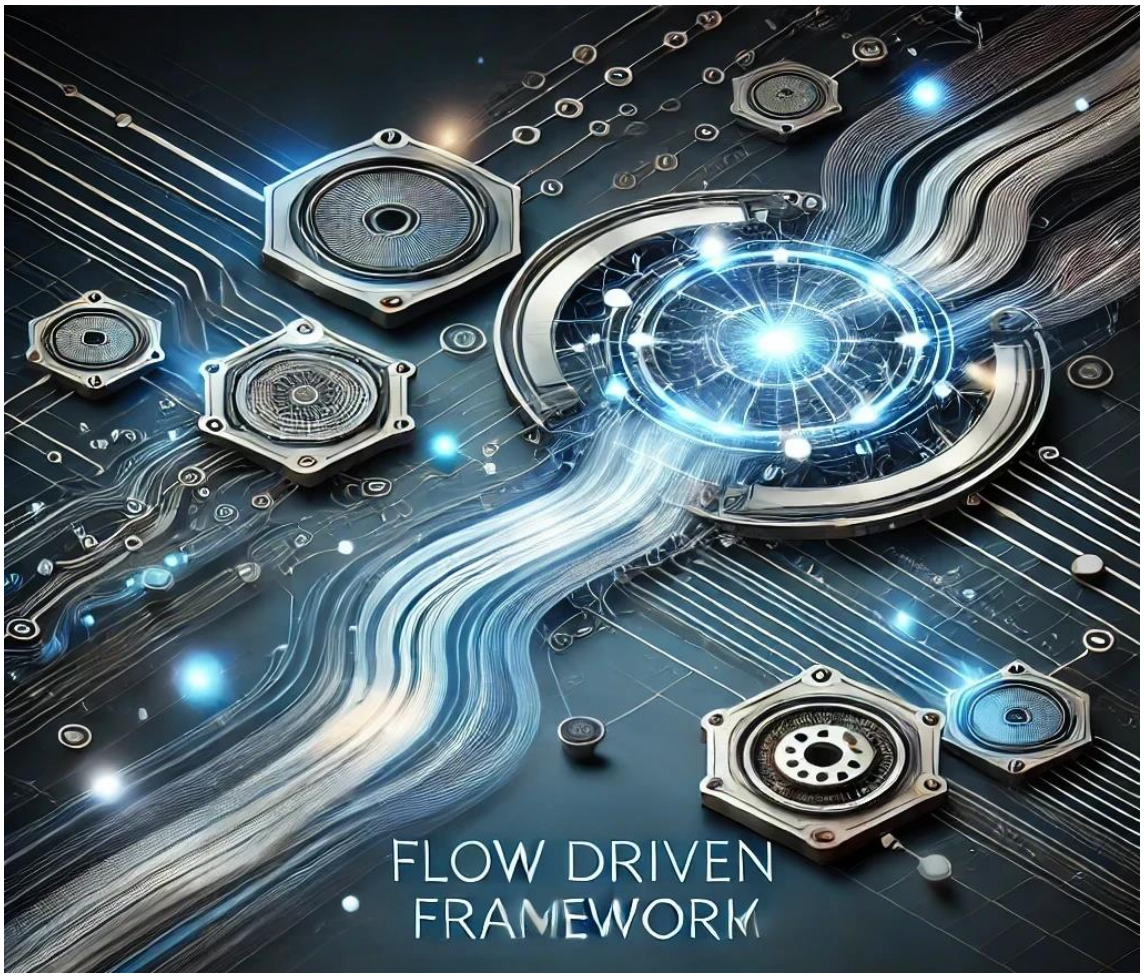

FLOW DRIVEN
FRAMEWORK

Contents

Overview

The **Flow Driven Framework** is a template to streamline the development of automation processes using UiPath. Inspired by UiPath's official **Robotic Enterprise Framework (REFramework),** it introduces enhanced capabilities to provide a structured approach for building automations by organizing business logic, data management, and reusable components into a clear and maintainable architecture. The **Flow Driven Framework** integrates dispatcher (producer) and performer (consumer) flows within a single workflow, simplifying the development process and reducing complexity.

Additionally, the framework features centralized exception handling, allowing developers to manage and respond to exceptions from any part of the workflow in a unified manner. The **Flow Driven Framework** also includes predefined workflows for tracking Key Performance Indicators (KPIs), providing built-in support for monitoring and improving process efficiency.

Key Features

- **Inspired by REFramework:** Built on the principles of the REFramework, with enhancements like integrated dispatcher and performer flows, and improved exception handling.
- **Integrated Dispatcher and Performer:** Combines dispatcher and performer flows into a single workflow for streamlined automation.
- **Customizable Execution Modes:** Allows the flow to be executed as a producer, consumer, or a hybrid of both, catering to diverse project requirements.
- **Centralized Exception Handling:** Unified approach to managing exceptions across all parts of the automation, enhancing reliability and maintainability.
- **Predefined KPI Workflows:** Built-in support for tracking and improving process performance through KPIs.
- **Global Configuration Management:** Utilizes a global config variable, reducing the need to import arguments across workflows and simplifying configuration management.
- **Native Application Closure Handling:** Automatically handles the closure of applications in case of errors, ensuring that resources are properly released and maintaining system stability.
- **Modular Design:** Promotes easy management and maintenance of complex automation projects through a clear separation of concerns.

- **Reusable Components:** Centralized management of reusable components like logging, error handling, and data management.
- **Scalability and Extensibility:** Designed to handle both simple and complex automation processes, making it suitable for a wide range of use cases, with the ability to extend as new requirements arise.

Use Cases

- **Require High Customization and Flexibility:** Suitable for automations that need to adapt to varying business requirements and workflows.
- **Integrate Dispatcher and Performer Flows:** Beneficial for projects that can leverage the combined producer and consumer flows within a single workflow.
- **Serve Simple (Linear) Projects:** Equally effective for straightforward automation tasks, providing a scalable solution that can grow with project complexity.
- **Involve Complex Business Logic:** Ideal for automations that need to separate business logic from other functionalities, enhancing clarity and maintainability.
- **Need Robust Error Handling:** Provides centralized exception management to ensure consistent and reliable error responses across the automation.
- **Require KPI Tracking and Optimization:** Equipped with predefined workflows for monitoring and improving process performance through KPIs.
- **Benefit from Global Configuration Management:** Simplifies configuration handling by using a global config variable, reducing the need for repetitive argument imports.
- **Demand Reliable Application Management:** Ensures that applications are properly closed in case of errors, maintaining system integrity and resource management.

Use Cases for Different Workflow Types

- **Linear Process Flow (Producer Flow)**
  A **linear process flow** is ideal for straightforward automation tasks that do not require complex branching or multiple workflows. In this scenario, the automation process follows a direct path from start to finish, making it suitable for processes where the sequence of actions is predetermined and does not vary.
    - **Example Use Case:**

An automation that generates a report based on predefined data. The automation retrieves the necessary information, formats it according to the report template, and outputs the final document in the desired format (such as PDF, Word or Excel). Since this process is straightforward and doesn't involve transaction processing or complex decision-making, the entire operation is managed within a linear process.

- **Producer Flow:**
  This type of flow handles the entire sequence of actions in a direct manner, from data extraction to processing and output. It is a streamlined approach for processes that do not require parallel processing or interaction with a queue.

- **Dispatcher Flow (Producer Flow)**
  The **dispatcher flow**, also known as the **producer flow**, is responsible for identifying and adding items to a processing queue. This flow is essential for automations that require processing multiple items in parallel or over a period of time, rather than handling them all in a single batch.
  - **Example Use Case:**
    An automation that scans emails for invoices and adds each invoice to a queue for further processing. The dispatcher identifies relevant emails, extracts the necessary data, and places each item in a queue for subsequent handling by a performer (consumer) process.
  - **Producer (Dispatcher) Flow:**
    The dispatcher identifies work items (such as transactions, documents, or tasks) and feeds them into a queue. This queue acts as a buffer, allowing for scalable processing where items are consumed and processed as resources become available. This flow is critical in scenarios where data or tasks need to be processed asynchronously or in a distributed manner.
- **Performer Flow (Consumer Flow)**
  The **performer flow**, also known as the **consumer flow**, is responsible for processing items from a queue. This flow consumes the items produced by the dispatcher, performing the necessary operations on each item in the queue.
  - **Example Use Case:**
    An automation that retrieves invoices from a queue and processes each one individually, such as by extracting data, updating records in a database, or generating reports. The performer handles each item in turn, ensuring that all tasks in the queue are completed efficiently.
  - **Consumer (Performer) Flow:**

The performer retrieves items from the queue and processes them according to predefined business rules. This flow is essential for handling tasks that have been queued by the dispatcher, ensuring that each item is processed correctly and in the order it was received.

Main Features

In addition to enabling **transactional processing**, the **Flow Driven Framework** offers several features that make it highly effective for building **stable** and **scalable** automation projects. Key features include:

- **Settings Management:**
  The framework allows the use of a global config variable, reducing the need to import arguments in every workflow. This ensures that configuration settings, such as file paths, credentials, and other parameters, are globally available and easy to manage across the entire project.
- **Logging:**
  Built-in logging mechanisms capture detailed information at every stage of the process, making it easier to monitor automation performance and identify issues. Logs can be customized to include specific data points, which helps in tracking the status of workflows and debugging when necessary.
- **Centralized Exception Handling:**
  The framework features a unified approach to exception handling. This ensures that errors occurring in any part of the process are handled gracefully and that appropriate actions, such as retries, notifications, or application shutdowns, are taken. By centralizing error handling, the framework ensures consistent and predictable responses to issues across all workflows.
- **Customization:**
  The Flow Driven Framework is highly customizable to meet various project requirements. It allows the execution flow to be tailored to function as a producer (dispatcher), consumer (performer), or a hybrid of both, depending on the needs of the automation. This flexibility makes it adaptable for both simple and complex workflows.
- **Transactional Processing:**
  Similar to the **REFramework**, the **Flow Driven Framework** supports transactional processing, allowing for handling of individual items or tasks in an automation. This is particularly useful for managing large volumes of work

that can be processed in parallel or sequentially, ensuring scalability and reliability.

- **KPI Tracking:**
  The framework includes predefined workflows for tracking **Key Performance Indicators (KPIs)**. This enables easy monitoring of automation performance and efficiency, allowing stakeholders to analyze and improve the process based on key metrics.
- **Application Closure Handling:**
  The framework has native functionality for **automatically closing applications** in case of errors. This ensures that resources are properly released and that systems return to a stable state even when unexpected issues occur, preventing potential disruptions in subsequent automation runs.

Settings

The **Flow Driven Framework** uses a centralized configuration file (Main.yml) to manage key settings and control how the automation behaves. This YAML-based configuration allows for clear organization of settings, making it easier to customize and manage automation projects across different environments. Below are the key sections and their purposes, based on the Main.yml file.

- Flow Configuration:
  The Flow setting determines the execution mode of the automation process. This can be customized to fit different workflow scenarios:
  - **Producer**: For dispatcher flows that add items to queues.
  - **Consumer**: For performer flows that process items from queues.
  - **Hybrid**: Combines both producer and consumer workflows in a single execution flow.

  This flexibility allows the automation to adapt to various project requirements, whether simple or complex.

```
# Determines the execution order of the states on the state machine.
# Can be Producer, Consumer or Hybrid (combines Producer and Consumer)
Flow: Hybrid
```

- **Orchestrator Folder:**
  The OrchestratorFolderName setting defines the folder in UiPath Orchestrator that the process will use. This helps manage assets, queues, and other resources within a specified Orchestrator folder.
  This setting is particularly useful when running automations in different environments, ensuring the correct folder is used during execution.

```
# Define the Orchestrator folder that the process will use.
# This setting will be used by default with Orchestrator Queue and with the Assets section that don't have a folder defined.
# This setting can be helpful when using studio in case the setted folder isn't the correct one.
#OrchestratorFolderName: FolderName
```

- **Process and Application Termination Settings:**
  The ShouldEndWithSuccess and ShouldThrowIfRunning settings control how the automation handles the final state and the closure of applications:
  - **ShouldEndWithSuccess**: Determines if the process should finish with a success status even if it encountered errors.
  - **ShouldThrowIfRunning**: Validates whether processes specified in ProcessesToKill should be checked after it executes the flow to close the appications.

```
# Define if the process should end the execution with a success status.
# Even if the process was not successfully runned.
ShouldEndWithSuccess: False

# Define if the CloseAllApps should validate if the programs in the ProcessesToKill area are running
ShouldThrowIfRunning: True
```

- **Queue Configuration:**
  The Queue section manages how the process interacts with transaction queues. It supports different queue types such as Orchestrator, CSV, or Excel, making it flexible for various data sources:
  - **Type:** Defines the type of queue used (Orchestrator, CSV, Excel).
  - **Name:** Specifies the queue name or file path.
  - **Sheet:** Relevant only for Excel queues, specifying the worksheet to be read.

```
# The queue settings
Queue:

    # Defines the queue type, it can be Orchestrator, CSV or EXCEL
    # By default, CSV uses semicolon ";" as separator, you can change this on Framework\Transactions\DataSources\GetFromCSV.xaml file
    Type: CSV

    # The queue name or the path (absolute or relative) to the CSV (*.csv) or EXCEL (*.xlsx/*.xls)
    Name: Data\Sample.csv

    # Required only for EXCEL Type to specify what sheet name must be read
#    Sheet: Sheet1
```

- **Retry and Failure Management:**
  This section cover how the Flow Driven Framework handles the exceptions. For more information see the chapter **Recovery and Retry Mechanism**.

```
# Settings regarding the behavior for local retries or process stop.
Recover:

    # The maximum number of local retries for the same transaction (Consumer phase only).
    # The local retry count will not be used with Orchestrator queue.
    MaxRetriesCount: 0

    # The maximum number of fails accepted before stop the process.
    MaxFailsCount: 2

    # Determines how the internal fails counter must be managed
    # - Default = Always reset after a successful transaction or flow change
    # - Balance = Decrements the fails counter for each successful transaction or reset it in flow change
    # - Never = Never resets the fails counter except in flow change
    FailsResetMode: Default
```

- **Merge Configuration:**

The Merge section allows for combining multiple configuration files, making it easier to manage large-scale automations by splitting configurations into logical segments. Sub-sections of the configuration are merged, with subsequent files overwriting earlier ones.

This feature enables the automation to handle modular configurations, improving maintainability and scalability.

```yaml
# List of configuration files to merge with this one.
# Each configuration file overwrites the values of the previous one. (Sub-sections are merged)
# All the configurations available on this file can be splitted in different files. The only requirement for this file is the Merge section.
Merge:
    - Framework\Config\Logs.yml
#    - AbsoluteOrRelativePathToYourFile.yml
```

- **Process and Asset Management:**
    - **ProcessesToKill:** Defines a list of applications that should be closed during the automation initialization and termination (e.g., web browsers or specific software).
    - **Assets:** Specifies assets to be loaded from Orchestrator during initialization.
    - **Credentials:** Specifies credentials to be loaded from Orchestrator when needed. This helps manage sensitive information and resources securely.

  These sections centralize the management of processes and secure resources, making it easier to configure and scale the automation as needed.

```yaml
# List of processes to kill in KillAllProcesses workflow.
#ProcessesToKill:
#    - msedge

# -----------------------------------------------------------------------------

# List of assets to load automatically from Orchestrator at initialization phase.
#Assets:
#    AssetName1: Orchestrator_AssetName
#    AssetName2: Orchestrator_AssetName
#    AssetName3: Orchestrator_AssetName

# -----------------------------------------------------------------------------

# List of credentials that will be used.
# The credentials will not be loaded automatically.
# This section is used to organize the credentials in a section.
#Credentials:
#    CredentialName: StoredCredentialName
```

- Screenshot Folder:

  The ScreenshotsFolder setting defines the location where screenshots will be stored during execution, typically used for logging or error reporting.

```yaml
# Folder to store the screenshots taken by TakeScreenshot workflow
ScreenshotsFolder: Data\Screenshots
```

Benefits of YAML-Based Settings Management

- **Modular Configuration:** The Main.yml file supports modular and scalable configurations by allowing settings to be split across multiple files.
- **Centralized Control:** With global settings like queues, retries, and process management, the automation is easier to control and adapt for different environments.
- **Flexible Queue Management:** Supports multiple queue types (Orchestrator, CSV, Excel) for flexibility in handling transactional data.
- **Simplified Error Handling:** Offers granular control over retries, failures, and process behavior in case of errors, enhancing robustness.
- **Simpler:** Simpler way to manage the configuration file.

Logging

Effective logging in an automation project provides multiple benefits, such as improved visibility into actions and events, easier debugging, and more meaningful auditing.

The Flow Driven Framework includes a robust logging structure that leverages various logging levels to track transaction statuses, exceptions, and transitions between different states. This approach ensures that important events are captured and can be reviewed for both real-time monitoring and historical analysis.

Key features of the logging system include:

- **Log Message Levels:** The framework uses different levels of the Log Message activity, such as Trace, Info, Warning, Error, and Fatal, to provide insights into the state of the automation process. Each log level serves a specific purpose, from general transaction tracking to detailed error reporting.
- **Transaction Status Logs:** Logs are generated at each stage of the transaction lifecycle, helping track whether transactions have succeeded, failed, or retried.
- **Exception Handling Logs:** When exceptions occur, detailed logs are produced to capture the error's context and source, aiding in faster debugging and resolution.
- **State Transition Logs:** Transitions between different states of the workflow are also logged, providing a clear view of how the process moves through its execution phases.

Most of the log messages consist of static parts that are predefined in the Logs.yml file. This file contains templates for common log entries and is read through the

Merge section of the Main.yml file, ensuring consistent and centralized management of log content.

Note that sensitive data should not be included in logs, since they are not encrypted and might lead to privacy issues if leaked.

```yaml
# Add your custom logs here, e.g.:
# BR01_PriceValue: The price must be greater than zero

# As alternative, you can create your own BusinessLogs.yml (e.g) file an reference it on Main.yml under Merge section:


# SYSTEM RESERVED!
# The below log messages keys prefixed with "Framework_" should be preserved due they are used by the framework.

# Static part of logging message. Calling Get Transaction Data.
Framework_GetTransactionData: Processing transaction number {0}

# Static part of logging message. No Data transition.
Framework_NoMoreData: Process finished due to no more transaction data.

# Static part of logging message. Error retrieving Transaction Data.
Framework_GetTransactionDataError: Error getting transaction data for transaction number {0}. {1} at Source {2}

# Static part of logging message. Processed Transaction succesful.
Framework_Success: Transaction Successful

# Static part of logging message. Processed Transaction failed with business exception.
Framework_BusinessRuleException: Business rule exception - {0}

# Static part of logging message. Processed Transaction failed with application exception.
Framework_ApplicationException: System exception - {0} at Source {1}

# Static part of logging message. Stop process requested from Orchestrator or Assistant.
Framework_StopProcessRequested: Stop process requested

# Static part of logging message. Maximum number of fails reached.
Framework_MaxFailNumberReached: Maximum number of fails reached. Stopping the process.

# Static part of logging message. The process was executed successfully
Framework_SuccessfulExecution: The process was executed successfully

# Static part of logging message. CSV or Excel read exception.
Framework_TabularQueueTypeException: Ensure to configure the activity options correctly to work with your tabular data queue

# Static part of logging message. State transition not recognized.
Framework_TransitionException: Transition from {0} to {1} not configured.
```

Exception Handling and Recovery

The **Flow Driven Framework** provides a robust exception handling mechanism designed to automatically manage failures, update transaction statuses, and handle unrecoverable exceptions in a structured way. This functionality is closely tied to logging, ensuring that all exception-related events are captured for easy analysis and debugging.

Exception Types

In the **Flow Driven Framework**, exceptions encountered during execution are categorized into two types:

- **Business Exceptions:** These exceptions arise when the business logic of the process cannot be fulfilled. For instance, if an expected file is missing, or input data is invalid, a business exception is raised. These exceptions need to be explicitly thrown by the developer using the Throw activity, typically with

a **BusinessRuleException**. They represent scenarios where the automation cannot proceed due to a violation of business rules.

  - **Example:**
    If an invoice processing workflow expects a required field but the field is missing, a BusinessRuleException can be thrown to signal the error and halt further processing of the transaction.

- **System Exceptions:** System exceptions occur when the process encounters an error in the underlying infrastructure, such as a network timeout, file access issue, or an unexpected application crash. These exceptions are not related to the business rules and are often recoverable through retries.
  - **Example:**
    A system exception could be triggered by a network error when attempting to download data or if a browser crashes while automating a web-based process.

State Machine FAILURE

The **FAILURE** state in the Main.xaml file plays a critical role in handling system and business exceptions. When an exception occurs, the flow transitions to the FAILURE state, which performs the following key actions:

- **Logging the Exception:** The FAILURE state captures detailed information about the exception, including the type of error, where it occurred, and the transaction that triggered it. This information is logged for analysis, making it easier to identify the root cause of the issue and improve the process in future runs.
- **Transaction Status Update:** Depending on the type of exception, the FAILURE state updates the transaction status:
  - **Business Exceptions:** The transaction is marked as failed without retry, as these errors require human intervention or business rule adjustments.
  - **System Exceptions:** The framework determines whether the transaction should be retried based on the configuration in the Main.yml file, such as MaxRetriesCount. If the retries are exhausted, the transaction is marked as failed.
- **Error Handling Logic:** The FAILURE state does not attempt to close any running applications (this responsibility is handled elsewhere in the framework, such as in specific workflows designed for application management). Instead, it focuses on ensuring proper error logging, updating transaction statuses, and determining whether further retries or error escalation is needed.

Recovery and Retry Mechanism

The framework includes configurable recovery mechanisms that determine how the process should respond to system exceptions and failures:

- **MaxRetriesCount:** Defines the maximum number of retry attempts for a transaction in the event of a system exception. If the transaction continues to fail after reaching this limit, it is considered unrecoverable and marked as failed.
- **MaxFailsCount:** This setting specifies the maximum number of allowed failures during the entire automation process. Once this limit is reached, the process stops to prevent further transactions from being processed under faulty conditions.
- **FailsResetMode:** Controls how the framework resets the failure counter after successful transactions:
    - **Default:** Resets the counter after a successful transaction or when switching between states (such as from Producer to Consumer).
    - **Balance:** Decreases the failure counter incrementally for each successful transaction.
    - **Never:** The failure counter is only reset when changing states, keeping track of failures consistently across multiple transactions.

Business vs. System Exception Handling

The framework handles business and system exceptions differently:

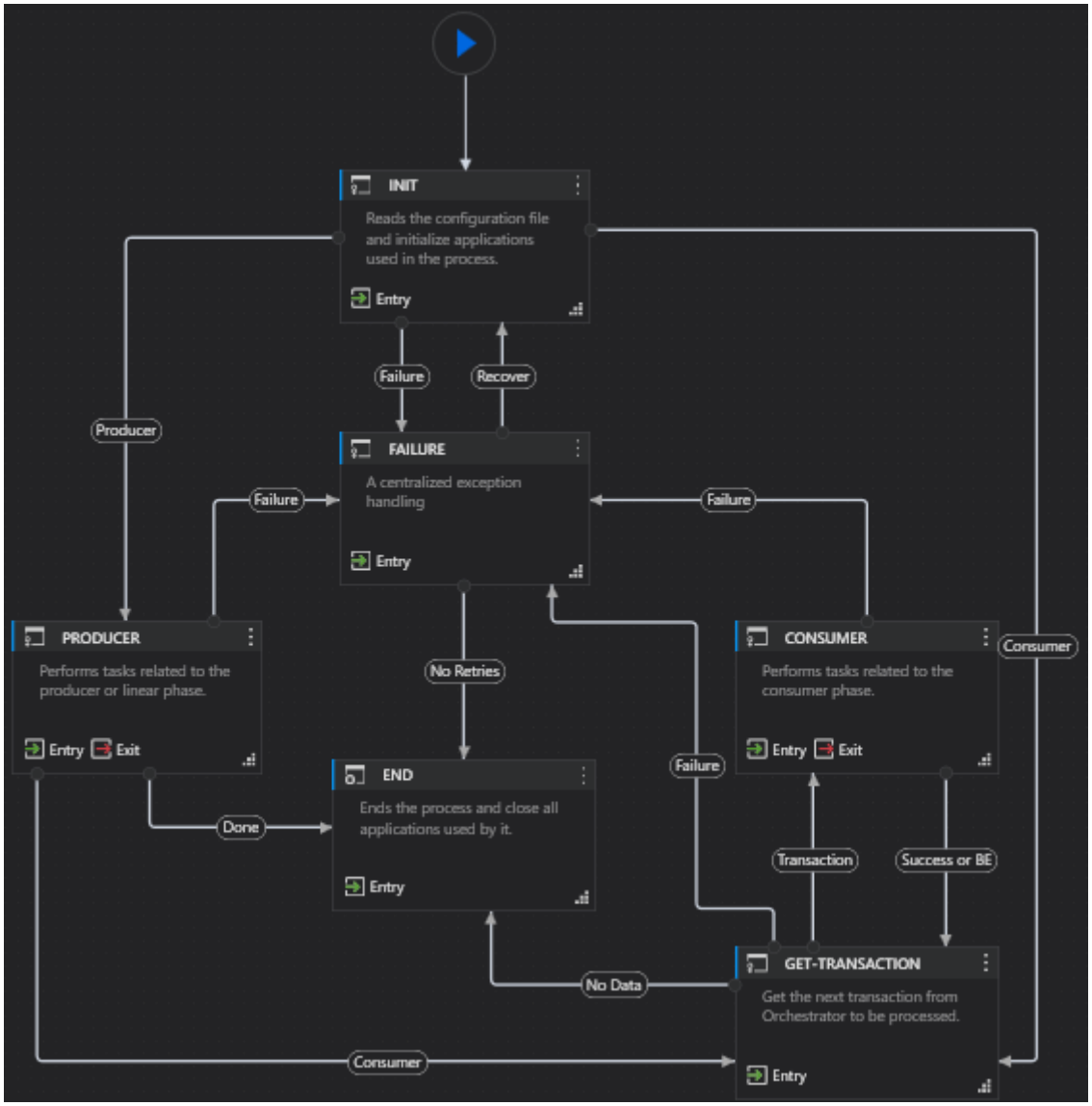
- **Business Exceptions:** These exceptions indicate a violation of business rules and do not trigger retries. They are logged and the transaction is marked as failed, as manual intervention or adjustments to the process logic are typically required.
- **System Exceptions:** These are often recoverable through retries. If a system exception occurs (e.g., a timeout or file access error), the framework attempts to retry the transaction up to the configured limit. If the error persists, the transaction is marked as failed.

Exception Handling Customization

The **Flow Driven Framework** offers flexibility in how exceptions are handled through configuration in the Main.yml file. This allows developers to:

- Adjust retry logic and failure thresholds.
- Log additional information about exceptions.
- Customize application and process closures in the event of unrecoverable failures.

Architecture



INIT State

The INIT state is responsible for initializing the process, reading configuration files, and preparing the environment. This includes loading necessary settings and starting applications.

- **Resets the BotException Variable:** Resets any prior exceptions to ensure a clean start.
- **Reads Configuration Files:** If this is the first run (when g_Config is Nothing), the state loads configuration settings from the specified YAML or Excel file using workflows such as InitAllSettings.xaml and UpdateSettings.xaml.
- **Starts Applications:** Calls workflows to initialize and start all required applications (via StartAllApps.xaml).

- **Error Handling:** If an error occurs, it will go to the FAILURE State to handle the failure accordingly.
- **Transitions:** Based on the outcome, the state transitions to either the PRODUCER or CONSUMER state, depending on the flow mode (e.g., Hybrid, Producer, or Consumer).

PRODUCER State

The PRODUCER state handles tasks related to adding items to a queue or processing transactions in a linear flow.

- **Stop Check:** First, it checks if a stop signal was received. If so, it logs the event and ends the process.
- **Executes Producer Workflows:** If no stop signal is received, the state attempts to execute the producer flow by invoking the Producer.xaml workflow.
- **Error Handling:** If an error occurs, it logs the exception and transitions to the FAILURE state.
- **State Transition:** On successful execution, the flow transitions based on the Flow type (either to the CONSUMER state in Hybrid mode or to GET-TRANSACTION in Producer mode).

CONSUMER State

The CONSUMER state is responsible for processing items from a queue, typically in a transaction-based flow (e.g., Orchestrator queue or CSV/Excel-based queue).

- **Executes Consumer Workflows:** It invokes the Consumer.xaml workflow to process each transaction.
- **Error Handling:** If an error occurs, it logs the exception and transitions to the FAILURE state.
- **State Transition:** After processing, it checks whether there are more items to process or if the transaction should stop. It transitions to GET-TRANSACTION or END depending on the outcome.

GET-TRANSACTION State

The GET-TRANSACTION state retrieves the next transaction to be processed, either from an Orchestrator queue or a CSV/Excel file.

- **Stop Check:** It checks if a stop signal was received, in which case it ends the process.
- **Get Next Transaction:** If no stop signal is received, it invokes the GetTransactionData.xaml workflow to retrieve the next transaction.

- **Error Handling:** If an exception occurs while retrieving the transaction, the state logs the error and transitions to the FAILURE state.
- **State Transition:** If no data is retrieved, the state transitions to END; otherwise, it moves to CONSUMER for processing.

FAILURE State

The FAILURE state provides centralized exception handling for the framework, managing both system and business exceptions.

- **Retry Mechanism:** If a failure occurs, it increments the retry count and checks whether the retry limit is reached. If retries are still allowed, the process transitions back to INIT to retry the transaction.
- **Fail Count Increment:** If retries are exhausted, the failure count is incremented, and the framework logs the failure.
- **Exception Logging:** It logs details of the exception, including whether it was a system or business exception.
- **Screenshot Capture:** If configured, it attempts to capture a screenshot of the current state to aid debugging.
- **Transition:** The state transitions to END if the maximum failure count is reached or retries are exhausted.

END State

The END state finalizes the process and closes any applications that were used.

- **Application Closure:** It invokes the FinalizeApps.xaml workflow to close all running applications.
- **Logging:** Depending on whether the process ended successfully or due to a system exception, it logs either a success or failure message.
- **Invoke OnEnd Workflow:** It calls the OnEnd.xaml workflow to handle any final cleanup actions or custom events.
- **Final Transition:** This is the final state of the state machine, which marks the process as complete.

Summary of Transitions

- **INIT → PRODUCER:** If the flow is in Producer or Hybrid mode, after initialization.
- **INIT → GET-TRANSACTION:** If the flow is directly in Consumer mode, after initialization.
- **PRODUCER → GET-TRANSACTION:** In Hybrid mode, after the Producer tasks are complete.

- **GET-TRANSACTION → CONSUMER:** If a transaction is available, it is processed by the Consumer.
- **CONSUMER → GET-TRANSACTION:** Once a transaction has been processed, the next one is retrieved.
- **Any State → FAILURE:** If an exception occurs in any state (except on the END State), the process transitions to the FAILURE state.
- **FAILURE → END:** If retries or failures exceed the limits, the process ends.
- **PRODUCER → END:** After successful execution when it's configured as Producer.
- **GET-TRANSACTION → END:** When no transactions remain.

Main Workflow Invocation Details

This topic covers the workflow invocations in the Main.xaml file.

State	*Workflow Path*	*Workflow File*	*Description*
INIT	*Framework\System\Initialization*	*InitAllSettings.xaml*	*Loads the global configuration for the automation (from YAML or Excel) required for the process initialization.*
	Framework\System\Initialization	*UpdateSettings.xaml*	*Updates specific settings such as queue name, flow type, or Orchestrator folder based on the provided arguments.*
	Framework\System\Applications	*FinalizeApps.xaml*	*Closes all running applications, ensuring that the automation starts in a clean state.*
	Framework\Actions\Applications	*StartAllApps.xaml*	*Initializes all required applications for the automation to run.*
PRODUCER	*Framework\Processes*	*Producer.xaml*	*Executes the **Producer** flow, typically adding items to the queue or performing linear operations.*
	Framework\System\Utils	*FailsReset.xaml*	*Resets the failure count to ensure that past failures do not affect future executions.*
FAILURE	*Framework\System\Utils*	*FailsReset.xaml*	*Resets the failure count after a recovery attempt.*
	Framework\Actions\Events	*OnSystemException.xaml*	*Handles system exceptions, allowing the automation to manage technical errors (unrelated to business rules).*
	Framework\Actions\Events	*OnBusinessException.xaml*	*Handles business rule*

			exceptions (BusinessRuleException) and logs them appropriately.
	Framework\System\Utils	*TakeScreenshot.xaml*	*Takes a screenshot when an exception occurs to aid in error analysis.*
GET-TRANSACTION	*Framework\System\Transactions*	*GetTransactionData.xaml*	*Retrieves the next transaction (from Orchestrator, CSV, or Excel) to be processed.*
CONSUMER	*Framework\Processes*	*Consumer.xaml*	*Executes the **Consumer** flow, processing the queue transaction items or tabular data.*
	Framework\System\Transactions	*SetTransactionStatus.xaml*	*Sets the transaction status after processing, indicating whether it was completed successfully, failed, or needs retry.*
END	*Framework\System\Applications*	*FinalizeApps.xaml*	*Closes all applications used during the automation execution.*
	Framework\Actions\Events	*OnEnd.xaml*	*Executes final actions when the process ends, such as cleaning up resources or generating final logs.*

Main State Machine Transitions

In the **Flow Driven Framework**, state transitions determine how the automation process moves between different states based on the success, failure, or completion of tasks. These transitions guide the flow from one state to another, depending on specific conditions such as error handling, retry mechanisms, and the status of transaction processing.

Each state machine transition has a specific condition that triggers the movement from one state to the next, ensuring that the framework can handle different automation scenarios efficiently, whether in **Producer**, **Consumer**, or **Hybrid** mode.

Below is a table that describes the transitions between the states and the conditions under which each transition occurs.

From State	*To State*	*Transition Condition*	*Description*
INIT	***PRODUCER***	***BotException Is Nothing AndAlso RuntimeFlow = "Producer"***	***Transitions to the Producer state if no exception occurred and***

			the flow is set to Producer.
	CONSUMER	*BotException Is Nothing AndAlso RuntimeFlow = "Consumer"*	*Transitions to the Consumer state if no exception occurred and the flow is set to Consumer.*
	FAILURE	*BotException IsNot Nothing*	*Transitions to the Failure state if an exception occurred during initialization.*
PRODUCER	*CONSUMER*	*BotException Is Nothing AndAlso RuntimeFlow = "Hybrid"*	*In Hybrid mode, transitions from Producer to Consumer after completing the producer tasks.*
	GET-TRANSACTION	*BotException Is Nothing AndAlso RuntimeFlow = "Producer" OrElse ShouldStop*	*Transitions to Get-Transaction when the Producer tasks are completed successfully or if a stop signal is received.*
	FAILURE	*BotException IsNot Nothing AndAlso Not ShouldStop*	*Transitions to Failure if an exception occurred during the Producer tasks.*
CONSUMER	*GET-TRANSACTION*	*BotException Is Nothing AndAlso Not ShouldStop AndAlso (TransactionItem IsNot Nothing OrElse TransactionRow IsNot Nothing)*	*Transitions to Get-Transaction after processing a transaction in the Consumer state.*
	FAILURE	*BotException IsNot Nothing AndAlso Not TypeOf BotException Is BusinessRuleException*	*Transitions to Failure if a system exception occurs during the Consumer tasks.*
GET-TRANSACTION	*CONSUMER*	*BotException Is Nothing AndAlso (TransactionItem IsNot Nothing OrElse TransactionRow IsNot Nothing)*	*Transitions to Consumer after successfully retrieving a transaction from the queue or data source.*
	FAILURE	*BotException IsNot Nothing AndAlso Not ShouldStop*	*Transitions to Failure if an error occurs while retrieving the next transaction.*
	END	*(BotException Is Nothing AndAlso TransactionItem Is Nothing AndAlso TransactionRow Is Nothing) OrElse ShouldStop*	*Transitions to End if no more transactions are available or a stop signal is received.*
FAILURE	*INIT*	*FailsCount <= g_Config.AsInt("Recover/MaxFailsCount")*	*Retries the process by transitioning back to Init*

			if the failure count is within the acceptable limit.
	END	*FailsCount > g_Config.AsInt("Recover/MaxFailsCount")*	*Transitions to End if the maximum allowed failures are reached, signaling that the process cannot recover.*

Main Variables

The **Flow Driven Framework** uses several variables to manage the automation process. These variables store information like transaction details, exception handling, and process control flags.

Variable Name	*Type*	*Scope*	*Description*
g_Config	***Autossential.Configuration.Core***	***Global***	***Stores the configuration data from the Main.yml file***
BotException	***System.Exception***	***Main***	***Stores any exceptions that occur during the execution of the process. It is reset in each new transaction.***
RuntimeFlow	***System.String***	***Main***	***Tracks the current flow being executed (e.g., Producer, Consumer, or Hybrid).***
FailsCount	***System.Int32***	***Main***	***Counts the number of failures encountered during the execution.***
ShouldStop	***System.Boolean***	***Main***	***Indicates whether a stop signal has been sent from Orchestrator to halt the process.***
TransactionItem	***UiPath.Core.QueueItem***	***Main***	***Represents the current transaction item being processed (for Orchestrator queues).***
TransactionNumber	***System.Int32***	***Main***	***Sequential counter of the current transaction. Increments after each transaction is processed.***
TransactionID	***System.String***	***Main***	***Holds the unique ID of the current transaction (for logging and tracking).***
TransactionRow	***System.Data.DataRow***	***Main***	***Stores the current row of data being processed in non-Orchestrator queues (e.g., CSV or Excel).***
TransactionData	***System.Data.DataTable***	***Main***	***Holds the tabular data when the queue type is CSV or Excel.***
RetriesCount	***System.Int32***	***Main***	***Tracks the number of retry attempts for the current***

			transaction.
ShouldMoveNext	***System.Boolean***	***Main***	***Determines whether the process should proceed to the next transaction. Defaults to True.***

Main Constants

The **Flow Driven Framework** also uses constants to manage fixed values that control the behavior of the framework. Constants are not typically modified during the execution of the process.

Constant Name	*Type*	*Default Value*	*Description*
Recover/MaxRetriesCount	***System.Int32***	***Defined in Main.yml***	***The maximum number of retries allowed for a transaction in the Consumer phase.***
Recover/MaxFailsCount	***System.Int32***	***Defined in Main.yml***	***The maximum number of failures allowed before the process halts.***
Framework_ApplicationException	***System.String***	***Defined in Logs.yml***	***Template for logging application exceptions that occur during execution.***
Framework_SuccessfulExecution	***System.String***	***Defined in Logs.yml***	***Template for logging successful executions in the automation process.***
Framework_StopProcessRequested	***System.String***	***Defined in Logs.yml***	***Message logged when a stop request is received from Orchestrator.***
Framework_GetTransactionDataError	***System.String***	***Defined in Logs.yml***	***Message logged when an error occurs while retrieving the next transaction.***
Framework_MaxFailNumberReached	***System.String***	***Defined in Logs.yml***	***Message logged when the maximum allowed number of failures has been reached.***

Main Arguments

The **Flow Driven Framework** uses arguments to pass data into and out of workflows. These arguments allow the framework to be flexible and adaptable, especially when dealing with different configuration files, queue names, and runtime settings.

Argument Name	Type	Direction	Description
in_Flow	System.String	In	Overwrites the flow configuration defined in the configuration files (e.g., Producer, Consumer, or Hybrid).
in_OrchestratorFolderName	System.String	In	Overrides the default folder name used for Orchestrator, if a different one is specified.
in_QueueName	System.String	In	Specifies a different queue name to be used during runtime, overriding the one defined in the configuration.
in_ConfigPathToMerge	System.String	In	Allows for the merging of an additional configuration file at runtime. Values from this file can override others.

Workflow Specifications

This section provides detailed descriptions of each workflow used in the **Flow Driven Framework**. Each workflow is an essential component of the framework, responsible for specific tasks such as initialization, transaction management, error handling, and process finalization. Understanding the purpose and structure of these workflows is key to customizing and extending the framework for different automation scenarios.

Main.xaml

- **Purpose:** The **Main.xaml** workflow is the central orchestrator of the **Flow Driven Framework**. It defines the overall structure of the automation process through a state machine, coordinating the flow between initialization, transaction processing, exception handling, and finalization. The main workflow manages transitions between different states based on predefined conditions, handling both linear and transactional processes in **Producer**, **Consumer**, or **Hybrid** modes.

Framework\System\Initialization\InitAllSettings.xaml

- **Purpose:** Loads the global configuration settings from YAML, which are used to drive the overall process. This workflow ensures that the necessary configurations such as queue names, folder paths, and other settings are available at runtime.
- **Key Actions:** Reads and stores the configuration data as global variables, enabling easy access throughout the automation process.

Framework\System\Initialization\LoadAssets.xaml

- **Purpose:** Loads assets from Orchestrator. Assets typically include values such as credentials, file paths, or other dynamic information that is used throughout the automation process.
- **Key Actions:** Fetches and loads assets stored in Orchestrator, allowing for centralized management of automation configurations and sensitive information (e.g., credentials).

Framework\System\Initialization\UpdateSettings.xaml

- **Purpose:** Updates specific configuration settings based on the provided arguments, such as in_QueueName, in_Flow, or in_OrchestratorFolderName. This allows the framework to adapt dynamically at runtime, depending on external inputs.
- **Key Actions:** Merges additional settings, applies argument overrides, and ensures the correct flow and queue configurations are in place.

Framework\System\Applications\StartAllApps.xaml

- **Purpose:** Initializes and starts all required applications necessary for the automation process. This is a crucial step for preparing the environment before starting the actual workflow.
- **Key Actions:** Opens applications (e.g., browsers, desktop apps) and verifies they are running before continuing with the automation.

Framework\System\Applications\FinalizeApps.xaml

- **Purpose:** Ensures that all applications opened during the process are properly closed when the automation is finished. This is important for resource management and avoiding issues in future runs.
- **Key Actions:** Closes applications, ensuring no leftover processes remain, which could interfere with subsequent executions.

Framework\System\Applications\CloseAllApps.xaml

- **Purpose:** Ensures that all applications used during the automation process are closed gracefully. This is essential to maintain system resources and ensure there are no lingering processes after the automation finishes.
- **Key Actions:** Closes all applications listed in the ProcessesToKill configuration. Ensures that no unwanted processes are left running that could interfere with future automations.

Framework\System\Applications\KillAllProcesses.xaml

- **Purpose:** Terminates all applications that are listed in the ProcessesToKill configuration. This is used as a last resort when applications cannot be closed gracefully and must be forcibly terminated.
- **Key Actions:** Identifies the processes to be killed (based on the ProcessesToKill list) and forcefully stops them. Ensures that the environment is cleaned up even in cases where graceful closure fails.

Framework\Processes\Producer.xaml

- **Purpose:** Executes the **Producer** flow, typically adding items to a queue for further processing. This workflow is critical for the initial stage in transaction-based processes or when performing linear automation steps.
- **Key Actions:** Gathers data and pushes it to the queue or performs tasks related to the producer's role in a **Hybrid** or **Producer** mode process.

Framework\Processes\Consumer.xaml

- **Purpose:** Executes the **Consumer** flow, responsible for processing the transaction items that have been retrieved from the queue. This workflow typically handles business logic associated with each transaction.
- **Key Actions:** Processes transaction data, performs validations, applies business rules, and marks transactions as successful or failed.

Framework\System\Transactions\GetTransactionData.xaml

- **Purpose:** Retrieves the next transaction to be processed, whether from Orchestrator queues or other data sources like CSV or Excel. This workflow manages how the data is pulled and prepares it for processing.
- **Key Actions:** Fetches the next available transaction item or data row, ensuring the correct flow of transactional data.

Framework\Transactions\DataSources\GetFromCSV.xaml

- **Purpose:** Retrieves transaction data from a CSV file. This is typically used in place of Orchestrator queues for smaller or simpler processes where data is stored locally in CSV format.
- **Key Actions:** Reads data from the specified CSV file, parses it into a DataTable, and makes it available for transaction processing. The separator (default ;) can be adjusted based on the file format.

Framework\Transactions\DataSources\GetFromExcel.xaml

- **Purpose:** Retrieves transaction data from an Excel file. This is used when the process data is stored in a structured format within Excel workbooks, providing an alternative to Orchestrator queues.
- **Key Actions:** Reads data from the specified Excel file and sheet, converts it into a DataTable, and makes the data available for transaction processing. It supports both .xlsx and .xls formats.

Framework\Transactions\DataSources\GetFromOrchestrator.xaml

- **Purpose:** Retrieves transaction data from an Orchestrator queue. This is the default method for handling transactional data in the framework when using Orchestrator.
- **Key Actions:** Connects to Orchestrator, fetches the next available transaction from the queue, and prepares it for processing. This workflow handles queuing logic such as marking items for retry or failure if an error occurs.

Framework\System\Transactions\SetTransactionStatus.xaml

- **Purpose:** Updates the status of each transaction after processing, indicating whether it was successfully completed, failed, or should be retried. This workflow is integral to transaction management in Orchestrator or other queue-based systems.
- **Key Actions:** Sets transaction status (e.g., success, business exception, or application exception) and handles retries if necessary.

Framework\System\Utils\FailsReset.xaml

- **Purpose:** Resets the failure count or retry count, depending on the configuration, ensuring that the automation doesn't continue in a failed state. It helps control the flow of error recovery.
- **Key Actions:** Resets the retry counter after a successful transaction or after switching between producer and consumer flows.

Framework\System\Utils\TakeScreenshot.xaml

- **Purpose:** Captures a screenshot during an exception or failure, helping to debug and trace the root cause of errors. This visual feedback can be crucial for identifying UI-related issues.
- **Key Actions:** Takes a screenshot of the current screen and saves it in a designated folder, specified in the configuration.

Framework\Actions\Events\OnSystemException.xaml

- **Purpose:** Handles system-level exceptions, such as infrastructure failures (e.g., timeouts, file access errors). This workflow needs to be used in case there are any specific tasks to be done when it encounters an application exception.
- **Key Actions:** Specific tasks related to the application exception (need to be done by the developer).

Framework\Actions\Events\OnBusinessException.xaml

- **Purpose:** Manages business rule exceptions, such as invalid data or missing input. This workflow needs to be used in case there are any specific tasks to be done when it encounters a business exception.
- **Key Actions:** Specific tasks related to the business exception (need to be done by the developer).

Framework\Actions\Events\OnEnd.xaml

- **Purpose:** Executes custom actions at the end of the process, typically used for final cleanup, logging, or notifications. This workflow ensures that any required tasks are performed before the process fully terminates.
- **Key Actions:** Logs the successful completion of the process or handles any finalization events specific to the automation project.

Framework\Actions\Events\OnStateTransition.xaml

- **Purpose:** Handles state transitions within the state machine, logging the transition between states and ensuring that workflows behave appropriately when moving between INIT, PRODUCER or CONSUMER.
- **Key Actions:** Logs state transitions and triggers any specific actions that need to occur when moving from one state to another.