Redefining a Process Engine as a Microservice Platform

Antonio Manuel Gutiérrez–Fernández(✉), Manuel Resinas, and Antonio Ruiz–Cortés

School of Computer Engineering, University of Seville, Seville, Germany
{amgutierrez,resinas,aruiz}@us.es

Abstract. In recent years, microservice architectures have emerged as an agile approach for scalable web applications on cloud environments. As each microservice is developed and deployed independently, they can be developed in the platform and programming language that best suite their purposes, using a simple communication protocol, as REST APIs or asynchronous event-based collaborations, to compose them. In this paper, we argue that process engines provide an excellent platform to develop microservices whose business logic involves complex work flows or processes so that a Business Process language can be used as high-level language to develop these services and a process engine to execute it. We identify the requirements for integrating a process engine in a microservice architecture and we propose how the communication and deployment in a microservice architecture can be handled by the process engine.

Keywords: Process engine · Event-based asynchronous communication · Microservice architecture

1 Introduction

The popularity of microservices architecture is increasing in software development. Opposite to monolithic designs or classic 3-layer web application, microservice architectures propose decomposing application into small components around business concepts. Although distributed systems are not a novelty, current technology stack with web environments, cloud platforms or persistence servers hinders monolithic application development and deployment. In short, the microservice architectural style is an approach to develop a single application as a suit of small services around business concepts (opposite to functional responsibilities in n-layer architectures), each running in its own platform and communicating with lightweight mechanisms [3]. The rationale behind the microservices architecture is that decomposing complex applications

This work has been partially supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programmes under grants TIN2015-70560-R, P12-TIC-1867, and P10-TIC-5906.).

into microservices allows evolving them independently which leads to more agile developments and technological independence between them [4]. However, the decomposition into microservices increases complexity of integration so the use of event-based asynchronous communication is encouraged to also decouple integration interfaces between services. A number of organizations such as Netflix or Twitter have moved to microservice architectures as their services grew. As a matter of fact, one of the advantages of microservice approach is it allows to choose the best suitable language to develop it. The decision making regarding the platform lies on the microservice domain, developers expertise or technical requirements.

In the last two decades, business process applications have been developed using process engines. These systems currently support the development of web front-ends, REST interaction and light deployments, such as a Java library (e.g. Activiti[1] or Camunda[2]). Traditionally, process engines have performed the orchestrator role in Service Oriented Architectures. However, in a microservice based application, this role is not strictly required as services are self choreographed [10].

In this paper, we propose redefining the role of a process engine as microservice platform instead of an orchestrator of services, for the development of services whose business logic is a workflow, such as purchase orders in ERP. To this end, we analyse the main required properties of a microservice and how a process engine can be adapted to provide them as a microservice.

In the next section, we introduce an example application and introduce the main features of a microservice architecture. In Sect. 3, we propose a methodology to design the microservice and its interfaces with a process engine and, in Sect. 4, we review other works related to services and process engines. Finally, conclusions and possible work extensions are described in Sect. 5.

2 Motivation and Background

2.1 Purchase Order Management

We introduce, as an example, the development of an application that manages Purchase Orders including their shipment logistics and bank payments. The application starts when an employee fulfills a new purchase order, waits for budget responsible validation, then deals with provider to request shipment, track and receive it, and, lastly, handles the related account payments. This application manages Purchase Order lifecycle and its related shipment and payment and includes a number of decision points close to business domain. This lifecycle can be quite straight-forward modelled as a Business Process, BP depicted in the Fig. 1, and deployed it in a process engine. Therefore, a process engine can fit as development platform to implement Purchase Order lifecycle.

[1] http://activiti.org/.
[2] https://camunda.com/.

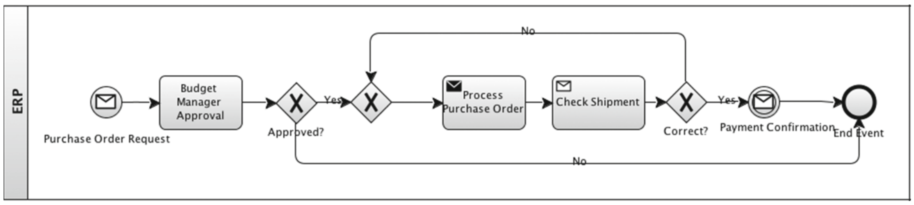

Fig. 1. Purchase order process

In current technology context, microservices architectures have provided an agile development pattern for scalable systems in cloud environments. These architectures are characterized by the following features.

Organized Around Business Capabilities. Monolithic systems become more difficult to maintain as they grow, so, following the single responsibility principle, microservice architecture approach proposes decomposing functionality into specific domain contexts to facilitate the development and testing stages and manage scalability. These separated domain contexts are named bounded contexts [2]. The decision about the granularity degree relies on the specific business domain concepts and development management (team size, expertise,...) so there are no fixed rules about decomposition scale but it should be enough small so a single team could be the only responsible for the development of a given microservice (and, e.g., in a development iteration, such as two weeks).

Decentralized Data Management. The microservice is responsible for managing the complete lifecycle of the business objects that fall in its bounded context (business logic, persistence,...) so that their details are encapsulated in the microservice and the interactions between objects in different microservices has to be performed with a public exposed interface.

Deployment in Isolation. Besides the encapsulation of the management of objects, each microservice has to be deployed in isolation, which decouples the development and execution platform between different microservices. Therefore, each microservice can be developed with the most suitable language for its purposes (considering technical or functional requirements). And, through this isolation, a microservice can be developed and updated avoiding that the related microservices (those that requires or provides operations from or to this microservice) require changes.

Interaction with Other Microservices. As we introduced, the dependencies between microservices are exposed with an application programming interface (API). Following the boundary context, this interface should be defined in terms of business concepts. In traditional service oriented architectures (SOAs), the interactions are commonly orchestrated through a mediator. However, in microservices architectures is preferred avoiding this orchestrated role so the services are self choreographed with a lightweight protocol, such as REST APIs. This choreography requires each microservice knows exact details of consumed operations (not only domain details but also endpoints or communication protocol). Furthermore, if they communicate with a synchronous pattern, microservices are dependent of consumed microservices (errors or too slow operations block execution). To avoid this coupling, event-driven asynchronous communication are preferred in microservices architectures [6]. The asynchronous approach requires a message broker which event consumers subscribe to and which handles emitted events to the related subscribers. With this approach, the blocking errors in consumed operations and the direct dependency between providers and requesters (event emitter and subscriber) are avoided.

Interaction with User Interface. Lastly, microservices architectures has to provide a proper handling of user interface. There are several approaches to develop the whole application frontend based on different microservices. On one side, we can develop a complete independent UI which uses the provided APIs by microservices to handle the business data. In this approach, UI component is highly coupled with the microservices and device-aware rendering is difficult as the provided APIs are business guided. Other approach is each microservice provides its own piece of User Interface and they are assembled to provide a full front-end. This approach requires a mechanism with templates systems or style sheets to provide a seamless appearance and it also is dependent on user client technological context where compositions could be harder to deal with (native applications or thick clients). And lastly, another approach is composing UI pieces from the different microservices in the backend, so a central server layer provides different user interfaces, for different user roles, devices, etc. This approach relies certain control in the UI management so it could handle logic it shouldn't.

2.2 Purchase Order Application as Microservice Architecture

Going back to our example, our application includes three business concepts - Purchase Orders, Shipments and Payments. Regarding the boundary contexts for a microservice architecture, we consider a decomposition into three microservices around these business concepts. The developers of these services should consider functional requirements, team expertise or technological aspects to implement them. As we introduced, business process modelling notation and process engine are, respectively, a suitable language and platform to develop Purchase Order logic. However, there are a number of challenges that have to be addressed in order for the process engine to perform as a microservice platform. First, the process engine has to support the management of the full lifecycle of Purchase Orders, including business logic and persistence (Point 1 in Fig. 2). Second, as the development and deployment is independent between different microservices, one

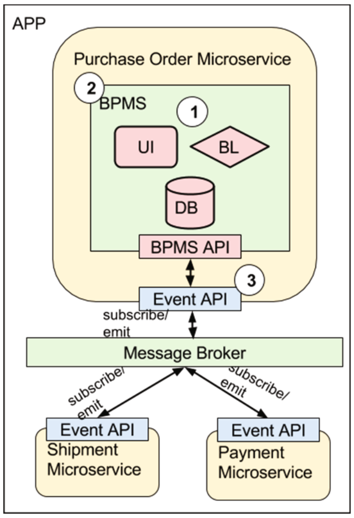

Fig. 2. Microservice architecture example

process engine has to be independently deployed and executed for each microservice where they perform as platform (opposite to traditional single deployment for a process-based application), as depicted in Point 2 in Fig. 2. Furthermore, process engines usually provide management and interaction interfaces but they have to be adapted or wrapped to communicate in asynchronous event-based ground (Point 3 in Fig. 2) and focused on business events (not to process events). And, last, the integration of user interfaces for the different microservices have to be addressed.

3 Building Microservices with a Process Engine

In this section, we argue how to address the introduced microservice features with a process engine as development platform for Purchase Order microservice.

3.1 Organized Around Business Capabilities

The example application introduced in previous section involves three business concepts: Purchase Order, Shipment and Payment. We depicted the business process for Purchase Orders which handles the full lifecycle of them. Shipment and Payment have their own processes, related to different business areas as logistics and accounting. Therefore, a decomposition into three microservices around each business concept fits with the leading single responsibility principle in the microservices architectures.

3.2 Decentralized Data Management

We have proposed developing Purchase Order microservice with a process engine as a platform. As we introduced, this implies the process engine has to manage the full lifecycle of Purchase Order objects. Process engines commonly self manage objects during business process instances and delete them after they finish (sometimes they store them just for keeping history traceability), encapsulating their persistence system (relational database, runtime memory,...). Therefore, the domain object managed by microservice has to be completely handled by one business process instance, as the Purchase Order in the Business Process in Fig. 1 (in other case, the system should be extended to manage the objects out of the process instance runtime or other development platform would be encouraged).

The properties for Purchase Order Data Object are depicted in Table 1.

Table 1. Purchase order data object

Reference	An unique string to identify purchase order (e.g.: 'A#432-2015')
Description	A string to humanly describe the content of the purchase order (e.g.: 'Monitor 22")
Date	A date value related to Order date (e.g.: '15/12/2014')
Status	An enumerated from {ordered, verified, requested, delivered, finished} to indicate the purchase order status (e.g.: 'ordered')
Payment	Payment information (bank account, price, date,...). This resource is managed by another microservice
Shipment	Shipment information (provider, receive date,...). This resource is managed by another microservice

In process engines, data are not a first-class citizen and they are simple managed as process variables without relationship between them. Therefore relating object requires manual handled references or extending the object model.

3.3 Deploying Process Engine Microservice

A microservice has to be self managed, i.e., each one is deployed and run independently. Therefore, each microservice has to be deployed with their own platform. Process engines have traditionally provided a platform to develop single full process oriented applications. However, in the last years, lightweight open source process engines, such as Activiti or Camunda, have appeared where their process engine can be deployed as a Java library in a external application.

In the context of cloud applications, a number of solutions propose deploying independent platforms through Virtual Machines or other approaches as Docker containers (chapter Deployment in [6]). Therefore, the deployment of an individual process engine to perform a single microservice (through virtualization or container), enables the required self-management. Figure 3 depicts an example stack of technologies to provide a process-engine-based microservice with a Java process engine (such as Camunda). The business process model and their related code to automate tasks (e.g. Java code in case of Camunda) is installed on a Camunda instance. This instance runs over a Java Virtual Machine and its single deployed on a Virtual Machine. Camunda stores execution data on a database which commonly would be deployed in a separate Virtual Machine. And the two Virtual Machines together form the microservice.

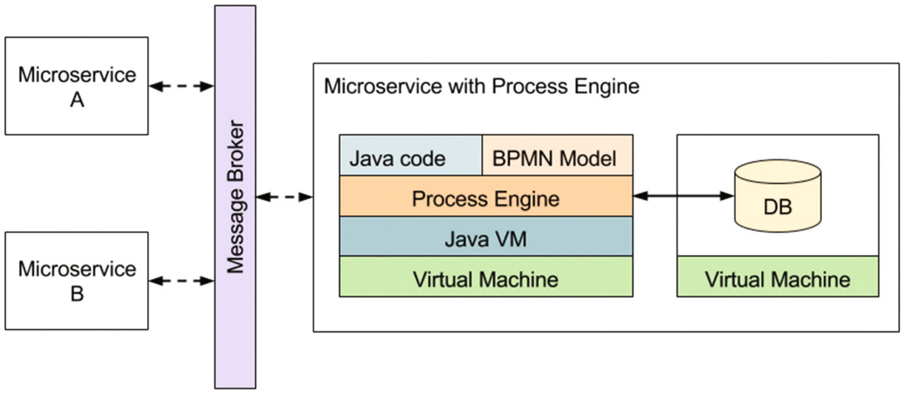

Fig. 3. Process engine microservice technology stack

As a result of this stack, there could be several process engines running simultaneously (one per each microservice) in the same application but the introduced overhead is not significant and it allows that different microservices evolve and scale independently.

We have also to point out that, in this work, we propose getting advantage of the process engine capabilities but Business Process Management Systems also commonly provide other functionality, such as a Process Modeller, that could be used in the processes definition but this is not deployed or included in the microservice.

3.4 Managing Microservice Asynchronous Communication

In order to develop an asynchronous event-based mechanism with a message broker, such as RabbitMQ -which fully support enterprise integration patterns- is proposed, although a similar simpler approach could be based on Atom standard and HTTP.

We study how to communicate with the process engine through a message broker to provide an asynchronous event-driven communication protocol, as depicted in Fig. 4.

In the figure, related to our example Purchase Order, an event "New Purchase Order (PO)" is emitted by an external service. Then it's propagated to the Purchase Order microservice (Step 1 in Fig. 4), which is subscribed to this event. This microservice, developed with a process engine, requires handling the event message as an operation provided by the process engine (Step 2 in Fig. 4). This operation is mapped to a process interaction, such as the creation of a "New Process Instance" (Step 3 in Fig. 4). The handling of "New Purchase Order" event message as a "New Process Instance" operation in the Process Engine is provided by an Event Mapper component. This component is part of the process engine microservice and would responsible for: (i) Subscribing to the microservice related events (e.g. "New Purchase Order" or "Payment Confirmation"), (ii) Map the incoming event message to the own operation provided by the process engine (e.g. "New Purchase Order" event message into "New Process Instance" operation and the outcoming requests from the process engines into event messages (e.g. "Process Purchase Order" request into "Process Purchase Order" event message). This map includes the routing and adapting of messages content.

Some process definition languages, such as BPMN, provide an event-based explicit mechanism to communicate process from different parties. There are a number of existing solutions to develop this Event Mapper component, such as Spring Integration or Mule, which provide mappers of different message protocols, so we can use them to different process engines (which are compatible with the supported message protocols). Modern process engines provide a synchronous management interface, commonly as a REST API, to create or interact with process instances, so it could also be used for several purposes (as support to user interface), but in this work, we focus on providing an asynchronous communication.

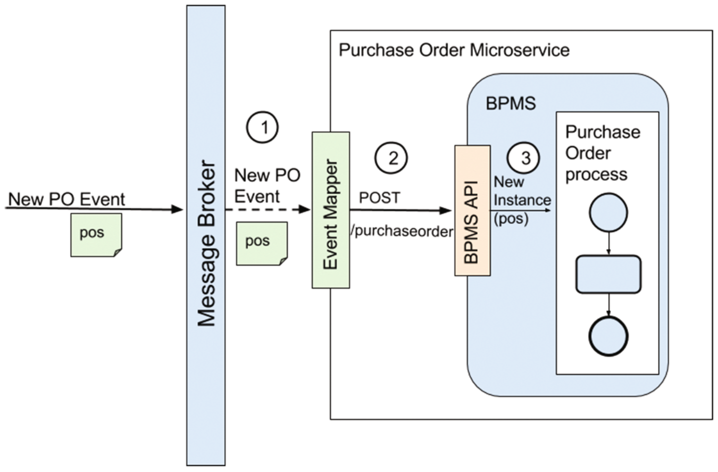

Fig. 4. Purchase order microservice with event-driven communication

Regardless the Event Mapper we use, to define a generic Event API and required mappings in the Event Mapper, we identify the required interactions with process engine that should be provided as events. These events relate with objects, regarding the possible states in their lifecycle: (i) Propagation of Object information, (ii) Creation of objects, (iii) Object updates. Therefore, the Event API has to meet the following requirements:

R1: Object information. Relevant changes in objects managed by the microservice process/es has to be notified through an event message.
R2: Creation of Objects. As without process instances there are no Objects, the API has to enable process instance creation.
R3: State-Machine awareness. The object state machine is provided by business process execution so the Event API has to enable interacting with process instances in those points that are waiting for external data, so the process instance can advance (and business object changes its state).

According to R1, all relevant changes in objects managed by business process have to be notified. This can be explicitly included in the process definition with the process language mechanisms or develop a generic mechanism of notifying any change on Data Objects. To support R2 and R3, we analyse tasks and data flow in business process for Purchase Orders. Regarding to R2, when a process instance starts there is no existing Data Objects so using events different to "New Purchase Order" should have no effect. If the process instance starting is related to the creation of a Data Object, then it is invoked by a New Data Object Event (i.e.: New Purchase Order Event). Regarding to R3, possible interactions with Object workflow are:

– An explicit business process event. If the business process explicitly waits for an external event, the object flow activates at the reception of a message (e.g., as intermediate event or as part of a reception task in BPMN).
– Implicit waiting task. On the one side, some tasks in process require from user interaction (User tasks). Modern process engines usually provide a web interface for this interaction so we can directly communicate with these interfaces through HTTP. On the other side, automatic tasks can also include active waiting for events (performed by programmatic mechanisms).

These stages in the proposed example are depicted with BPMN in Fig. 5. In the example, when a process is instantiated (1), a Data Object for a Purchase Order (PO) is created (from required Reference and Description values) and its Status is initiated to 'ordered'. After budget manager checks PO, the Data Object is updated in (2) to change the Status value to 'approved' or 'cancelled'. If the PO is approved, there is a request for shipment and when shipment is received in (3), the PO Object is updated again to change the Status to 'received' and to relate it to shipment Data Object.

Considering the previous discussion about Message, these three events can be related to external Messages. Purchase Order creation in (1) relates to message "New Purchase Order". Purchase Order update in (2) and (3) relates to

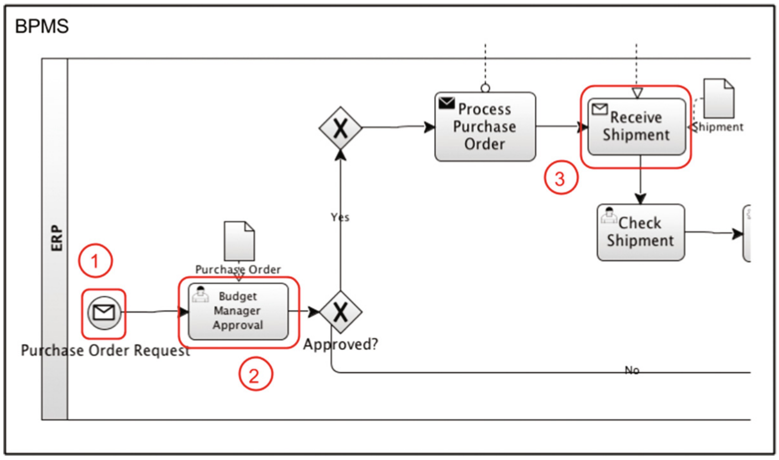

Fig. 5. Process flow interactions

Table 2. Mapping from events to process instance stage

Business object event	Process stage
New purchase order event	Start Instance
Budget manager approval	User Task
Shipment received	Receive Event
Purchase order update	Any stage

"Budget Manager Approval" and "Shipment Received" messages. After any creation or update of Purchase Order objects, corresponding event is notified with a "Purchase Order Update" message. A summary map for these relationships is depicted in Table 2. As deleting a Data Object is not consistent with a process workflow, we do not consider this event in our proposal.

3.5 Developing User Interface

In an event asynchronous communication pattern, the frontend can be developed using this event interface. Therefore, the frontend has to be aware of Purchase Order lifecycle to provide a consistent interface (e.g.: Interface to create a new Purchase Order has to provide all the required data fields for a new Purchase Order). To support any screen related to purchase order management, the user interface should subscribe to all "Purchase Order Update" events and emit an event after each "Purchase Order" updating ("Shipment Received" or "Budget Manager Approval" events) or creation ("New Purchase Order" event) through the user interface.

Process engines already feature user interface mechanisms to manage process instances, interact with tasks, authorization, etc. However, these facilities are thought to be used in a single process engine application and not integrated in a composed application. Therefore, the frontend has to be fully developed.

4 Related Work

As far as we know, there is no approach related to use a process engine as a platform to implement microservices. A number of research works have explored managing business processes as services. In [9], a REST API is proposed to consume a business process as a REST Resource. This idea is extended in [5] to provide a scalable architecture for business process services, considering runtime aspects. While both of them provide a REST API to interact with business processes, their goal is managing them as resources while our goal is to use business process model as the 'programming language' to manage -other- resources.

Other papers propose to use business processes as services orchestrators[3]. In this respect, Pautasso et al. [8] propose using BPEL to compose REST services and Bellido et al. [1] define control-flow patterns for REST services managed by business process. The base of these proposals is to use processes for their capabilities to provide a specific function, orchestration, in a service architecture while our proposal focuses on providing a generic purpose platform to develop microservices.

And, at last, Overdick [7], proposes extending BPEL to manage the state machine of resources in REST way. They get advantage of BPEL as event handler to control HTTP operations on the resources. So, while our approach is to fully develop suitable microservices with processes, this work only provides an extension to BPEL in order to capture and manage REST requests.

5 Conclusions and Future Work

In this paper, we have shown how to encapsulate business processes and process engines as programming language and platform to develop a microservice. Specifically, we first analyse the characteristics of microservice architectures to define the requirements for a process engine to implement a microservice. With these requirements, our proposal proposes different approaches to handle these requirements.

This work is a first approach to support a novel platform for the industry microservice architectures but there are several lines of future work. First, this approach has to be extended to more complex aspects such as interactions between processes in the same microservice or responsibility delegation events. Second, we focus on the interface to handle business objects, but common microservices also usually provide a management interface to monitor execution or to control configuration properties. And, last, this approach has to be applied to real scenarios to validate its feasibility.

[3] http://www.bpm-guide.de/2015/04/09/orchestration-using-bpmn-and-microservices/.

References

1. Bellido, J., Alarcón, R., Pautasso, C.: Control-Flow patterns for decentralized RESTful service composition. ACM Trans. Web (TWEB) **8**(1), 5 (2013)
2. Evans, E.J.: Domain-Driven Design: Tacking Complexity in the Heart of Software. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
3. Fowler, M.: Microservices, March 2014. http://martinfowler.com/articles/microservices.html
4. Fowler, M.: Microservices trade-offs, January 2015. http://martinfowler.com/articles/microservice-trade-offs.html
5. Gambi, A., Pautasso, C.: RESTful business process management in the cloud. In: 2013 ICSE Workshop on Principles of Engineering Service-Oriented Systems (PESOS), pp. 1–10, May 2013
6. Newman, S.: Building Microservices. O'Reilly Media, Incorporated, Sebastopol (2015). https://books.google.es/books?id=1uUDoQEACAAJ
7. Overdick, H.: Towards resource-oriented BPEL. In: Gschwind, T., Pautasso, C. (eds.) Emerging Web Services Technology, Volume II. Whitestein Series in Software Agent Technologies and Autonomic Computing, pp. 129–140. Birkhäuser, Basel (2008)
8. Pautasso, C.: RESTful web service composition with BPEL for REST. Data Knowl. Eng. **68**(9), 851–866 (2009)
9. Pautasso, C., Wilde, E.: Push-enabling RESTful business processes. In: Kappel, G., Maamar, Z., Motahari-Nezhad, H.R. (eds.) ICSOC 2011. LNCS, vol. 7084, pp. 32–46. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25535-9_3
10. Richards, M.: Microservices vs Service-Oriented Architecture. O'Reilly Media, Incorporated, Sebastopol (2015)