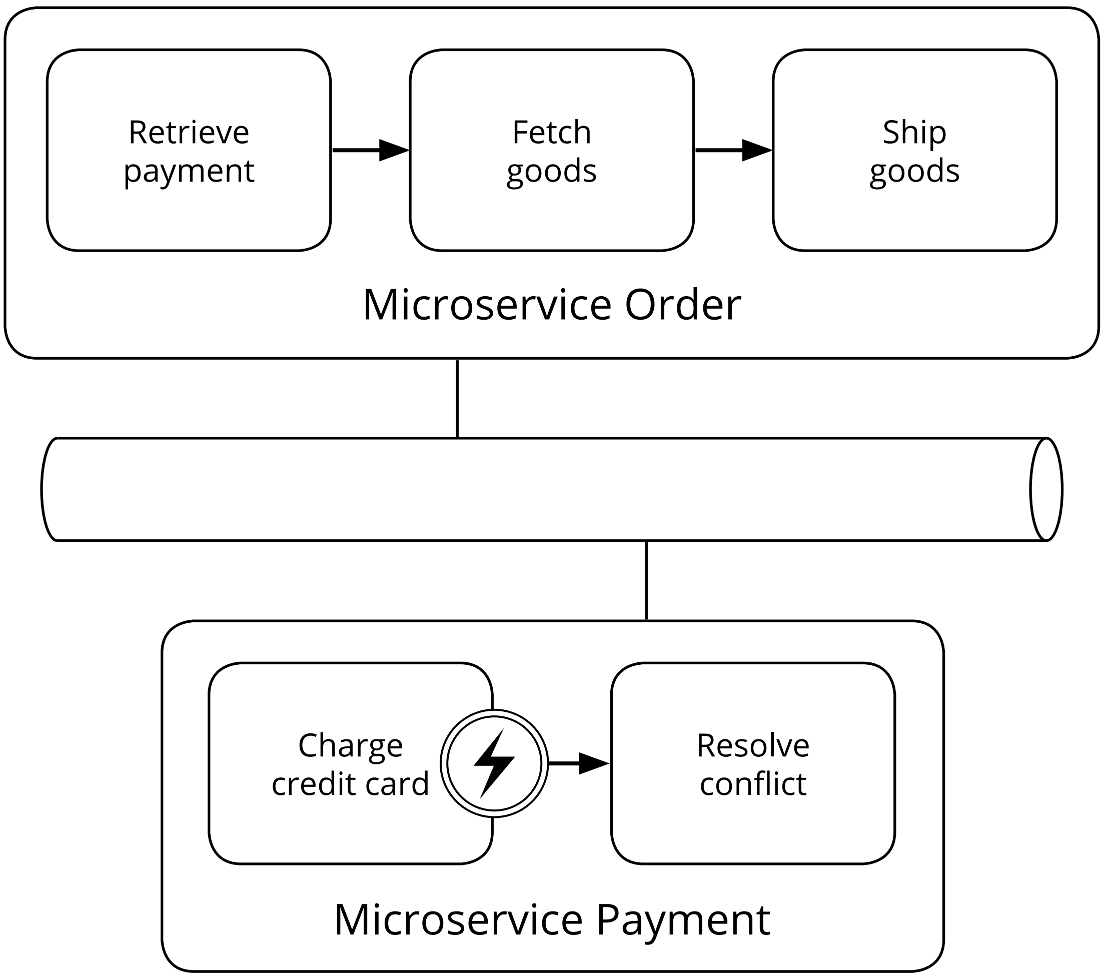
Microservices and BPM



May 2017 – camunda.com

Contents

Microservices

Microservice architectures gain a lot of popularity due to the ever-increasing complexity of systems. Microservices split the overall system into individual parts (the microservices) each focused on a single business capability ("do one thing and do it well"). While this sounds a bit like Service Oriented Architecture (SOA) it differs fundamentally in the way microservices are being developed, deployed and operated as well as how the individual services integrate into the overall architecture. The main motivation for SOA was to reuse ("built for reuse"), for microservices on the other hand it is team organization and independent exchangeability of individual components ("built for replacement"). Microservices address the problem to scale software development. Software structure always reflect the team structure building it which is known as Conway's Law. With microservices you define services and therefore also team boundaries around meaningful business or domain capabilities, which is known as Bounded Context.[1] One team is responsible for the whole technological stack required. Within that context this team has autonomy to do whatever it takes to implement the requirements at hand and to reduce coupling to other microservices. Effort to coordinate with other teams is reduced as much as possible to gain an overall flexible organization.

Hence individual microservices should work independently. This independence relates to various aspects:

- **Independent life cycle:** It must be possible to develop microservices independently by different teams. It must also be possible to start and stop microservices or to make changes independent of others. There should not be a need to coordinate your deployments with other teams.
- **Stable Interfaces:** Microservices provide the environment with stable interfaces which must not be broken during updates. If incompatible changes at an interface are necessary, they should be depicted by versioning the interface.
- **Communication:** If one microservice needs to communicate with other microservices in order to fulfill its task, the calling service must expect that the communication partner may not be able to answer the question immediately. That is why asynchronous messaging or feeds are used often. Other patterns also exist like the bulkhead architecture for example where the outage of one service does not influence upstream services.
- **Robustness and fault tolerance:** Individual microservices must continue to run even if other services in the overall system cause problems. In many cases it is better that one individual user of the system sees an error than letting the entire system break down in an uncontrollable manner.
- **Local data storage:** Microservices often keep local copies of data that they need in order to fulfill their service which basically enables other characteristics mentioned here.

1 https://martinfowler.com/bliki/BoundedContext.html

- **Independent scalability:** Not every microservice needs the same resources. It must be possible to provide each service with the resources needed without affecting others.

The six points mentioned are intended as guidelines. Not every microservice must necessarily satisfy each of these criteria or meet the challenges mentioned in the same way.

Business Processes Stretch Across Multiple Microservices

When looking at end-to-end business processes, they typically stretch across multiple individual microservices which therefore have to collaborate in order to achieve the desired business outcome. This is where the rubber meets the road. In this whitepaper we will use the example of a simple order fulfillment service involving payment, inventory and shipping microservices.

There are different possible communication styles in microservice architectures often mixed within a single company. We do not want to discuss all communication styles and their pros and cons in this whitepaper, instead we want to highlight two popular approaches: Let your microservices talk REST with each other or use messages put on a message or event bus. These approaches have slightly different challenges. Let's start with the trend more recently adopted.

Asynchronous Communication

The heart of this approach is that microservices exchange asynchronous messages. While this can be done using a message broker it is also en-vogue to use an event-bus for this. Often you do not use dedicated queues or topics for one communication channel but have one "big pipe" you push all events onto. As a result, microservices just emit events and don't have to care who is picking it up. Also they just consume events without knowing where they come from. This results in a high degree of de-coupling.

We do want to concentrate on the resulting event chain that carries out the overall business process. In this architecture, there does not have to be any central brain in control which frees up the overall architecture from central components. For example Martin Fowler said that you do not need an orchestration engine in microservice architectures.[2]

Let's look at an example of a simple order fulfillment process where you could imagine the microservices and event chain shown in Figure 1.

2 https://martinfowler.com/articles/microservices.html

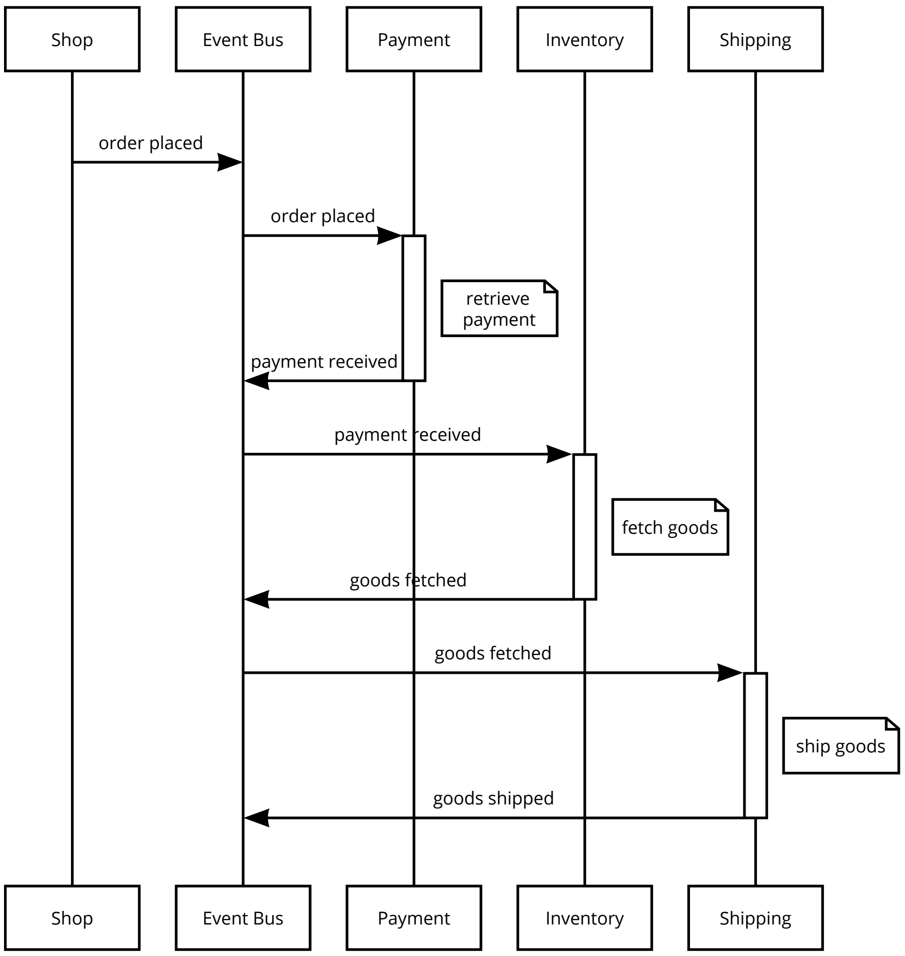


Figure 1: Event flow of a simple order fulfillment system

This looks good at first glance but reveals problems shortly after.

Limitations of Event Chains

The *payment* microservice listens to the *order placed* event. That means every time you have a new use case requiring payment you have to touch the *payment* service. This is very bad for the overall goals of team autonomy. Assume you want to build and deploy a new service which can sell downloadable artifacts. Now you have to coordinate the deployment with the *payment* team as they have to start listening to an event you emit, e.g. *downloadable item purchased*.

This can be solved by introducing an **event command transformation**.[3] This pattern reflects that you sometimes have to issue messages which are commanding another service to do something. In the example above, *payment* should listen to a *retrieve payment* command, this improves de-coupling.

3 Described in more detail in https://blog.bernd-ruecker.com/why-service-collaboration-needs-choreography-and-orchestration-239c4f9700fa

As you can create similar examples with the other events in the chain it becomes evident that it makes sense to introduce an individual business domain for the overall order fulfillment logic. In the example we recommend an *order* microservice, which does all necessary transformations which can be seen in Figure 2.

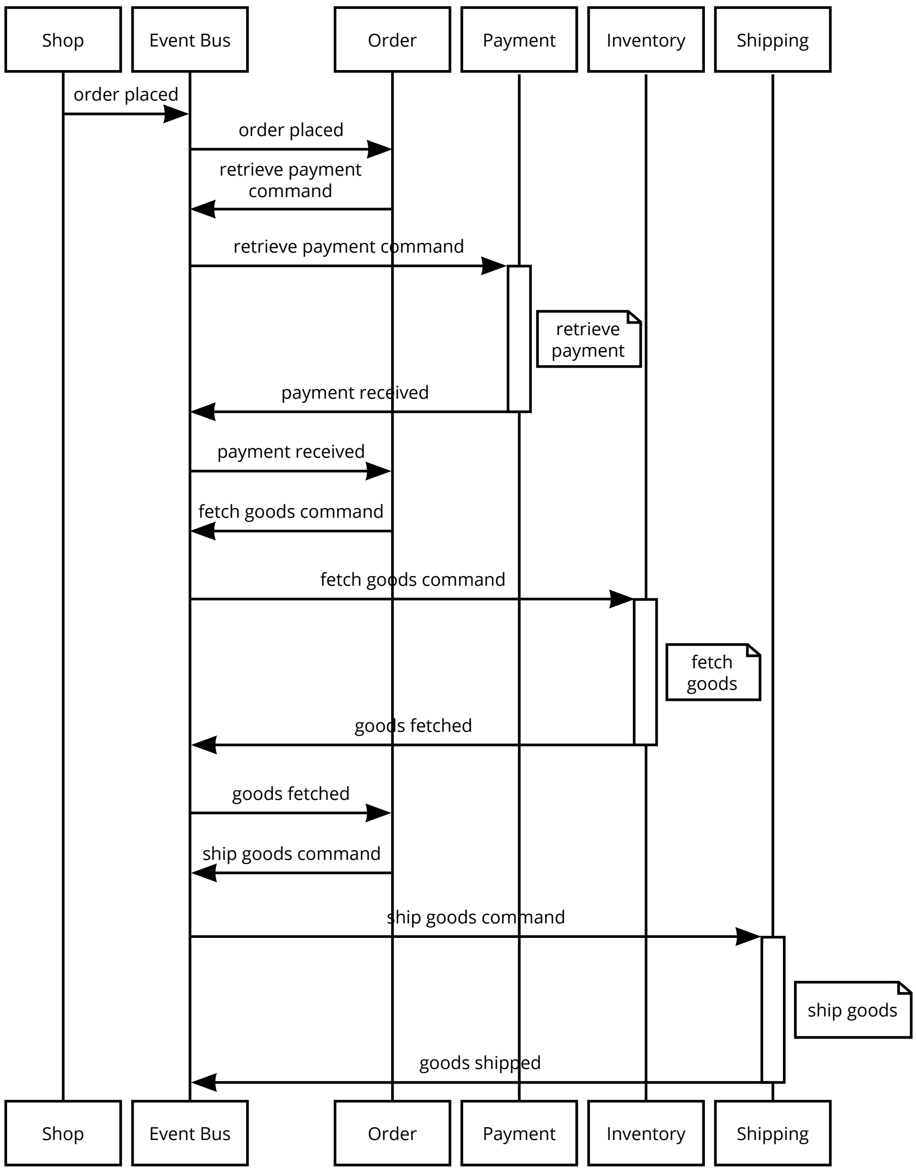


Figure 2: Event flow when using event command transformation

Now this order service can also decide on all the related requirements to the overall business process. So, if you want to first fetch all goods before you retrieve the payment there is one single microservice where you can easily achieve that. One team can implement and redeploy it on its own. Coordination between teams is not necessary. Without this order service you would have to touch at least two services and coordinate a joined release in order to make the very same change: *Inventory* has to listen to *order placed* and *payment* has to listen to *goods fetched*.

Sometimes it is argued that this microservice is a single-point-of-failure for the order fulfillment. We do not see this as a proper argument as every microservice should **be** exactly the single-point of everything for one certain business capability. If *payment* is down, no payments will be done. If *order* is down, no order will be fulfilled. It would not be possible anyway if at least one important service in the chain is down – or would you ship your order if you cannot retrieve payment or fetch the goods?

It is the responsibility of the *order* team to achieve the degree of high-availability needed for the business requirement at hand. And as we do asynchronous communication, events will not be lost if you have downtime. You could even see the positive side of things: Whenever you need to do an emergency stop of orders (which is more common than you might think) you know exactly where to do that.

Synchronous Remote Calls

A different approach is to use synchronous remote calls, typically via REST. The basic call chain is comparable. The argument to introduce the *order* microservice is now even stronger because without it you force the payment service to know the concrete *inventory* interface.

Additionally, implementing the *order* microservice gets a bit harder as you have to tackle unavailability of remote services. This involves state handling and a retry mechanism. While this can be solved by different means including messaging again we see many projects struggle with this in real-life. This is why we often leverage state machines (to be introduced shortly) for this use case.

Long Running Processes Require State

This brings us to another requirement you need to address with the order anyway. Let's assume that payment is done by charging your credit card. And when the card is rejected we do not cancel the whole order but send information to the customer to update his credit card and give him seven days to do so. If he updates the credit card in time, the payment can still be retrieved successfully. The order process has to wait for the payment for a potentially long time which immediately requires persisting the state of every order.

Persistence can be tackled by multiple approaches, typical solutions involve custom entities, actor frameworks or very simple state handling frameworks.[4] But in real-life projects we experience a lot of subsequent requirements as soon as you persist state. What if the customer doesn't respond within seven days? So, you must track time and timeouts. You also need some monitoring on ongoing order processes and proper reporting in place.

A workflow engine is perfectly qualified to handle the persistent state. It can also handle the flow of events (or to be precise: the flow of event command transformations as explained above) in a sophisticated

4 See also https://blog.bernd-ruecker.com/how-to-implement-long-running-flows-sagas-business-processes-or-similar-3c870a1b95a8

manner. The flow can even be visualized graphically as shown in Figure 3, even though engines like Camunda allow to express the flow via pure Java code without any graphical modeling required.

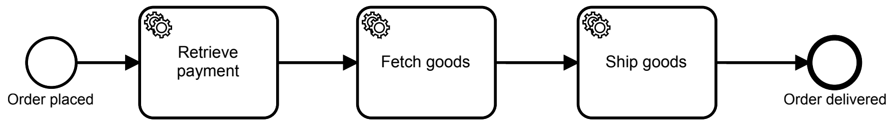


Figure 3: Graphical visualization of the event flow using BPMN

Advantages of Using a Workflow Engine

When using a workflow engine, you experience several benefits:

- **Explicit processes:** The processes become explicit instead of being buried somewhere in your code and therefore can be changed much easier.
- **State handling:** The workflow engine handles the persistence of each single order instance.
- **Transparency of status:** The status of a process instance can easily be checked by asking the workflow engine. Monitoring can take place in the graphical diagram directly.
- **Visibility:** The graphical model can be used to discuss the process, might it be between business stakeholders and IT, between developers, between developers and operations among others.

Martin Fowler also recognized the importance of visibility and transparency as he recently wrote: *"Event notification is nice because it implies a low level of coupling, and is pretty simple to set up. It can become problematic, however, if there really is a logical flow that runs over various event notifications. The problem is that it can be hard to see such a flow as it's not explicit in any program text. Often the only way to figure out this flow is from monitoring a live system. This can make it hard to debug and modify such a flow. The danger is that it's very easy to make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow, and thus set yourself up for trouble in future years."*.[5]

Despite these generic advantages there are a couple of notable features of workflow engines that can solve problems very eminent in microservice architectures. Let's quickly dive into a few of them.

5 https://martinfowler.com/articles/201701-event-driven.html

Timeout Handling

The workflow engine can track time and automatically take additional action or switch to another path in the flow if some message does not arrive in time, as shown in Figure 4.

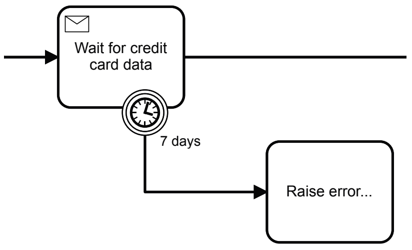


Figure 4: Timeout handling in BPMN

Message Correlation and Coordination

Sometimes several messages that belong to the same process instance must be "merged" in the process instance. With the support of the workflow engine this is easy as the workflow engine can decide what needs to happen next based on the current persistent state and the incoming message. Communication patterns such as *message sequences, message synchronization, waiting for messages with timeouts and mutual exclusion of messages* are already solved in BPMN, as shown in Figure 5.

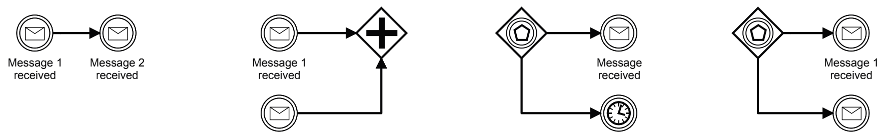


Figure 5: Solution of various communication patterns in BPMN 2.0

Error Handling

Whenever an error occurs you can specify the behavior, for example you can take another path or imply some retry mechanism especially when doing synchronous calls. This is shown in Figure 6.

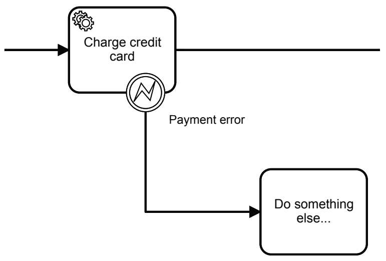


Figure 6: Error handling in BPMN

Business Transactions

BPMN knows the concept of compensation. Compensation is for situations when a process hits a problem and needs to undo steps which were already carried out earlier. This makes it easy to implement the so called Saga pattern very well known in distributed systems.[6] A classic example is a trip booking whereby multiple services are called as shown in Figure 7. The Saga needs to store state and can benefit from leveraging the features of a workflow engine. You find the full source code online.[7]

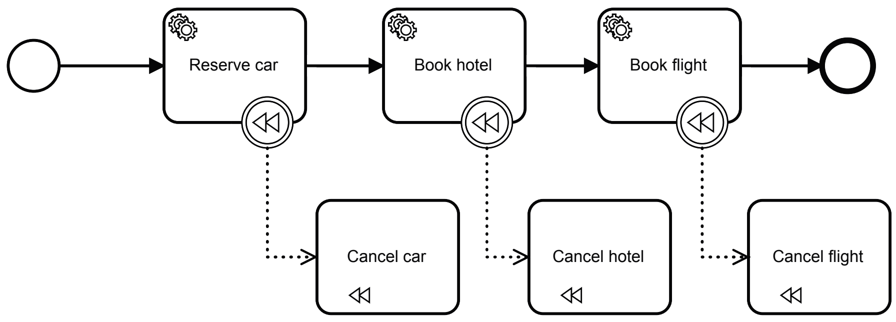


Figure 7: Business transactions and compensations in BPMN

Misconceptions on Workflow Engines

When proposing workflow engines as recommended in this whitepaper, you might be faced with reluctance to do so in microservice architectures. Typically, this is a result of the following misconceptions:

- Processes violating the bounded context: “When you model the end-to-end process it includes parts which are owned by different microservices, so you should not mess around in their territory”.
- Central controller: “A workflow engine is a central tool which is not only a single point of failure and a limitation in scalability but also forces teams to use a certain technology or to adapt to centrally governed changes like upgrading the engine”.
- Developer-adverse and heavyweight tools: “Teams building microservices should be autonomous in their tool decisions and operate the solutions they pick. BPM tools are heavyweight and take weeks to setup, this can never be managed by a microservice team and the developers will not select it anyway”.
- Too expensive (the same argument, but with money): “Our microservices run virtualized and BPM Suites are licensed by cores and you have to license all cores of the host machine for every service which is not affordable”.
- Do not run in the cloud: “We cannot deploy the inflexible BPM Suite to our (probably on premise) cloud environment”.

6 https://blog.bernd-ruecker.com/how-to-implement-long-running-flows-sagas-business-processes-or-similar-3c870a1b95a8

7 https://github.com/flowing/flowing-trip-booking-saga/

These reasons are misconceptions based on errors made in the past with BPM or based on an outdated view on workflow technology. The solutions are:

- properly distributed ownership of the business process to the microservices,
- lightweight engines,
- cloud-ready technology and license models,
- thoughtful wording.

Let's dive deeper into how Camunda resolves this.

Distributed Ownership of the end-to-end Process

The microservice community knows "God services" as anti-pattern.[8] It is about services that are very powerful and do all the work and just delegate to dumb CRUD services to save data. This should be avoided as you always have to change the God service for everything. There is no real decoupling. BPM practitioners are indeed often guilty of modeling God-like monolithic end-to-end process models as the example in Figure 8.

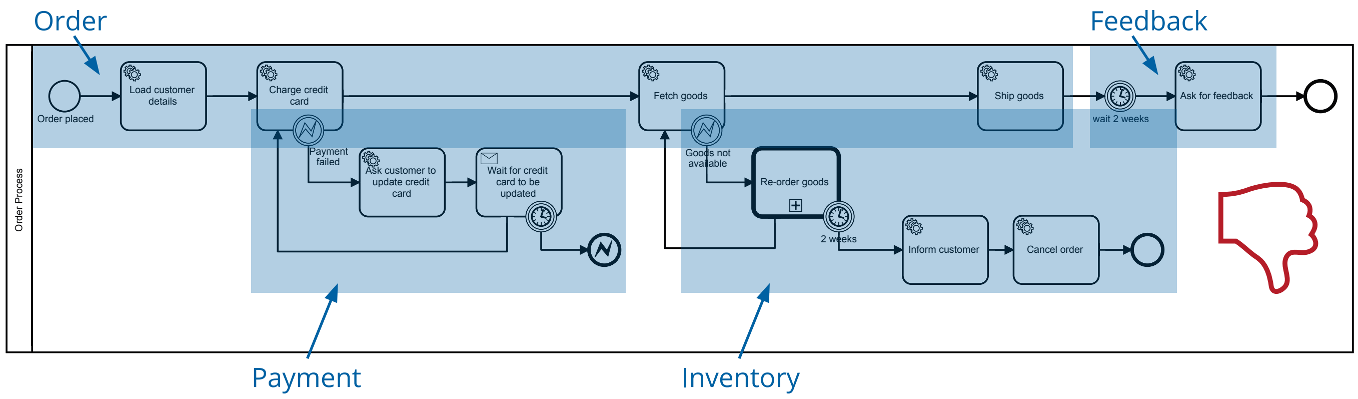


Figure 8: "Monolithic" order process violating microservice ownerships

To be fair, in many organizations this way of modeling is very valid and will work. So it is not bad per se! But it requires a process owner being responsible for the whole scope which might be the case if the company is "monolithically thinking" (which again is not necessarily a bad thing). But when you do take microservices seriously, this model is a no-go as you violate bounded contexts.

Instead you have to split the process into proper pieces which clearly belong to one microservice as shown in Figure 9. So the details of the *payment* process are a black box for the *order* fulfillment expert.

8 http://shop.oreilly.com/product/0636920033158.do

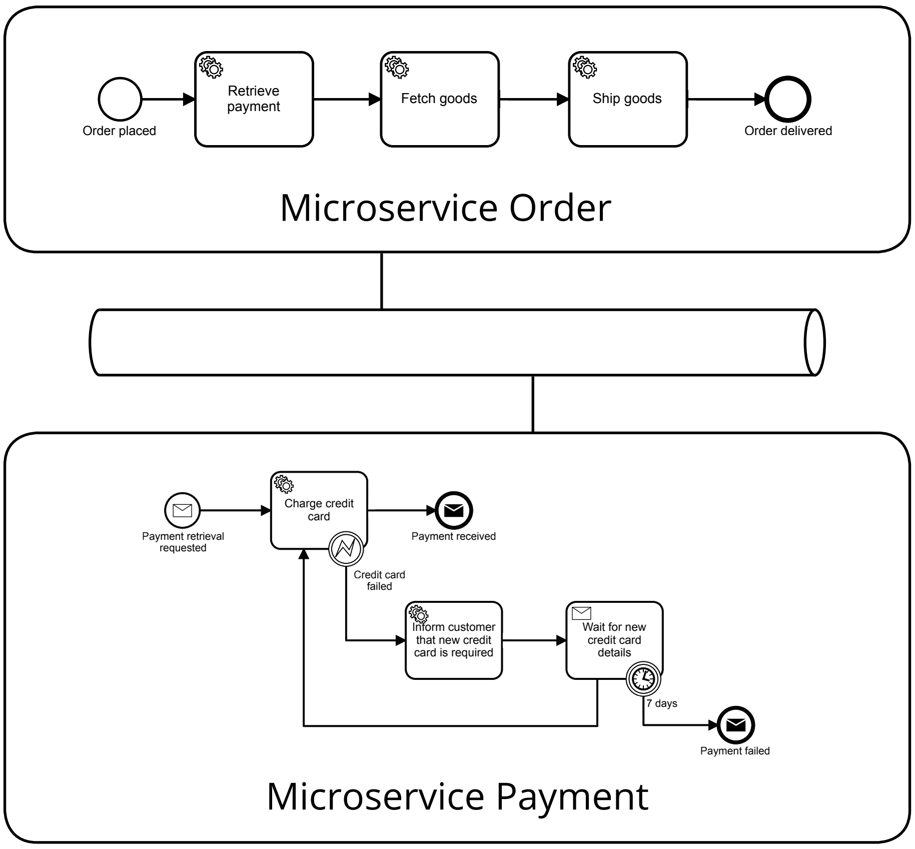


Figure 9: Split the overall process into pieces owned by the right microservice

There is a cool detail about BPMN by the way: You can still model the end-to-end process as a so called collaboration diagram if you want to have the big picture visible at least for discussions or requirements engineering, indicated in Figure 10. And we indeed have these discussions very often in early project phases, even when applying microservices. This is not about coordinated deployments or some other hard dependencies but a fruitful discussion raising the awareness of the overall business goal within every team. During implementation, the ownership of the parts has to be clearly taken by the corresponding microservice teams.

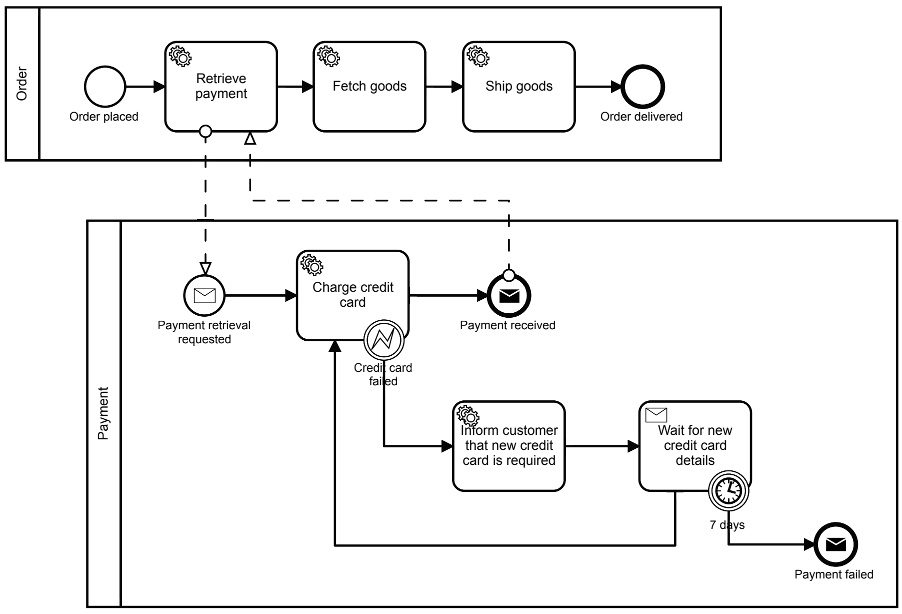
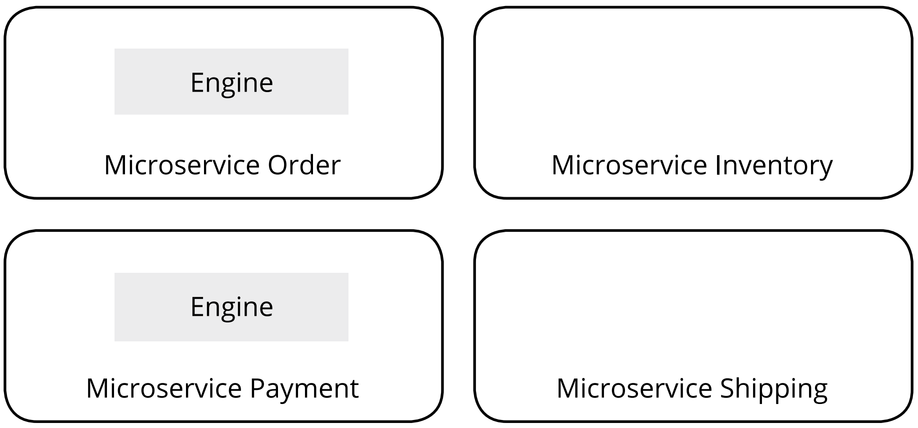


Figure 10: BPMN can also visualize service collaborations

Note that the communication between the processes is not really done by means of BPMN messages as drawn here as *order* should not even know that *payment* also runs a BPMN process.

Lightweight and Embeddable Engine

Most developers think of workflow engines being part of big proprietary zero-code BPM suites. But this is not true. Workflow engines can be very lightweight and feel like a development library. They can run embedded in the microservice. For example, you can easily setup a microservice using Spring Boot and the Camunda engine being started as part of that. This way every microservice has its own engine. The team owning that service can autonomously decide on the tool and version they want to use. The engine itself starts up very fast and can also be used to run automated unit tests in-memory without requiring any external dependency.



There is another important aspect. With Camunda you are not forced to model processes graphically but can express the very same thing in code. The order fulfillment might be simply expressed by:

```
camunda.getRepositoryService().createDeployment()
      .addModelInstance(Bpmn.createExecutableProcess("order")
        .startEvent()
        .serviceTask().name("Retrieve payment").camundaClass(RetrievePaymentAdapter.class)
        .serviceTask().name("Fetch goods").camundaClass(FetchGoodsAdapter.class)
        .serviceTask().name("Ship goods").camundaClass(ShipGoodsAdapter.class)
        .endEvent()
        .done()
      ).deploy();
```

That’s it. Graphical layout is done automatically by Camunda (version >= 7.7). This approach might sound strange to BPM-aficionados but developers are sometimes scared by graphical models as they do not see what is hidden behind them and expect hard to understand weird magic. The code above shows clearly that there is no hidden complexity and helps to get your developers onboard. Once they get familiar with BPMN and start to do more complex flows projects, they often switch to model graphically. The important aspect is, it is up to you.

Note that the order fulfillment example described in this whitepaper is available as running source code.[9]

Cloud-Ready Technology

Camunda can run embedded within a Spring Boot process. That allows you to deploy the engine as part of your microservice in a lot of different cloud environments. Another approach would be to run the Camunda engine as part of a Java Container like Tomcat or Wildlfy and distribute it as Docker container. And there are much more possible approaches due to the lightweight core of the Camunda engine which allows great flexibility for deployment options. Additionally, Camunda supports multi-tenancy which allows even more options. There is a whitepaper dedicated to the topic of multi tenancy.[10] So don’t worry about the specific cloud environment you have in mind.

Cloud-Ready License Model

Camunda uses a transaction based license model. This model is completely independent of number of servers, cores, environments, users, or the like. This fits perfectly in the world of microservices and the cloud. So, there is no problem to run an engine per microservice. It just seems that other workflow vendors seldom offer real cloud-ready models triggering the misconceptions mentioned earlier.

9 https://github.com/flowing/flowing-retail

10 https://network.camunda.org/whitepaper/19

Misconceptions and Wording

We want to reveal one “trick” we often apply successfully: We do not use the acronym BPM when talking to developers. We might not even use *business process*. We much prefer the term *workflow* or even shortened to *flow*. These words are less biased and therefor generate less rejections. The same goes for *orchestration* where we for example just talk about *service collaboration* or *implementing the flow*. And there is some truth in that anyway as with the flows we describe in this paper you might “only” implement a business transaction on collaborating services, there might not be any business process or workflow (often meaning human tasks involved).

Monitoring and Tasklist

There are two challenges when you run multiple engines within your microservice landscape:

- **How to implement proper process monitoring?** You typically want to have one place where you find information about all running processes and collaborations.
- **How to create only one tasklist UI for the end-user?** The user usually wants to see all his tasks at once regardless of which service generated them.

There is one approach to solve these challenges easily: The various engines point to the same database as shown in Figure 11. Then you can use the out-of-the-box monitoring and tasklist tools to work with this database. This approach is worth considering because of two Camunda features supporting it. First Camunda can work **deployment aware**. This means that every engine knows the processes deployed locally – and only touches them. So the *payment* service will not touch *order fulfillment* processes even if it can see them through the database. Secondly Camunda has **rolling upgrade** capabilities which means that you can run two versions of the engine on the same database. Let’s assume you upgrade the database to 7.8, then your microservices can still run 7.7 – and all services can subsequently upgrade to 7.8. So there is no need to touch all microservices at one point in time. And of course, Camunda also supports some clustered databases, so you do not have to introduce a single-point-of failure.

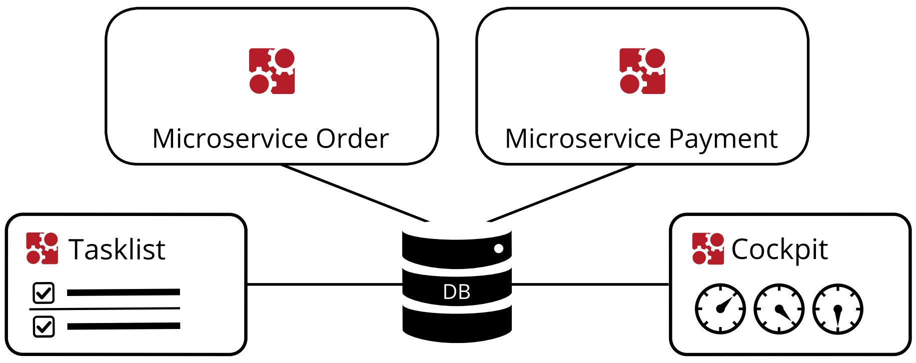


Figure 11: Multiple engines can share the database

The central database violates some microservice principles. However, we have customers using this approach as they gain an easy setup but keep a sufficient degree of de-coupling, so it's a good deal. But if you do not want to have that central database that is also possible. For the tasklist you push tasks to a central tasklist which could either be some third-party tool, some home-grown solution or another Camunda instance responsible to create and handle human tasks. The implementation might also be hidden in a *human task* microservice. Note that there are Camunda Best Practices available on how to integrate with an external tasklist from an engine.

Typically there is already a central monitoring system in place in microservice landscapes. Often this is based on the Elastic stack or similar tools. It is now easy to push all relevant events from all engines to it and provide a central overview which might link back to the right cockpit instance for details or operator actions, as visualized in Figure 12. You will not get a BPMN visualization in these tools though. If required it is not hard to build your own BPMN visualizations in such stacks using the lightweight bpmn.io[11] JavaScript framework.

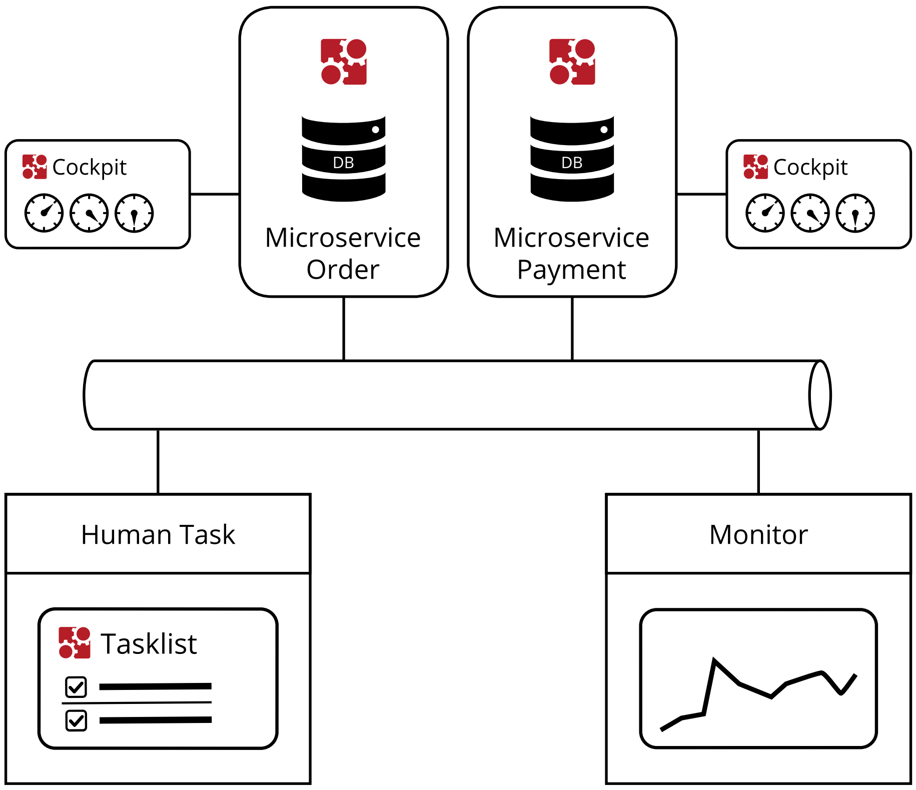


Figure 12: Multiple engines can easily push tasks or history events to central components

11 http://bpmn.io/

Conclusion

In microservice architectures, end-to-end business processes are carried out by collaborating microservices. Hence the overall business process must be distributed to various microservices according to the bounded contexts and business capabilities. However, as pure event chains increase coupling in a very unfavorable way, introducing proper event command transformations becomes essential. Often these transformations within the end-to-end process do not fit well in the ownership of any of the participating microservices. Then it is favorable to introduce an own bounded context and microservice for it.

The collaborations are often long running and require state handling. This is where a workflow engine fits perfectly in the microservice toolset as it helps the development team of one microservice to do a better job and also deliver added value like graphical visualizations, tooling or powerful features around timeouts, failure handling or compensation. With Camunda you get a lightweight workflow and BPM product that is flexible enough to cover different requirements and does not stand in the way of microservices. You can embed the engines into your microservices and are not forced to run any central component. The developer-friendly engine allows you to define flows programmatically, combine it easily with normal code and motivates you to write automated unit tests.

Transparency and business-IT alignment are important goals that should not be thrown overboard when applying microservices. You might have to overcome some common misconceptions about workflow or BPM in your company, but that “challenge” is definitely worth it.

Links & Literature

1 https://martinfowler.com/articles/microservices.html

2 Described in more detail in https://blog.bernd-ruecker.com/why-service-collaboration-needs-choreography-and-orchestration-239c4f9700fa

3 See also https://blog.bernd-ruecker.com/how-to-implement-long-running-flows-sagas-business-processes-or-similar-3c870a1b95a8

4 https://martinfowler.com/articles/201701-event-driven.html

5 https://blog.bernd-ruecker.com/how-to-implement-long-running-flows-sagas-business-processes-or-similar-3c870a1b95a8

6 https://github.com/flowing/flowing-trip-booking-saga/

7 http://shop.oreilly.com/product/0636920033158.do

8 https://github.com/flowing/flowing-retail

9 https://network.camunda.org/whitepaper/19

10 http://bpmn.io/

Imprint

Europe / Asia

Camunda Services GmbH
Zossener Str. 55
10961 Berlin
Germany

Phone: +49 (0) 30 664 04 09 - 00
E-Mail: info@camunda.com
www.camunda.de

America

Camunda Inc.
44 Montgomery St, Suite 400
San Francisco, CA 94104
USA

Phone: +1.415.548.0166
E-Mail: info@camunda.com
www.camunda.com