MICROFLOWS

Author: Dr. Javier Navarro-Machuca,
IO Connect Services

contact@ioconnectservices.com
www.ioconnectservices.com

TABLE OF CONTENTS

Introduction 3

Common implementation problems in system integrations 4

What is a microflow? 7

Anatomy of a microflow 8

The principal components of a microflow 8

Inter-flow communication 10

Migrating a legacy integration flow to microflows 11

Addressing common implementation problems in system integrations with microflows 13

Summary 16

References 16

INTRODUCTION

In this post, I introduce the concept of Microflows. A Microflow is the combination of a Microservice and a well-defined transaction integration flow implemented with an Integration Framework Library.

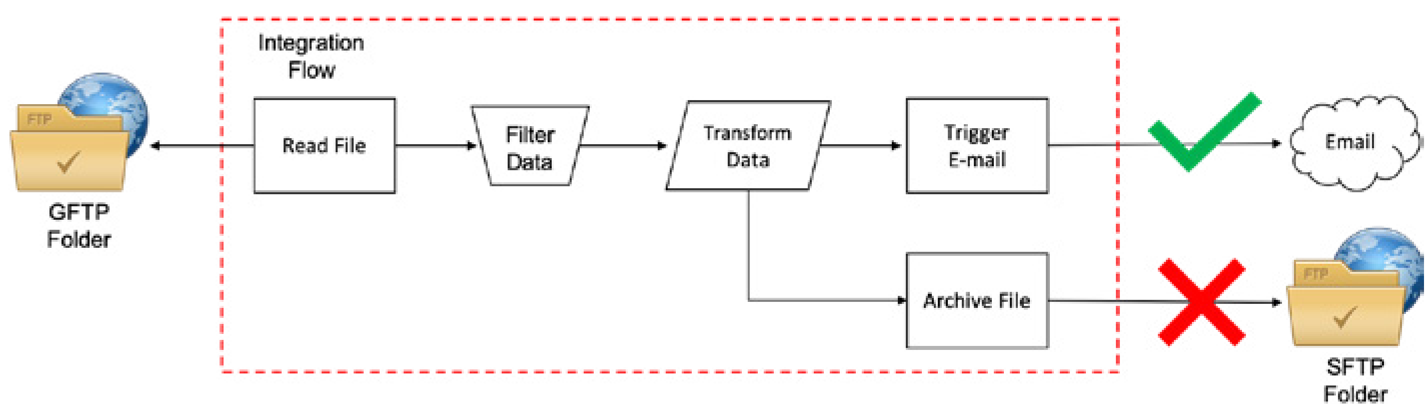
The concept of Microservices has been trending in the last couple of years. In my point of view, Microservices is a way to streamline some of the principles of the Service Orientation Architecture paradigm (SOA): service loose coupling, service abstraction, service reusability, service autonomy, service statelessness, service composability. On the other hand, governance complexity increases since you end up with many software assets that you need to maintain, version, and standardize. Therefore, service discoverability and standardized service contract principles are extremely important in order to achieve a good level of service governance.

For many programmers and software architects, SOA is a synonym of ESB (enterprise service bus). An ESB tool is mainly a compound of SOA patterns that can be used together to integrate services and monolithic applications that usually do not implement the same communication protocols and define a different data model. These differences usually bring the burden of endpoints call orchestrations, data model and data format transformations, protocol bridging and service transaction management [2] entirely to the ESB layer. Mule and Apache Camel are good lightweight ESBs and integration libraries options to avoid the high cost of big and fat ESB platforms.

COMMON IMPLEMENTATION PROBLEMS IN SYSTEM INTEGRATIONS

Bad transaction design
In many cases – especially in implementations with ESB platforms – the entire integration flow has no proper transaction design; the flow is composed of many activities that do not share the ACID transaction properties or do not compensate transaction routines in case that any activity fails. This problem introduces data inconsistency across the applications that receive information handled by the flow, creating a lot of complexity in compensating for the system failures.

Weak or missing fault tolerance controls
Bad transaction design usually lacks fault tolerance controls. Sometimes, well-design transactions implement weak fault tolerance mechanisms, leaving all the responsibility to the ACID properties to guarantee data consistency and durability across applications, especially in distributed systems. A good example of this is when one of the transaction activities needs to keep data into a system or application that cannot be enrolled in the ambient transaction, such as in the two-phase commit protocol. Another common example is when the integration transaction rolls back without providing a way to automatically retry the execution or to notify a compensating process for further retry or to alert a person via email, log system or a dashboard.



In this integration flow, the trigger email activity executed successfully while the archive file activity failed. The lack of fault tolerance controls like a retry and bad transaction design leaves an inconsistent state of the data.

Lack of metrics definitions for alerting controls

Logging process activity in integrations is a common practice to record successful or unsuccessful executions in software applications. In many cases, the logs are not further processed or analyzed automatically by other applications to raise alerts when an unexpected result or a severe problem is found.

It is common that the logs are referenced as a reactive measure (several hours or days after) to find any information that can help with identifying the root cause of the problem. Defining the right metrics to log is no easy feat; it is important to analyze and classify what the data points are and how they can provide us with the right metrics for an application to log. Good data metrics will facilitate the automation of raising alerts to either prevent or rapidly react to a problem.

Shared or global configuration dependencies

This recurrent practice is found when there are several applications sharing the same application host. One example is when there is more than one application exposing web services via common ports 80 for HTTP or 443 for HTTPS. In this case, one single component is configured at the server level globally to be shared by all the web service apps. This convention breaks the autonomy of the applications since, if the configuration needs to be changed to fulfill the requirements of any one application, it might affect all the rest, forcing us to test any other application that would be impacted. Test automation is a good way to mitigate this risk. This is something that is usually desired but rarely carried out.

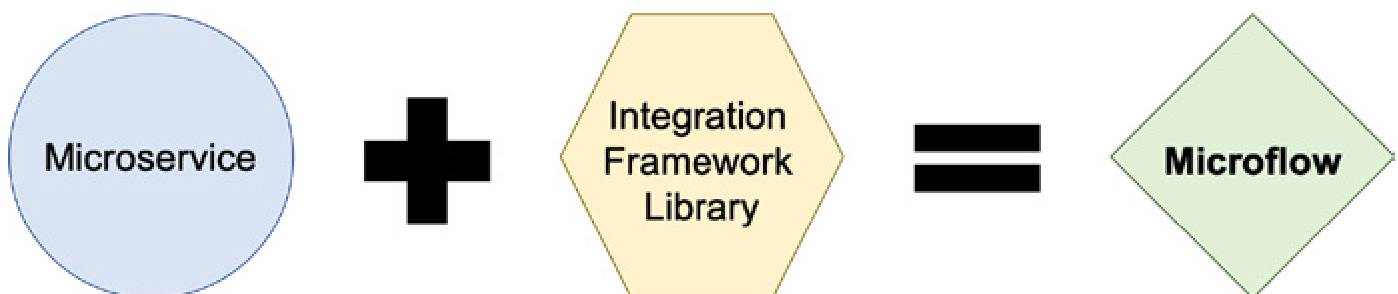
Monolithic scalability

This is found when several integration flows are implemented in one single application where the flows are executed concurrently. If there is a need to scale one of the flows within the application due to load increase or usage, the entire application will be deployed to satisfy this demand. This might introduce several problems if not all the flows were designed to run with several instances in parallel. Some of the few unwanted result examples are data inconsistency, excessive computer resource consumption or random errors that are difficult to trace.

Despite being very common in system integrations, the previous problems are not unique to integration flows, in fact, there are many best practices and guidelines to avoid them in application development, such as SOA principles and Microservices.

WHAT IS A MICROFLOW?

A Microflow is the combination of a Microservice and a well-defined transaction integration flow implemented with an Integration Framework Library.



The main reason for using an Integration Framework Library like Mule or Apache Camel is to take advantage of the implemented 65 integration patterns [3]. The implementation of the patterns is proven, tested, and maintained in these integration libraries, so we do not need to reinvent the wheel and custom implement them when we want to use a Microservices approach instead of an ESB.

ANATOMY OF A MICROFLOW

Microflows promote service autonomy and abstraction; it exposes a well grain-defined public interface to communicate with external applications, like other Microflows. Like Microservices, it is key to set a good application boundary for our Microflows. Containers are a good artifact that can help us to achieve the application division that we are looking for.

Docker is very popular in the containerization world, and we can find images of almost all application runtimes and servers. Therefore, it is very tempting to utilize an App Server like Mule Server or Apache Server to host our integration applications. I do not recommend this practice since it can break the desired application independence. If we host more than one app in the Docker App Server image, we are practically taking the common integration problems to the next level. Instead, we should host a lightweight application process in our containers, where the main dependency is the application run-time – like JRE or .NET. Docker OpenJDK [4] is a good image to use for our purpose.

THE PRINCIPAL COMPONENTS OF A MICROFLOW

Integration Framework Library

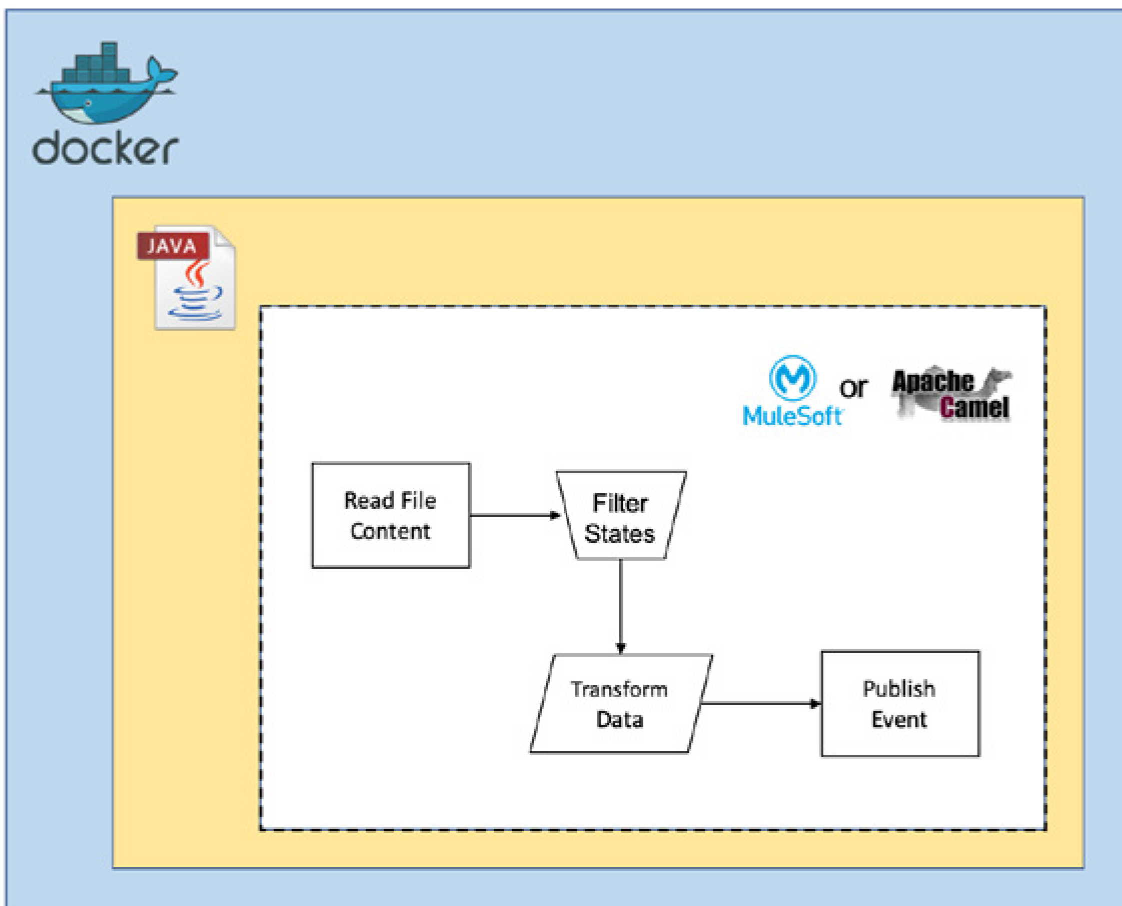
This library must provide a good compound of enterprise integration patterns. Integration flows should use these patterns for integration solutions "ranging from connecting applications to a messaging system, routing messages to the proper destination, and monitoring the health of the messaging system." [3]

Application Package

This is the package of code libraries—compiled or to be interpreted — that will be executed by the run-time system.

Container Image

A container image is the basis of containers. The containers are instances of these images. Docker' glossary defines an Image as "the ordered collection of root file-system changes and the corresponding execution parameters for use within a container run-time. An image typically contains a union of layered file- systems stacked on top of each other" [5].



This image shows a blueprint of the anatomy of a Microflow. The integration flow is implemented in an Integration Framework Library like Mule or Apache Camel, wrapped in a JAR file, and hosted in a Docker Java image.

INTERFLOW COMMUNICATION

How to achieve inter-service communication is a common discussion when designing Microservices. This should not be foreign to Microflows either. Many purists may say that the best way to communicate services should be via HTTP with RESTful interfaces. When talking about system integrations, especially when uptime is not a quality that all systems, applications, and third parties have, there is a need to guarantee successful data exchange delivery. While in Microservices the arguments focus mainly on sync vs async communication, Microflows do it in terms of system availability and SLAs. Messaging patterns fit most of the system integration needs very well, regardless of the communication protocol used.

To make our integrations more resilient, we need a buffer between our services when transmitting data. This buffer will serve as the transient storage for data messages that cannot be processed by the application destination yet. Message queues and event streams are good examples of technologies that can be used as transient storage. The Enterprise Integration Patterns language defines several mechanisms that we can implement to guarantee message delivery and how to set up fault tolerance techniques in case a message cannot reach its destination.

A Microflow should not be limited to one simple message communication exchange, in many cases, we need to expose different channels for integration, allowing the consumer to decide what message channel exchange fits better its integration use case. I recommend that for every Microflow, you expose an HTTP/S endpoint and a message queue listener as the entry inbound components of the flow.

MIGRATING A LEGACY INTEGRATION FLOW TO MICROFLOWS

To migrate legacy implementations of integration flows to Microflows, it is necessary to have a good understanding of transaction processing [6] and better yet, experience. Transaction processing will help us with identifying indivisible operations that must succeed or fail as a complete unit, and any other behavior that would result in data inconsistency across the integrating systems. These identified indivisible operations are the transactions that we will separate to start crafting our Microflows. Each single transaction must fulfill the ACID properties [7] to provide reliable executions. There are some design patterns that can facilitate the transaction design and implementation like the Unit of Work [8].

System integrations commonly exchange data among applications distributed in different servers and locations, where no single node is responsible for all data affecting a transaction. Guaranteeing ACID properties in this type of distributed transactions is an important task. The two-phase commit protocol [9] is a good example of an algorithm that ensures the correct completion of a distributed transaction. One main goal when designing Microflows is that one Microflow's implementation handles one single distributed transaction as a whole.

Database management systems and message brokers are technologies that normally provide the mechanisms to participate in distributed transactions. We should take advantage of this benefit and always be diligently investigating what integrating systems or components can be enlisted in our Microflow's transaction scope. File systems and FTP servers are commonly not transaction friendly, for this purpose we need to use a compensating transaction [10] to undo a failed execution to bring the system back to its initial state. We need to consider what our integration flow must do in case the compensating transaction also fails. Fault tolerance techniques are key to maintaining system data consistency in this corner scenario

Dead letter queues and retry mechanisms are artifacts that we should always consider as ways to improve our fault tolerance in our transaction processing. If we are creating Web APIs, our APIs must provide operations that we can use to undo transactions.

In summary, these are the steps to follow when migrating a legacy integration flow app to Microflows. The steps are not limited to Microflows migration since they can be used to design Microflows integrations from the green field:

1. Identify all the indivisible transactions in the implementation
2. Separate each transaction in its own flow
3. Promote each transaction to a Microflow
4. Identify what activities and integrating components can enlist to a distributed transaction
5. Define a compensating transaction for each integrating component that cannot enlist to a distributing transaction
    1. Analyze what compensating transactions must be promoted as Microflows
6. Communicate Microflows via channels that can enlist to distributed transactions (via two-phase commit protocol or message acknowledgments) and that provide message reliable delivery like message queues, stream events, etc.

Bad transaction design

To address this problem, it is necessary to carry out steps 1 through 3 of the Microflows migration steps. First, we need to identify all the indivisible transactions by leveraging design techniques like state machine diagrams. Each state usually represents one activity that needs to be executed in an integral fashion to meet the post-conditions needed to move to the next state. If any of the conditions are not met, the integration flow must undo any partial execution and move back to the original state. Second, we separate each indivisible transaction in its own flow, which will facilitate working on the integrating activity in isolation. This step facilitates the design and development of good practices like unit testing and user acceptance testing.
Finally, we promote each transaction to a Microflow to deploy in our solution environments. This will help us to treat any Microflow independently to better maintain and support it.

Monolithic scalability

With Microflows, we do not need to redundantly deploy our whole integration blueprint to handle load peaks or to provide high availability to the application consumers. Microflows support high availability since they can scale horizontally and we can cherry-pick the strategy to scale each one independently: a Microflow with a synchronous web service interface can be set in a cluster with a minimum of X instances running for availability purposes, whereas a Microflow that listens to a message queue can scale based on computer resources usage or queue length.

The intrinsic transnational design promoted by Microflows helps substantially with improving fault tolerance by making our integrations more resilient to error recovery due to the ACID properties. In many cases, this is not enough, and we need to

put other mechanisms in place to assure that our transaction will be executed successfully. Some examples of fault tolerance mechanisms and patterns are redundancy, error escalation to dead letter queues and poison queues, and compensating activity, among others.

One major advantage of using an integration framework library for the core development of Microservices, is that a big subset of the implemented 65 enterprise integration patterns facilitates (if not yet implemented entirely) the correct application of many fault tolerance controls.

Lack of metrics definitions for alerting controls

Another advantage of using queues as a mechanism to communicate Microflows is that we can easily set up monitor and alert controls on the queue itself. Alerts can be set up based on message longevity (e.g., alerting when a message has been in the queue for more than X hours), queue length (e.g., alerting when there are more than Y number of messages in the queue), etc. These alerts will tell us when something is wrong in our integrations such as when a third-party system is not working. Dead letter queues are very useful for this purpose; we can trigger alerts as soon as a DLQ contains one or more messages. Many monitoring tools offer plug-ins to set up alerts on the integration components based on resource limit usage.

Business based alerts must not be forgotten either; we should be able to send notifications to stakeholders when a transaction presents a problem based on business values conditions. The design principles of Microflows facilitate the implementation of business alerts since we can focus on it in isolation, based on the use case that such Microflow implements, on what notifications need to be sent for that given integration transaction.

Shared or global configuration dependencies

Microflows promote process and resource configuration and access autonomy. Each Microflow instance is responsible for accessing computer resources as needed to achieve the successful execution of the integration transaction. One Microflow

may be polling an FTP server in a higher frequency than the rest. This is a good example of why the practice of creating global configurations for computer host consumption is not recommended, otherwise, we might be forced to share the global configuration that is not optimal for the transaction needs. Microflows can be tuned and maintained in isolation, without having a significant impact on each other.

SUMMARY

Microflows are the result of applying Microservices design principles to system integrations flows that are implemented with an Integration Framework Library. The practice of Microflows moves system integrations away from a centralized ESB orchestration approach to a more distributed and decentralized choreography of independent transactions that form a wholly cohesive system integration solution. Several common integration problems were discussed and addressed by using Microflows principles. Specific examples of Microflows implementations targeting the pointed integration problems will be covered in future articles.

REFERENCES

[1] *https://en.wikipedia.org/wiki/Service-orientation_design_principles*

[2] *http://soapatterns.org/*

[3] *http://www.enterpriseintegrationpatterns.com/*

[4] *https://hub.dockevr.com/r/_/openjdk/*

[5] *https://docs.docker.com/engine/reference/glossary/#image*

[6] *https://en.wikipedia.org/wiki/Transaction_processing*

[7] *https://en.wikipedia.org/wiki/ACID*

[8] *http://wiki.c2.com/?UnitOfWork*

[9] *https://en.wikipedia.org/wiki/Two-phase_commit_protocol*

[10] *https://en.wikipedia.org/wiki/Compensating_transaction*