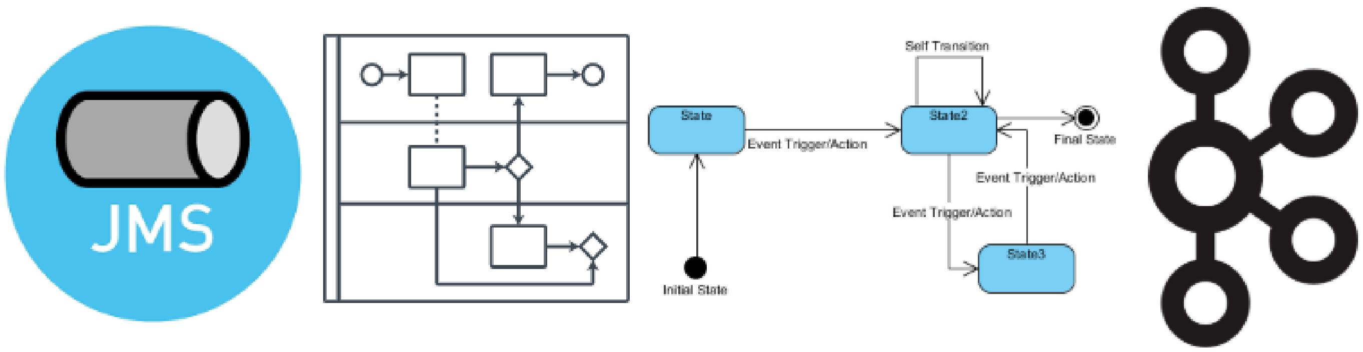
Tricky messaging, part two: business processing

Victor Alekseev Follow
Jul 21 · 37 min read



To begin with, let’s remember what we have already achieved in the previous article **“Synchronous transport and not only”** and what we can use as a technical basis.

We are currently able to:

- Make synchronous and asynchronous calls through JMS queue
- Minimize the blocked time of threads on both sides, as well as the transaction time on the database
- Provide scalability and fault tolerance, significantly reduce client-server cohesion
- Implement “weakened” pseudo-sessions (partitioning, runtime sharding), where requests from the same user are serviced possibly by the same server

In this article, I’d like to look at more specific message handling patterns and anti-patterns that are often used when processing large pieces of data. A typical use-case is the processing of complex structured documents, such as financial or customs documents.

Runtime sharding

This last feature may seem not very useful in a situation where a group of stateless servers handles independent ready to process documents or requests coming from the user interface.

However, this is not entirely true, because to process each document, we need not only its own data, but also a lot of data from the business context in which the operation is requested.

For example, if we process payment requests as a feature of some budgeting system, we need to load from the database before processing each request:

- The data of the categories for which the transaction is requested. Some budgetary limits are likely defined for each category. Also, we can meet with the possibility of limiting the types of requests accounted for by these categories. Usually, such the category system is hierarchical, and we will also need to load not only one but also all parent categories.
- For additional verification of the user's electronic signature under the request, we have to load their rights regarding the requested transaction. It needs to find out whether the user is allowed to sign requests of this type for a specified amount of money in the context of his organization.
- Users may go on holiday from time to time, in which case they will delegate their signature rights to their colleagues. Accordingly, we need to download the user's signed "delegated authority act" and adjust the current permissions set available to him.
- Administrators can define different signature templates for different types of documents and different categories — sets of signature types whose presence is required to process the document further. For example, the author of a document may perform the "first" signature and their line manager the "second" signature. Alternatively, there may be responsible supervisors in an organization with the right

of power signature, allowing the document to be drawn up and sent for processing alone.

This list is quite far from completion but gives an idea of how much data may need to be read from the database to process a request to buy an ordinary cleaning rag. Suppose we are dealing with a multi-tiered holding company, where the same persons may have different responsibilities in different organizations and may subordinate to each other in various combinations. In that case, it becomes pretty difficult to load all this volume of descriptions every time.

That is why we try to cache the data loaded from the database as much as possible.

Data classification

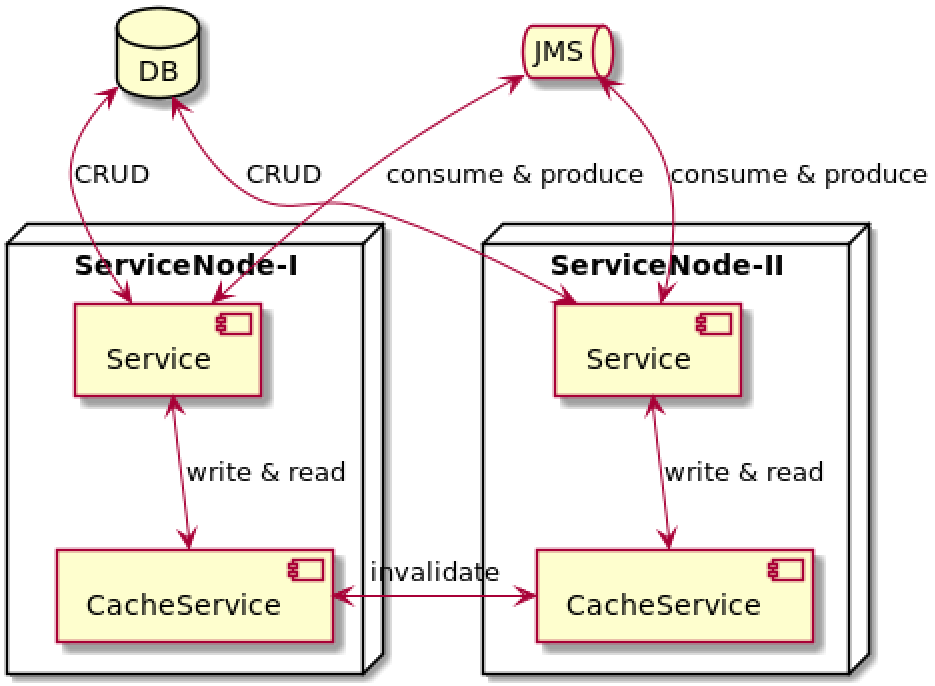
Regarding the caching process, we can distinguish the following common categories of data:

- Permanently cached data. These are usually constants within the system source code or values defined in the application configuration. That's why, they are often system or technical by nature. For example, a timeout in the processing of a document.
- Long-term cached data or "common dictionaries". Typically, this is relatively infrequently updated data, which is reloaded into memory periodically based on a TTL condition long enough. For example, a list of holidays or the organizational structure of a holding company.
- Operational dictionaries, which contains dynamically changeable data that are not the subject of business transactions. For example, user rights to carry out operations on a particular category. Usually, such data is stored with a small TTL or invalidated when a new value is written to the database
- Non-cacheable data. This is usually business transaction data, such as the documents being processed themselves. It also includes data reflecting the business state of the system as a whole, which is rapidly updated as individual transactions occur. For example, it may be current balances of budgetary limits for specific categories.

System design

Typically, data is cached by Infinispan-like solutions that provide out-of-the-box support for JPA second-level cache. The cache itself can either be embedded into the application or deployed side-by-side (look at the “sidecar” Kubernetes pattern, for example) on the same cluster node to ensure the small response times using the loopback network interface. The second option is more flexible, as it effectively differentiates the amount of resources (memory / threads, for example) available to the service itself and for the cache that supports it. The cache instances usually communicate together to ensure that updated by one service’s data is evicted from all instances of the cache.

Thus we can often encounter the following business document processing architecture:



This all works fine as long as we have little data or the entire amount of cacheable frequently used data fits into the memory of one caching service. If we exceed this amount and keep processing documents on random nodes in the cluster, the cache becomes highly inefficient. It happens because the caching process has to clear memory for each non-cached data item to load it, discarding any other data items selected by some strategy. Unfortunately, to process the following query, this data item may with some probability be needed again, leading to additional database queries, CPU and memory usage.

The remedy is runtime sharding when documents requiring the same context are mainly processed on the same server nodes.

Some notes about

In conclusion, there are just a few points to be mentioned:

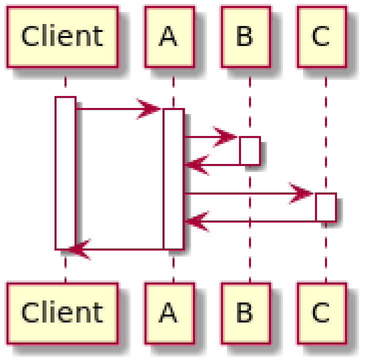
- I want to especially emphasize that this is not data sharding but sharding of computing resources. All instances of the service have access to the same data storage.
- At any time, all this request traffic relating to one data context can forward automatically to another service instance. Unfortunately, it consequently causes a massive update of the cache data.
- If it is not a problem for us to be locked to ActiveMQ, we can achieve the same result without the complex processing and using the standard **JMSXGroupId** message header.

Another common problem with this scheme is ensuring that the cache instances communicate with each other. This problem is particularly painful and may be met often in the case of cloud infrastructure. There are also situations where, due to security requirements, we need to locate individual service instances in separated specific secure network segments.

In enterprise solutions, we can fix such issues by the organization a dedicated topic in which application services publish requests for data invalidation. All of the service's instances are also subscribed to this topic. On receipt of the message, they invoke the invalidation operation on the corresponding cache instance.

Raw queue-based processing

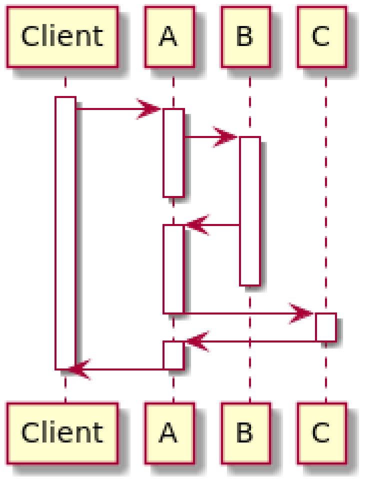
When a client calls a single service doing all the query processing alone, all is well: the thread lock time and transaction duration in the database are minimal. But as soon as we come to a distributed system, things get a bit worse. For example, if service A needs to call services B and C in turn and use the results, we have the following scenario:



Thus service A is blocked while service B is called (just as the client was blocked before while service A is called). So, if the client's request is handled by the whole tree of calls to different services of the system, then we benefit only when we call leafy endpoint services.

Splitting calls

To fix this situation, we can split each operation into several parts, which activate each other in sequence by forwarding messages. For example, if service A calls service B and C in turn, we may observe the following sequence of actions:



This architecture provides us with the following benefits:

- Services B and C are still called optimally as leaf services
- The called operation of service A is split into three parts, each of which is optimized in terms of blocking the involved resources.

- Even the most “creative” developer would find it challenging to apply a distributed transaction here (quite harmful anti-pattern).
- The fact that JMS does not retain the order of close messages in the message queue is irrelevant for us in this case. It happens because requests to services B and C cannot be swapped in principle.

However, the scheme is not without some disadvantages too:

First of all, the number of queues in the system is increasing dramatically.

- This in itself is not usually a serious problem, but each queue is an asynchronous transfer of control from one fragment of code to another. As a result, once a certain amount of functionality has been achieved, it becomes tough to understand what is calling where and what sequence of calls is behind each business operation. It is not only not convenient, but it is also not scalable as the number of developers, and the amount of functionality implemented grows.
- Over time, the application becomes a tangle of queues through which everyone calls everyone. Inevitably there are queues that no one writes to them. Worse, there are also queues that people write to, but no one reads. As business processes usually operate with durable queues, such orphans inevitably slow down the overall data processing.

The second problem is more severe and deal swith the fact that service A needs to pass the context of the original call between the individual parts of the asynchronously activated code.

This context can include:

- at least the source message identifier to use as the correlation header of the final response
- some data from source request, which can only be applied after receiving a response from B
- data from response B, which is not needed per se, but maybe needed if service C returns an error and the operation on service B has to be rolled back

Since a message to service B can be sent by one instance of service A and received by another, we have the following ways to pass the context:

- transferring the context within the message. This approach is quite elegant since we thus have some kind of distributed processing stack trace at each message processing point, This stack trace describes all previous services in the call chain. However, this approach can only be applied if the volume of transmitted data is small enough. In the enterprise world, this is not too often, although it does happen from time to time.

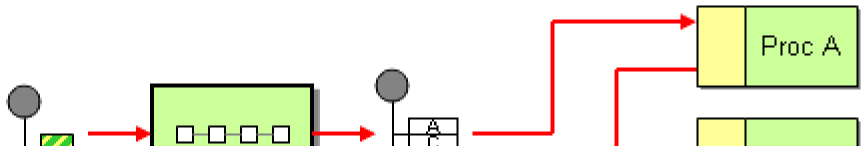
- storing the context is in some shared repository. In the enterprise world it is usually a database, but we can use the distributed cache too. On receiving the next reply, a fragment of the service application code loads the business process context from the database, applies data received in response, updates the context, saves it, and sends the subsequent request.

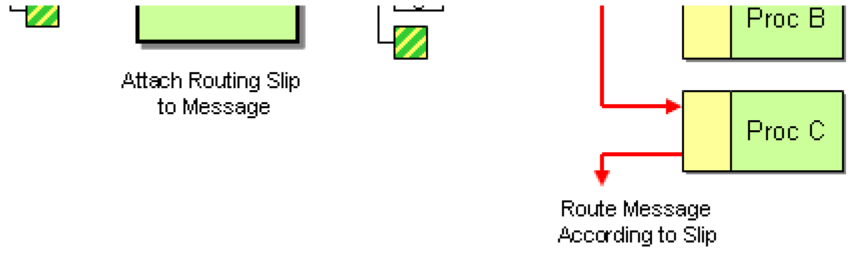
The third difficulty with this approach is a possible call timeout. Service A can passively and endlessly wait for a response from service B. If service B does not return an answer for some reason, then service A will never know about it and will not be able to take any compensatory action. It is especially painful when the response from service C is lost, and therefore it is not possible to cancel the operation already performed at service B. To cope with this complication, we can, for example, log the messages sent in the stored context of the business process execution and periodically poll the repository to find any responses not received in time.

A less severe and fundamental problem is that the duration of a synthetic method call on service A cannot now be calculated automatically. Advanced tools like Java Melody will not help us. We are forced to mark business method boundaries scattered over different fragments of source code by ourselves.

The chain of calls

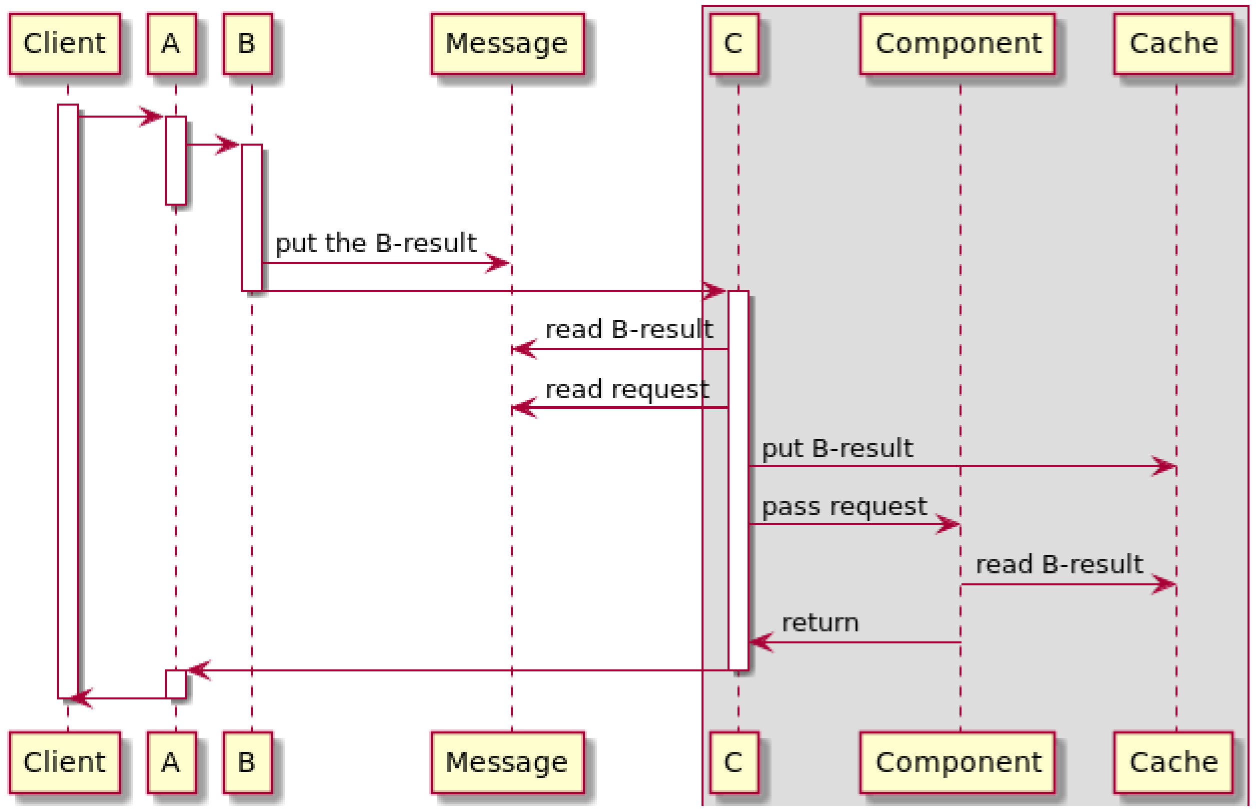
In some cases where the result of one service is used as the only argument for the next service, we can slightly optimize the interaction by applying the “Routing Slip” pattern.




Proc B
Proc C
Attach Routing Slip to Message
Route Message According to Slip

The thoughtless application of this template makes the services dependent on each other, which is undoubtedly a bad practice. However, you can do something more tricky:

- Separated independent requests to services B and C are placed in the original request from service A.
- A special section for intermediate responses is also provided as a standard part of the message. Service B places its response in this section before routing the request to service C (and only if such routing is requested).
- Service C receives the request, places all responses from the section into its local cache, and executes the request. When service C needs to perform a call to service B, it first searches the cache for the already available result of the request. If unsuccessful (when something calls the code, not in the context of the pattern under discussion), it starts the remote call.



Thus, we can call the methods of services B and C as before. Following this approach, it is possible to put inside a whole list of services to which the message should be consecutively sent and more advanced routing logic with conditional transitions and error handling logic. We will come back to this pattern in the next article about the user interface, when we will discuss the possibility of simplifying the code by partially eliminating event-driven development on the client side.

Also, this pattern is ideal for implementing cross-concepts, such as validation of data formats, authorization of a transaction requested by the client, verification of a set of signatures certifying transaction data, auditing, etc. In this case, the invoked services may not depend at all on the previously invoked infrastructure services or depend on them minimally.

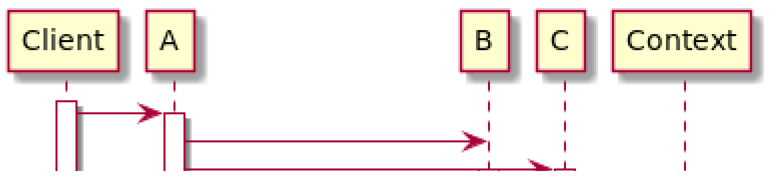
In some approximation, this design is a distributed implementation of the well-known "chain of responsibilities" pattern. Each service mentioned in the receipt can:

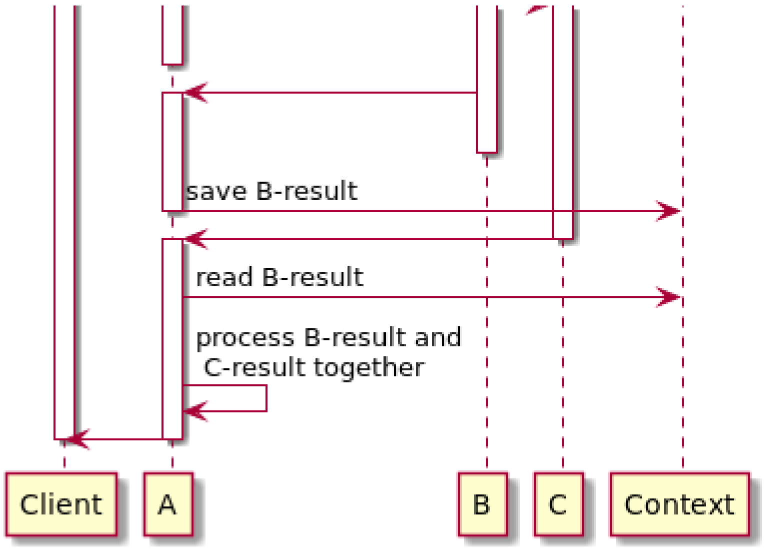
- return a response to the client
- pass the request to the next handler
- modify the receipt in terms of the set of further handlers or data/metadata transmitted to them

This template is also helpful for testing individual services because instead of providing mocked services-dependencies, you can provide their mocked responses immediately.

Parallel calls

If service A needs to make several independent calls to services B and C, we can perform these operations in parallel. We have to apply the same pattern as discussed in the previous article when dealing with large file transfers. In this case, the situation is more straightforward because we do not need to invent an operation context for intermediate storage of results specially — we already have one.





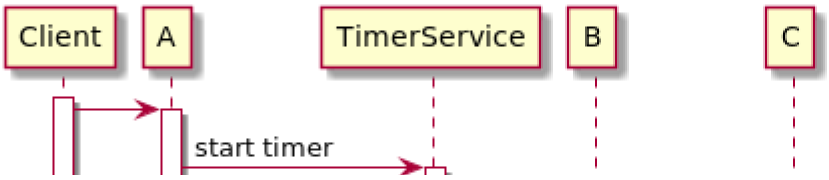
Queues and timers

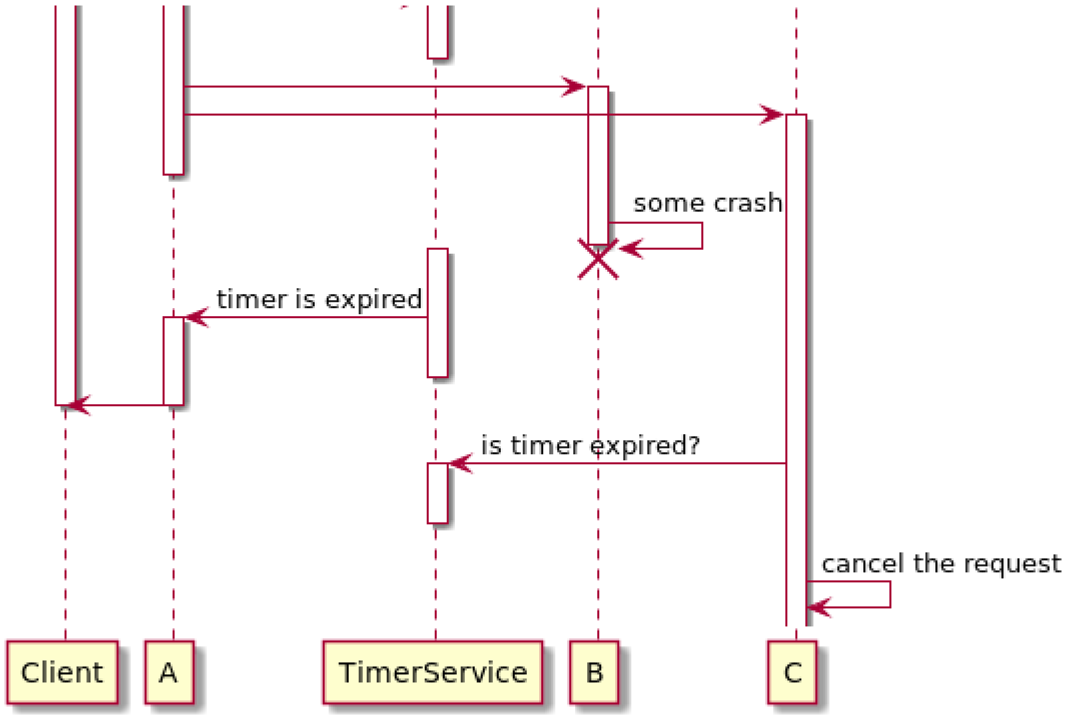
The last pattern in this group is to interrupt or initiate some kind of activity by a timer. The service can send a message to itself in the simplest case, adding the ActiveMQ-specific **AMQ_SCHEDULED_DELAY** header.

However, there are also more complicated situations where:

- MQ broker does not support such a feature at all
- We want to be able to cancel or alter the timer
- If the timer is triggered, we want to send more than one message to different services
- Other services should be able to find out if any timer has expired or not

So we can introduce a specialized service that stores the context of the operation and schedules internal events to activate them in time. It can achieve such effect based on continuous polling of the database (possible, but rather bad design) or by sending itself scheduled activating messages through JMS (In this case, we use the JMS broker as a cluster-level scheduler) or using a specialized solution like Quartz Enterprise Job Scheduler.





Scope of usage

Overall, the above interaction templates cover approximately 80% of the functionality required to implement business processes.

If your system uses only a few business processes with 2–5 asynchronous steps in each, you probably do not need more advanced technologies and frameworks.

Most likely, most of the rather technical services that will be accessed by instances of BPMN-based processes and state machines belong to this category.

The splendor and the misery of BPMN

The issues of applying BPMN is not directly relevant to the topic of this article. Nevertheless, it is worth discussing to clarify the fact that we don’t use BPMN in some cases at all. Generally speaking, we may consider activating the following task by the scheduler as a kind of message transmission. In this sense, BPMN-based applications are undoubtedly message-driven. Also, messaging is widely used in the implementation of BPMN processes as a means to reduce transaction times and increase system throughput.

Once upon a time, BPMNs were only very high-end, very advanced, and supplied only by large and reputable vendors for installation into limitless enterprise landscapes. At that time there were no questions of using them for ordinary applications at all. About ten years ago, our team asked some quite honorable vendor representatives how much hardware we need to process a couple of million financial documents a day. They swallowed nervously, looked away, and started talking about how great and easy it could be for any manager to draw almost any business process with only a mouse.

There were already solutions like jBPM and Activity, but they were raw and required a lot of work to be adequately cooked. Nevertheless, it was already a giant step forward compared to enterprise BPMN servers. For more details about a comparison between these two breeds of engines, see my article **“Some casual notes about BPM-based applications. Oracle, Camunda, and others…”**.

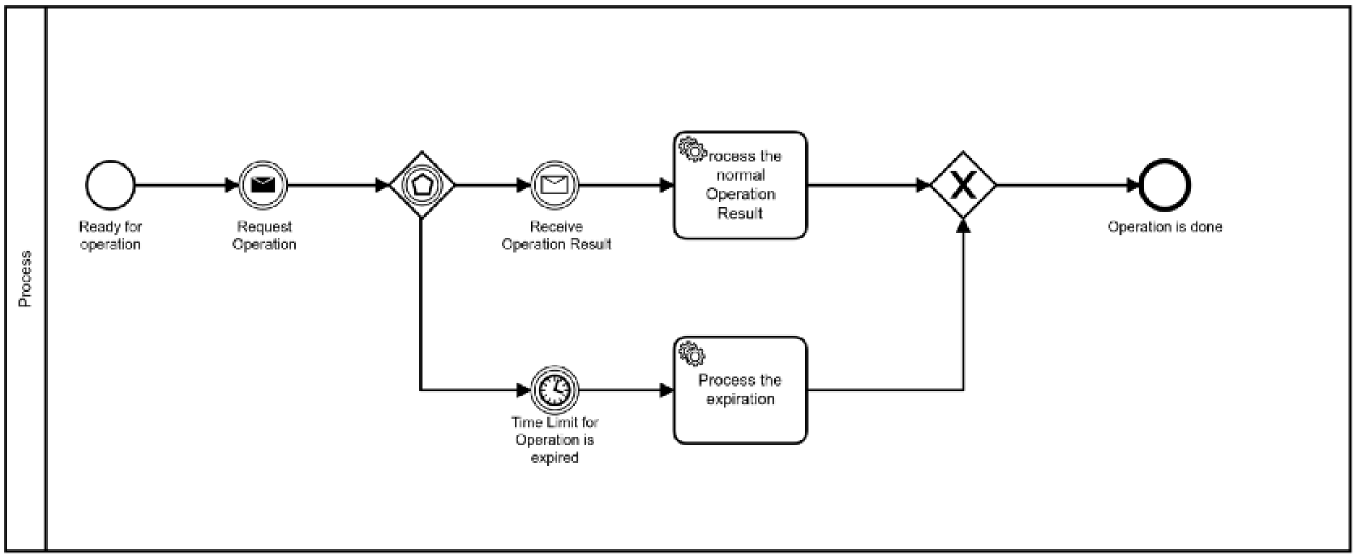
Finally, a clear, embeddable, transparent, lightweight, and ready-made Camunda appeared on the scene. We can use this server out of the box without much fear and the need to add lizard skin and virgin blood at moonless midnight. It would seem that this silver bullet could solve all the problems listed in the previous part of the article. However, applying the BPMN approach itself, even in such a lightweight style, introduces several fundamental pitfalls. Mainly because a typical BPMN engine is, in essence, a very advanced scheduler implemented on top of polling a relational database.

Embedded BPMN engine

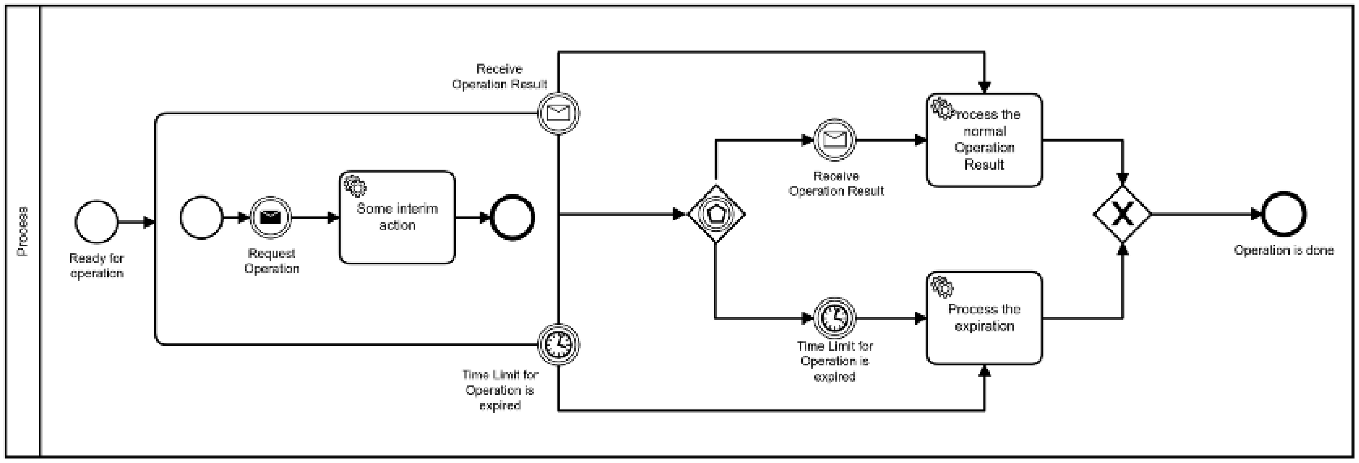
If we embed the engine inside the service:

- Business operations will be executed in the same transaction spawned by the scheduler. If we have performance issues or, even more so, run remote calls, the scalability of the engine as a whole suffers greatly.
- Since the engine database is located in the same schema as the service database, there is no way to quickly implement an administrative interface that deals with all services simultaneously.
- As the scheduler continuously polls the database for getting ready jobs to run, resources are consumed regardless of whether the system has a workload or not.

A typical example of handling events in a BPMN process description is shown in the following diagram. In this case, we do not care about the nature of the events and what technology is used. It is the fragment of the implementation of the process, located in a single service with a BPMN engine.



The only important point is that no other activities should be located between the message sending operation and waiting for a response. Otherwise, the response can come too early when the BPMN engine is not expecting it yet. We can overcome this limitation by using a bounded sub-process.



So, during the post-request business activity, we can handle the response by the bounded interrupting message or timer catching events activities. As far as the bounded process has been completed, we can apply the traditional approach. The disadvantage of this approach is the unwieldiness of the processing scheme, overloaded with technical details. It is a classic anti-pattern of BPMN-based programming.

Another standard solution is implementing a dedicated pure technical process, which saves the received message in its context and tries to wait until the target recipient process has finally reached the ready point. But in general, the need for such exotic designs is quite rare.

A frequest anti-pattern is storing, especially permanently, the payload from an incoming message in the database stored context of a process instance and dealing with it by the next task. A process instance context is designed to store mainly parameters, flags and identifiers. Placing large amounts of data in it slows down the BPMN engine significantly. So we have to put data into separate storage and save the only identifier into the message.

Another anti-pattern is sending to the process instance the message without getting the prior notification about the process's readiness to receive and use it. From a business point of view, this is often acceptable. For example, if the client may send the following message only a few days after the first one. At this moment, the process has no doubt already reached the desired state and ready to receive the next message. However, testing such processes becomes extremely difficult. So it makes sense to add notifications to the scheme, even if they don't make much sense from a business perspective. In addition, we can use such notifications to collect statistics or do an incident investigation.

BPMN engine as a separate service

Pay attention that the above BPMN patterns is not easily applicable outside embedded deplyment case. It happens because the some standard message-relating technology is not a feature of BPMN specification. Of course, we can modify the Camunda server or publish groovy code to send messages together with the BPMN definition. But it means a customization and moving towards the previous approach.

At the same time, we can consider the publication of the **ExternalTask**, which becomes available to the performers, as sending a message to some logical "topic".

When we locate the business code outside the BPMN engine:

- We continue to poll the base continuously, but the problems of scalability and administration are being significantly reduced.
- We have a separate single service to deploy, administer, configure, and so on. Worse yet, this service is at the very borderline between infrastructure and application services. From this moment, no one is responsible for it: neither the system administrators nor the developers. We got the classic orphan, whom everyone prefers to touch as little as possible to avoid being responsible for the consequences.
- If we want to keep our services generic, we must implement additional services — “task performers”. These performers, tightly coupled with the process definitions, pull external tasks from the BPMN service and execute them at the request of generic services. The system becomes much more distributed and complex structured.
- Business code and scheduler code are now executed in different transactions, and we have to deal with partially inconsistent data. It is not a critical problem but an additional severe discomfort for developers and analytics. Of course, we can modify the **ExternalTask** feature to use the JMS transport, which will publish the tasks’ descriptions and receive the answers from the performers. In this case, we will have proper transactionality on each side. But this means the necessity to implement non-trivial customization.
- If we need to transfer a significant amount of data from one performer to another, we cannot more temporarily put this data into a process context to be automatically deleted when the transaction is committed. Instead, we need to introduce another service to store the payloads temporarily. This service also usually brings the responsibility of format validation and conversion. Thus, the execution of an external task may involve interaction with as minimum three data sources at once: the BPMN engine, the payloads storage, and some service’s database. The risk of errors on the part of application developers increases accordingly.

Fundamental BPMN related issues

In addition, the implementation of BPMN processing in an application brings the following fundamental problems:

- BPMN is not only a one another technology but a specific development methodology that resides in the middle between application architecture and business architecture. Accordingly, even the granularity of breaking down the process into individual tasks is far from a purely technical problem. For example, we often encounter situations where many tasks of a highly detailed process are performed together one after the other in a single transaction. The result is not clear to business analysts on the one hand and has no technical meaning or benefit on the other.
- The process description and its implementation exist separately as XML description and Java / Groovy code, respectively. In the process of evolution, this bundle constantly tends to fall apart. The only way to ensure consistency of the descriptions is through extremely detailed integration tests that require carefully prepared data and resources to keep it up-to-date.
- The inevitable evolution of the system and support for long-lived processes means that we are dealing with versioning in four dimensions: the version of the BPMN description, the Java code version, the version of the data, stored in the process instance (the processes duration can be days and months), and the database schema version. This fact usually turns out to be an unpleasant discovery for most developers. This problem is not unique to BPMN, but here it is noticed later and causes more confusion.
- The very notion of a process consisting of individual activities is not entirely transparent to both the developers and users of the system. As a rule, we operate with some business objects, and it's intuitive for them to ask in what status or state the object is. BPMN isn't about the state of the data object. So it's often redundant and makes it difficult to understand what's going on with some data instances. One way or another, the user needs to see some state of the business entity in the interface. So we can't do away with states in any case and have to simulate them somehow on top of the activity model. In most cases, it is often much easier to deal with a state-based model from the beginning.

Let me try to describe a typical service, where the usage of BPMN, in my humble opinion, is redundant:

- One or two processes, each containing only several asynchronous operations. Although divided into separate tasks, the rest of the activity is performed synchronously, together in a single transaction.
- The tasks are mainly technical, and from the perspective of business analysts and administrators, their content is not very clear.
- Initially, there were many more processes and tasks, but they inevitably started merging and increasing granularity as we optimized performance. Such refactoring usually does some positive effect because the scheduler spends a lot of resources on running process instances, selecting individual tasks for execution, working with the context stored in the database, etc. Such simplification is usually not a problem because from the beginning there is not much sense in their separate existence.
- As we move towards the first release, developers usually disable process history collecting for optimization purposes. After that we don't have any information about the performance of individual business steps. No one usually thinks about the future and how incidents will be investigated during the actual operation — a very reckless decision.
- In operating the system, no one pays much attention to optimizing existing business processes.
- BPMN scheme acts as an alternative way to encode the business logic, partially understandable to non-programmers. And, of course, beautiful and accurate pictures are a joy to managers of any level.

The obvious advantages

However, BPMN has the bright advantage of serving as a means of communication between all involved in the development process. Especially if we have:

- multiple complex processes shared among many groups of services
- they are designed by a separate team of dedicated analysts.

In the case of complex projects, this capability becomes critical to success.

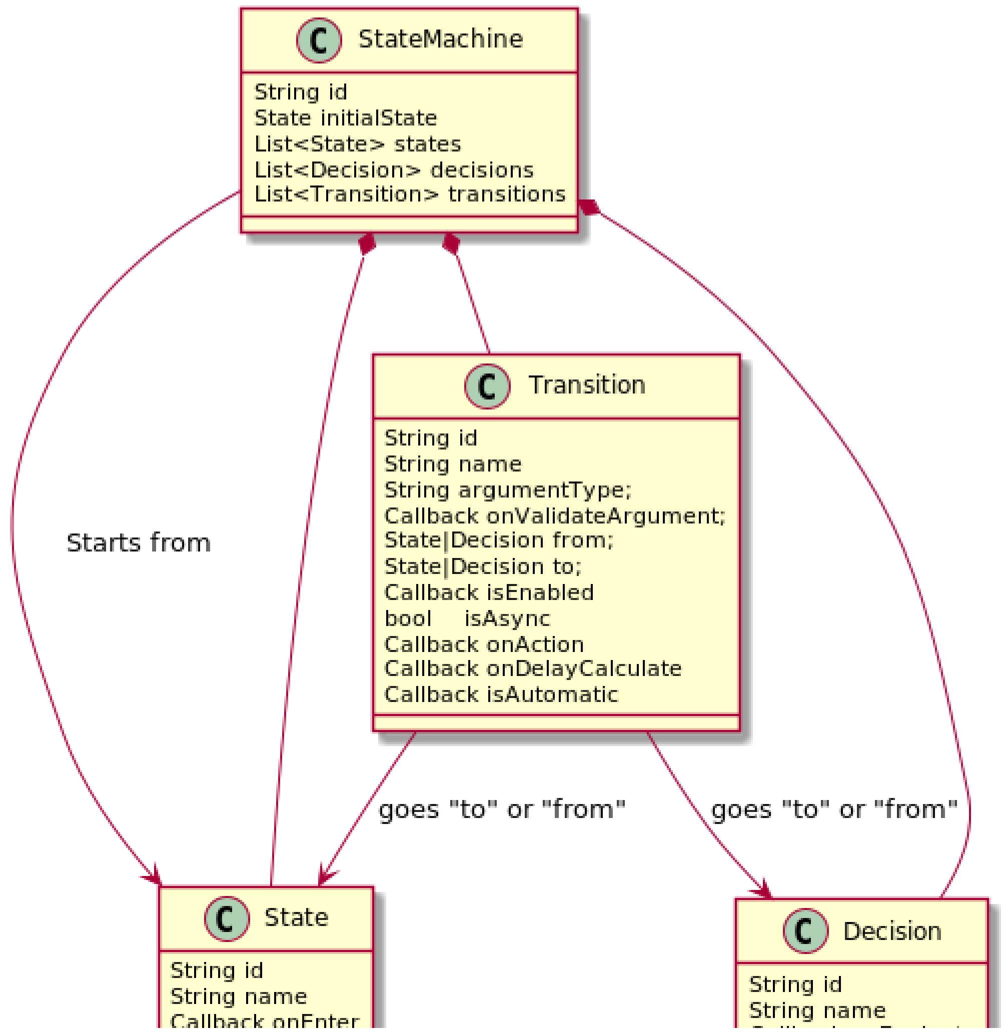
State Machine

Metadata and principles

Compared to BPMN, the description of the state machine is extremely compact and consists mainly of the following basic elements:

- **StateMachineDesc** — the container for other elements of metadata
- **State** — one of the possible states
- **Transition** — the possible transition from one state to another
- **Decision** — the ability to choose between several possible transitions
- **Callback** — some fragment of executable code, which defines the behavior the state machine

The following diagram illustrates the relationship of the listed elements to each other:



The ideology of the state machine is also quite simple and trivial.

- We have to perform all operations on documents/aggregates only by activating the corresponding transitions, whose callbacks, in turn, perform queries to the database or make any other changes to the stored data.
- Each aggregate /document stores as part of its data the identifier of the corresponding state machine instance.
- To perform operations on an aggregate/document, any application services, first of all, asks the **StateMachineService** for the current state and a list of available transitions. After that, it requests execution of one of the available transitions with passing the required arguments.

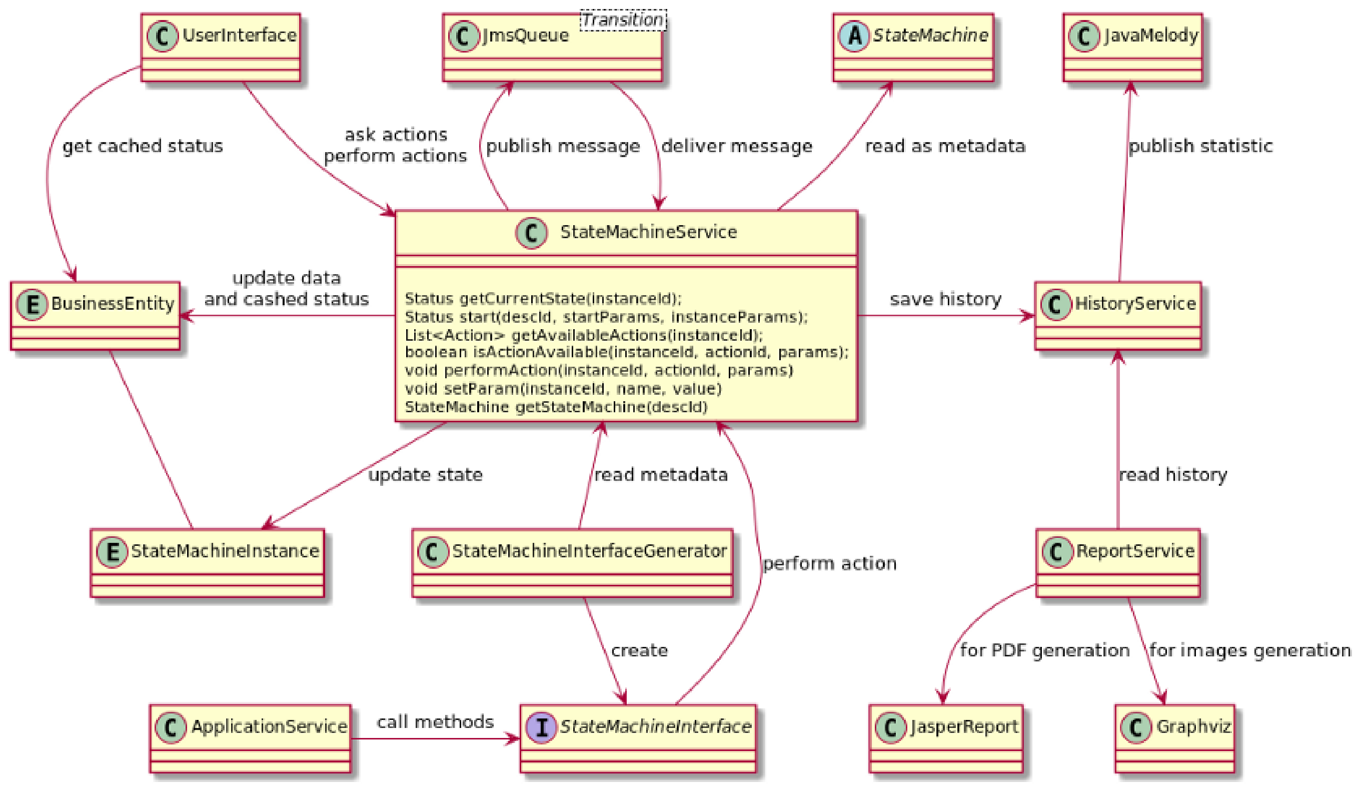
System design

The following main application components normally interact with the state machine service:

- **UserInterface** — reads by **StateMachineService** the list of available transitions and asks to perform some transition
- **StateMachineInterfaceGenerator** — during runtime builds (by JavaPoet, for example) the classical Java interface, which presents all possible transitions as methods
- **StateMachineInterface** — represents the access to some state machine as a java interface. All application code, execpt for UserInterface, has to call operations on the state machine through this interface
- A set of JMS and corresponding infrastructure — one per each transition
- **HistoryService** — stores the history of transitions for future analysis. Also it publishes all statistical information on the timing of operations in something like JavaMelody

- **ReportService** — generates two sets of artifacts. Firstly, it generates different kinds of reports about the history of process's instances (usually by means of something like JasperReport). Secondly, it provides the graphical representation of the state machine's metadata (usually by means of Graphviz or something like this)

The next schema illustrates the relationship between different components:



```
Status getCurrentState(instanceId);
Status start(descId, startParams, instanceParams);
List<Action> getAvailableActions(instanceId);
boolean isActionAvailable(instanceId, actionId, params);
void performAction(instanceId, actionId, params)
void setParam(instanceId, name, value)
StateMachine getStateMachine(descId)
```

Benefits and objections

The implementation of such functionality usually consists of 10–30 classes only and brings the following benefits:

- The engine is quite simple, lightweight, and transparent to the developers.
- Pure event-driven architecture, eliminating the need to poll the database constantly
- It consumes minimum resources and can be easily embedded in any service. The latter makes it much easier to integrate with any software components already in place.
- All operations are transactional with minimal overhead costs

- Since each asynchronous transition corresponds to a separate queue, the “heat map” of the data processing is built automatically
- AOP on the generated interface and JavaMelody like tools enable the collection of all relevant statistical information for analysis
- Horizontal scalability and fault tolerance are provided out of the box by the design
- Due to the simplicity of the metadata structure, it is easy to develop a DSL that allows both the state structure and the code that implements the callbacks to be described simultaneously in the same source code fragment. It is easier to reuse common fragments of code/description in different state machines.
- It remains possible to use the structure of the state machine as a means of knowledge sharing between developers, analysts, and managers
- From the administrators’ and users’ point of view, it allows quick answers to two fundamental questions: “what is the state of the object” and “what can be done with it.”
- Since the structure of the state machine description is very simple, it is easy to customize it both declaratively and programmatically. And we can do this with about the same ease at different levels: for a particular deployment, for a specific tenant, and even for a concrete instance of the entity being processed.

The main objection to using such an approach is usually the same: JMS does not keep the message’s order. I want to especially emphasize hear that this objection is entirely irrelevant in this case:

- Each transition can be initiated only by the event sent by the previous state. So there is only one message addressed to the state machine instance that can be in the queue at any time. The relative order of the messages addressed to different instances usually is not so essential for us.
- By the way, in the case of the BPMN engine, the situation is usually much worse in this respect. The order in which the individual activities are carried out in parallel process branches is not generally defined. Furthermore, two activities may try to update the process state in parallel during completion. In this case, one of them will

receive **OptimisticLockingException** and will be automatically restarted. If it has already performed a remote call without using a distributed transaction (eliminating distributions transactions is a common approach because they are not reliable and kill system throughput), it will attempt to update the remote resource twice.

Usage of Kafka

Unlike the first article, where we found the usage of Kafka as a transport rather complex and not flexible enough, the situation here is much better. It happens primarily because we do not need any tricky patterns based on filtering the messages by headers, etc.

In addition to high throughput and the ability to be used as backup data storage, Kafka provides us out-of-the-box with the feature of partitioning incoming messages. What does this mean?

- Runtime sharding — if we set the message business keys correctly, messages with the equivalent business keys will mainly be processed by the same servers. In this way, we can use caching technology effectively.
- There is no asynchronous processing at all. No two messages from the same partition can be processed simultaneously by two pieces of code executed by different or the same servers. This non-asynchronous character makes it extremely easy to implement features such as assembling a big message from fragments or checking for duplicate messages.
- Strict sequence of message processing is guaranteed. As far as we already saw, in the context of a state machine's usage, this circumstance is not significant.

Let's consider the implications of these and some other circumstances that we have to deal with on the implementation of business processes.

Strictly sequential message processing

There are some limitations, which are useful to keep in mind:

- First of all, we have to understand that only messages from a single partition are consistently processed. Messages from different partitions of the same or different topics can be processed in any order.
- Furthermore, since data is physically stored on different servers in the cluster, messages sent simultaneously to different partitions may end up being processed with a very significant and unpredictable relative delay.
- Also, a possible unavoidable disorder of messages is feasible if messages to the Kafka topic get through a construction like HTTP gateway. Because of the structure of the JVM, we cannot say anything about the relative order in the Kafka topic of messages that arrive at the input of the servlet container at close times. It is because they are handled by different threads, the events within which do not have a fixed mutual order. Moreover, a failure can happen in a distributed system at any time, followed by a re-request of the HTTP operation. All this will also change the order of Kafka messages compared to the order of the original HTTP requests.

Strictly sequential message processing means that in case of any continuous or permanent errors, the “poisoning” message blocks the whole processing stream of a single partition. Apart from the case of JMS, we cannot simply postpone its subsequent retrieval for some time, using the same or a dedicated queue and the corresponding functional header. Accordingly, we need to build a whole complex system of interval-dedicated topics and corresponding consumers that will allow us to successively defer messages that cannot be processed right now for progressively longer intervals of time. Another possible approach is to set up a repository for deferred messages based on a relational database, but in this case, the throughput will be lower.

We also have to maintain some context for postponing messages. If we have deferred one message relating to some data aggregate, we must also defer all subsequent messages to be applied after it, keeping the logical consequence. Accordingly, error handling implementation becomes even more complex and most likely again similar to be supported by some relational data storage.

In more detail, these techniques are discussed in the article **“Exceptions and Retry Policy in Kafka,”** and its implementation is challenging.

Kafka does not scale well in terms of increasing the number of topics

From the cluster's point of view, each new topic results in a group of dedicated replication processes as far as monitoring, and redistribution of actual metadata across the cluster activities. Thus, to keep the optimal count of threads on each broker servers, we have to add more nodes to the cluster.

From Kafka's client point of view, many topics being listened to means a significant additional consuming resources by all execution threads. The set of these threads contains ones, implemented by an original Kafka client, Spring containers, monitoring infrastructure and so on. Roughly speaking, we can assume that each additional topic brings a dozen new threads per partition into the application.

Being in some way limited by the count of topics, we may encounter some difficulties when trying to map each asynchronous transition of each state machine to a separate topic, as in the case of JMS. Accordingly, we can only follow the approach — "one topic per one state machine." Unfortunately, the "heat map" of the data processing is not easily available more in this case. In addition, long-running and fast transitions will now be executed by the same set of threads and slow each other down. Moreover, a single rarely executed operation can significantly slow down the processing of all other documents.

The situation becomes even more unpleasant if we deal with many tenants and have to guarantee a specific bandwidth for each of them. In this case, we have to use separate topics for each combination of tenant + state machine.

As we can see, despite our efforts, the number of topics continues to overgrow as our project reaches commercial success.

Scalability

Kafka itself is exceptionally scalable in terms of the amount of data that we can conduct through it. Unfortunately, the same cannot be said about consumers: the degree of its scaling is limited by the number of the topic's partitions. If message processing is fast, it's not a problem (for example, when we only transform / enrich data or gather some statistics). But if we need to execute several database queries to process each message, the number of partitions becomes a severely limiting factor. As operations involving the database are usually blocking operations. So, to use the database efficiently, we need more threads. The slower the database queries run, the more threads we need to efficiently handle the query thread.

We could, of course, send the message immediately upon receipt to the separated thread pool for further processing. It means going back to problems related to asynchrony, for example, when checking for duplicates in an incoming message stream.

Also, it makes much more difficult to maintain the partition's offset for the group. Let's imagine that we read messages 100th, 101th, and 102th and process the 101st with an error. In this case, we can't automatically save offset 102 and continue processing. We need to save offset 100 and remember that we have already processed the 102nd message.

In this situation, we are beginning to think whether it is not easier to save everything received "as is" in the database and then deal with each individually. One less request to the database, one more — the degree of headache is about the same.

Transactions

A much more severe problem with Kafka's usage in this context is that it implements transactions in a highly unusual way. I have already written about this in my article **"Transactional integration of Kafka with databases."** However, apart from those already described from that time onwards, I have found a few very unpleasant others. All of these are mainly explained by the way transactional behavior is implemented in Kafka. It was implemented on top of the existing message storage and transfer mechanism. Therefore it is not a core principle of the architecture, but some additional and optional functionality.

In a nutshell, how the transactionality is achieved:

- To mark the end of a transaction, the broker writes technical messages (**Cx/Rx**) to the partition. These messages consume offsets, but apart from usual messages (**Mx**), they cannot be read by consumers.
- Transactions exist only in the consumer's imagination as a function of the whole message's stream. For each message individually, its transactional status cannot be ascertained — it is determined by the existence of previous/subsequent technical messages.

- The transactional consumer reads the sequence and filters uncommitted messages. But it has to read messages in the same order as they were sent. Therefore, when it encounters an open transaction, it simply stops and waits until a technical message is finally written. This technical message has to indicate that the transaction has been committed or rolled back. To be completely sure, we can read in the documentation: “Further, the consumer does not need to any buffering to wait for transactions to complete. Instead, the broker does not allow it to advance to offsets which include open transactions.”

Numerous amusing effects arise from this architecture:

- Therefore, if we look at the contents of such a topic with some tool like **Kafka Tool**, we will see some strange gaps in the sequence. The content of the partition with the maximum offset 105 looks something like this: **…[M100][M102][M104]…** — at the end we can see six offsets and only three messages. Technical messages like **[C103] and [C101]** are not visible for us. Such delta is not terrible unless you are trying to calculate the number of messages in a topic by the difference in offsets. Monitoring tools like Grafana across all partitions show a seemingly strange lag of at least one message.

- Let’s say service A is configured as **read_committed** and reads service B’s transactional messages. If service B or its incoming requests are broken, it starts rolling back transactions. In this case service A really has nothing to read: after the 100th committed message: **[M100][C101][M102][M103][R104][M105][M106][R107]**. But rollbacked messages are already in the partitions (and, BTW, visible for the usual consumer), and they consume offsets. So after a while, admins start to cry: “Service A has stopped consuming messages, see immediately on Grafana, there is a gap of many thousands of dollars, fix it urgently!!!”. In reality, service A is in good state, but we still have a big headache because Grafana can’t distinguish good gaps (messages that really shouldn’t be consumed because they’re not committed) from bad gaps (messages that are not consumed yet).

- If one of the services starts a transaction and hangs/slows down, then at least one partition will be temporarily locked. In this case service A really again has nothing to read: after the 100th committed message: **[M100][C101][M102][M103][M104][M105]**. The consumer will have to wait until the transaction is committed/rollbacked to move on. Again we have a gap and no any tools to understand what is actually happening. If a slowing down service typically sends multiple messages during a transaction, it has a chance to block the whole topic. It is a serious potential problem for topics with many services/code fragments writing into them simultaneously.

Once again, the conclusion I have drawn for myself is that Kafka is not for enterprise systems. It is twice as true if we are going to have transactional data processing. It is for large streams of non-transactional data, some small part of which would not be painful to lose. There are several examples of such streams: technological logs or information about user behavior.

So what should we do if we do want to use Kafka in this context? The answer is obvious — don't use transactions at all. What should we do instead? Generally, it is recommended to save messages to a dedicated table and apply a CDC solution like Debezium.

In some cases, you can do with a more simple design for sending messages:

- The sender saves the message to the database and commits the transaction. Next, he synchronously sends Kafka a message in the same thread and deletes the message from the database.
- If a thread or the whole service dies, the message stays in the database. Accordingly, you need a monitor that selects and resends messages that have been in the database for longer than a specific period.
- This approach can be optimized by accumulating identifiers of sent messages in a global collection and deleting them in batches asynchronously to the sending process. The number of duplicate messages sent will increase accordingly.
- If the receiver is resistant to duplicates, that's enough. If not, we need a mechanism that will, before resending a message, check that it is not in the outgoing topic by calling Kafka.
- The receiving party acts similarly with the exact order of operations. It first performs a duplicate check of the received message and its business processing. And only after

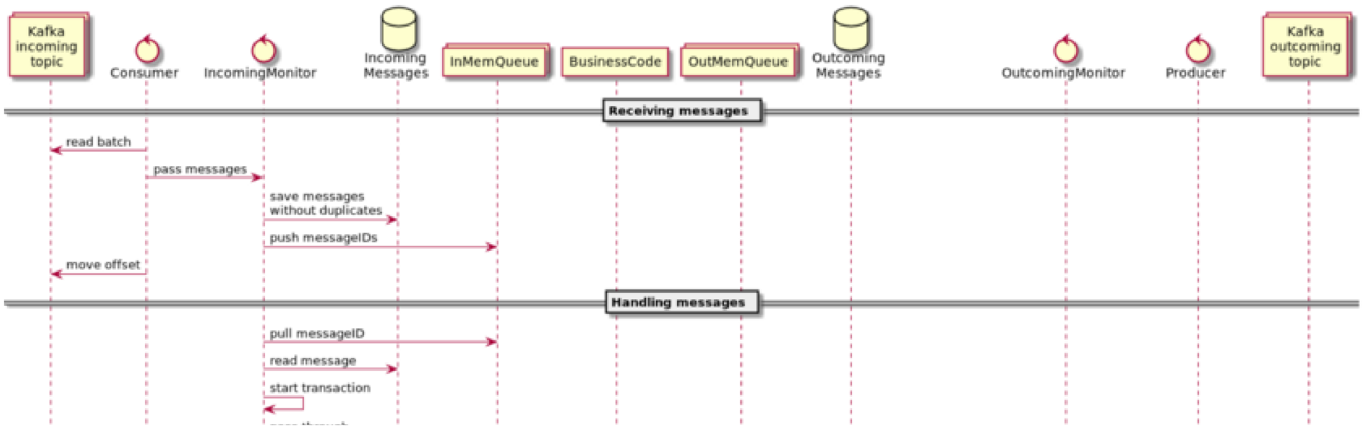
that, it performs an acknowledgment of the received message. Thus, in the case of a crash, some of the messages will be received twice but filtered out at the start of processing.

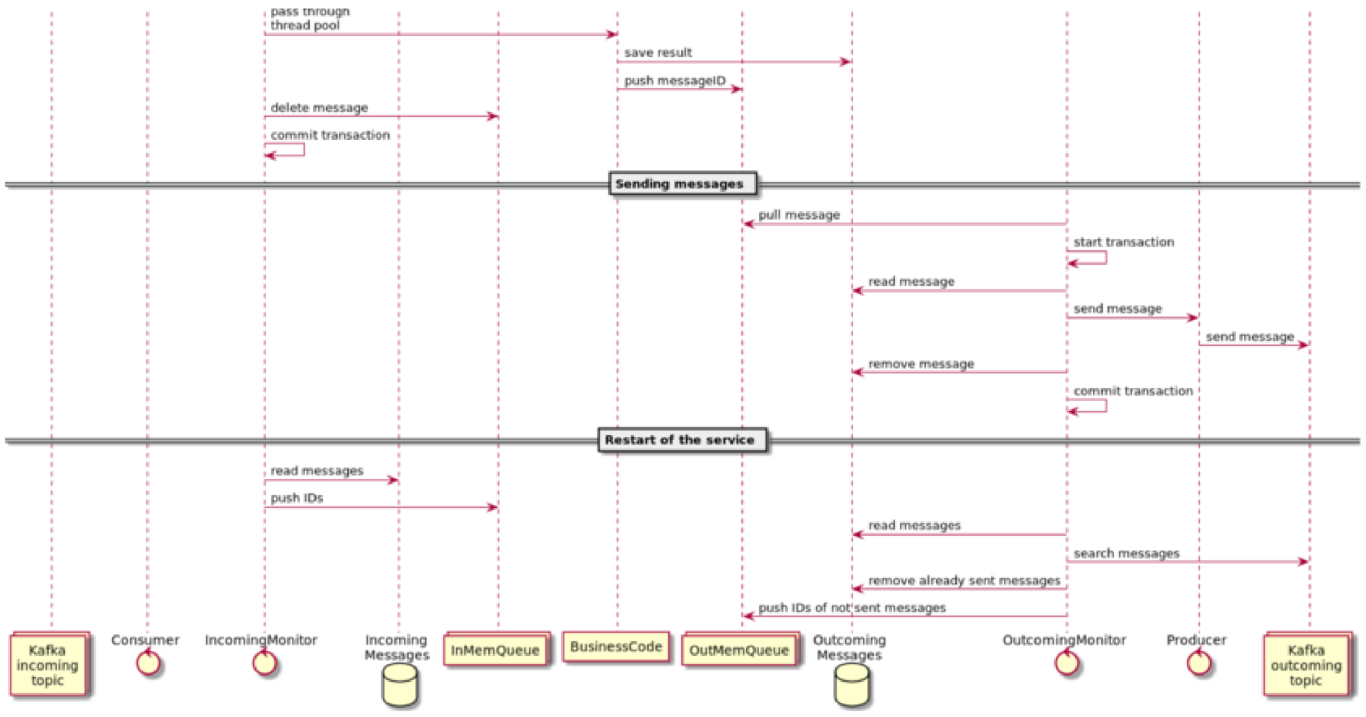
All together

The previous sections gradually lead us to the idea that the best way is to organize two transactional repositories inside the service for incoming and outgoing messages and a mechanism for transferring messages between them to/from Kafka topics.

- If we want to avoid constantly polling these repositories, we need to implement two queues in memory. Message IDs will be pushed into them when we will store messages in the repositories.
- The listeners of these queues (incoming/outcoming monitors) will be responsible for reading messages from repositories by primary keys and processing them: passing to business handlers or sending into outcoming Kafka topic.
- When processing incoming messages, instead of a simple queue, we can organize a more complex index-based structure, allowing us to reorder / delay / re-process the messages already received from Kafka before processing.
- In the case of a service restart, we just need to re-initialize the queues data, basing on the repository data and on the messages, which we read from the outcoming topics
- to improve efficiency when working with the network and disks, it is recommended to use batching when working with the database and the Kafka topics

The following diagram can illustrate this processing flow:





The principal advantages of this approach are:

- Inbox and outbox databases are very compact, and we do not need to poll them all the time. If necessary, we can put them in the memory of the server. We can use advanced table-based queueing mechanisms (available in PostgreSQL or TimelineDB, for example) to optimize dealing with such tables.
- We are free to change the order of message processing, re-group them, deffer, split / join, etc.
- We can update the message's properties arbitrarily to implement any complex processing schemes.
- All work with messages is performed entirely in a transactional way, in the traditional sense of the term. And it does not require developers to know specific technologies, patterns, the architecture of Kafka, additional API, etc.

This design also gives us the following exciting features:

- The entire Kafka-related infrastructure, including monitors, can be implemented and deployed together with the business service as universal auxiliary sidecar containers. It makes the business service code very simple, easy to understand, and to test.

- In the initial phase of application development, when all services are only potentially marked up and to speed up prototyping, we can forget about Kafka at all and add this functionality as cross-conception later as needed.

Summarizing the above circumstances, we can conclude that the specifics of working with Kafka is in full accordance with the principles declared by its creators: "Smart Broker vs Dumb Broker". But the foolishness of the broker in this context, in my opinion, is slightly overestimated.

- First, we get two distributed products (Kafka + ZooKeeper) in the system at once, each of which needs to be deployed, administered, and monitored.
- Second, the automatic distribution of data and load across the cluster Kafka is far from always being stable and optimal. This product is exceptionally far from the "deploy and forgets" ideology.

Since, in Kafka's case, the broker is dumb, the client must be the opposite smart. It means that client developers must also be reasonably competent and spend considerable extra effort, which is not justified in all cases.

Kafka as an archive or backup storage

Does this mean that Kafka is practically useless for us in the context of implementing business processes? It turns out that no: it is a handy tool but in a completely unexpected role.

One of the fundamental disadvantages of JMS is that messages from the queue are immediately deleted after consumption by listeners. Keeping them in a database-based archive is relatively slow and expensive, especially considering that we will never reread 99% of them. In addition, databases and the S3 file storage considered an alternative do not have the necessary throughput.

At the same time, there are practical requirements for the system to be auditable. It means, among other things, the ability to trace in detail the entire flow of information produced by a particular process instance. A similar requirement can be made as the "reproducibility" of a particular process over a considerable period. Strictly speaking, this is a tough functionality to implement because over time in the system, literally, everything is changing — the source code, the contents of dictionaries, data structure, and so on. Therefore it is almost impossible to rerun an instance of the process, giving it an identical environment and data. But this usually does not require the process behavior to be reproduced programmatically at all. We can provide the auditor with all the necessary data consumed or produced by the process instance for manual step-by-step execution with validation of the results.

In this case, Kafka proves to be very useful to us as a high throughput information store, into which we can dump all the process execution data: initial data, incoming and outgoing messages, payloads of user's tasks and external tasks, the state of the process before a branching point or calling another process and so on. So, we can throw in all the information that could be useful later in auditing, reproduction, or simply parsing incidents. In this context, we use Kafka as a high-performance logging system.

A logical question arises, why can't we just log all messages and use Elasticsearch + Kibana? The answer is very sad and straightforward — since business messages can weigh megabytes of data, we risk killing Elasticsearch with a huge amount of data, most of which we don't want to index at all.

If we put all messages in Kafka, we can access them only by a unique pointer to the message. This pointer consists of the following components: the name of the topic, the partition number, and the offset of the message in the partition. We have access to individual messages by pointer and a stream of messages starting from some pointer. Thus, we can store the vast amount of data in Kafka and send for indexing into Elasticsearch only the metadata of each message, with a pointer back to the Kafka stream.

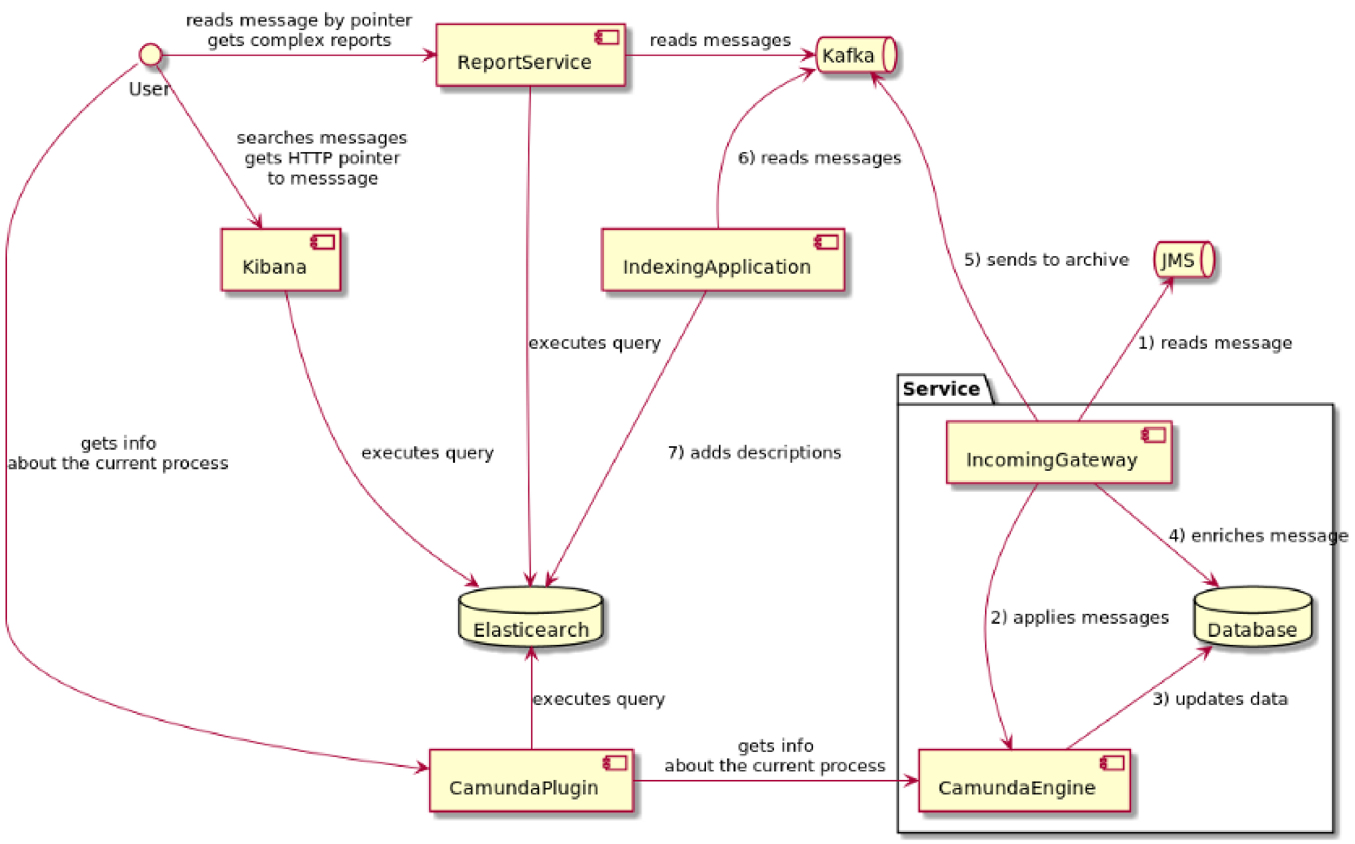
The infrastructure for this approach consists of the following main components:

- The code in the service, which interacts with the BPMN engine: receives the message from the JMS queue, searches for the corresponding process instance, passes the message to the process instance to be executed. After, it enriches the message with the process ID, the result of the operation performed, and other business-related data. In the end, it sends a message to Kafka's topic, which acts as archival storage.
- Indexing application: consumes messages from archive topic, calculates the pointer to message, creates JSON document with the message's metadata (including

pointer, of course), sends JSON document to Elasticsearch for indexing.

- The application administrator, through Kibana, can search documents with message descriptions by various criteria. Each description includes a pointer to the messages, automatically generated by Kibana UI as an HTTP link to the reporting service. The Elasticsearch and Kibana manual describes this as “Manage index pattern data fields”. In the simplest case, the user, by clicking on this link, gets from the report’s service a full view of the message with all the data, business/transport headers, etc.
- An administration plugin of the Camunda UI that outputs in the context of the current process instance a complete list of messages related to it with links to the reporting service. It runs by querying Elasticearch.
- The reporting service: can generate a range of reports, from the content of a single message to a complete description of activity related to a single process. On the one hand, it uses Elasticearch to search for messages. On the other hand, it directly reads Kafka tokens to access the messages themselves.

The following diagram can illustrate this scheme of interaction:



In principle, it is not very far from here to usage Kafka storage as event-sourcing backup storage. Such storage can be used for recreating, in case of unexpected problems, the primary database from the stream of messages sent to the processes. Nevertheless, this imposes significant limitations on the implementation of the service.

- All data-related operations must be performed exclusively through a business process.
- All additional data used in processing (such as dictionaries, for example) must either be unchanged or stored in the versioned form.
- All executable code must also be versioned. And this is probably the most challenging requirement to implement.

Conclusion

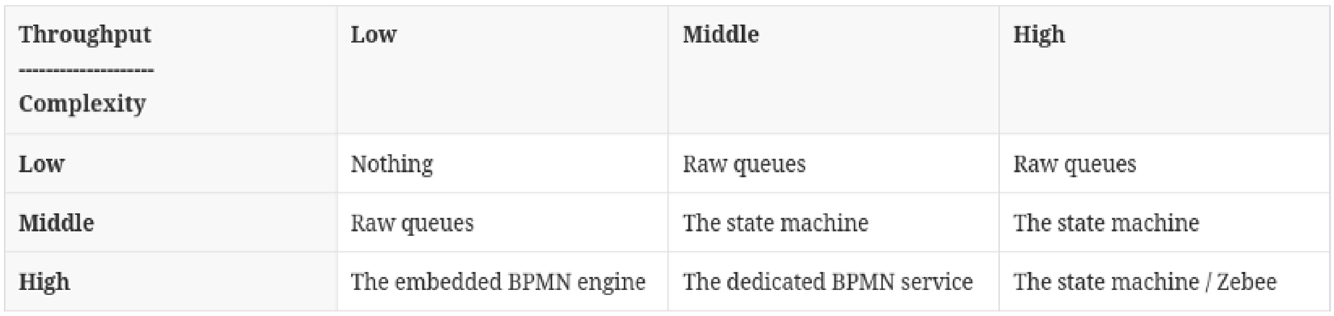
So we looked at several approaches to implementing business processes based on messaging:

- nothing — we don't have to use messaging, even though it is stunning, fashionable, and enterprising
- raw queues
- the embedded BPMN engine
- the dedicated BPMN service
- the state machine.
- Zebee — the experimental BPMN engine from Camunda. Its main difference is that it does not use a relational database to store data but a partitioned distributed storage like Kafka. I have no experience with this system, so I can only mention it as a possible and very promising alternative.

Each variant is designed to reduce the system's complexity by bringing in some extra work, code, methodology, infrastructure, etc. What to apply in each case is a matter of discussion.

The general principle, it seems to me, is as follows: the complexity of the approach must not exceed the complexity that we want to eliminate. The treatment must not be more poisonous than the disease.

If we consider only two dimensions of the system, productivity, and complexity of business processes, I would recommend the following approach:



Throughput ------------------- **Complexity**	**Low**	**Middle**	**High**
Low	Nothing	Raw queues	Raw queues
Middle	Raw queues	The state machine	The state machine
High	The embedded BPMN engine	The dedicated BPMN service	The state machine / Zebee

I would also like to repeat again that these approaches are practical only in the context of enterprise applications that actively use the database to process message content.

In contrast, the next article will focus on how to use messages when implementing a user interface. It won't be as big as this one, as I rarely have to deal with this development aspect.

Nevertheless, there are a few things to keep in mind that can make life a lot easier for application developers.

Buy me a coffee