MuleSoft

WHITEPAPER

# Reactive programming:

## New foundations for high scalability in Mule 4

# Table of contents

# Introduction

Mule developers build integration solutions that must cater to high volumes of realtime and batch data. In some cases the size of payloads can be larger than memory. In others API solutions may have to handle high rates of API calls per second. Horizontal scalability, whereby Mule instances are deployed across multiple virtual and physical machines, is costly. Thus a single Mule instance must utilize to the maximum underlying processing and storage resources as it handles traffic. Its ability to handle more traffic with more underlying resources is what we refer to as vertical scalability.

Mule 4 addresses vertical scalability with a radically different design using reactive programming at its core. With the aim to maximize throughput our engineers have focussed their efforts on achieving higher concurrency through non-blocking operations and a much more efficient use of CPU, memory, and disk space.

In this whitepaper you will learn the reactive approach and non-blocking code execution, as well as an innovative approach to thread management and streaming capabilities, all in the context of Mule 4. You will also learn how to extend Mule 4 with your own modules in a way that exploits these exciting new capabilities. Finally, you will see the underlying reactive Spring Reactor framework used at the core of Mule 4 and how to use it in your extensions.

# What is Mule 4?

Mule 4 is the newest version of Mule runtime engine which uses reactive programming to greatly enhance scalability. A Mule application is an integration application which incorporates areas of logic which are essential to integration. These logical domains include:

- Connectivity
- Transformation
- Enrichment
- Validation
- Routing
- Error Handling
- Security

A Mule application is developed declaratively as a set of flows. Each flow is a chain of event processors. An event processor processes data passing through the flow with logic from one of the said domains. Mule 4 uses reactive programming to facilitate non-blocking execution of the event processors. This has a significant impact on a Mule application's ability to scale the amount of events it can handle concurrently. Each event processor belongs to a module. Modules are added to Mule runtime engine as required by Mule applications. Mule 4 is entirely extensible through custom modules. The Mule SDK, a new addition to the Mule 4 ecosystem, enables you to extend Mule with new modules. Custom modules benefit from the same scalable reactive capabilities of Mule runtime engine.

# Reactive programming

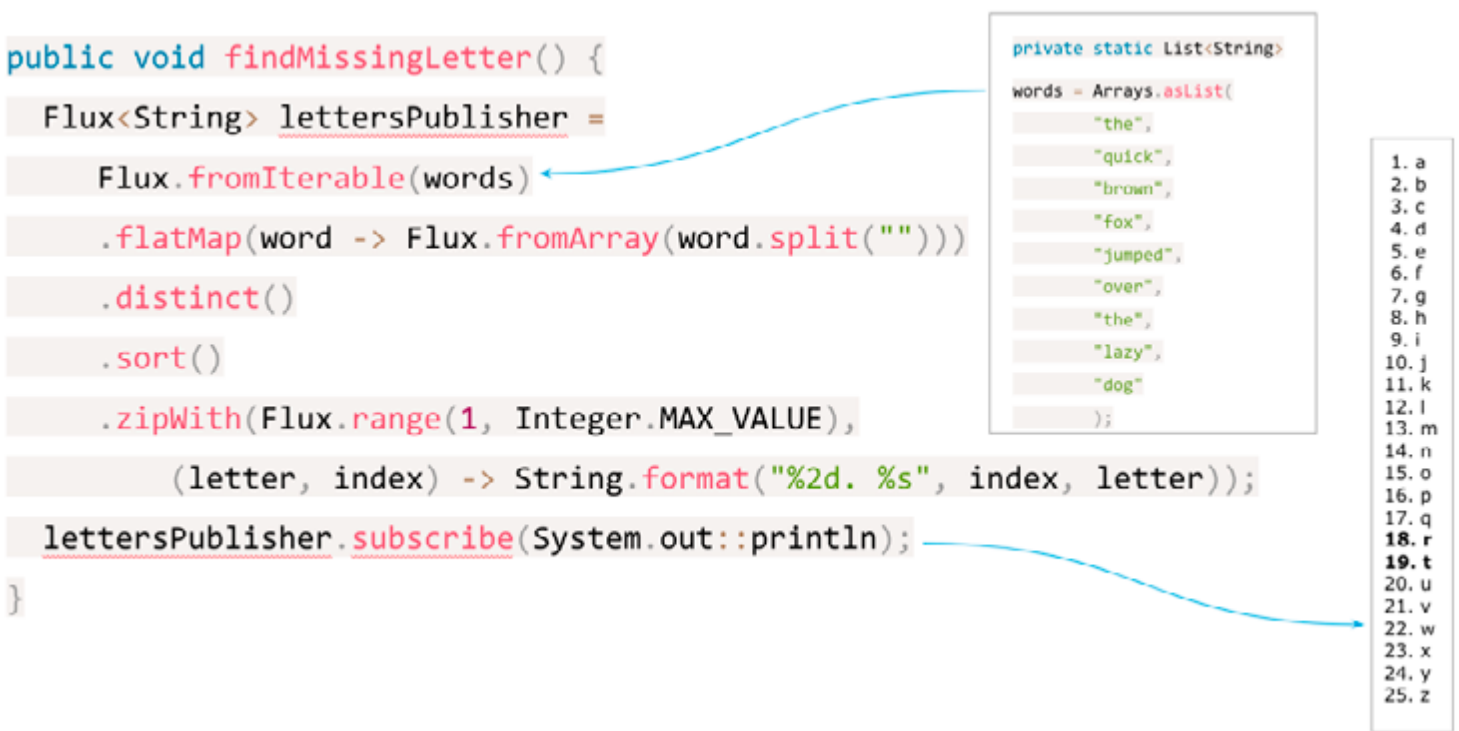
The advent of single page web apps and native mobile apps has fueled users' appetite for fast and responsive applications. We want data to be made available now. We have become accustomed to push notifications. Architectures in the back-end have evolved to cater to front-end responsiveness and drive elasticity and resiliency through event driven microservices. Reporting requirements have evolved from entirely retrospective analysis to include real-time predictive analytics made possible by event stream processing.

Reactive Programming (Rx) arose in the .Net community to give engineers the tools and frameworks needed to cater to this appetite. It quickly spread to other languages. It became JavaRx in the Java world and Spring has fully embraced it with Project Reactor. Reactor now lies at the heart of Mule 4's internal architecture.

Reactive merges the best from the Iterator and Observer design patterns, and functional programming. The Iterator pattern gives the consumer the power and control over when to consume data. The Observer pattern is the opposite: it gives the publisher power over when to push data. Reactive combines the two to get the beauty of data being pushed to the subscriber when it is available but in a way that protects the subscriber from being overwhelmed. The functional aspect is declarative. You can compare functional composition of operations to the declarative power of SQL. The difference is that you query and filter data in real time against the stream of events as they arrive. This approach of chaining operators together avoids the callback hell which has emerged in asynchronous programming and which makes such code hard to write and maintain.

# Spring Reactor Flux

Try out the online tutorial on Reactor. Here's an example from an article published on InfoQ:




The Flux API represents a stream of events. We create a flux from the array of words: *["the", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog"]*. As we process each word in the array we form another flux from all the letters in the word and merge each new flux into one for all the words using the *.flatMap()* operation. We then purge out repeating letters, sort them and merge them with a flux of incrementing integers *[ "1. a", "2. b", ... ]* using the *.zipWith()* operation. A key takeaway is that all of the above is declarative. Through functional composition we can declare how we want processing to be done without actually implementing it. But nothing happens till we subscribe. Subscribing to the final flux produces the stream of strings we see in the output.

# Reactive programming in Mule 4

Consider Mule flow to be a chain of (mostly) non-blocking publisher / subscriber pairs. The source is the publisher to the flow and also subscriber to the flow as it must receive the event published by the last event processor on the flow. The HTTP Listener, for example, publishes an event and subscribes to the event produced by the flow. Each event processor subscribes and processes in a non-blocking way.

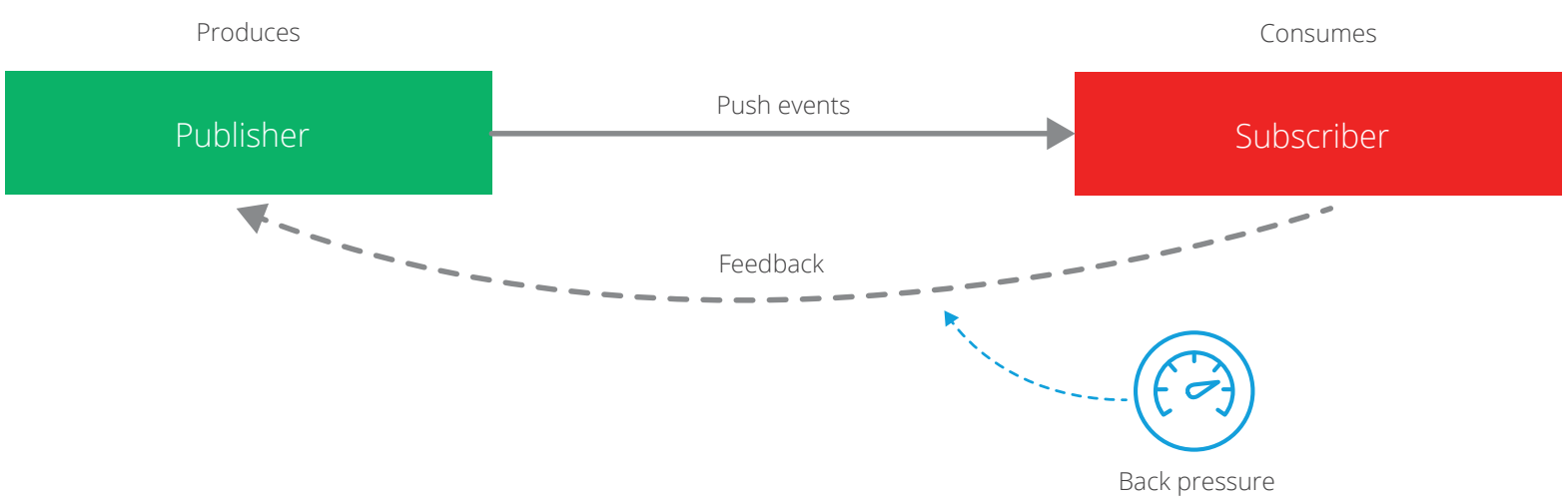



**Figure 1:** The HTTP listener publishes an event and subscribes to the event produced by the flow.

# Back pressure

Consider for a moment a natural phenomenon called back pressure which occurs in confined spaces like pipes. It opposes the desired flow of gases through that space. Mechanical engineers have to design the likes of exhaust pipes with this in mind so that the gas can move through the confined space with the least amount of impedance.

This forms the basis of a metaphor key to the reactive programming model. In order to allow the subscriber to consume the event stream without being overwhelmed by too many events, the subscriber must be able to apply back pressure, in other words signal to the producer to "please slow down."

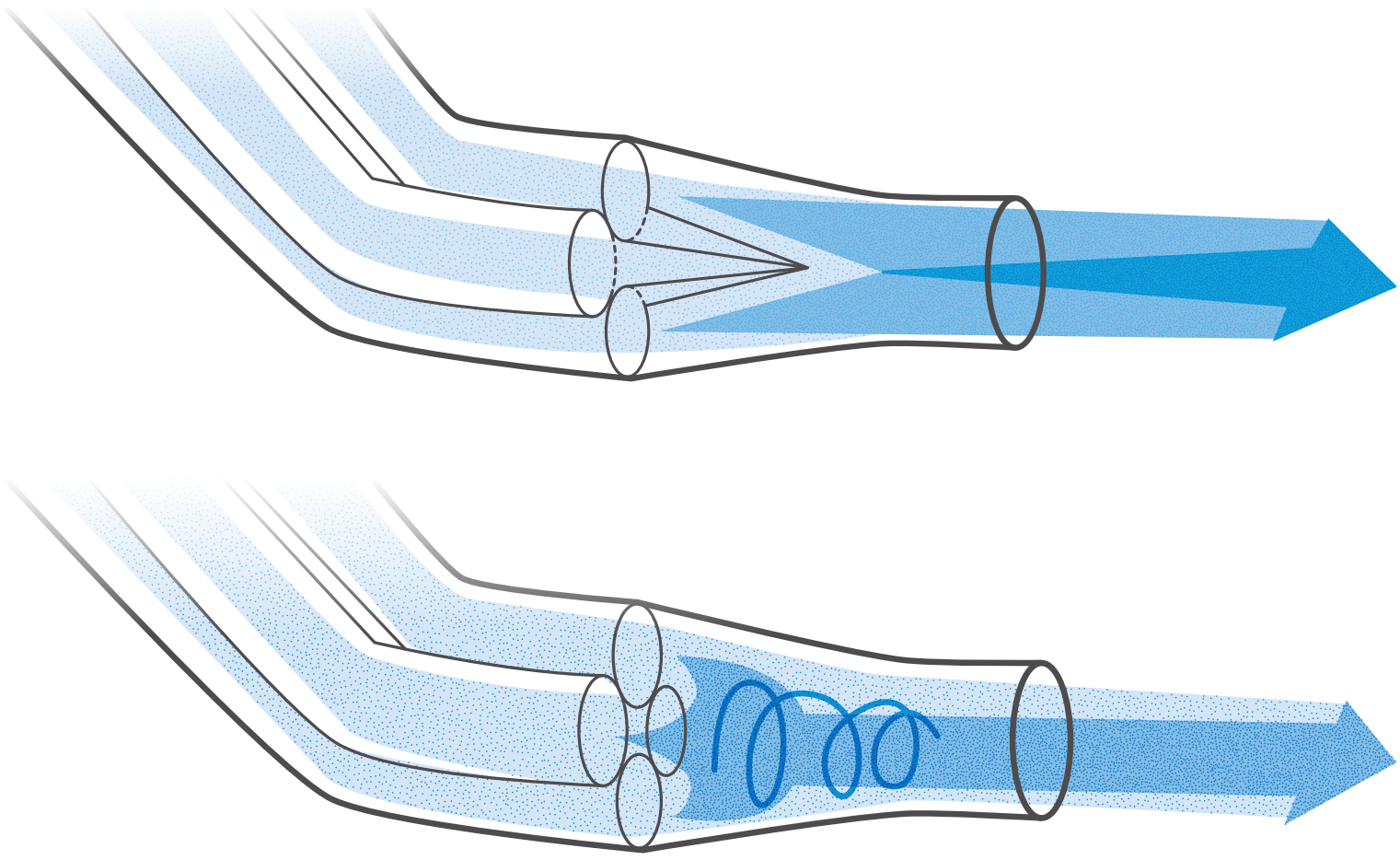


**Figure 2:** Back pressure occurs in confined spaces like exhaust pipes. It opposes the desired flow of gases through that space.

## Automatic back pressure in Mule 4

Mule 4 applications are automatically configured so that the event source receives a back pressure signal when all threads are currently executing and no free threads remain in a required thread pool. In practical terms this will trigger the HTTP Listener, for example, to respond with a 503–"Server busy", and the JMS Listener will not acknowledge receipt of a message. OutOfMemory errors are avoided as a result of this configuration.

## Manual back pressure in Mule 4

Mule developers can also configure each event processor to signal back pressure to the event source through the "maxConcurrency" attribute. This configuration affects the number of events that can pass through the event processor per second.

# Non-blocking operations in Mule 4

Non-blocking operations have become the norm in Mule 4 and are fundamental to the reactive paradigm. The wisdom here is that threads which sit around waiting for an operation to complete are wasted resources. On the flip side thread-switching can be disadvantageous. It is not ideal that each event processor in a Mule application flow execute on a different thread as this will increase the latency of the flow execution. However the advantage is that a Mule flow can cater to greater concurrency because we keep threads busy with necessary tasks.


**Figure 3:** Non-blocking operations can be likened to highway tolling systems.

A simple metaphor for non-blocking operations in a Mule flow is the tolling systems on highways. Visualize a highway with multiple lanes filled with cars and a toll stop crossing the width of the highway. Some lanes go through the manual tolls. Others go through the e-tolls. Each lane on the highway is like a

thread. Each car is like an event being processed by those threads. A car stopped at a manual toll is like an event going through a blocking operation. The database module's read operation, for example, blocks and waits for responses to come back from the database server. That's a bit like going through the manual toll. You stop your car, you pay, and the cars have to backup behind you while you wait for your transaction to be processed. The e-tolls in contrast allow you to drive straight through. These are like the non-blocking HTTP Requester. It sends off requests on one thread but that same thread is immediately freed up and does not wait for a response from the server. The net-effect is greater concurrency and throughput–more traffic flowing on the highway!

# Thread management and auto-tuning in Mule 4

Mule 4 eradicates the need for manual thread pool configuration as this is done automatically by the Mule runtime.

## Centralized thread pools

Thread pools are no longer configurable at the level of a Mule application. We now have three centralized pools:

- CPU_INTENSIVE
- CPU_LITE
- BLOCKING_IO

All three are managed by the Mule runtime and shared across all applications deployed to that runtime. A running Mule application will pull threads from each of those pools as events pass through its processors. The consequence of this is that a single flow may run in multiple threads. Mule 4 optimizes the execution of a flow to avoid unnecessary thread switches.

## HTTP thread pools

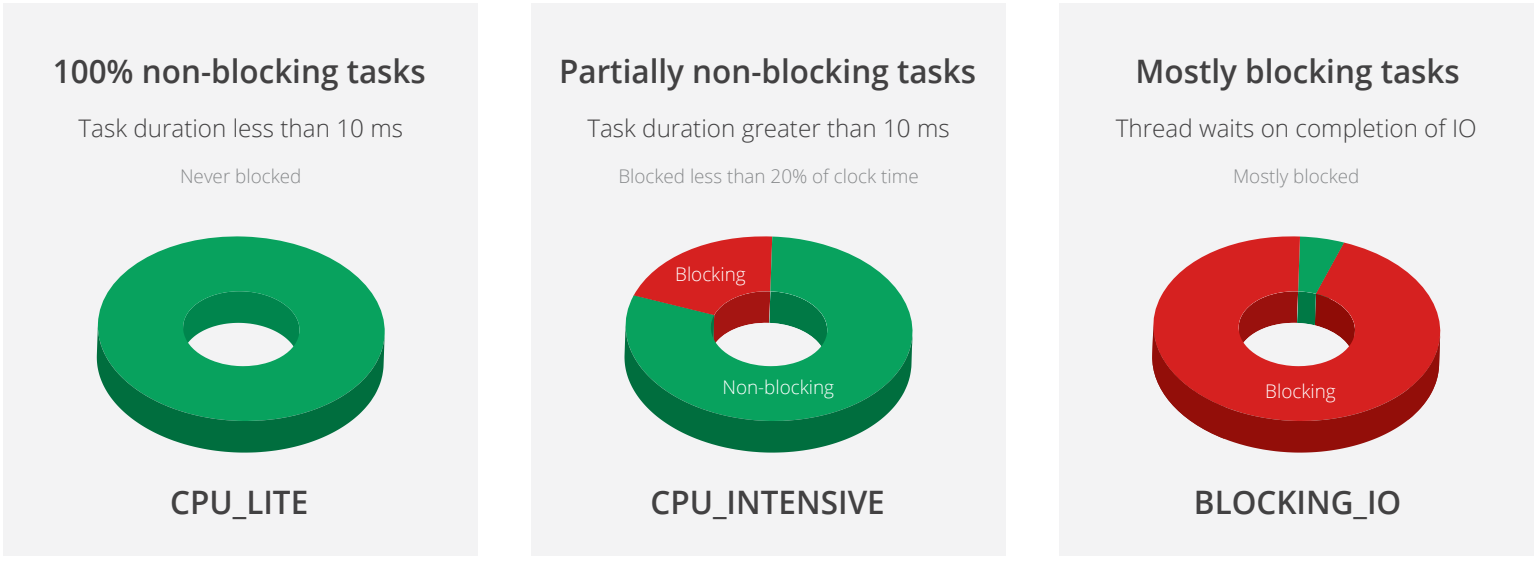
The Mule 4 HTTP module uses Grizzly under the covers. Grizzly needs selector thread pools configured. Java NIO has the concept of selector threads. These check the state of NIO channels and create and dispatch events when they arrive. The HTTP Listener selectors poll for request events only. The HTTP Requester selectors poll for response events only.

There is a special thread pool for the HTTP Listener. This is configured at the Mule runtime level and shared by all applications deployed to that runtime. There is also a special thread pool for the HTTP Requester. This is dedicated to the application that uses a HTTP Requester. So 2 applications on

the one runtime both using a HTTP Requester will have one selector pool each for that HTTP Requester. If they both use a HTTP Listener they will share the one pool for the HTTP Listener.

## Thread pool responsibilities

The source of the flow and each event processor must execute in a thread that is taken from one of the three centralized thread pools (with the exception of the selector threads needed by HTTP Listener and Requester). The task accomplished by an event processor is either 100% nonblocking, partially blocking, or mostly blocking.




The CPU_LITE pool is for tasks that are 100% non-blocking and typically take less than 10ms to complete. The CPU_INTENSIVE pool is for tasks that typically take more than 10ms and are potentially blocking less than 20% of the clock time. The BLOCKING_IO pool is for tasks that are blocked most of the time.

## Thread pool sizing

The minimum size of the three thread pools is determined when the Mule runtime starts up.

| Pool | Min | When? | Max | Increment |
|---|---|---|---|---|
| CPU_LITE | #cores | Mule startup | 2 x #cores | 1 |
| CPU_INTENSIVE | #cores | Mule startup | 2 x #cores | 1 |
| BLOCKING_IO | #cores | Mule startup | #cores + mem–245760)/5120) | 1 |
| GRIZZLY (shared) | #cores | Deployment of first App using HTTP Listener | #cores + 1 | 1 |
| GRIZZLY (dedicated) | #cores | Deployment of each App using HTTP Requester | #cores + 1 | 1 |

The minimum size of the shared Grizzly pool for the HTTP Listener is determined upon the deployment of the first app to the Mule runtime that uses an HTTP Listener. The size of the dedicated Grizzly for the HTTP Requester pool is determined upon deployment of each app that uses a HTTP Requester.

In all cases the minimum number of threads equals the number of CPU cores available to the Mule runtime. Growth towards the maximum is realized in increments of one thread as needed.

The maximum size of the BLOCKING_IO thread pool is calculated based on the amount of usable memory made available to the Mule runtime. This is determined by a call the Mule runtime makes when it boots to Runtime.maxMemory().

For a Mule runtime sitting on a 2 core / 1 Gig machine or container the following table shows what the minimum and maximum values are for each thread pool.

| Example | Pool | Min | Max |
|---|---|---|---|
| 2 core CPU<br>1 Gig RAM | CPU_LITE | 2 | 4 |
| | CPU_INTENSIVE | 2 | 4 |
| | BLOCKING_IO | 2 | 151 |
| | GRIZZLY | 2 | 3 |

## Thread pool scheduler assignment criteria

In Java the responsibility of managing thread pools falls on the Scheduler. It pulls threads from the pool and returns them and adds new threads to the pool. Each of the five pools we described above has its own Scheduler. When a Mule 4 app is deployed each of its event processors is assigned a Scheduler following the criteria outlined in the following table.

| Scheduler | Event Processors |
|---|---|
| CPU_INTENSIVE | › DataWeave (just the transform processor)<br>› Scripting Module |
| BLOCKING_IO | › All blocking IO modules (e.g. Database)<br>› Transactional scopes |
| GRIZZLY (shared) | HTTP Listener |
| GRIZZLY (dedicated) | HTTP Requester |
| CPU_LITE | All other event processors, scopes and routers |

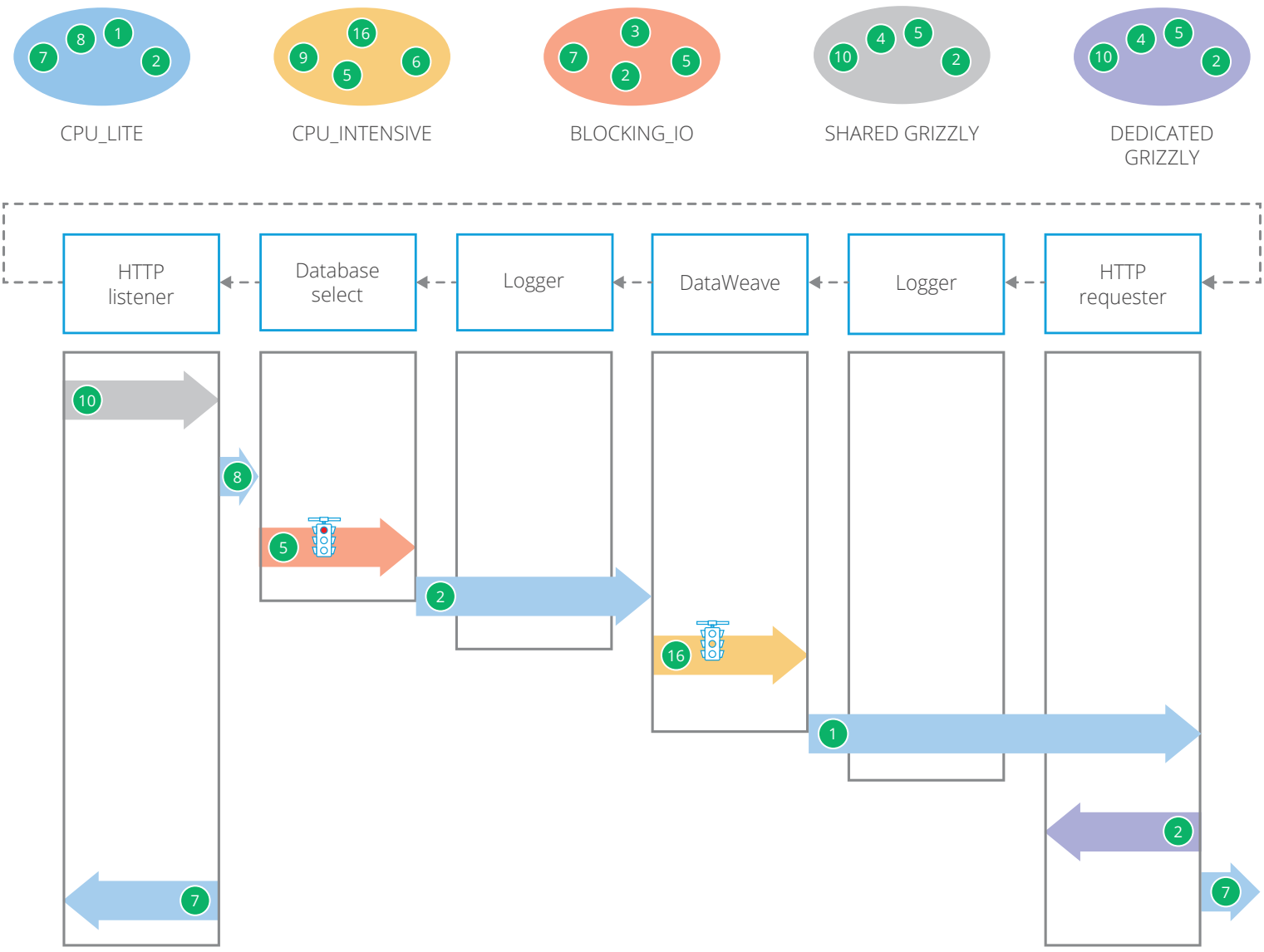
An important consideration is the handoff between each event processor. That is always executed on a CPU_LITE thread.

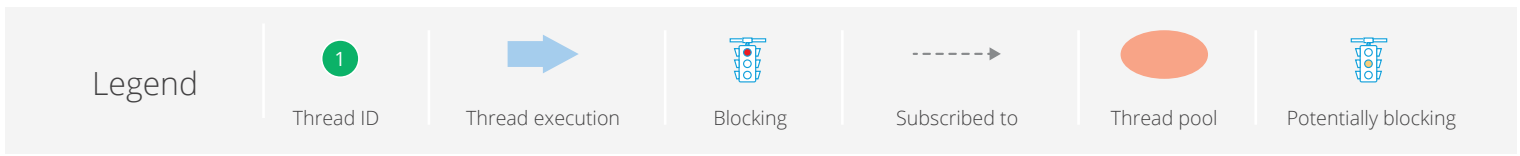
Over time our engineers will enhance modules to make their operations non-blocking.

## Mule runtime example consumption of thread pools

The following diagrams show how threads are assigned in various types of Mule flow. Watch out for the red traffic light, which denotes a blocking operation (BLOCKING_IO). The amber traffic light denotes potential partial blocking (CPU_INTENSIVE). The space or handoff between each event processor is non-blocking and catered to by a CPU_LITE thread. Nevertheless, the optimization in the thread schedulers will avoid unnecessary thread switching so a thread from a given pool can continue to execute across processors as we shall see.

# Typical thread switching scenario



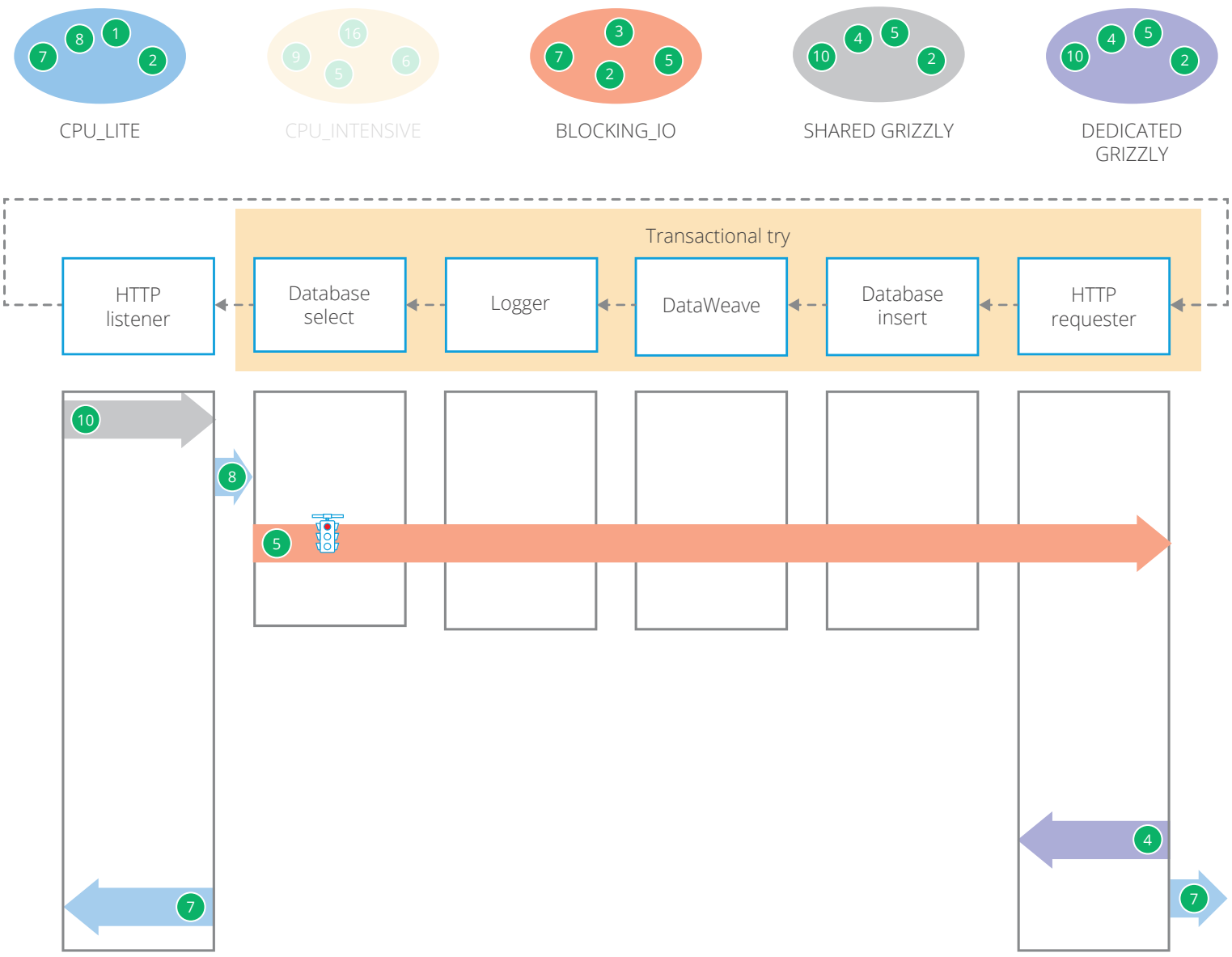



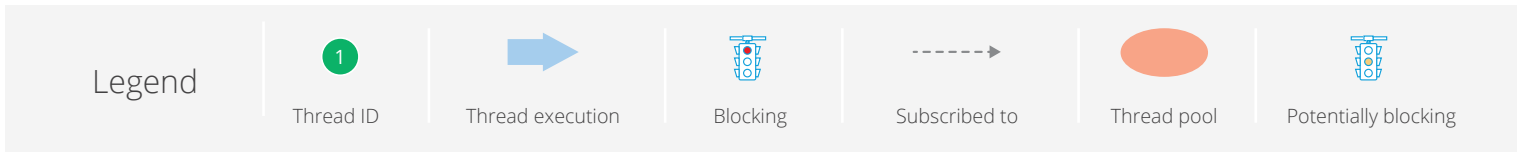

In this first scenario:

- *SHARED_GRIZZLY Thread #10* receives the HTTP Listener request.
- *CPU_LITE Thread #8* caters to the handoff between the HTTP Listener and the Database select operation.
- *BLOCKING_IO Thread #5* must make the call to the database server and then wait for the result set to be sent back.
- A thread from CPU_LITE is needed for the Logger operation but Scheduler optimization allows for *CPU_LITE Thread #2* to also be used for the handoff before and after it.

- *CPU_INTENSIVE Thread #16* executes the DataWeave transformation. DataWeave always takes a thread from this pool regardless of whether blocking actually occurs.
- A similar optimization occurs on the second Logger and handoffs with *CPU_LITE Thread #1* also making the outbound HTTP Requester call.
- *DEDICATED GRIZZLY Thread #2* receives the HTTP Requester response.
- There is an optimization on the response after flow completion: *CPU_LITE Thread #7* does the handoff back to the flow source (HTTP Listener) and also executes the response to the client.

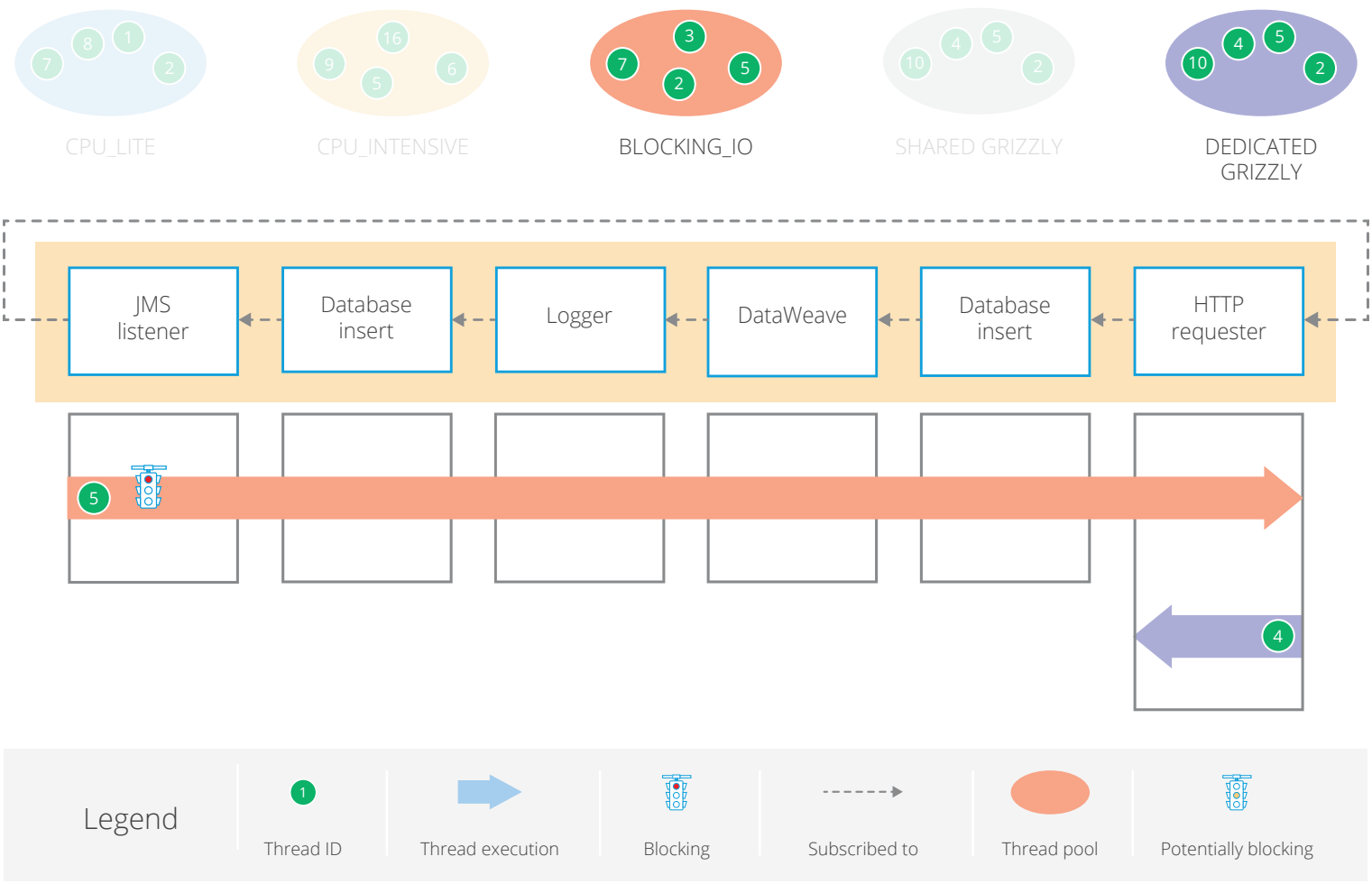
# Try scope with Transaction



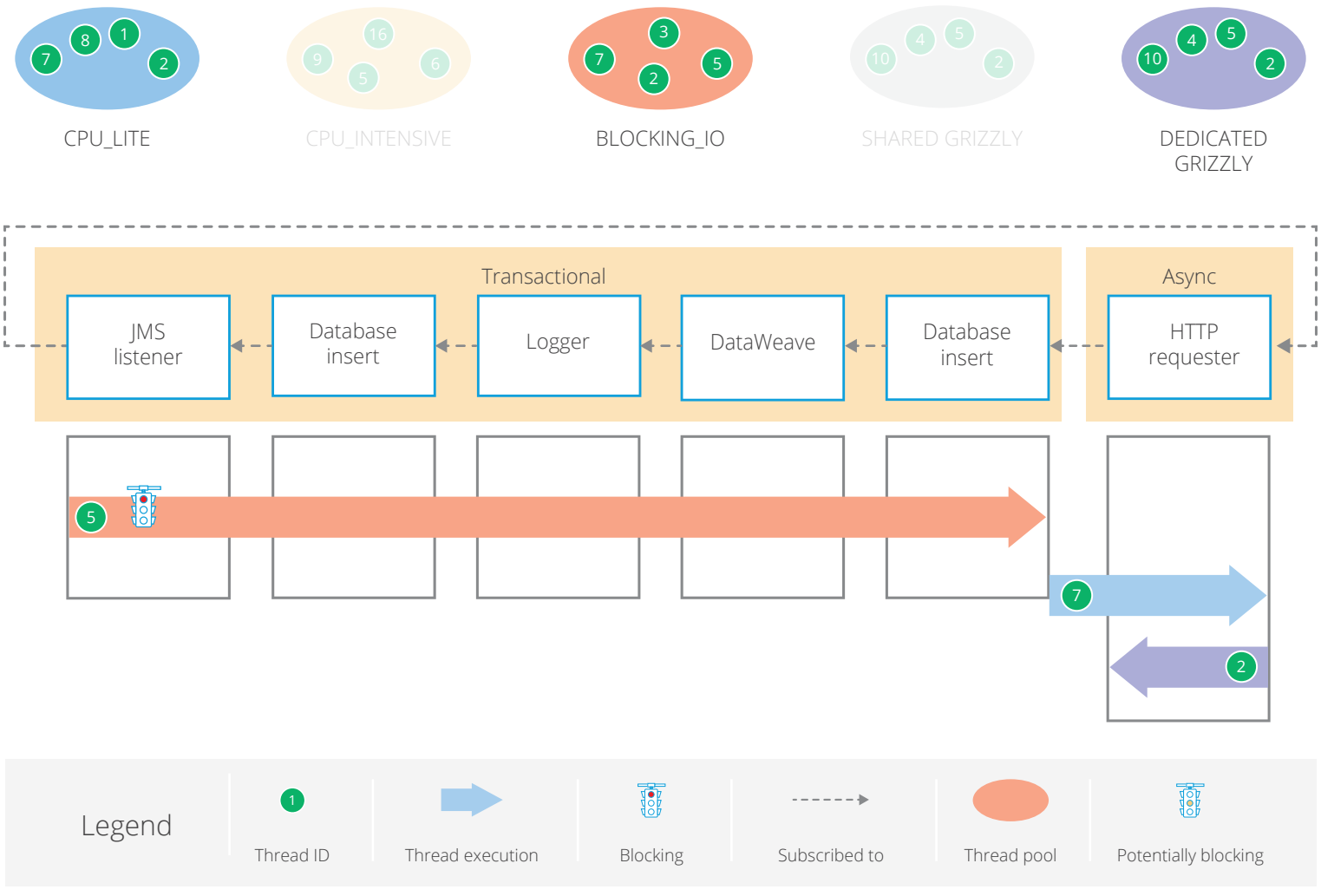

Here the Transactional Try scope mandates the use of a single thread. This will always be from the BLOCKING_IO pool regardless of what type of operations are contained within the scope.

# JMS Transactional




In this scenario the whole flow is transactional and requires a single thread from BLOCKING_IO up to the HTTP request.

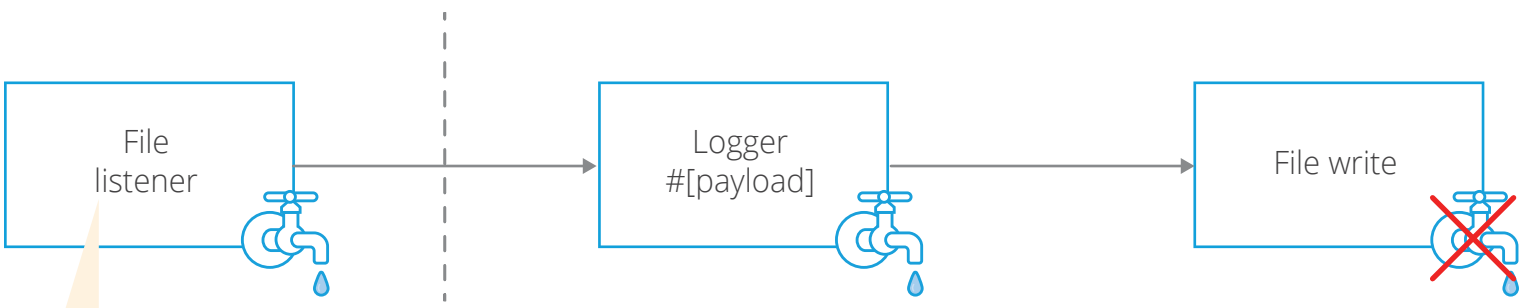
# JMS transactional with Async scope




In this scenario the Async scope ends the transaction and normal thread selection applies.

# Streaming in Mule 4

Mule applications have always been able to process input streams. Mule 4 now enables an already-consumed stream to be consumed by a subsequent event processor. Instead of having to read a stream into memory for repeated processing of the payload, Mule 4 event processors can consume the same stream both in sequence and in parallel. This is the default setting. There's no need for explicit configuration. To get a Mule 4 application to behave in the same way as a Mule 3 application you have to manually configure the source of the stream with the <non-repeatable-stream/> configuration.




```
<non-repeatable-stream/>
```

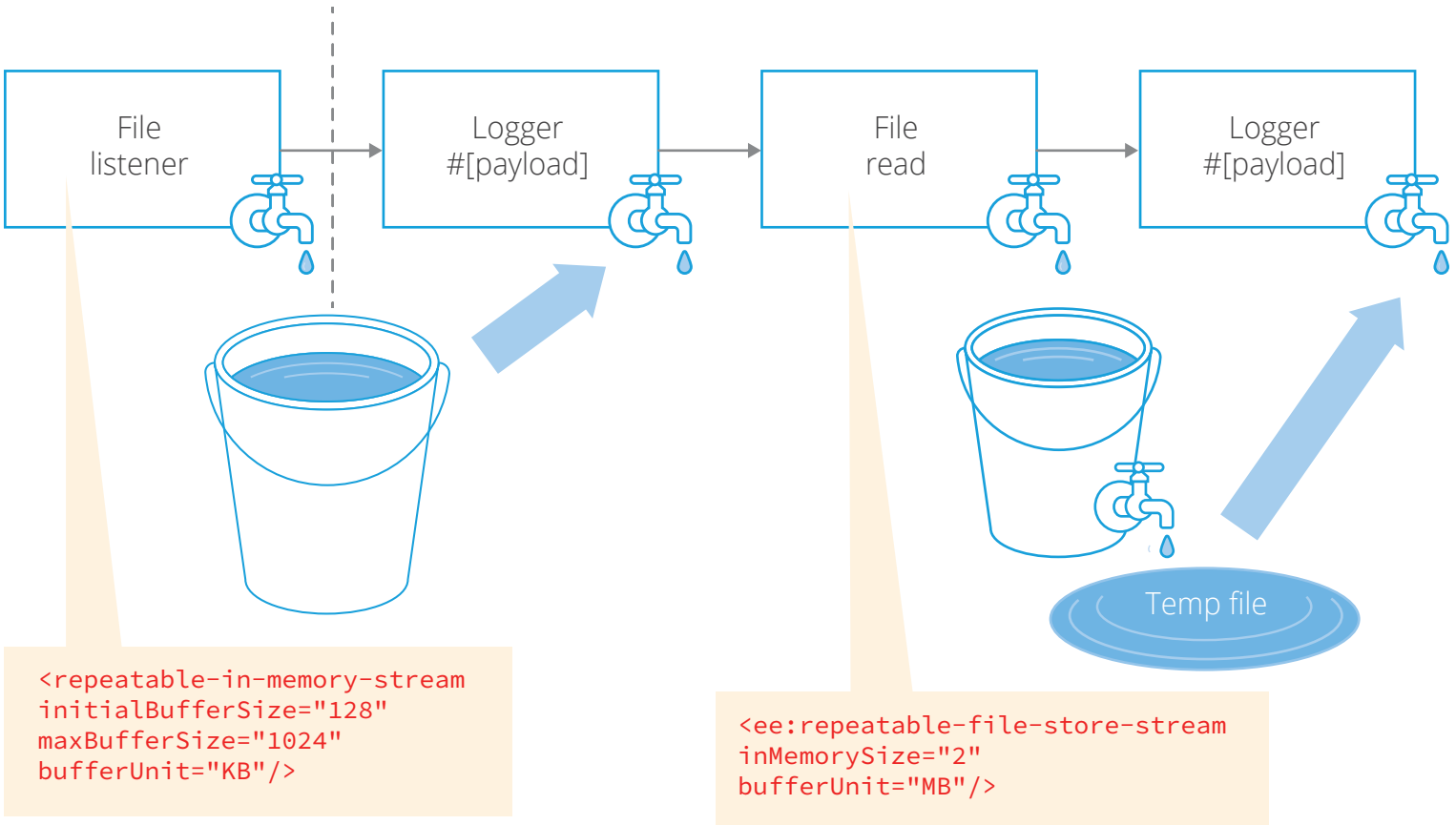
In the above scenario the File Listener will pass a non-repeatable stream to the chain of event processors. Note that because the Logger consumes the stream it is no longer available to the File write operation. An error will occur because it tries to consume the stream after it was already consumed by the previous Logger operation.

## Repeatable and concurrent streams

Mule 4 introduces repeatable and concurrent streams so that the above scenario will work by default. A stream can be consumed any amount of times in a flow. Concurrent stream consumption is also possible so that the multiple branches of a scatter-gather scope can each consume a stream the scope may receive on a flow.

## Binary streaming

Binary streams have no understanding of the data structure in their stream. This is typical of the HTTP, File, and SFTP modules.

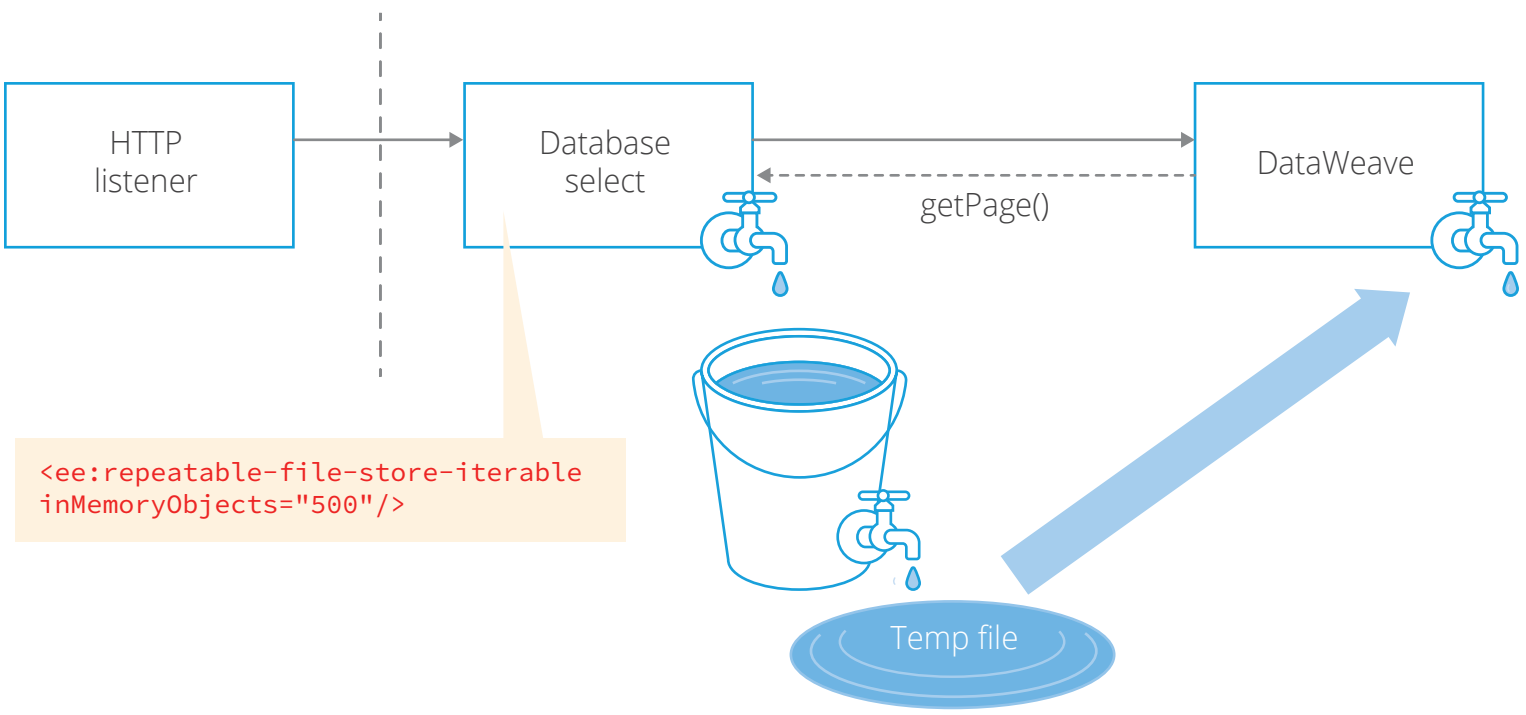



There are two ways to configure repeatable binary streams for event processors in Mule 4: in-memory and file-based. The in-memory configuration allows you to start with an initial size buffer which will increase in size to a max. The file-based configuration allows you to configure the invariant size of a memory buffer which will be filled before any overflow is stored to a temp file. This is transparent to the Mule developer. In the scenario above the developer doesn't worry about how much of the memory buffer was used or whether any data was stored to file.

## Object streaming

Object streams are effectively Object iterables. They are configurable for operations that support paging, like the Database select operation or the Salesforce query operation. The configuration is similar to the binary streams and also applied by default. Those operations that support paging return a *PagingProvider* (more on this when we explore the

Mule SDK). The next event processor in the flow will transparently call the *PagingProvider.getPage()* method as it iterates through the stream of objects.




In the above example the Database select operation passes a *PagingProvider* reference to the DataWeave event processor. DataWeave will call the *PagingProvider.getPage()* method as it iterates through the stream of objects. The configuration on the Database module establishes how many objects are stored in memory before the rest are persisted to a temp file.

## Tuning strategy for streaming

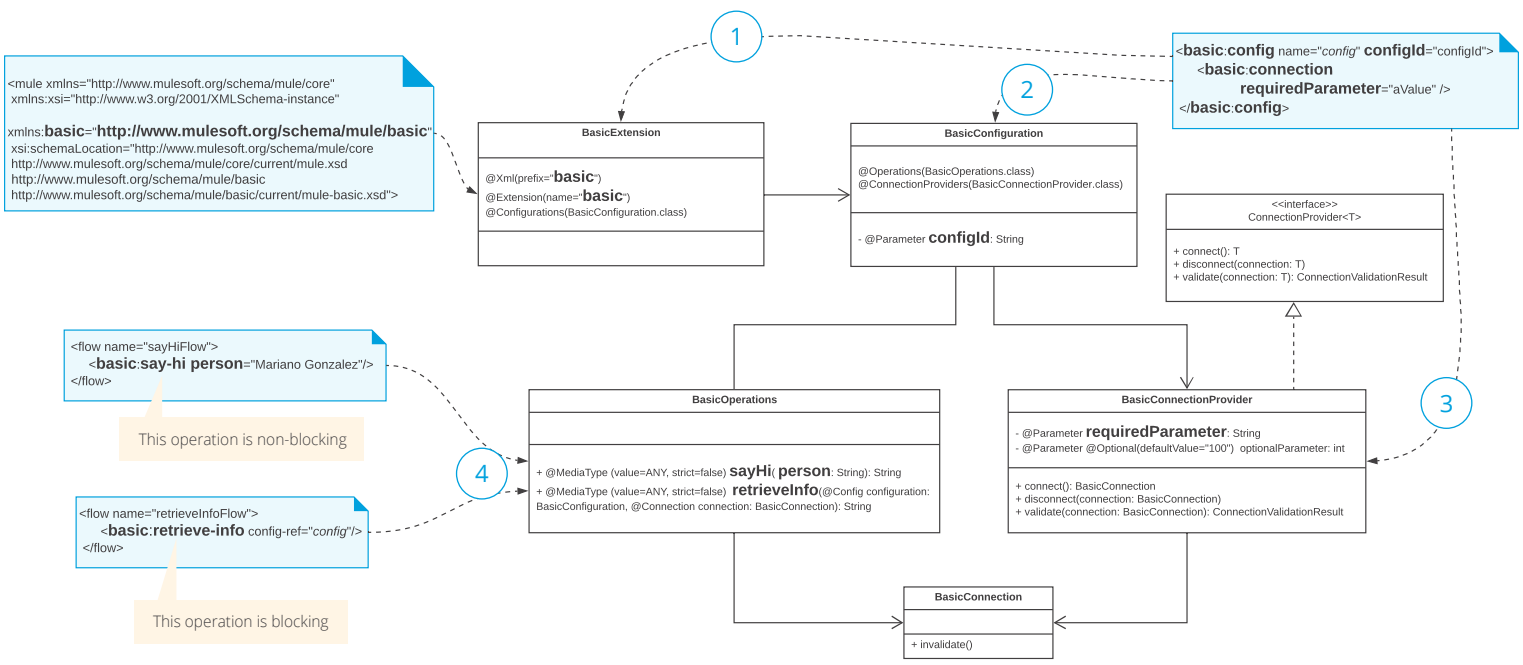
Because there is no global streaming configuration for the Mule application (or the Mule runtime) you do well to calculate the product of your max expected concurrency and the totality of the in-memory buffers for the application. You should then limit the amount of events being processed by a streaming operation using the maxConcurrency configuration. When the maxConcurrency is reached back pressure will be triggered to the flow source.

# Scalability features in Mule 4 SDK

Mule 4 is fully extensible. The SDK allows you to add modules that are treated as first-class citizens alongside the out-of-the-box modules. The SDK exploits the same thread management and streaming configurations described above.

## SDK landscape

A module is a collection of independently configurable components: sources, functions, operations, routers and scopes. The components are optionally configurable to accept parameter values particular to their use in a Mule App, and which are applied either at deployment time or at runtime through Dataweave expressions. A component can have multiple configurations or none. A module configuration may require connections to external systems. This is often the case for operations.




The SDK comes with a Maven archetype to get you started. The above paints a picture of the classes of the *Basic* module it generates. None of the class names are important. Their association is realized through annotations.

1. The *BasicExtension* class defines the basic namespace through the use of annotations. It uses the **@Configurations** annotation to refer to its optional set of configurations. You can have zero or multiple configurations on the one module.

2. The *BasicConfiguration* class has an **@Parameter** annotated configId field which facilitates unique identification of the configuration for the Mule app to which the module is deployed. External connectivity is optional in modules. The BasicConfiguration does require connectivity and has an **@ConnectionProvider** annotation naming the *BasicConnectionProvider* class, which will act as a factory of the *BasicConnection* class. The class also has an **@Operations** annotation pointing to the *BasicOperations* class.

3. The *BasicConnectionProvider* class has **@Parameter** annotated fields that appear in the configuration. You can also use annotations that affect optionality and how Studio is configured.

4. The *BasicOperations* class has two public methods. All public methods are separate operations. The signatures of these methods is important to determine which thread pool schedulers are assigned to them and how streaming will work.

## Thread pool scheduler assignment criteria for operations

The SDK infers the types of execution of an operation automatically. It sees all operations falling into either blocking or non-blocking categories. So either the CPU_LITE scheduler or the BLOCKING_IO scheduler will be assigned to the operation upon deployment.

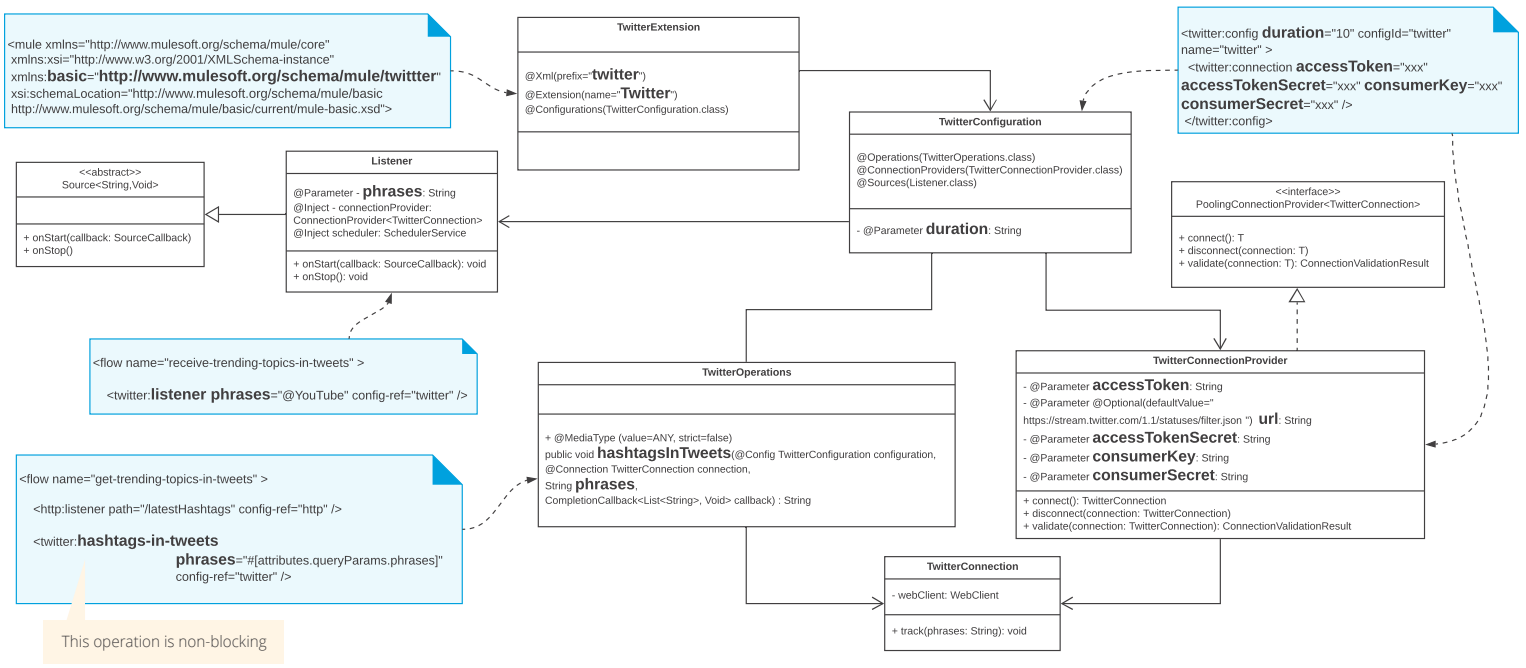
1. If there is no @Connection annotated parameter in the method then the operation is considered non-blocking and the CPU_LITE scheduler will be assigned.
2. If there is an @Connection annotated parameter in the method then:
    a. If the signature has a void return type and has a CompletionCallback parameter then the operation is considered non-blocking and the CPU_LITE scheduler will be assigned.
    b. If the signature has a return-type then the operation is considered blocking and the BLOCKING_IO scheduler will be assigned.

There is also a method level annotation, eg. @Execution(CPU_INTENSIVE), you can use to manually determine the scheduler that should be assigned. This will override the above automatic inference.

## Example Twitter module

Let's say we want to build a module that consumes the Twitter streaming API to receive a stream of tweets that match a particular set of phrases but returns only a list of hashtags to the next event processor on the flow.

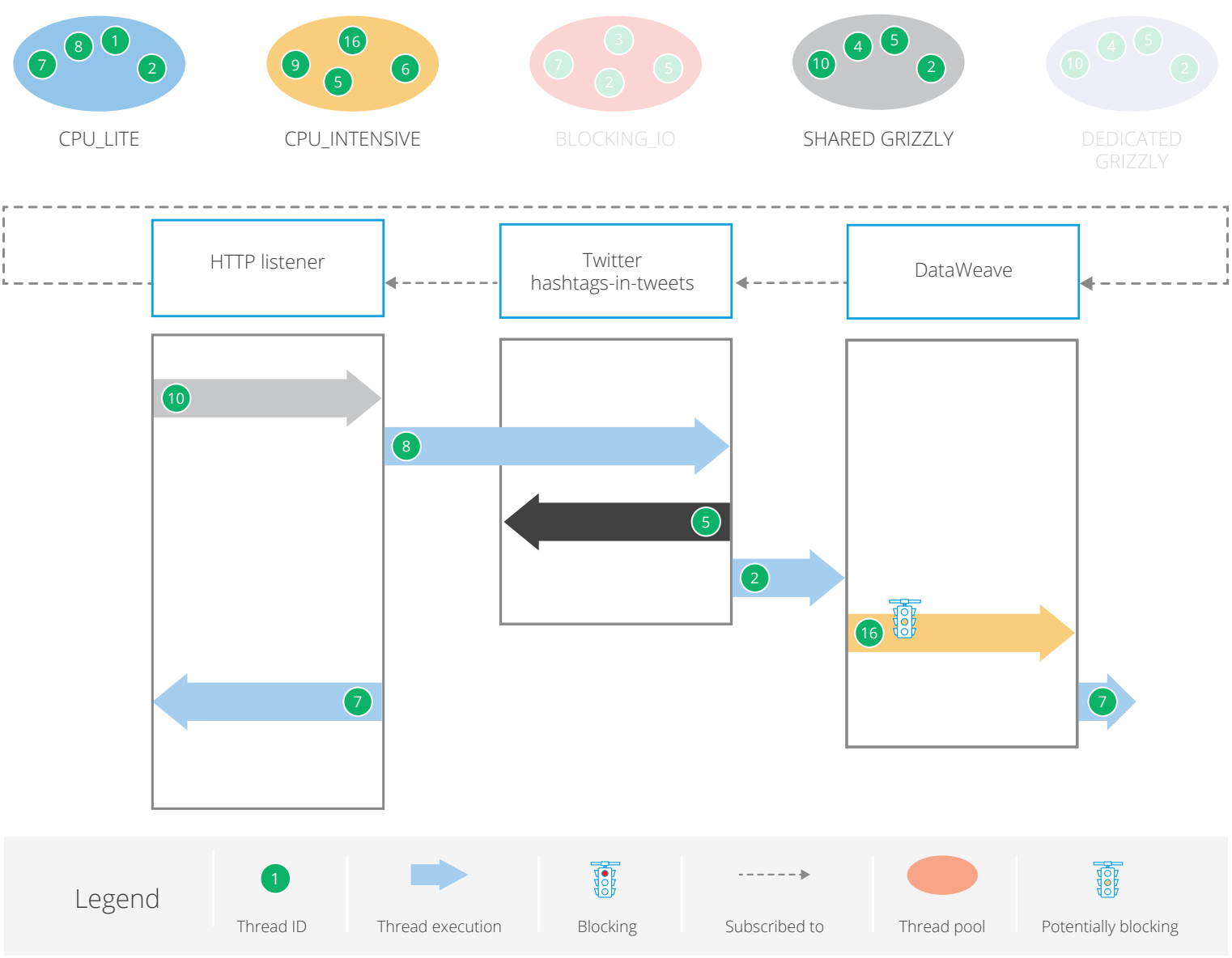
We can make the *hashtags-in-tweets* operation non-blocking by setting its return type to void and including a CompletionCallback in its parameter list. We'll use this to send the hashtags back.




The operation implementation is worth a glance as we can utilize the Spring Reactor SDK we described in the beginning. It subscribes to tweets matching the phrases parameter, then creates a new flux of hashtags found in those tweets for 10 seconds, removes duplicates, sorts them and returns them as a List.

```
public void hashtagsInTweets(@Config TwitterConfiguration configuration,
                             @Connection TwitterConnection connection, String phrases,
                             CompletionCallback<List<String>, Void> callback) {
connection.track(phrases) // flux of tweets
        .doOnError(e -> callback.error(e))
     .map(tweet -> tweet.getText()) // to flux of their texts
     .flatMap(text -> { // to flux of hashtags in texts
        List<String> allMatches = new ArrayList<String>();
        Matcher m = HASHTAG_PATTERN.matcher(text);
        while (m.find()) allMatches.add(m.group());
        return Flux.fromIterable(allMatches);
})
     .take(Duration.ofSeconds(configuration.getDuration())) // limit
     subscription to x seconds
     .distinct() // to flux of only unique hashtags
     .sort() // to flux of sorted hashtags
     .collectList() // to list
     .subscribe(list -> callback.success(Result.<List<String>, Void>builder().
     output(list).build()));
}
```

The operation is assigned the CPU_LITE scheduler. Note how the callback builds a response using *Result.builder()* inside the function you pass to *the .subscribe()* method. This will be executed asynchronously when the response comes back from Twitter. The thread it executes on will be determined by the library you choose to make the HTTPS call to Twitter.




## Binary and object streaming

All operations which have InputStream parameters or return an InputStream can exploit the binary streaming configurations described above. To exploit Object streaming in your operation you need to return a *PagingProvider<Connection, String>* interface so the next event processor on the flow can call its *getPage()* method. The following module consumes the CKan API. The important detail here is that it keeps track of the

pageIndex across the *getPage()* calls. When *getPage()* returns an empty List it won't be invoked anymore.

```
public class CKanOperations {

     @MediaType(value = ANY, strict = false)

     public PagingProvider<CKanConnection, String> packageList() {

         return new PagingProvider<CKanConnection, String>() {

             private int pageIndex = 0;

             @Override
             public List<String> getPage(@Connection KanConnection connection) {
                 List<String> packages = null;
                 packages = connection.packageList(this.pageIndex);
                 this.pageIndex += packages.size();
                 return packages;
             }

             @Override
             public void close(CKanConnection connection) throws MuleException {}

             @Override
             public Optional<Integer> getTotalResults(CKanConnection arg0) {
                 return Optional.empty();
             }

             @Override
             public boolean useStickyConnections() {
                 return false;
             }

         }

}
```

# Learn more

Mule is highly scalable both horizontally and vertically. The topics we covered here address vertical scalability. Automatic tuning of deployed Mule applications allows them to execute in a non-blocking fashion that exploits all resources available to the Mule runtime. Threads are assigned to the application in a way that maximizes concurrency. Memory and disk are utilized in a way that facilitates stream processing of larger than memory payloads. All of these efficiencies are fully exploitable in the Mule SDK if you wish to author your own modules. Try Mule 4 today to see for yourself how to address vertical scalability in an innovative and effective way.

# About MuleSoft

## MuleSoft, a Salesforce company

MuleSoft's mission is to help organizations change and innovate faster by making it easy to connect the world's applications, data, and devices. With its API-led approach to connectivity, MuleSoft's market-leading Anypoint Platform™ empowers over 1,600 organizations in approximately 60 countries to build application networks. By unlocking data across the enterprise with application networks, organizations can easily deliver new revenue channels, increase operational efficiency, and create differentiated customer experiences.

For more information, visit **mulesoft.com**