに、**Config.xlsx** ファイルの **Constants** シートに構成する静的な情報が含まれます。

ロボットが生成するメッセージに含まれる通常のログ フィールド（ロボット名やタイムスタンプなど）に加えて、追加のカスタム ログ フィールドを使用して、各トランザクションに関する

データを追加できます。処理対象の新しいトランザクションを取得するとき、**GetTransactionData.xaml** ファイルでカスタム ログ フィールドである **TransactionId**、**TransactionField1**、および **TransactionField2** の値を定義できます。

図 4 に、**SetTransactionStatus.xaml** ファイルの一部を示します。このファイルで、**[ログ フィールドを追加]** アクティビティを使用してカスタム フィールドをログ メッセージに追加します。**[メッセージをログ]** アクティビティを使用した後、追加されたフィールドは **[ログ フィールドを削除]** アクティビティによって削除します。これにより、単一のトランザクションに関する追加フィールドが確実に 1 回のみ出力されることになるため、このようなデータを使用した集計に影響を与えるおそれがある、ログの重複が防止されます。

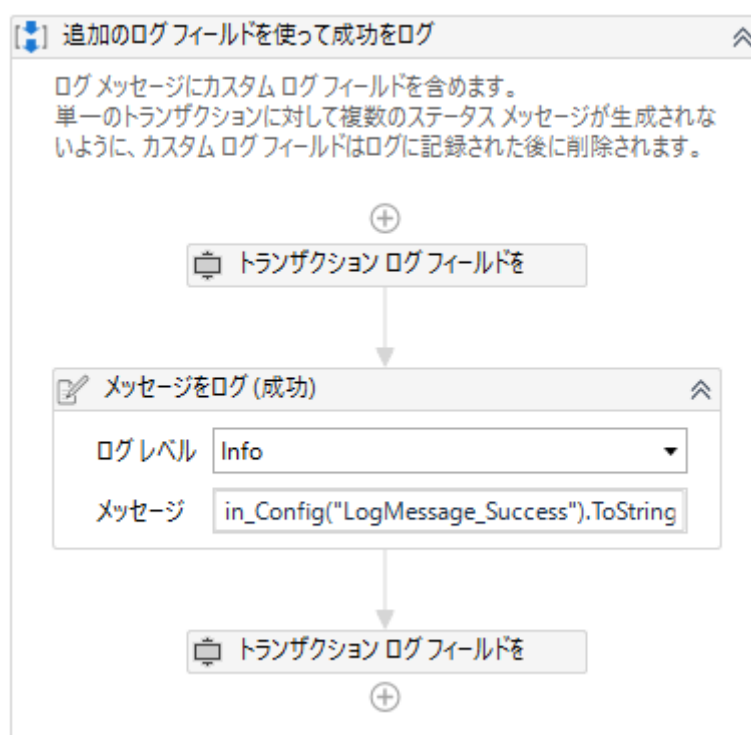


図 4: カスタム ログ フィールドの追加と削除

カスタム ログ フィールドの使用は任意ですが、トランザクションの補足情報を追加しておけば、デバッグやトラブルシューティングに役立つ可能性があります。

また、これらのカスタム ログ フィールドは業務報告に活用できます。たとえば、請求書をトランザクションとして扱うプロセスで、請求書番号を **TransactionId** フィールドに割り当てることができます。また、請求日を **TransactionField1** に、合計を **TransactionField2** に割り当てることができます。このようなデータを基に生成されるログを使用して、1 か月のうち多

数の請求書が処理された日や、特定期間内に処理された総合計額を示す視覚資料を作成できます (図 5)。

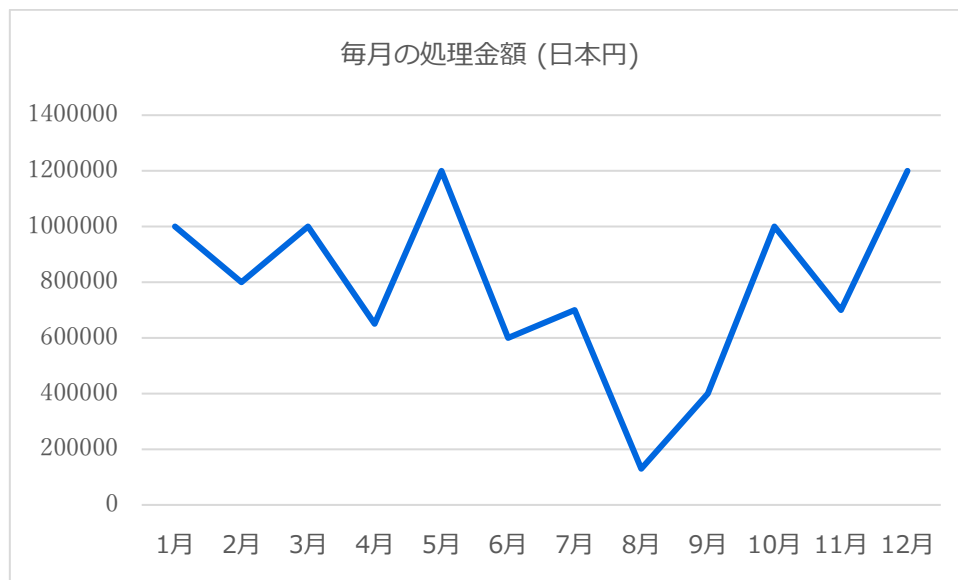


図 5: カスタム ログ フィールドを使用したレポートの例

ログは暗号化されず、漏えいした場合はプライバシーの問題につながる可能性があるため、機密データをログに含まないでください。

## 例外処理および例外からの回復

REFramework には堅牢な例外処理スキームが備わっており、例外から自動的に回復したり、回復不可能な例外が発生した場合は、トランザクションの状態を更新して実行を正常終了したりできます。この機能はログ機能と密接に関連しており、例外に関するすべての情報が適切に記録されるため、分析と調査に活用できます。

フレームワークの実行中に発生する例外は、次の 2 つのカテゴリに分かれます。

- ビジネス例外:** この種の例外は **BusinessRuleException** クラスによって実装されます。自動化対象の業務プロセスのルールに関連して問題が発生した場合に、この例外をスローする必要があります。たとえば、ファイル付きのメールを受信するプロセスにおいてメールにファイルが添付されていない場合、プロセスを続行できません。この場合、開発者は **【スロー】** アクティビティを使用して **BusinessRuleException** をスローできます。この例外はプロセスのルールに反する問題があったことを示します。**BusinessRuleException** はワークフローの開発者が明示的にスローする必要があります。

ます。フレームワークやアクティビティによって自動的にスローされることはありません。

- **システム例外:** プロセス自体のルールに関連しない例外は、システム例外とみなされます。システム例外の例としては、タイムアウトしたアクティビティが挙げられます。ネットワーク接続が低速であったり、ブラウザのクラッシュのためセレクトアが見つからなかったりすると、タイムアウトが発生します。

例外のカテゴリ、つまりビジネス例外かシステム例外かに応じて、トランザクションをリトライするかどうかが決まります。ビジネス例外の場合、通常、トランザクションは自動的にリトライされません。ビジネス ルールに関連する問題には人の介入が必要なためです。その一方でシステム例外の場合、一時的な問題が原因でエラーが発生した可能性があります。また、同じトランザクションのリトライが人の介入なしで成功する可能性があります。

フレームワークにおけるビジネス例外とシステム例外の概念は、Orchestrator における**ビジネス例外**と**アプリケーション例外**に対応します。実際、トランザクションのソースが Orchestrator キューである場合、システム例外の場合のリトライ回数は Orchestrator に直接設定できます。Orchestrator を使用しない場合、「設定」で説明したように、リトライの構成は **Config.xlsx** ファイルに指定します。

## アーキテクチャ

REFramework はステート マシンのワークフローとして実装されています。このワークフローでは、実行の特定の状況を表すステートを定義します。特定の条件に応じてあるステートから別のステートに実行が遷移することで、プロセスのステップを表すことができます。

## ステート

REFramework のステートは以下のとおりです。図 6 をご覧ください。

- **初期化:** 構成ファイルを読み取り、プロセスで使用するアプリケーションを初期化します。初期化が成功すると、実行は **[トランザクション データを取得]** ステートに遷移します。エラーが発生した場合は **[プロセスを終了]** ステートに遷移します。トランザクションの処理中にシステム例外が発生した場合、エラーからの回復が試行されます。具体的には、アプリケーションを再初期化できるように、使用されているすべてのアプリケーションを閉じて **[初期化]** ステートに戻そうとします。

- **トランザクション データを取得:** 処理対象の次のトランザクションを取得します。処理対象のデータがそれ以上なかったり、エラーが発生したりすると、実行は **[プロセスを終了]** ステートに遷移します。新しいトランザクションが正常に取得されると、**[トランザクションを処理]** ステートで処理されます。
- **トランザクションを処理:** 単一のトランザクションを処理します。処理の結果は、成功、ビジネス例外、またはシステム例外です。システム例外の場合、現在のトランザクションの処理は自動的にリトライできます。結果がビジネス例外である場合はトランザクションがスキップされ、**[トランザクション データを取得]** ステートで新しいトランザクションの取得が試行されます。現在のトランザクションの処理が成功した場合も **[トランザクション データを取得]** ステートに戻り、新しいトランザクションが取得されます。
- **プロセスを終了:** プロセスを終了し、使用されているすべてのアプリケーションを閉じます。

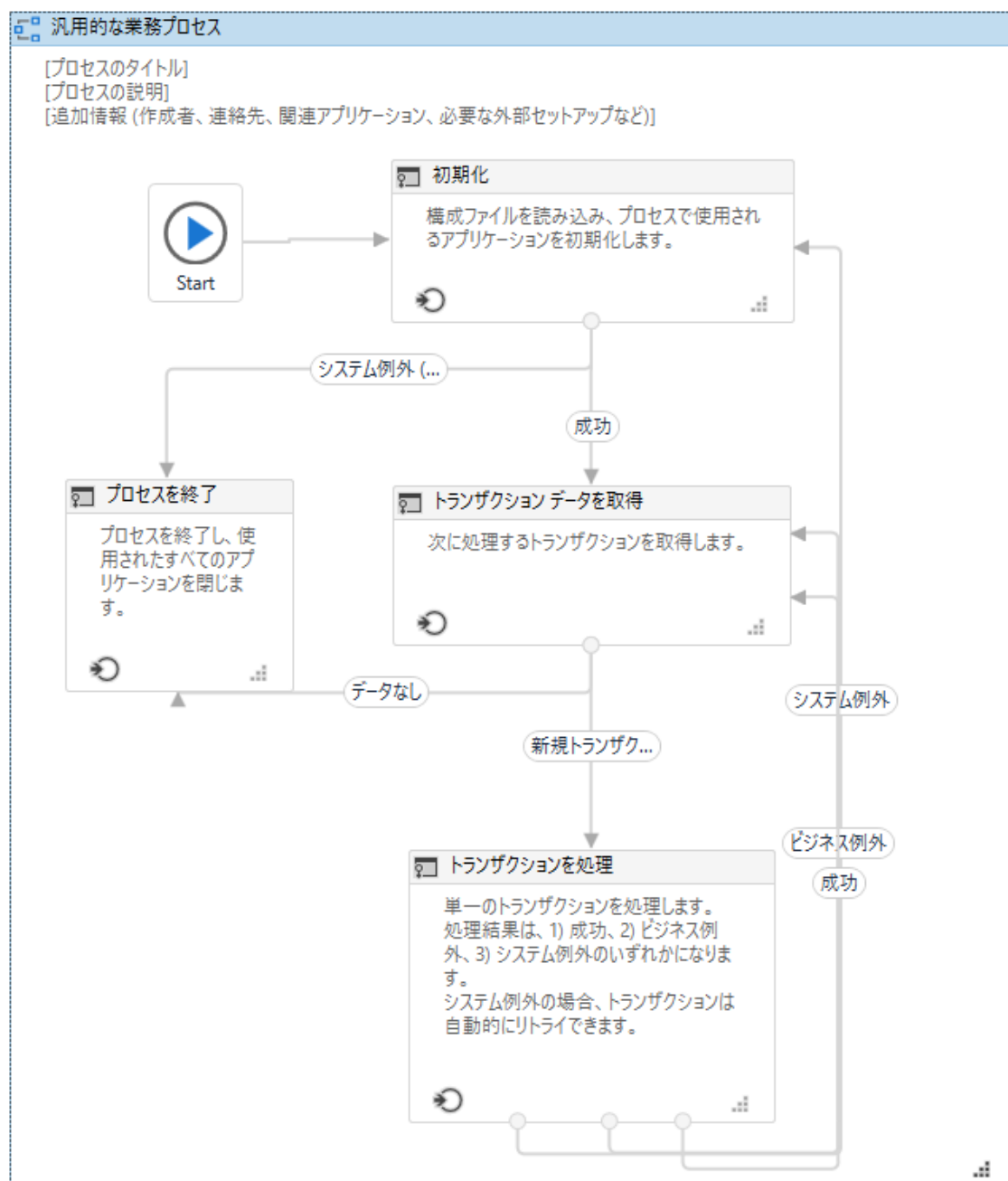


図 6: REFramework のステートを含むステート マシン

各ステートから、表 2 に示すワークフローを呼び出します。詳しくは、「ワークフロー」をご覧ください。

表 2: 各ステートで呼び出されるワークフロー

ステート	呼び出されるワークフロー
初期化	InitAllSettings.xaml KillAllProcesses.xaml InitAllApplications.xaml
トランザクション データを取得	GetTransactionData.xaml
トランザクションを処理	Process.xaml SetTransactionStatus.xaml <ul style="list-style-type: none"> <li>• RetryCurrentTransaction.xaml</li> <li>• TakeScreenshot.xaml</li> <li>• CloseAllApplications.xaml</li> <li>• KillAllProcesses.xaml</li> </ul>
プロセスを終了	CloseAllApplications.xaml KillAllProcesses.xaml

## 共有変数

表 3 に、**Main.xaml** ファイルで宣言され、各ステートで呼び出されるワークフローに引数として渡される変数を示します。

表 3: 共有変数

名前	既定の型	説明
TransactionItem	QueueItem	処理対象のトランザクション アイテムです。この変数の型は、プロセスにおけるトランザクションの種類に合わせて変更できます。たとえば、スプレッドシートのデータを <b>DataTable</b> に読み込んで処理する場合、 <b>TransactionItem</b> の型を <b>DataRow</b> に変更できます（「実際の例 2: 表形式データを使用する」の例を参照）。また、トランザクションが処理対象の画像ファイルへのパスである場合、 <b>TransactionItem</b> の型を <b>String</b> に変更できます。
SystemException	Exception	ステート間の遷移中に、 <b>BusinessRuleException</b> 以外の例外を表すために使用します。
BusinessException	BusinessRuleException	ステート間の遷移中に、自動化対象のプロセスのルールに反する状況を表すために使用します。
TransactionNumber	Int32	トランザクション アイテムのシーケンシャル カウンターです。
Config	Dictionary(String,Object)	プロセスの構成データを格納するディクショナリ構造（設定、定数、およびアセット）です。

名前	既定の型	説明
RetryNumber	Int32	システム例外の場合に、トランザクション プロセスのリトライ試行回数を制御するために使用します。
TransactionField1	String	オプションで、トランザクション アイテムに関する追加情報を含めるために使用します。
TransactionField2	String	オプションで、トランザクション アイテムに関する追加情報を含めるために使用します。
TransactionID	String	情報提供とログのために使用します。 この ID はトランザクションごとに一意にすることをお勧めします。
TransactionData	DataTable	スプレッドシートから取得した後など、トランザクションを <b>DataTable</b> に保存する場合に使用します。

**Main.xaml** で呼び出され、ほとんどすべてのワークフローに渡される重要な変数が **Config** ディクショナリです。この変数は **[初期化]** ステートで **InitAllSettings.xaml** ワークフローによって初期化され、**Config.xlsx** ファイルで宣言されるすべての構成を保持します。ディクショナリであるため、**Config** の値には「Config("Department")」や「Config("System1\_URL")」のように指定して、キーからアクセスできます。**Config.xlsx** ファイルの **Description** 列の値はディクショナリに含まれないことに注意してください。

## ワークフロー

ここでは、REFramework を構成するワークフローについて、その概要、目的、引数などを詳しく説明します。該当する場合は、トランザクション アイテムの型が **QueueItem** でないときに修正すべき箇所についても説明します。

## Framework¥InitAllSettings.xaml

このワークフローは **Framework** フォルダにあり、プロジェクトで使用される **Config** (構成) デクショナリを初期化、事前設定、および出力します。設定と定数はローカルの構成ファイル **Data¥Config.xlsx** から読み込まれ、アセットは Orchestrator から取得されます。設定や定数と同じ名前のアセットが存在する場合、アセットの値が優先されます。表 4 に、**InitAllSettings.xaml** で使用される引数を示します。

表 4: InitAllSettings.xaml の引数

引数	説明	既定値
in_ConfigFile	設定、定数、およびアセットを定義する構成ファイルのパスです。	"Data¥Config.xlsx"
in_ConfigSheets	構成ファイルに含まれる、設定および定数が記載されているシートの名前です。	{"Settings","Constants"}
out_Config	プロセスの構成データ (設定、定数、およびアセット) を格納するデクショナリ構造です。	既定値なし

構成ファイルが見つからないなど、このワークフローの実行中に例外が発生すると、**[初期化]** ステートの **[トライ キャッチ]** アクティビティで検出され、**[プロセスを終了]** ステートに実行が遷移します。

## Framework¥KillAllProcesses.xaml

設定を初期化した後で主要なプロセスを開始する前に、システムを確実にクリーンな状態にするためのアクションを実行できます。このアクションは **[プロセスを強制終了]** アクティビティを使用して実行します。このアクティビティにより、業務プロセスで使用するアプリケーションの Windows プロセスが強制的に終了されます。プロセスを強制終了すると、ファイルに保存する前のデータが失われるなど、望ましくない結果が生じるおそれがあることに注意してください。**KillAllProcesses.xaml** ワークフローは **Framework** フォルダにあり、このようなクリーンアップ ステップを実装できます。

このワークフローの名前には「KillAllProcess (すべてのプロセスを強制終了)」とありますが、使用されるすべてのプロセスを必ずしも強制終了する必要はありません。また、システムをクリーンな状態に戻すには、他のステップを使う方が適切である場合もあります。最終的には、クリーンアップのステップは業務プロセスの要件によって左右されます。

## Framework¥InitAllApplications.xaml

**InitAllApplications.xaml** ワークフローは **Framework** フォルダにあり、プロセスの実行中に操作するアプリケーションを初期化できます。**[アプリケーションを開く]** や **[ブラウザーを開く]** などのアクティビティを含めることも、ログインや認証などのアクションを実装する他のワークフローを呼び出すこともできます。

表 5 に、このワークフローが受け取る唯一の引数である **Config** (構成) デictionary を示します。**Config** には、Web アプリケーションの URL など、特定のアプリケーションの起動に必要なデータを格納できます。

表 5: InitAllApplications.xaml の引数

引数	説明	既定値
in_Config	プロセスの構成データ (設定、定数、およびアセット) を格納する Dictionary 構造です。	既定値なし

## Framework¥GetTransactionData.xaml

このワークフローは **Framework** フォルダにあり、指定されたソース (Orchestrator キュー、スプレッドシート、データベース、メールボックス、または Web API) からのトランザクション アイテムの取得を試行します。

残りのトランザクション アイテムがない場合は引数 **out\_TransactionItem** を「Nothing」に設定し、**[プロセスを終了]** ステートに遷移します。使用される引数については表 6 をご覧ください。

表 6: GetTransactionData.xaml の引数

引数	説明	既定値
in_TransactionNumber	トランザクション アイテムのシーケンシャル カウンターです。	既定値なし
in_Config	プロセスの構成データ (設定、定数、およびアセット) を格納するディクショナリ構造です。	既定値なし
out_TransactionItem	処理対象のトランザクション アイテムです。	既定値なし
out_TransactionField1	オプションで、トランザクション アイテムに関する追加情報を含めるために使用します。	既定値なし
out_TransactionField2	オプションで、トランザクション アイテムに関する追加情報を含めるために使用します。	既定値なし
out_TransactionID	情報提供とログのために使用します。この ID はトランザクションごとに一意にすることをお勧めします。	既定値なし
io_TransactionData	スプレッドシートから取得した後など、トランザクションを <b>DataTable</b> に保存する場合に使用します。	既定値なし

繰り返しのないプロセスなどでトランザクションが 1 つしかない場合は、**[条件分岐]** アクティビティを追加して引数 **in\_TransactionNumber** の値が「1」である (そのトランザクションが最初で唯一のものである) かどうかを確認し、トランザクション アイテムを **out\_TransactionItem** に代入します。**in\_TransactionNumber** の値が「1」以外のときは、**out\_TransactionItem** を「Nothing」に設定します (図 7)。

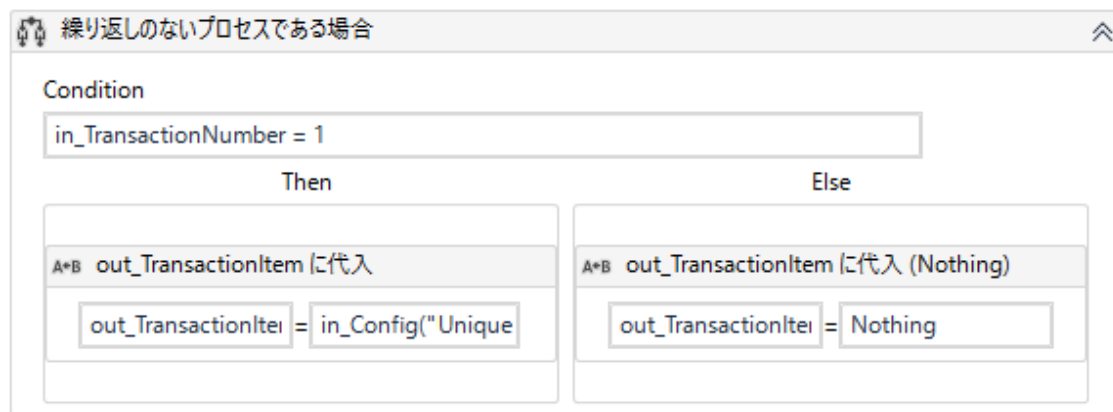


図 7: 繰り返しのないプロセスの場合の GetTransactionData.xaml の構成

Orchestrator キュー以外をソースとするトランザクションが複数ある場合は、引数 **in\_TransactionNumber** をインデックスとして使い、処理対象の適切なトランザクションを取得します。残りのトランザクションがなくなったら、**out\_TransactionItem** を「Nothing」に設定し、プロセスを終了します。

**GetTransactionData.xaml** ワークフローは、既定では Orchestrator キューの使用を前提とします。そのため、最初のアクティビティでは新しいトランザクションを Orchestrator キューから取得しようと試みます。この状況については「実際の例 1: キューを使用する」の例で説明します。

Orchestrator キューを使用しない場合は、**[トランザクション アイテムを取得]** アクティビティを適切なロジックに置き換えてください。たとえば、トランザクションが **DataTable** の行である場合は、現在のトランザクションに対応する行をここで取得します。この場合の例については「実際の例 2: 表形式データを使用する」をご覧ください。

最後に、主にログと可視化のため、このワークフローの省略可能なステップで、トランザクション アイテムの補足情報を追加して利用できるように注目しましょう。たとえば、あるプロセスのトランザクション アイテムが請求書である場合は、**out\_TransactionID** に請求書番号、**out\_TransactionField1** に請求日、**out\_TransactionField2** に請求金額を代入できます。カスタム ログ フィールドを使用したログについて詳しくは、「ログ」をご覧ください。

## Process.xaml

**Process.xaml** ワークフローは、業務プロセスの主要なステップを呼び出すために使用します。一般的に、このような主要なステップは複数のサブワークフローを使って実装します。主な引数

は **in\_TransactionItem** で、処理対象のデータを表します。引数 **in\_TransactionItem** の既定の型は **QueueItem** であり (表 7)、**DataRow**、**String**、**MailMessage** などの他の型を使用する場合は変更する必要があります。

表 7: Process.xaml の引数

引数	説明	既定値
in_TransactionItem	処理対象のトランザクション アイテムです。	既定値なし
in_Config	プロセスの構成データ (設定、定数、およびアセット) を格納するディクショナリ構造です。	既定値なし

処理中に **BusinessRuleException** がスローされた場合は、現在のトランザクションはスキップされます。別の種類の例外が発生すると、現在のトランザクションはリトライの構成に従ってリトライされます。

## Framework¥SetTransactionStatus.xaml

**SetTransactionStatus.xaml** ワークフローは **Framework** フォルダーにあり、各トランザクションのステータスを設定しログに記録します。ステータスは、成功、ビジネス例外、およびシステム例外の 3 つのいずれかになります。

プロセスのルールに反する状況はビジネス ルール例外に分類され、**BusinessRuleException** オブジェクトで表されます。トランザクションの処理は中止されます。例外を引き起こした問題が解決されるまで結果は同じになるため、トランザクションはリトライされません。たとえば、メールの添付ファイルを読み取るプロセスで送信者がファイルを添付していなかった場合は、ビジネス ルール例外とみなされます。この場合、トランザクションを直ちにリトライしても、結果は変わりません。

その一方で、**BusinessRuleException** と異なる型の例外はシステム例外に分類されます。この種類の例外が発生したときは、プロセスに関連するアプリケーションを閉じてから再度開いた後で、トランザクション アイテムをリトライできます。この動作は、自動化対象のアプリケーションの問題 (システムのフリーズなど) で例外が発生したのであれば、アプリケーションの再起動で問題が解決する可能性があるという考えに基づいています。

トランザクションのソースが Orchestrator キューである場合は、**[トランザクションのステータスを設定]** アクティビティを使ってステータスを更新します。また、リトライ メカニズムも Orchestrator によって実装されます。

Orchestrator キューを使用しない場合は、スプレッドシートの特定の列に書き込むなどしてステータスを設定できます。このような場合、リトライ メカニズムはフレームワークにより制御されるので、リトライ回数は構成ファイルに定義します。

**SetTransactionStatus.xaml** ワークフローの最後に、**io\_TransactionNumber** がインクリメントされます。これにより、次に処理するトランザクションが取得されます。

**SetTransactionStatus.xaml** のその他の引数について詳しくは、表 8 をご覧ください。

表 8: SetTransactionStatus.xaml の引数

引数	説明	既定値
in_Config	構成データを格納するディクショナリ構造です。	既定値なし
in_SystemException	ステート間の遷移中に使用する例外変数です。	既定値なし
in_BusinessException	ステート間の遷移中に使用する例外変数です。	既定値なし
in_TransactionItem	処理対象のトランザクション アイテムです。	既定値なし
io_RetryNumber	この変数は、システム エラーの際に処理のリトライを試行する回数を制御します。	既定値なし
io_TransactionNumber	トランザクション アイテムのシーケンシャル カウンターです。	既定値なし
in_TransactionField1	トランザクション アイテムの情報をオプションで追加できます。	既定値なし

引数	説明	既定値
in_TransactionField2	トランザクション アイテムの情報をオプションで追加できます。	既定値なし
in_TransactionID	情報提供とログのために使用するトランザクション ID です。	既定値なし

## Framework¥RetryCurrentTransaction.xaml

**RetryCurrentTransaction.xaml** ワークフローは **Framework** フォルダーにあり、フレームワークのリトライ メカニズムを管理します。システム例外が発生すると **SetTransactionStatus.xaml** で呼び出されます。

リトライ方法は **Config.xlsx** に定義される構成に基づきます。「設定」で説明したとおり、**MaxRetryNumber** 定数の値が「0」に設定されている場合、リトライは Orchestrator で管理されます。**MaxRetryNumber** の値が「0」を超える場合、リトライはフレームワークによってローカルで管理されます。**RetryCurrentTransaction.xaml** の引数について詳しくは、表 9 をご覧ください。

表 9: RetryCurrentTransaction.xaml の引数

引数	説明	既定値
in_Config	プロセスの構成データ（設定、定数、およびアセット）を格納するディクショナリ構造です。	既定値なし
io_RetryNumber	システム例外の場合に、トランザクション処理のリトライ試行回数を制御するために使用します。	既定値なし
io_TransactionNumber	トランザクション アイテムのシーケンシャル カウンターです。	既定値なし

引数	説明	既定値
in_SystemException	ステート間の遷移中に、ビジネス例外以外の例外を表すために使用します。	既定値なし
in_QueryRetry	リトライ プロシージャを Orchestrator キューで管理するかどうかを示すために使用します。	既定値なし

## Framework¥TakeScreenshot.xaml

このワークフローは **Framework** フォルダにあります。画面全体のスクリーンショットをキャプチャし、拡張子を PNG として引数 **in\_Folder** (表 10) に指定されたフォルダーに保存します。

表 10: TakeScreenshot.xaml の引数

引数	説明	既定値
in_Folder	スクリーンショットを保存するフォルダーのパスです。	既定値なし
io_FilePath	撮影するスクリーンショットのパスと名前を指定するオプションの引数です。	既定値なし

トランザクションの処理中に例外が発生すると **TakeScreenshot.xaml** が呼び出されます。このワークフローは既定ですべてのプロセスで使用されますが、特に、ユーザーの操作を伴わないプロセスの実行中に発生する問題のデバッグに役立ちます。人間がロボットを監視せず、したがって問題が発生した時点で確認できない場合も、問題解決の手がかりが得られます。

## Framework¥CloseAllApplications.xaml

このワークフローは **Framework** フォルダにあり、プロセスを終了して使用されたアプリケーションを閉じるために必要なプロシージャを実行します。**OpenAllApplications.xaml** と

同様に、アクティビティを直接このワークフローに配置することも、サブワークフローを呼び出してシステムからログアウトするなど、より複雑なステップを実行することもできます。

## フレームワークを使用する

REFramework は UiPath Studio のプロジェクト テンプレートとして提供されています (図 8)。テンプレートを使用して作成されたプロジェクトには、フレームワークを構成するすべてのファイルが自動的に含まれます。

### テンプレートから新規作成



#### オーケストレーション プロセス

サービスのオーケストレーション、人間の介入、および実行時間の長いトランザクションによりプロセスを実装します。



#### バックグラウンド プロセス

ユーザーによる操作が不要でバックグラウンドで実行できるプロセスを作成します。同じロボットで複数のバックグラウンド プロセスを同時に実行できます。



#### Robotic Enterprise Framework

大規模展開のベスト プラクティスに準じたトランザクション ビジネス プロセスを作成します。



#### トリガー ベースの有人の自動化

マウスまたはキーボードのユーザー イベントにตอบสนองして自動化プロセスをトリガーします。



#### トランザクション プロセス

フローチャート ダイアグラムとしてプロセスをモデル化します。

図 8: UiPath Studio の [ホーム] 画面にあるテンプレート メニュー

## フレームワーク ファイルへの変更

プロジェクトを作成した後、自動化対象のプロセスの要件に応じて、次のファイルを変更する必要があります。

### Data¥Config.xlsx

プロセスごとに異なる、必要な設定、定数、およびアセットを追加する以外に、次の変更を加えます。

1. **logF\_BusinessProcessName** 設定の値を、プロセスの名前に合わせて変更します。  
この値はログの記録に使用され、このプロセスの実行時にフレームワークによって生成

されるすべてのログ メッセージに含まれます。

2. トランザクションのソースが Orchestrator キューである場合は、**OrchestratorQueueName** 設定の値を Orchestrator に定義されているキューの名前に合わせて変更します。プロセスでキューを使用しない場合は、この行を削除して **Constants** シートの **MaxRetryNumber** の値を「0」より大きい整数に変更すると安全です。これは、システム例外で失敗したトランザクションをロボットがリトライする回数を示します（詳しくは、「設定」をご覧ください）。

## Main.xaml

まず、プロセスのトランザクションの種類に従って、**TransactionItem** 変数の型を設定します。既定の型は **QueueItem** ですが、たとえば、行を Excel ファイルから読み取る場合は **DataRow** に、メール アカウントからメールを取得する場合は **MailMessage** に変更できます。

キューを使用する場合、さらに変更する必要はありません。ただし、型を変更した場合は、**GetTransactionData.xaml**、**Process.xaml**、および **SetTransactionStatus.xaml** ワークフローも更新する必要があります。これらのワークフローでは変数 **TransactionItem** の型を **QueueItem** と想定しているためです。このような更新の方法の例については、「実際の例 2: 表形式データを使用する」をご覧ください。

上記のワークフローを調整した後で、関連する **[ワークフロー ファイルを呼び出し]** アクティビティから渡す引数も更新する必要があります。**GetTransactionData.xaml** は **[トランザクション データを取得]** ステートで呼び出され、**Process.xaml** と **SetTransactionStatus.xaml** は **[トランザクションを処理]** ステートで呼び出されます。引数を更新するには、図 9 に示すように **[ワークフロー ファイルを呼び出し]** アクティビティの **[引数をインポート]** ボタンをクリックして、調整した引数に渡す変数を入力します。

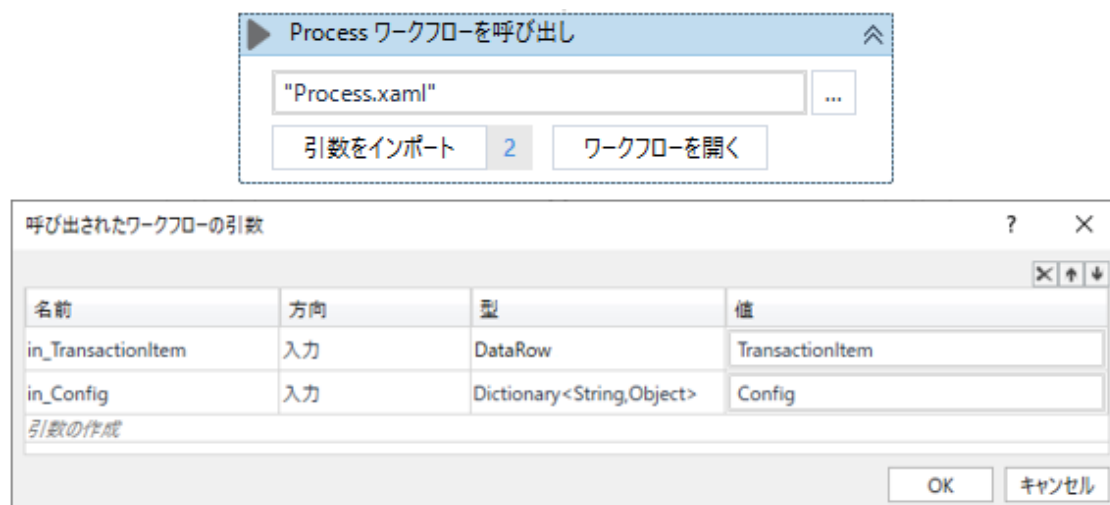


図 9: 呼び出すワークフローの引数の更新

## Framework¥GetTransactionData.xaml

Orchestrator キューを使用する場合、トランザクションは既定で含まれる **【トランザクション アイテムを取得】** アクティビティによって取得されます。また、**GetTransactionData.xaml** ワークフローを変更する必要はありません。

トランザクション アイテムの型が **QueueItem** 以外である場合は、**out\_TransactionItem** 引数の型をトランザクション アイテムの型 (**DataRow** や **MailMessage** など) に合わせて変更します。新しいデータ ソースを定義するには、このワークフローの最初のアクティビティである **【トランザクション アイテムを取得】** を、適切なデータ取得方法に置き換えます。たとえば、**【範囲を読み込み】** アクティビティを使用してスプレッドシートからデータを取得し、**io\_TransactionData** 引数に保存します。その後、**Main.xaml** の **【トランザクション データを取得】** ステート内でこのワークフローを呼び出す **【ワークフロー ファイルを呼び出し】** アクティビティに、新しい型の **out\_TransactionItem** が反映されていることを確認します。

データ ソースを定義したら、トランザクション アイテムを取得するステップを含める必要があります。トランザクションが 1 つしかない場合は、引数 **in\_TransactionNumber** の値が「1」である (そのトランザクションが最初で唯一のものである) かどうかを確認し、トランザクション アイテムを **out\_TransactionItem** に代入します。**in\_TransactionNumber** の値が「1」以外のときは、**out\_TransactionItem** を「Nothing」に設定します (図 10)。

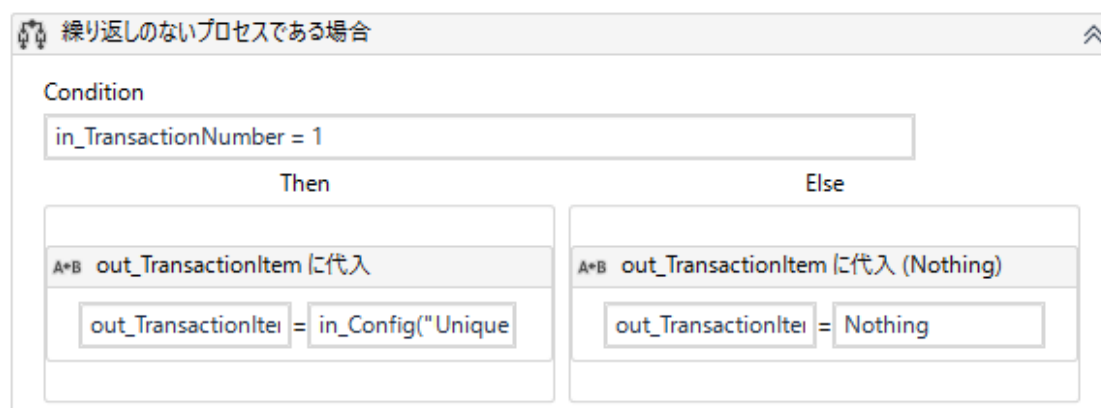


図 10: 単一のトランザクションを処理するプロセスの構成

トランザクションが複数ある場合は、引数 **in\_TransactionNumber** をインデックスとして使い、処理対象の適切なトランザクションを取得します。残りのトランザクションがなくなったら、**out\_TransactionItem** を「Nothing」に設定し、プロセスを終了します (図 11)。

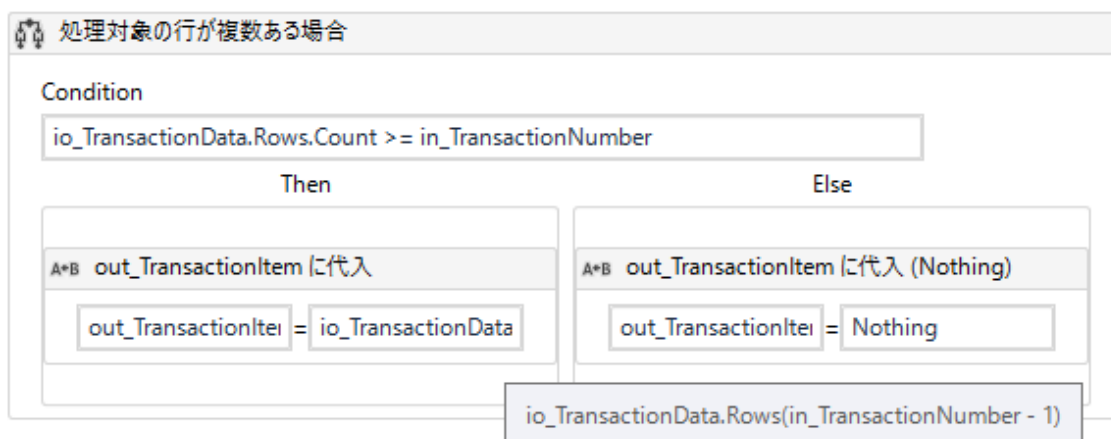


図 11: 複数のトランザクションを処理するプロセスの構成

オプションで、このワークフローの最後の **[トランザクション情報をログ フィールドに追加]** シーケンスで **[代入]** アクティビティを使用して、トランザクション アイテムの情報を追加できます。たとえば、請求書処理の自動化に関するレポートを作成するには、「ログ」で説明したように、**out\_TransactionID** に請求書番号を、**out\_TransactionField1** に請求日を、**out\_TransactionField2** に合計額を保存できます。図 12 をご覧ください。



図 12: カスタム ログ フィールドの構成

## Process.xaml

Orchestrator キューを使用する場合、**Process.xaml** に特別な変更を加える必要はありません。このワークフローでは、各トランザクション アイテムに **in\_TransactionItem** 引数を使用してアクセスできます。たとえば、請求書処理の自動化プロジェクトでは、「in\_TransactionItem.SpecificContent("InvoiceNumber")」で請求書番号を取得し、「in\_TransactionItem.SpecificContent("TotalAmount")」で合計額を取得することが考えられます。

Orchestrator キューを使用しない場合は、**in\_TransactionItem** 引数の型を **Main.xaml** の変数 **TransactionItem** の型に合わせて変更します。その後、**Main.xaml** の **[トランザクションを処理]** ステート内でこのワークフローを呼び出す **[ワークフロー ファイルを呼び出し]** アクティビティに、新しい型の **in\_TransactionItem** が反映されていることを確認しま

す。

## Framework¥SetTransactionStatus.xaml

「Framework¥SetTransactionStatus.xaml」で説明したとおり、このワークフローは **Process.xaml** ワークフローの実行後に呼び出され、処理ステップの結果に従ってトランザクションのステータスを設定します。

プロセスのデータ ソースが Orchestrator キューである場合は、キュー アイテムのステータスは既定で **【トランザクションのステータスを設定】** アクティビティによって更新されます。このワークフロー ファイルの変更は不要です。

Orchestrator キューを使用しないプロセスの場合、**in\_TransactionItem** 引数の型を調整するだけでなく、適切なステップを実装してトランザクションのステータスを設定する必要があります。その後、**Main.xaml** の **【トランザクションを処理】** ステート内でこのワークフローを呼び出す **【ワークフロー ファイルを呼び出し】** アクティビティに、新しい型の **in\_TransactionItem** が反映されていることを確認します。

トランザクションのステータスの追跡が不要な場合は、**in\_TransactionItem** 引数の型を **QueueItem** のままにして、**Main.xaml** の **【トランザクションを処理】** ステートの **【ワークフロー ファイルを呼び出し】** アクティビティの対応する引数に、値「Nothing」を渡すだけにすることができます。図 13 をご覧ください。

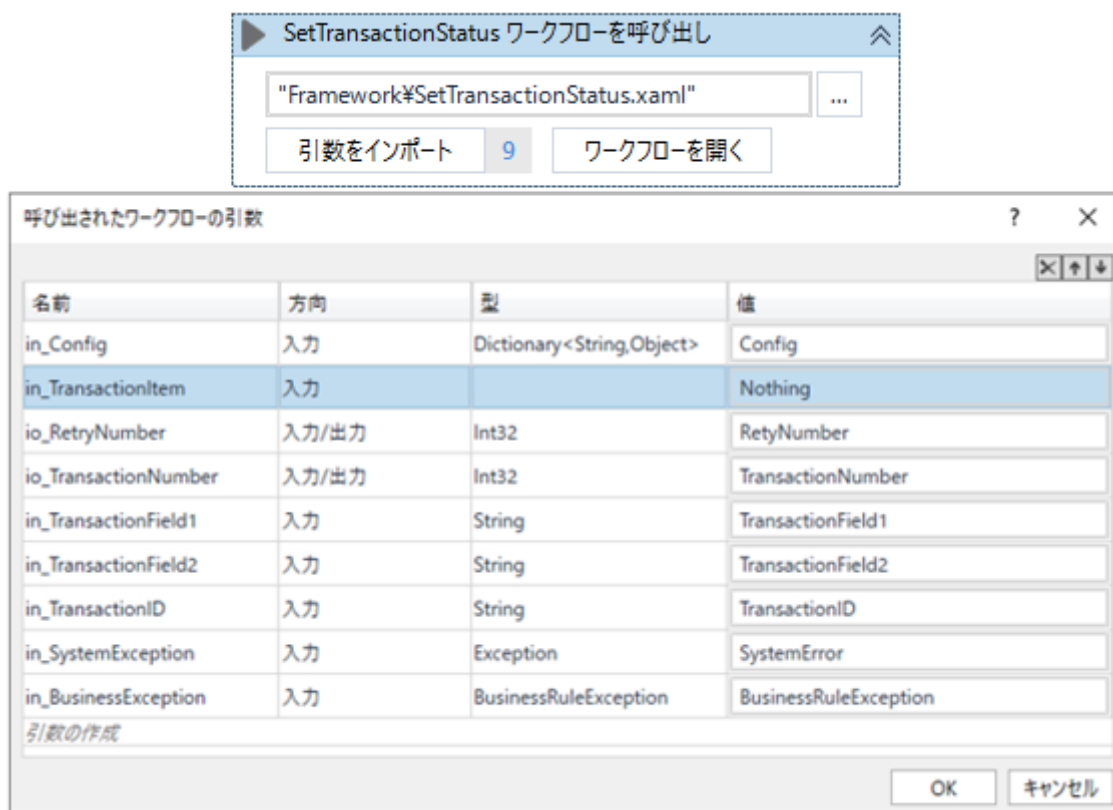


図 13: SetTransactionStatus.xaml を呼び出すときの引数の構成

## 実際の例 1: キューを使用する

最初の例では、Invoices（請求書）という名前の Orchestrator キューに、請求書番号、日付、および合計額などの請求書に関するデータが含まれている場合を考えてみましょう（図 14）。

詳細

コメント

履歴

トランザクション: d3a755cb-2d3f-42a4-a37f-382f88bca032

▼ 固有データ: オブジェクト

InvoiceNumber: 1551

InvoiceDate: Thu Sep 20 2018 09:00:00 GMT+0900 (Japan Standard Time)

InvoiceAmount: 26831

出力データ: 空

分析データ: null

状態	リビジ...	処理期限	リトラ...	延期	開始	終了	ロ...	例外
成功	なし	0			3ヶ月前	3ヶ月前	MrRobot	

項目 10

1 - 10 / 0

<<

<

>


>>

図 14: Invoices (請求書) キューに含まれるキュー アイテムの詳細

「フレームワーク ファイルへの変更」の説明に従って、請求書の処理を次の手順で実装できます。

1. **Data¥Config.xlsx** ファイルの **Settings** シートで、**OrchestratorQueueName** パラメーターの値を「Invoices」に変更し、**logF\_BusinessProcessName** パラメーターの値を「InvoiceProcessingSample」に変更します。
2. **InitAllApplications.xaml** ファイルで、プロセスで使用するアプリケーション（請求書登録システムなど）の起動とログインを実装するワークフローを呼び出します。
3. **CloseAllApplications.xaml** ファイルで、ログアウトと使用したアプリケーションの終了を実行するワークフローを呼び出します。必要に応じて、**KillAllProcesses.xaml** ファイルを使用して追加のクリーンアップ ステップを実行します。
4. **Process.xaml** ファイルで、実際の請求書処理ステップの実装に必要なワークフローを呼び出します。処理ステップには、登録システムの適切な画面にアクセスしたり、**[クリック]** や **[文字を入力]** などのアクティビティを使って各請求書を登録したりする処理が含まれます。

上の手順からわかるとおり、データ ソースとして Orchestrator キューを使用する場合は、フレームワークに必要な変更は多くありません。フレームワークは構成ファイル内のキュー セットと自動的に通信し、1 回に 1 つのトランザクション アイテムを取得し、処理の結果に従ってアイテムのステータスを更新します (図 15)。



キユー > Invoices

Default クラシック フォルダー

Default: キユー > トランザクション

検索

ステータス: 成功 リビジョン: すべて 優先: すべて ロボット: すべて レビュー担当者: すべて 例外: すべて

Reset to defaults

ステータス	参照	リビジョン	優先	処理期限	延期	開始	終了	ロボット	レビュー	例外
0 選択した行										
成功	なし	ノーマル				3ヶ月前	3ヶ月前	MrRobot		
成功	なし	ノーマル				3ヶ月前	3ヶ月前	MrRobot		
成功	なし	ノーマル				3ヶ月前	3ヶ月前	MrRobot		
成功	なし	ノーマル				3ヶ月前	3ヶ月前	MrRobot		
成功	なし	ノーマル				3ヶ月前	3ヶ月前	MrRobot		
成功	なし	ノーマル				3ヶ月前	3ヶ月前	MrRobot		

図 15: キユー アイテムの更新されたステータス

## 実際の例 2: 表形式データを使用する

2 番目の例では、図 16 に示すように、Excel スプレッドシートに保存されている請求書データを使用します。スプレッドシートの各行に 1 件の請求書に関するデータが含まれているため、データを **TransactionData** 変数に読み込み、**TransactionItem** の型を **DataRow** に変更する必要があります。そのため、次の変更を加えます。

	A	B	C	D
1	Number	Date	Total	Processed
2	1009	2018/04/05	2952	
3	8824	2018/10/21	64125	
4	3803	2018/01/04	80269	
5	6550	2018/01/10	91526	
6	4433	2018/10/01	28177	
7	3867	2018/04/04	17614	
8	7329	2018/05/31	99886	
9	4162	2018/07/30	65745	
10	5222	2018/08/09	46538	
11	8523	2018/04/26	49064	
12	5850	2018/09/19	98002	
13	9663	2018/04/05	45369	
14	3360	2018/12/12	2294	
15	1018	2018/06/10	55155	
16	4327	2018/09/05	46983	
17	1398	2018/08/07	6911	
18	9568	2018/05/16	94112	
19	2961	2018/12/13	43388	

図 16: スプレッドシートの請求書データ

- 最初の例のように、**Data¥Config.xlsx** ファイルの **Settings** シートで、**logF\_BusinessProcessName** の値を「InvoiceProcessingSample」に変更します。ただし、データソースが Orchestrator キューではないため、**OrchestratorQueueName** の行を削除します。新しい設定パラメーター **SampleDataFilepath** を追加し、その値として処理対象の請求データを含む入力 Excel ファイルのパスを指定します。たとえば、「Data¥Input¥InvoiceSampleData.xlsx」(図 17) です。

A	B
Name	Value
logF_BusinessProcessName	InvoiceProcessingSample
SampleDataFilepath	Data¥Input¥InvoiceSampleData.xlsx

図 17: Config.xlsx の Settings シート

- Data¥Config.xlsx** ファイルの **Constants** シートで、**MaxRetryNumber** の値を「0」より大きい整数に変更します。「設定」で詳しく説明したように、この値はシステム例外の発生時に処理をリトライする回数を示します。この例では「2」に変更します。
- Process.xaml** ワークフローで、引数 **in\_TransactionItem** の型を **QueueItem** から **DataRow** に変更します。また、実際の請求書処理ステップを実装するワークフ

ローを呼び出します。処理ステップには、登録システムの適切な画面にアクセスしたり、**【クリック】** や **【文字を入力】** などのアクティビティを使って各請求書を登録したりする処理が含まれます。

4. **GetTransactionData.xaml** ワークフローで **out\_TransactionItem** の型を **DataRow** に変更し、さらに、既存の **【トランザクション アイテムを取得】** アクティビティを削除します。この例では Orchestrator キューを使用しないためです。この場合、トランザクション アイテムを正しく取得するには 2 つのチェックが必要です。これらは次のように実装します。

- a. **【条件分岐】** アクティビティを追加して、データ ソースが「io\_TransactionData Is Nothing」の条件で初期化されたかどうかを確認します (図 18)。初期化されていない場合は、**【範囲を読み取り】** アクティビティを使用して、構成ファイルに定義されているパスで指定された Excel ファイルからスプレッドシートを読み取ります。このとき、パスは「in\_Config("SampleDataFilepath").ToString」で参照します。

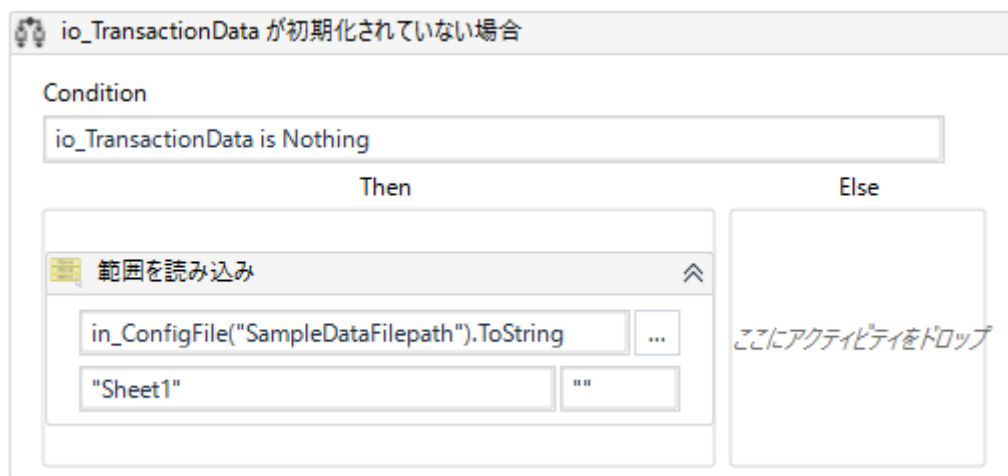


図 18: データ ソースの初期化 (io\_TransactionData)

- b. その後、**GetTransactionData.xaml** ワークフローが実行されるたびに 1 行を取得するロジックを実装する必要があります。これを行うには、別の **【条件分岐】** アクティビティを追加して、条件「io\_TransactionData.Rows.Count >= in\_TransactionNumber」を設定します。この条件によって、処理対象の行があるかどうかを確認します。未処理の行がある場合は **【代入】** アクティビティを使用して、適切な行を現在のトランザクション アイテムに設定します。

このとき、適切な行は「io\_TransactionData.Rows(in\_TransactionNumber - 1)」で参照します。引数 **in\_TransactionNumber** を使って現在処理中の行を追跡することに注意してください。未処理の行が残っていない場合は、**out\_TransactionItem** に「Nothing」を代入します (図 11)。フレームワークによる新しいトランザクションの取得の試行を防ぐため、「Nothing」を代入する必要があります。

5. **SetTransactionStatus.xaml** ワークフローを次のように変更して、トランザクションのステータスが、入カスプレッドシートの **Processed** 列で追跡されるようにします (図 16)。まず、**in\_TransactionItem** の引数の型を **QueueItem** から **DataRow** に変更します。処理の結果に従って入力 Excel ファイルの **Processed** 列を更新するため、次のステップを実装します。

- a. **[成功]** シーケンスで **[TransactionItem が QueueItem の場合 (成功)]** という名前の **[条件分岐]** アクティビティを削除し、代わりに **[セルに書き込み]** アクティビティを追加します。このアクティビティのプロパティで、**[テキスト]** プロパティに「"はい"」を、**[ブックのパス]** プロパティに「in\_Config("SampleDataFilepath").ToString」を、**[セル]** プロパティに「"D"+(io\_TransactionNumber+1).ToString」を設定します (図 19)。「D」は **Processed** 列を指します。「io\_TransactionNumber+1」によって表見出しをスキップし、正しい行に書き込みます。

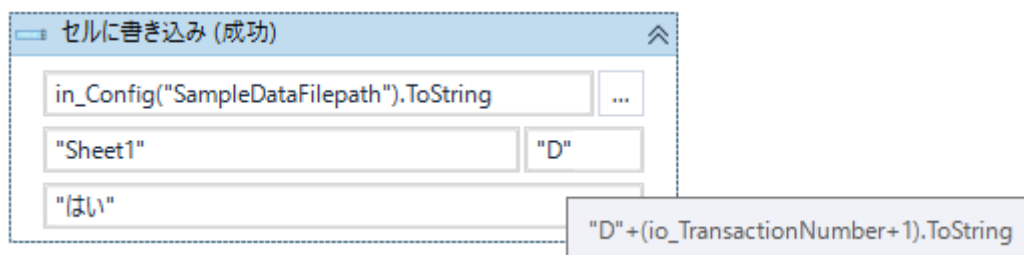


図 19: [セルに書き込み (成功)] アクティビティの構成

- b. 前の手順と同様に、**[ビジネス例外]** シーケンスで **[TransactionItem が QueueItem の場合 (ビジネス例外)]** という名前の **[条件分岐]** アクティビティを削除し、代わりに **[セルに書き込み]** アクティビティを追加します。このアクティビティのプロパティの値はトランザクションが成功した場合と同じですが、**[テキスト]** プロパティに「"いいえ (ビジネス ルール例外)"」を設定する必要があります。

- c. 最後に、**[システム例外]** シーケンスで **[TransactionItem が QueueItem の場合 (システム例外)]** という名前の **[条件分岐]** アクティビティを削除し、代わりに **[セルに書き込み]** アクティビティを追加します。ここでも、**[値]** プロパティを除き、トランザクションが成功した場合と同じプロパティ値を使用します。**[テキスト]** プロパティは「"いいえ (システム例外)"」に設定する必要があります。
6. **Main.xaml** ワークフローで、変数 **TransactionItem** の型を **QueueItem** から **DataRow** に変更します。この変更によって、ワークフローのさまざまな個所にアラートが表示されることに注意してください。これは、新しい型が **[ワークフロー ファイルを呼び出し]** アクティビティに定義済みの引数の型と互換性がないことを示しています。次の **[ワークフロー ファイルを呼び出し]** アクティビティの **[引数をインポート]** ボタンをクリックして、**TransactionItem** 変数を対応する引数に設定します。
  - a. **[トランザクション データを取得]** ステートの **[GetTransactionData ワークフローを呼び出し]** アクティビティ
  - b. **[トランザクションを処理]** ステートの **[Process ワークフローを呼び出し]** アクティビティ。これは、**[トライ キャッチ]** アクティビティの **[Try]** ブロックで実行します。
  - c. **[トランザクションを処理]** ステートの **[SetTransactionStatus ワークフローを呼び出し]** アクティビティ。これは、**[トライ キャッチ]** アクティビティの **[Finally]** ブロックで実行します。
7. **InitAllApplications.xaml** ワークフローで、プロセスで使用するアプリケーション (請求書登録システムなど) の起動とログインを実装するワークフローを呼び出します。
8. **CloseAllApplications.xaml** ワークフローで、ログアウトと使用したアプリケーションの終了を実行するワークフローを呼び出します。必要に応じて、**KillAllProcesses.xaml** ワークフローを使用して追加のクリーンアップ ステップを実行します。
9. **Process.xaml** ワークフローで、実際の請求書処理ステップを実装する必要なワークフローを呼び出します。処理ステップには、登録システムの適切な画面にアクセスしたり、**[クリック]** や **[文字を入力]** などのアクティビティを使って各請求書を登録したりする処理が含まれます。

要約すると、上記の手順では、Excel ファイルから請求書に関するデータを読み取り、ファイルの各行をトランザクションとして使用します。トランザクションの処理後、処理の結果（成功、ビジネス例外、システム例外）に応じて **Processed** 列を更新します。

最後に、メール (**MailMessage**) やファイルへのパス (**String**) など、他の種類のトランザクションにも同じステップを適用できることに注意してください。

## テスト フレームワーク

REFramework には、ワークフローの自動テストを容易にするテスト機能も含まれています。1 つずつテストして結果を手動で確認する代わりに、ワークフローの予測される結果（成功、ビジネス例外、およびシステム例外）を指定して、実際の結果と一致するかどうかを確認できます。

### 構成

テストの実行は自動化できますが、テスト自体の準備は重要なポイントです。

テストを準備する方法の 1 つは、引数に既定値を使用し、テスト ケースに応じて手動で規定値を変更することです。ただしこのアプローチでは、毎回手動で変更する必要があるためにさまざまな入力のテストが難しくなるだけでなく、デバッグしにくい問題が発生することがあります。たとえば、開発者が引数に既定値を設定したことを忘れて、引数を指定せずに **【ワークフロー ファイルを呼び出し】** アクティビティを使用した場合、ワークフローの実行結果は成功になりますが、実際の処理データではなく、既定値に基づく誤った値が返されることになります。

もう 1 つのアプローチは、テスト対象のワークフローを呼び出す補助的なテスト ワークフローを作成することです。引数として渡すさまざまな値をループを使って制御しながらワークフローを呼び出すことができるため、さまざまなケースを容易にテストできます。

### 使用例

前述の 2 番目のアプローチに基づいて、ACME System 1 (<https://acme-test.uipath.com/>) という Web システムにログインするワークフローのテストを作成する必要があるとします。ACME System 1 にログインするワークフローの実装例として、**System1\_Login.xaml** を図 20 に示します。

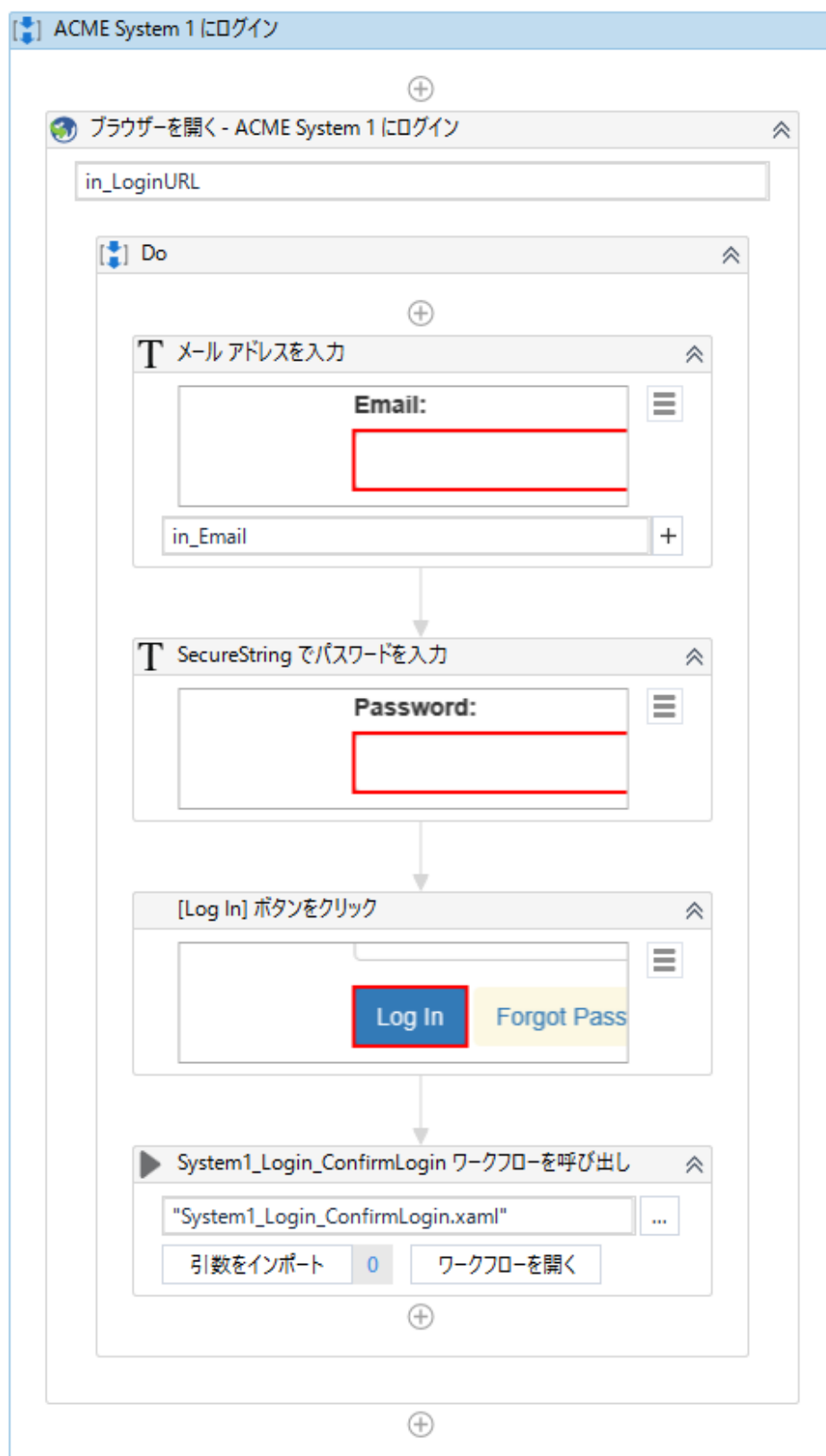


図 20: System1\_Login.xaml

「例外処理および例外からの回復」で説明したとおり、プロセスの通常の実行を妨げる状況を示すために **BusinessRuleException** 例外をスローするのは、開発者の責任です。

そのような例外処理のため、この例では新しいフローチャート ワークフローを作成して **System1\_Login\_ConfirmLogin.xaml** という名前を付け、図 21 に示すステップを追加します。ダッシュボード ページが見つからない場合は、エラー メッセージを示すダイアログを探します。ダイアログが存在する場合は渡した資格情報に問題があることを意味するため、**[クリック]** アクティビティによってダイアログが閉じられた後に **BusinessRuleException** をスローする必要があります。ログインに失敗したにもかかわらずエラー メッセージのダイアログが存在しない場合、**Exception** をスローしてシステム例外が発生したことを示します。

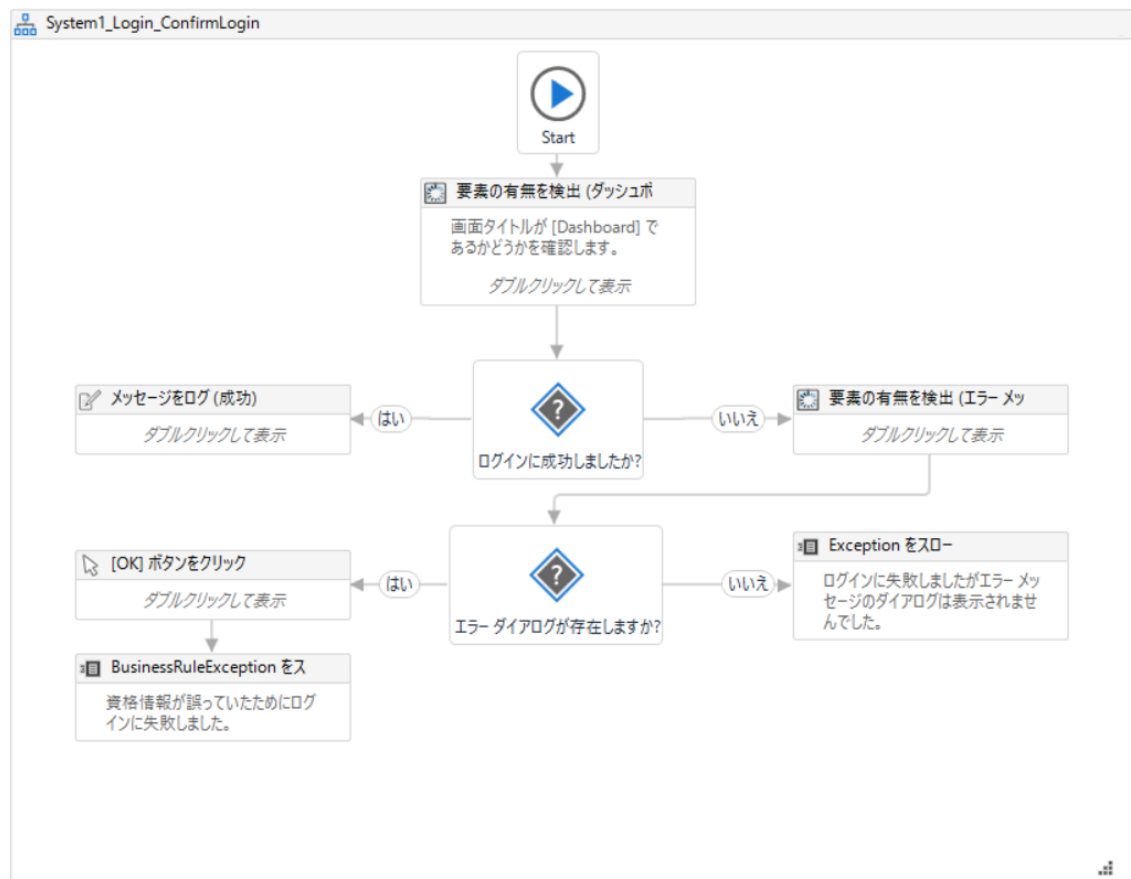


図 21: System 1 のログイン確認

最後に、図 20 の下部に示すとおり、**System1\_Login.xaml** で **[クリック]** アクティビティの後に **[ワークフロー ファイルを呼び出し]** アクティビティを配置して、**System1\_Login\_ConfirmLogin.xaml** を呼び出します。

**System1\_Login.xaml** ワークフローを作成したら、次のテスト ワークフローを追加します。

1. **System1\_Login\_Success\_Test.xaml**: ログインに成功したのでダッシュボードに遷移します。
2. **System1\_Login\_IncorrectCredential\_Test.xaml**: メールまたはパスワードが正しくないためログインに失敗しました。**BusinessRuleException** をスローします。
3. **System1\_Login\_SystemFailure\_Test.xaml**: サーバーまたはブラウザーの問題のためログインに失敗しました。

次に、**System1\_Login.xaml** を呼び出して望ましい結果を生成する引数を渡すことにより、テスト ワークフローを実装します。**System1\_Login\_Success\_Test.xaml** で引数として正しい資格情報を受け取ります (図 22)。**System1\_Login\_IncorrectCredential\_Test.xaml** で空の資格情報または誤った資格情報を受け取ります。**System1\_Login\_SystemFailure\_Test.xaml** をテストするには、Web システムの応答を遅延させたり **SelectorNotFoundException** などの例外を発生させたりする接続の問題など、システム例外を発生させる可能性のあるシナリオをエミュレートする必要があります。

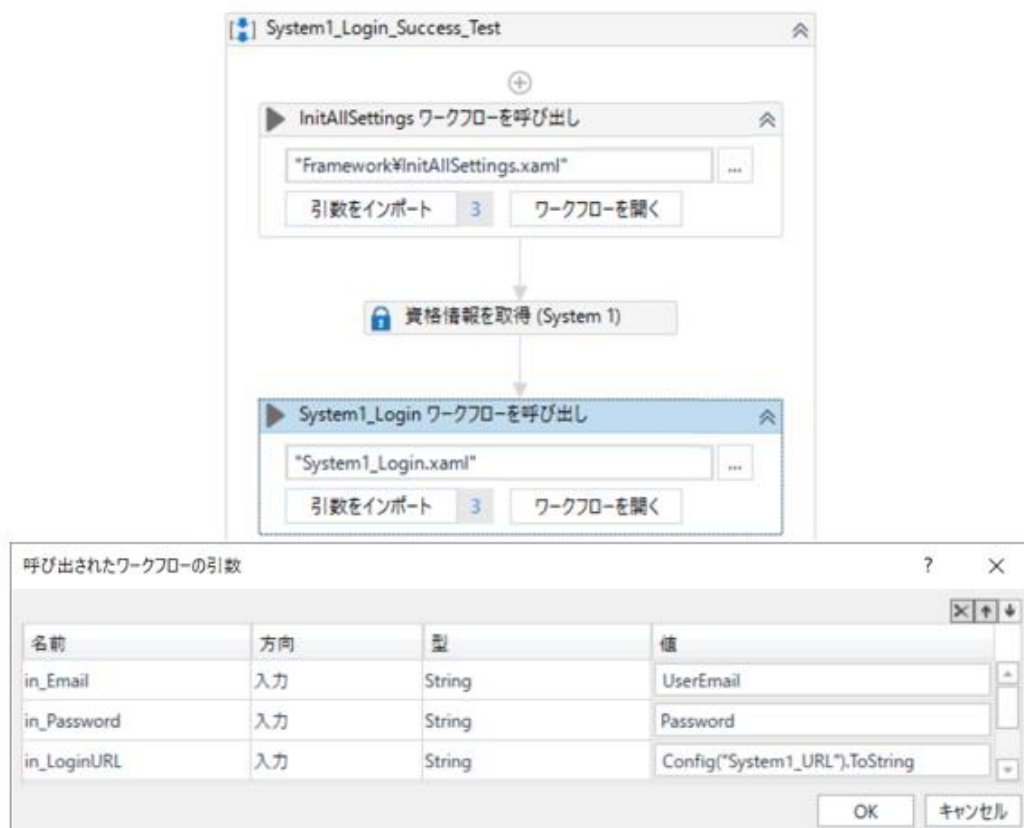


図 22: System1\_Login\_Success\_Test.xaml の実装

テスト ワークフローを作成したら、**Tests¥Tests.xlsx** (図 23) の **Tests** シートに追加します。このシートは、フレームワークによって実行されるテストの一覧です。テストを実装するファイルの名前以外に、**ExpectedResult** 列に、ワークフローの予測される実行結果として「Success」、「BusinessException」、または「SystemException」を入力します。

	A	B
1	<b>WorkflowFile</b>	<b>ExpectedResult</b>
2	Tests\System1\System1_Login_Success_Test.xaml	Success
3	Framework\CloseAllApplications.xaml	Success
4	Tests\System1\System1_Login_IncorrectCredential_Test.xaml	BusinessException
5	Framework\CloseAllApplications.xaml	Success
6	Tests\System1\System1_Login_SystemFailure_Test.xaml	SystemException
7	Framework\CloseAllApplications.xaml	Success

図 23: 実行するテストのリスト (Test¥Tests.xlsx の Tests シート)

テストの実行結果が後続のテストに影響しないように、システムを各テストの開始前の状態に戻す必要があります。この例でシステムを開始前の状態に戻すには、テストで開いたブラウザーのウィンドウを閉じる必要があります。そのため、**CloseAllApplications.xaml** ワークフローに **[ブラウザーにアタッチ]** アクティビティを追加し、その内部に **[タブを閉じる]** アクティビティを追加します。タブが閉じるように設定するには、ブラウザー ウィンドウを表示し、セクターを「<html title='ACME System 1\*'/>」に変更します。これにより、ログインに失敗したときに、ダッシュボード画面とログイン画面のどちらでもタブを閉じることができます。最後に、**Tests¥Tests.xlsx** ファイルにリストされている各テストの後に **Framework¥CloseAllApplications.xaml** を追加します。

その後、**RunAllTests.xaml** ワークフローを実行すると、定義済みのテストを自動的に実行できます。結果は **Tests.xlsx** の **Results** シートに書き込まれます (図 24)。さらに TestLog.txt ファイルが開き、テストの実行ログが表示されます。

	A	B	C	D
1	<b>WorkflowFile</b>	<b>ExpectedResult</b>	<b>Status</b>	<b>Comments</b>
2	Tests\System1\System1_Login_Success_Test.xaml	Success	PASS	
3	Framework\CloseAllApplications.xaml	Success	PASS	
4	Tests\System1\System1_Login_IncorrectCredential_Test.xaml	BusinessException	PASS	
5	Framework\CloseAllApplications.xaml	Success	PASS	
6	Tests\System1\System1_Login_SystemFailure_Test.xaml	SystemException	PASS	
7	Framework\CloseAllApplications.xaml	Success	PASS	

図 24: テスト結果 (Test¥Tests.xlsx の Results シート)

**Tests.xlsx** に指定した順序でテストが実行されることに注意してください。これは、たとえば特定のアプリケーションとの相互作用をテストする必要があるが、そのアプリケーションを開くワークフロー (**InitAllApplications.xaml** など) を以前に実行していない場合に影響があります。そのような場合セクターが見つからないため、システム例外になる可能性があります。



最後に、同じワークフローを複数回テストする場合は、**Tests.xlsx** にそのテストを繰り返し指定する必要があることに注意してください。

## 拡張機能の配布とサポート

REFramework は MIT ライセンスの下で利用可能です。テンプレートとして UiPath Studio または <https://github.com/UiPath/ReFrameWork> 経由で配布されます。

お客様およびパートナー企業様におかれましては、個々のユース ケースや特定の種類のトランザクションに対してフレームワークを適用する際、拡張機能の手順をご理解のうえ、ユーザー様のニーズに合わせた変更を実装されるようお願いいたします。スプレッドシートのデータを使用した拡張例については、「実際の例 2: 表形式データを使用する」をご覧ください。また、REFramework をベースとするテンプレートを UiPath Go! からダウンロードできます (<https://go.uipath.com/>)。