Ausgabe 05/2020

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 05

Java aktuell

iJUG Verbund
www.ijug.eu

Microservices
Performancetests, Service Meshes, MicroProfile GraphQL und mehr

Javas Geheimnisse
Weniger bekannte Features und Eigenheiten

Deep Learning
Einblick in das Trend-Thema

Klein aber oho: MICROSERVICES

JavaLand

16. - 18. März 2021
in Brühl bei Köln

Save the Date

Hybride Veranstaltung

Was die JavaLand als Plattform für Wissentransfer und Networking ausmacht, kannst du im Phantasialand oder online erleben. Als Teilnehmer entscheidest du selbst, welche Variante du wählen möchtest.

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu

Saga-Orchestrierung mit Imixs-Microservices

Ralph Soika, Imixs GmbH

Jedes moderne Projekt setzt heute auf eine Microservices-Architektur. Damit lassen sich einzelne Funktionen besser kapseln und schneller in kleinen Teams umsetzen und betreiben. Jeder, der in einem größeren Projekt arbeitet, merkt aber auch hier, dass die Komplexität nicht sinkt. Gerade wenn unterschiedliche Services in einem zusammenhängenden Geschäftsprozess koordiniert werden müssen, kann es schnell unübersichtlich werden. Saga-Orchestratoren bieten hierfür eine interessante Lösung.

Eine Kernidee der Microservices-Architektur ist es, die fachlichen Anforderungen auf einzelne „Micro"-Services zu verteilen. Über die Vorteile wurde bereits viel geschrieben. Ein Aspekt, der bei diesem neuen Architekturstil immer wieder auftaucht, ist die Verteilung der Verantwortlichkeiten. Dies ergibt durchaus Sinn, da dadurch auch die Komplexität von großen Datenstrukturen aufgelöst werden kann. An dieser Stelle kommt das Konzept des *„Bounded Context"* zum Einsatz.

Domain-driven Design

Die Idee hinter dem Begriff *„Bounded Context"* stammt ursprünglich aus dem *Domain-driven Design (DDD)* und wurde bereits im Jahr 2003 entwickelt. Die Idee hier ist es, die Fachlogik möglichst Technologie-neutral in einem Domänen-Modell zu beschreiben, um so die Auf-

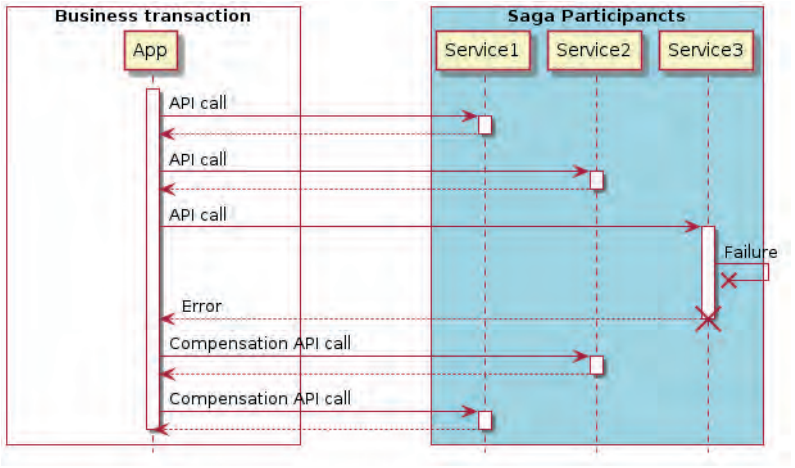


Abbildung 1: Das Saga-Pattern (© [Imxis-Workflow])

teilung in kleinere Blöcke besser zu unterstützen. Konkret bedeutet das, dass sich die einzelnen Teams zusammen mit den Fachbereichen Gedanken darüber machen, welche fachlichen Daten in einen Microservice gehören und welche nicht. Der einzelne Microservice trägt damit gleichzeitig die Verantwortung für seine Daten. Diese werden dazu oft in einer „lokalen" Datenbank unter Verwendung des Transaktionsprinzips verwaltet.

Verteilte Geschäftsprozesse

Problematisch wird die Sache mit der Verantwortung erst dann, wenn Daten zwischen einzelnen Services ausgetauscht werden müssen. Der Grund für den Austausch der Daten und die damit verbundene Verknüpfung der einzelnen Microservices untereinander sind hier nicht die Daten, sondern der dahinter liegende Geschäftsprozess. Und dieser lässt sich so gar nicht in eine Datenbank-orientierte Sichtweise zwängen. Geschäftsprozesse beschreiben den eigentlichen Zweck einer Anwendung aus Business-Sicht. Es geht hier weniger um die Daten, sondern vielmehr darum zu beschreiben, wie ein bestimmtes betriebswirtschaftliches Ziel erreicht werden kann. Sei es der Verkauf von Produkten aus einem Onlineshop, die Abwicklung von Serviceleistungen auf einer Internetplattform oder die Steuerung von Produktionsabläufen in technischen Systemen. In jedem Fall müssen hierzu Daten zwischen den einzelnen Services ausgetauscht werden.

Nun stellt sich schnell die Frage, wer in solch einem verteilten System die Verantwortung für die Datenkonsistenz über die Grenzen einzelner Microservices hinweg behält. Beispielsweise muss bei einem Onlineshop die Aktualisierung des Lagerbestandes mit der Logistik und der Zahlungsabwicklung synchronisiert werden, um nicht Gefahr zu laufen, am Ende inkonsistente Daten in einzelnen Systemen zu haben. Das sogenannte „Saga-Pattern" *[1]* ist hier ein gängiges Entwurfsmuster, um solche Abhängigkeiten zu beschreiben und zu implementieren.

Das Saga-Pattern

Eine Saga beschreibt eine Folge lokaler Transaktionen innerhalb einer Microservices-Architektur mit dem Ziel, Daten über mehrere Services hinweg zu aktualisieren. Die Saga beschreibt zum einen die Reihenfolge der Aktualisierungen und zum anderen, wie in einem Fehlerfall bereits festgeschriebene Daten nachträglich korrigiert und damit die Datenkonsistenz des gesamten Systems sichergestellt werden kann.

Konkret bedeutet das, dass, sobald innerhalb eines Geschäftsvorgangs der Aufruf eines einzelnen Microservice fehlschlägt, die Saga die bereits festgeschriebenen Änderungen einzelner Microservices explizit rückgängig machen muss. Dabei spielt es keine Rolle, ob der Fehler technisch oder fachlich bedingt ist. Dies wird in *Abbildung 1* veranschaulicht. Der Geschäftsvorgang besteht aus einer Reihe verschiedener Service-Calls. Schlägt der letzte Aufruf fehl, müssen die vorangegangenen Änderungen der Service-Calls 1 und 2 wieder korrigiert werden. Dies wird als Kompensation von Transaktionen bezeichnet und ist ein wichtiges Konzept, um die Datenkonsistenz innerhalb eines lang laufenden Geschäftsvorgangs aufrechtzuerhalten.

Choreografie oder Orchestrierung?

Eine Saga-Implementierung besteht also aus einer logischen Abfolge einzelner Service-Calls. Doch wer koordiniert diese Aufrufe? Hier gibt es zwei mögliche Lösungsansätze: die Choreografie und die Orchestrierung.

Choreografie

Bei der Verwendung von Choreografie erfolgt die Koordinationslogik über Events. Dazu abonniert ein Saga-Teilnehmer die Events der jeweils anderen Teilnehmer und veröffentlicht daraufhin selbst eigene Events. Für die Kommunikation können diese dann entweder direkt an den nächsten Microservice versendet werden oder über einen zentralen Nachrichtenbroker – beispielsweise eine Kafka-Queue – gesammelt und asynchron verteilt werden. Auf den ersten Blick scheint diese Art der Koordinationslogik einfach realisierbar. In der Praxis ist sie jedoch meist schwer zu verstehen. Der Grund ist, dass es im Code keine einzelne Stelle gibt, die die Saga vollständig beschreibt. Vielmehr verteilt sich die Umsetzung der Saga auf die einzelnen Services und deren Implementierung von Ereignissen. Ändert sich der dahinter liegende Geschäftsprozess und somit die Saga, müssen unter Umständen mehrere Microservices gleichzeitig

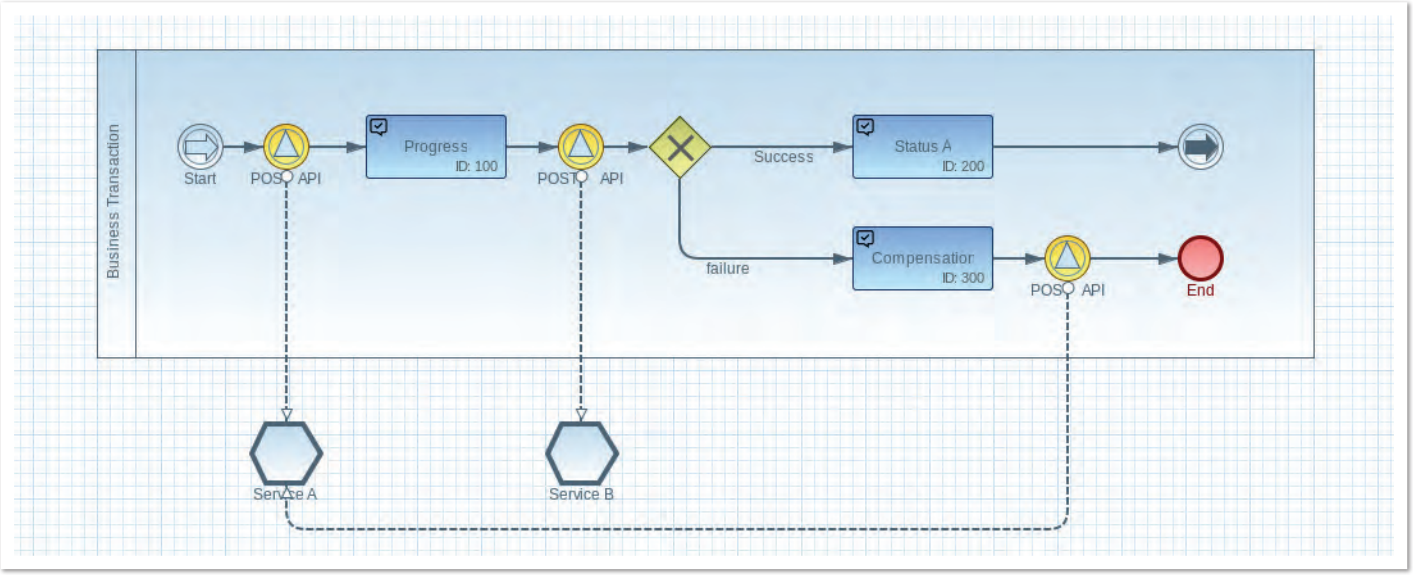


Abbildung 2: BPMN-Modell (© [Imixs-Workflow])

angepasst werden. Ein weiteres Problem sind zyklische Abhängigkeiten, die bei komplexer werdenden Geschäftsprozessen schnell entstehen können.

Orchestrierung

Die Orchestrierung stellt eine alternative Möglichkeit dar, eine Saga zu implementieren. Bei der Verwendung von Orchestrierung ist es die alleinige Verantwortung eines Saga-Orchestrators, die Saga-Teilnehmer darüber zu informieren, was zu tun ist. Der Saga-Orchestrator tritt hierbei selbst als Microservice auf, der mit den Teilnehmern wahlweise synchron oder asynchron kommuniziert. Bei jedem neuen Prozessschritt ruft dieser einen bestimmten Teilnehmer auf. Der Vorteil: Der Saga-Orchestrator ruft die Teilnehmer direkt über ein API auf und muss nicht über Ereignisse von außen informiert werden. Infolgedessen ist der Orchestrator von den Teilnehmern abhängig und nicht umgekehrt. Die Teilnehmer haben selbst keine Kenntnis von der Saga. Es gibt weniger Kopplung und auch keine zyklischen Abhängigkeiten.

Model-driven Design mithilfe von BPMN

Wie kann nun das Saga-Pattern innerhalb einer Microservices-Architektur konkret mit einem Saga-Orchestrator umgesetzt werden? Da es sich hier letztendlich um die Beschreibung eines Geschäftsprozesses handelt, ist ein modellbasierter Ansatz hier naheliegend.

Die Business Process Modelling Notation (BPMN) *[2]* hat sich für die Beschreibung von Geschäftsprozessen mittlerweile als Defacto-Standard etabliert. Mithilfe von BPMN lassen sich zum einen fachliche Abläufe beschreiben, zum anderen bietet BPMN aber auch die Möglichkeit, technische Umsetzungsdetails in einem Modell zu hinterlegen. Dadurch wird das fachliche Modell für ein Softwaresystem ausführbar und ist damit sehr gut für die Umsetzung des Saga-Patterns geeignet.

Imixs-Workflow

Das Open-Source-Projekt Imixs-Workflow *[3]* bietet mit seiner Microservices-Implementierung eine einfache Lösung, um das Saga-Pattern innerhalb einer Microservices-Architektur umzusetzen.

Die Imixs-Workflow-Engine kann selbst als ein Microservice über ein RESTful API angesprochen werden. Das Unterprojekt „Imixs-Microservice" *[4]* stellt dazu einen Docker-Container bereit, der mit Kubernetes oder Docker-Swarm schnell in eine entsprechende Architektur aufgenommen werden kann.

Nach dem erfolgreichen Setup des Service kann mit der Modellierung einer Saga mithilfe des BPMN-Modeler „Imixs-BPMN" begonnen werden.

Die Modellierung

Imixs-Workflow ist eine Event-orientierte Workflow-Engine. Das heißt, die einzelnen Zustände der Saga werden als Tasks beschrieben. Die Übergänge von einem Zustand zum nächsten erfolgen mithilfe von Events. Durch die Modellierung entsprechender Gateways kann nun der Ablauf im Erfolg oder Fehlerfall entsprechend modelliert werden. *Abbildung 2* zeigt ein vereinfachtes Beispiel für eine Onlinebestellung.

Sobald eine neue Order erstellt wurde, wird über das Event „Update Stock" zunächst der Lagerbestand durch einen API-Call am Stock-Service aktualisiert. Anschließend befindet sich die Saga im Status „Payment". Hier wird ein weiterer API-Call durchgeführt, um den Bezahlvorgang über den Payment-Service abzuschließen. Schlägt dieser fehl, wird über ein Gateway eine „Compensation"-Task eingeleitet. Diese leitet über einen erneuten API-Call gegen den Stock-Service die Korrektur des zuvor gebuchten Lagerbestandes ein.

Dieses stark vereinfachte Beispiel zeigt, wie die Koordination des eigentlichen Geschäftsprozesses mit einem Model-driven Design beschrieben werden kann. Ein weiterer Vorteil von Imixs-Workflow als Saga-Orchestrator ist die Tatsache, dass der jeweilige Zustand innerhalb der Saga sowie sämtliche bereits erfolgten Schritte persistiert werden. Es kann also immer auch rückwirkend der genaue Ablauf eines Geschäftsprozesses nachvollzogen werden. Gerade in Hinblick auf Revisionssicherheit und Compliance-Richtlinien ist diese Funktionalität bei komplexen Geschäftsabläufen eine zwingende Voraussetzung.

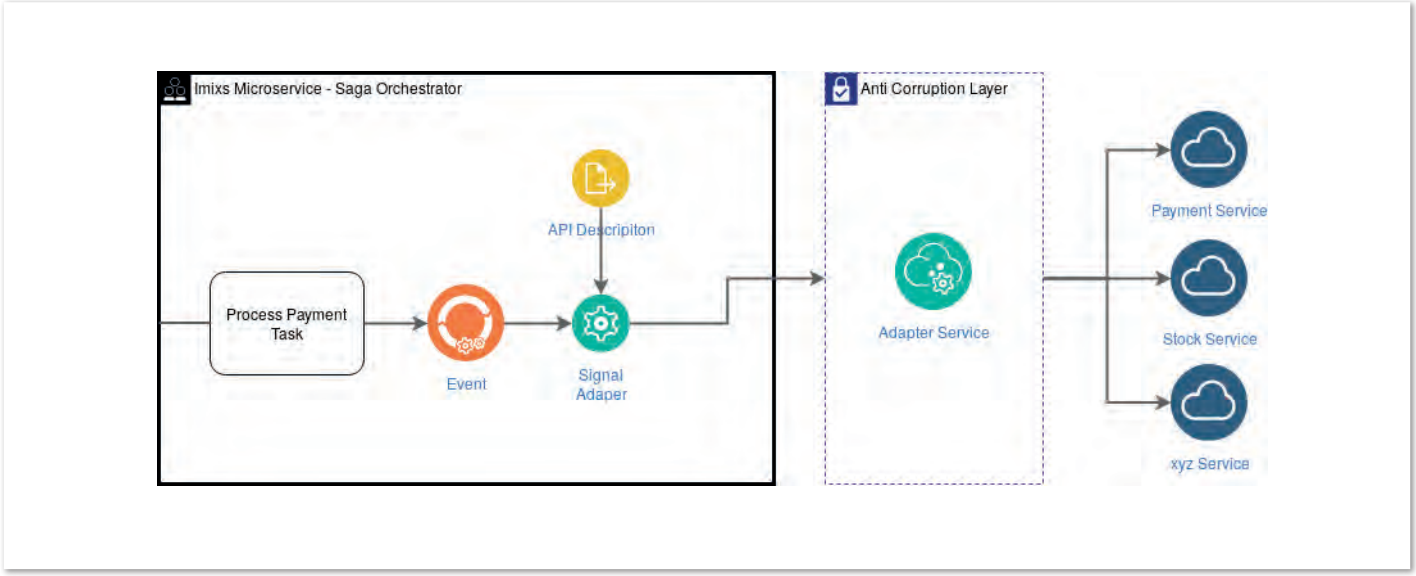


Abbildung 3: Anti-Corruption Layer (© [Imixs-Workflow])

Plug-ins und Adapter

Imixs-Workflow bietet über Plug-ins und Adapter die Möglichkeit, zusätzliche Funktionen und Dienste bereitzustellen. Diese helfen, den Implementierungsaufwand deutlich zu reduzieren.

Das Imixs-Workflow-Adapter-API *[5]* ist ein Erweiterungsmechanismus, um den Verarbeitungslebenszyklus eines BPMN-Ereignisses anzupassen. Eine Adapterklasse kann so an bestimmten Stellen innerhalb der Saga einen externen Microservice aufrufen und Daten senden oder empfangen. Das Ergebnis solch eines Aufrufs kann direkt für die weitere Prozessverarbeitung ausgewertet werden.

Listing 1 zeigt die Implementierung eines Adapters anhand eines Beispiel. Der Adapter erhält durch die Workflow-Engine Informationen zum aktuellen Vorgang sowie zum aufgerufenen Event. Diese Daten können verwendet werden, um den eigentlichen API-Call an den externen Service auszuführen und das Ergebnis entsprechend zu adaptieren. Nachfolgende Calls können so auf diese Daten zugreifen – beispielsweise auf eine Transaktions-ID.

Schlägt der Aufruf fehl oder kann dieser fachlich nicht ausgeführt werden, wird eine AdapterException ausgelöst. Diese kann nun in der weiteren Verarbeitung über das BPMN-Modell abgefangen und entsprechende Kompensationen können eingeleitet werden.

Das Anti-Corruption-Layer-Pattern

Auch bei dem hier gezeigten Saga-Orchestrator entstehen letztendlich Abhängigkeiten zu anderen Services, wenn deren APIs direkt über eine Adapter-Klasse implementiert werden. Um diese Abhängigkeit aufzulösen, kann das sogenannte *Anti-Corruption-Layer-Pattern* eingesetzt werden. Es stellt eine Zwischenschicht zwischen dem Saga-Orchestrator und dem eigentlichen API-Call dar und übersetzt die Daten zwischen zwei Systemen *(siehe Abbildung 3)*.

Der eigentliche API-Aufruf wird damit in einen eigenen Microservice gekapselt. Dadurch trennt man die beiden Schichten und der Saga-Orchestrator kann nun unabhängig von weiteren Services entwickelt werden. Sollte sich der Geschäftsprozess dahinter ändern oder

```
public class StockAdapter implements SignalAdapter {
  public ItemCollection execute(
     ItemCollection doc,
     ItemCollection event) throws AdapterException {

  Client client = ClientBuilder.newClient();
  try {
    result=client
      .target(REST_URI)
      .path(doc.getItemValue("productId"))
      .request(MediaType.APPLICATION_XML)
      .post(Order.classEntity.entity(order,
        MediaType.APPLICATION_XML));
     // adapt result
     ...
  } catch (ResponseProcessingException e) {
    throw new AdapterException(
      StockAdapter.class.getSimpleName(),
      ERROR_API_COMMUNICATION,
      "Failed to call rest api!");
  }
}
```

Listing 1: Adapter-Beispiel

kommen weitere APIs dazu, können diese Änderungen nun direkt im BPMN-Modell vorgenommen werden, ohne dass der Orchestrator-Service selbst neu deployt oder angepasst werden muss.

Die eigentliche Konfiguration des API-Aufrufs im Modell erfolgt dazu über ein API-Description-Objekt. Dieses kann eine XML- oder JSON-Struktur mit den entsprechenden Informationen aufweisen und ist Teil der BPMN-Modellierung.

Fazit

Die Komplexität innerhalb einer Microservices-Architektur kann schnell erheblich anwachsen, wenn im Zuge eines Geschäftsprozesses eine Vielzahl von Services mit unterschiedlichen Daten (Bounded Context) koordiniert werden müssen. Das Saga-Pattern bietet hier eine elegante Möglichkeit, die Zusammenhänge und Abläufe zu beschreiben.

Das Open-Source-Projekt Imixs-Workflow stellt mit dem Imixs-Microservice eine elegante Möglichkeit bereit, einen Saga-Orchest-

rator selbst als Microservice zu betreiben. Dabei können komplexe Sagas mithilfe von BPMN modelliert werden. Das Plug-in- und Adapter-API von Imixs-Workflow erlaubt die Einbindung beliebiger Services und eine Umsetzung auch komplexer Abläufe über Gateways und Business Rules.

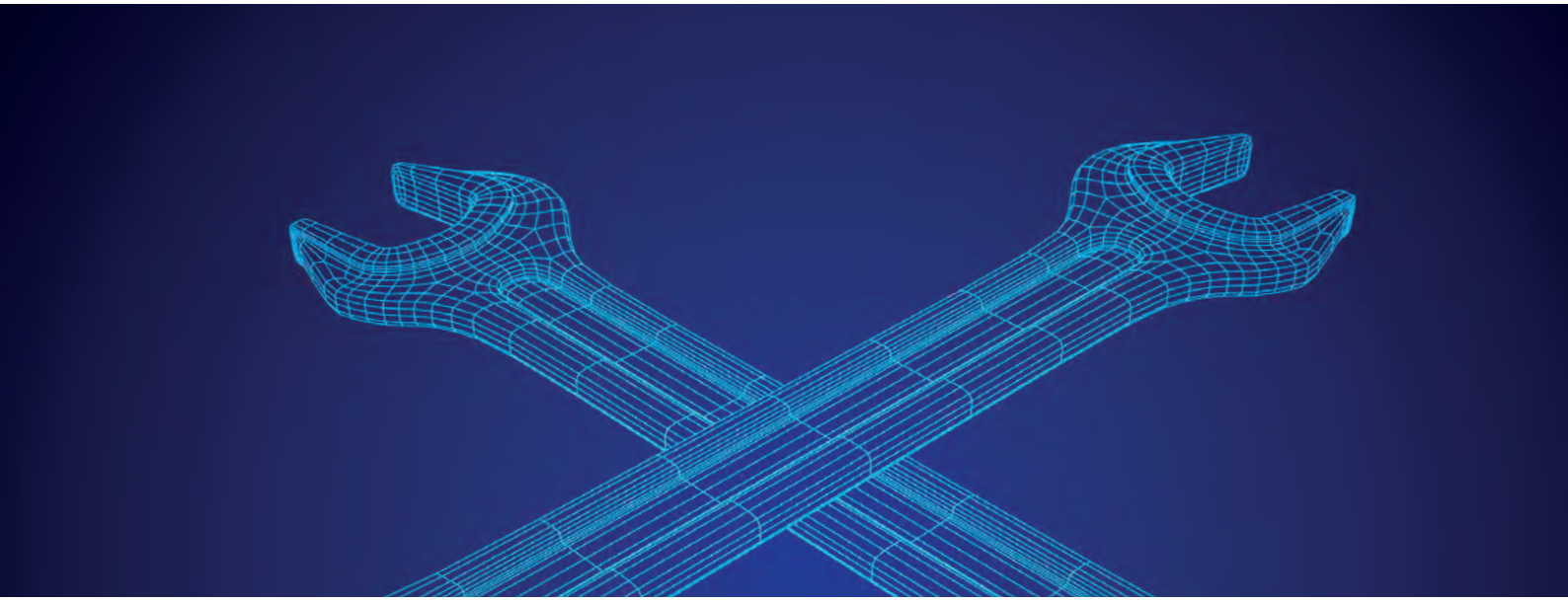
Durch den Einsatz eines Anti-Corruption-Layers kann der Saga-Orchestrator vollständig von den einzelnen Services entkoppelt werden. Dadurch werden feste Abhängigkeiten zwischen den einzelnen Services vermieden und Geschäftsprozesse lassen sich ohne Anpassung der Implementierung anpassen.

Quellen

[1] *https://microservices.io/patterns/data/saga.html*
[2] *https://www.omg.org/spec/BPMN/2.0/*
[3] *https://www.imixs.org*
[4] *https://github.com/imixs/imixs-microservice*
[5] *https://www.imixs.org/doc/core/adapter-api.html*

Ralph Soika
Imixs GmbH
ralph.soika@imixs.com

Ralph Soika ist Projectlead im Open-Source-Projekt Imixs-Workflow und Geschäftsführer der Imixs GmbH. Er berät seit mehr als 20 Jahren Unternehmen bei der Einführung von Geschäftsprozess-Management-Lösungen und Workflow-Architekturen. Ralph Soika entwickelt moderne Microservices-Architekturen und ist aktives Mitglied im Open-Source-Projekt Eclipse BPMN2 Modeler.



Service Mesh: eine Infrastruktur für Microservices

Jörg Müller, INNOQ

Service Meshes lösten als Technologie in den vergangen ein bis zwei Jahren ein hohes Interesse aus. Der Hauptgrund dafür sind die gestiegenen Anforderungen an die Infrastruktur durch den Trend zu Microservices-Architekturen. Können diese Anforderungen durch diese neue Infrastruktur gelöst werden?

2. + 3. DEZEMBER 2020

BERLINER EXPERTENSEMINAR

DOAG

NICOLAI PARLOG

Java after eight

KURSÜBERBLICK

In diesem Kurs werden die Java-Versionen 9 bis 15 beleuchtet. Dabei liegt der Fokus auf neuen Sprachfeatures wie Sealed Classes, Records, Text Blocks, switchExpressions und var, aber auch neue und erweiterte APIs sowie JVM- und Performance-Verbesserungen werden theoretisch vorgestellt und mit praktischen Übungen untermauert. Darüber hinaus werden der Migrationspfad von Java 8 zu 11 bzw. 15, der neue Release-Zyklus und die aktuelle Situation um Distributionen und Support besprochen.

ZIELGRUPPE

Erfahrene Java-Entwickler, die sich theoretisch und praktisch auf die neuen Java-Versionen vorbereiten möchten.

VORAUSSETZUNGEN

Einige Jahren praktische Erfahrung mit Java-Entwicklung und sicherere Java-8-Kenntnisse.

www.doag.org/go/expertenseminar_parlog