Dynamische Workflows in der Cloud

Juni 25, 2020

Autor: Jürgen von Hirschheydt

Bei der Entwicklung von Software Systemen für den Einsatz im Cloud Umfeld kommt man in der heutigen Zeit an einer Microservice-orientierten Architektur fast nicht mehr vorbei, ganz unabhängig davon in welchem Grad diese umgesetzt werden.

Damit gehen unweigerlich einige zusätzliche Fragestellungen einher, die unter Umständen einen schwerwiegenden Einfluss auf die letztendliche Architektur und Implementierung haben können, wie etwa:

- Größe / Abgrenzung der Microservices
- Domain-driven Design
- Kommunikation zwischen den Services
- State und Persistenz
- Skalierbarkeit

Zusätzlich zu den mehr technisch orientierten Anforderungen gesellen sich auch fachliche und wirtschaftliche Aspekte hinzu, die ebenfalls eine entscheidende Rolle spielen können:

- Time-to-Market Zeiten
- Customizing Möglichkeiten

Je nach Gewichtung dieser Kriterien kann die finale Lösungsarchitektur unterschiedlich aussehen, einige Grundkonzepte ziehen sich jedoch wie ein roter Faden durch alle Lösungen.

INHALTSVERZEICHNIS

1. MICROSERVICES ALS KOMPONENTE

Microservices stellen letztlich die Bausteine dar, aus denen sich das Gesamtsystem zusammensetzt. Dabei ist es relevant die richtige Balance zwischen der strengen Umsetzung des Patterns im Gegensatz zu notwendigen Freiheitsgraden hinsichtlich Performance, Latenzen und fachlicher Abgrenzung zu finden.

Je kleiner und genauer die Services geschnitten sind, desto mehr einzelne Services sind an der Ausführung eines Prozesses beteiligt - womit die Latenzzeiten in der Kommunikation der Services untereinander steigt. Andererseits sollten die Services auch nicht zu groß geschnitten werden, da ansonsten der eigentliche Vorteil des modularen Komponenten-basierten Aufbaus verloren geht.

Unabhängig davon sollten Microservices aber immer self-contained und stateless sein, da nur so eine Skalierbarkeit und Wiederverwendung gewährleistet werden kann. Sofern zusätzliche Daten über die Eingangsparameter hinaus erforderlich sind, muss sich der Microservice um die Beschaffung und Verwaltung dieser Daten selber kümmern. Somit bleibt der Microservice nach außen hin weiterhin stateless und die gleiche Instanz kann in unterschiedlichen Prozessen wiederverwendet werden.

PERSISTENZ / DATENERHEBUNG / STATES

Üblicherweise kommen hier Datenbanken zum Einsatz, wobei die Anwendung und Art der gespeicherten Daten über die genaue Technologie entscheidet. Handelt es sich mehr um transaktionale Daten, die mutmaßlich besser in einer SQL-basierten Datenbank untergebracht sind, oder um Dokumente, Graphen oder Media Assets.
Es gibt für jeden Anwendungsfall eine passende Technologie.

Aus dem Microservice Pattern heraus sollte jeder Service seine eigene Datenbank bzw. einen eigenen Datenbankkontext haben, wobei diese Anforderung oftmals die erste ist, die in einem Projekt aufgegeben wird. An dieser Stelle sollte kurz erwähnt werden, dass es auch legitim sein kann, diese Informationen in einem BLOB Storage unterzubringen.
State Informationen, die ein Service für die Ausführung benötigt, müssen ebenfalls in der Datenbank persistiert werden.

2. MICROSERVICES: KOMMUNIKATION

Da sich ein einzelner Prozess aus mehreren Services zusammensetzt, müssen diese miteinander kommunizieren können. Wenn zum Beispiel der Versand einer E-Mail in einem separaten Service implementiert wurde, dann muss diesem Service mitgeteilt werden:

- Wer sind die Empfänger?
- Was ist der E-Mail-Inhalt?
- Gibt es Anhänge?
- usw.

Historisch stellte dies niemals ein Problem dar, da einfach eine Library Funktion zum E-Mail-Versand aufgerufen wurde, die diese Informationen als Parameter übergeben bekam. Dies erweist sich im Cloud Umfeld als nicht so simpel, da durch Skalierung und mögliche asynchrone Abarbeitung ein solcher Aufruf nicht ohne Umstände erfolgen kann. Ein Grund dafür ist, dass der aufrufende Code keine Kenntnis davon hat, wo die dafür notwendige Instanz ist.

Das Problem lässt sich lösen, indem eine zusätzliche Instanz dieser Funktion aufgerufen wird und die Verantwortung für das notwendige Routing, sowie das Hoch- bzw. Herunterfahren der notwendigen Instanzen der Cloud Infrastruktur überlassen wird.

MESSAGE BROKER

IEine weitere Alternative hierzu ist ein Message Broker, dem lediglich eine Nachricht übergeben wird, welche wie ein Job Ticket alle notwendigen Informationen enthält. Aus dieser Nachricht in der Queue (oder Topic) wird dann ein Event erzeugt, welches den auszuführenden Microservice antriggert. Die sich daraus ergebenden Vorteile sind vielfältig. Im Einzelnen werden folgende Vorteile näher beleuchtet:

- Keine Code-level Abhängigkeiten der Komponenten
- Garantierte Auslieferung der Nachrichten, auch im Fehler- bzw. Offline-Fall
- Gezielte Skalierung möglich

Keine Code-level Abhängigkeiten

Im obigen Beispiel besteht die direkte Abhängigkeit darin, dass der Aufrufer den E-Mail-Versender kennen muss, um ihn anzusteuern. Änderungen am E-Mail-Versender müssen also in allen aufrufenden Code-Teilen angepasst werden (⍰ Refactoring). Geht man aber den Weg über den Message Broker, so verlagert sich diese Abhängigkeit auf die Nachricht und deren Format, die in dem Message Broker übergeben wird.

Das Wissen um die ausführende Instanz, also den E-Mail-Sender, ist hier nicht mehr relevant.
Durch diese Entkoppelung können nun die Komponenten zum E-Mail-Versand zu jedem beliebigen Zeitpunkt völlig isoliert vom Rest des Systems geändert werden, solange die neuen Komponenten das bestehende Nachrichtenformat verarbeiten kann.

Garantierte Auslieferung / Offline Fall

Besonders im transaktionalen Umfeld ist es wichtig zu garantieren, dass jede getriggerte Aktion auch durchgeführt wird. Historisch war hier eine sehr umfangreiche Fehlerbehandlung notwendig, während bei einem Message Broker in der Verbindung mit asynchronen, stateless Microservices ein Großteil dieser Fehlerbehandlung entfällt.

Wenn also die Komponenten zum E-Mail-Versand einen Fehler beinhaltet, oder gar nicht zur Verfügung steht, verbleiben die Nachrichten in der jeweiligen Warteschlange und können zu einem späteren Zeitpunkt von einer aktualisierten Komponente verarbeitet werden. De facto werden die Emails im obigen Beispiel also später, aber garantiert, ausgeliefert.
Es sollte allerdings sichergestellt sein, dass der eingesetzte Message Broker über ein sogenanntes „Dead Lettering“ sowie einen „Exponential Backoff“ verfügt.

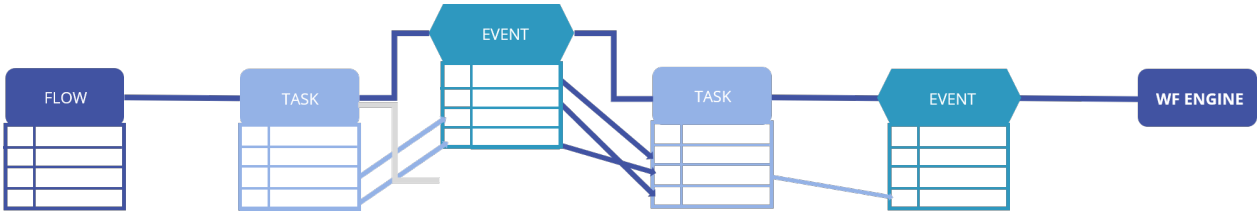
Gezielte Skalierung

Da ein Message Broker zu jedem Zeitpunkt die Anzahl der sich in den jeweiligen Warteschlangen befindlichen Nachrichten kennt, kann hier eine gezielte Skalierung der daran geknüpften Services eingestellt werden. Ein Versand von E-Mails hat sicherlich eine geringere Priorität als das Verbuchen eines Kontostandes, so dass für die Verbuchungen schneller zusätzliche Instanzen bereitstehen sollen, während beim E-Mail-Versand die Skalierung einen weniger starken Anstieg haben soll.
Hier wäre beispielsweise ein Verhältnis zwischen der Anzahl der Nachrichten zu auszuführenden Instanzen denkbar.

3. WORKFLOW STEUERUNG

Da durch die Entkopplung der Abhängigkeiten zwischen den einzelnen Microservices das Wissen um den Prozessablauf gewissermaßen aus dem Code entfernt wurde, muss der Prozess auf eine andere Art und Weise definiert werden.

Dementsprechend fehlt der Ablaufplan, wann welche Komponente etwas zu erledigen hat.
Hier kommt der Workflow ins Spiel, der definiert, wann welche Komponente aufgerufen werden soll:



Dabei werden Tasks durch Microservices abgebildet und die Events entstehen aus den Nachrichten innerhalb des Message Brokers. So lassen sich komplexe Prozesse und Datenströme „leicht" modellieren. In der Umsetzung gibt es nun mehrere Entscheidungen zu treffen, die mit ihren eigenen Fallstricken aufwarten.

Einerseits kann jeder Task sein gewünschtes Folgeevent direkt triggern, indem die entsprechende Nachricht in die Queue gelegt wird (Beispiel: „Versende diese E-Mail"). Das bedeutet allerdings auch, dass der aufrufende Microservice über Prozesswissen verfügen muss, das streng genommen außerhalb des Microservices liegt.

Auf der anderen Seite kann ein Microservice auch nur eine Art an Statusnachrichten absetzen, ohne sich direkt darum zu kümmern, wer diese verarbeitet. In unserem Beispiel würde der Bestellvorgang eines Shops eine Nachricht absetzen „Auftrag erstellt" und die notwendigen Bestelldaten an das Event anfügen. Von hier aus können nun mehrere Komponenten dieselbe Nachricht abarbeiten, wie etwa:

- „Lagerbestand verändern und verbuchen"
- „Rechnung erzeugen"
- „E-Mail mit Auftragsbestätigung an den Kunden versenden"

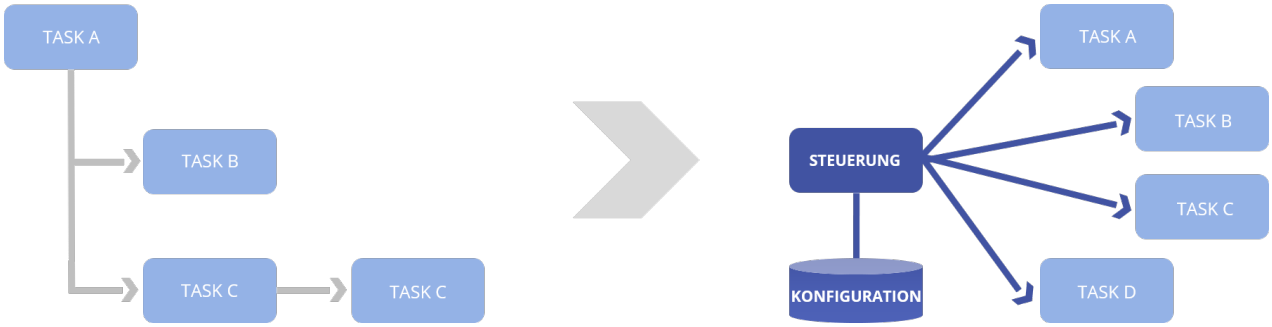
Sollte nun zu einem späteren Zeitpunkt der Prozess dahingehend geändert werden, dass Aufträge ab einem bestimmten Bestellwert zu einem Key-Account Manager gesendet werden sollen, so muss lediglich ein weiterer Event-Empfänger erstellt werden, der die Nachricht empfängt, die Kriterien prüft und die Bestellung intern weiterleitet.

Wichtig ist hierbei, dass die bestehenden Prozesse davon nicht betroffen sind (!)

Es kann so isoliert getestet, deployed und das System angepasst werden, ohne dass die bestehenden Prozesse erneut durch den Testzyklus laufen müssen.

4. WO BLEIBT DIE DYNAMIK?

Durch ein paar Umstellungen im Ablauf der Nachrichtenbearbeitung kann man erreichen, dass alle Nachrichten durch eine zentrale Komponente laufen, die dann entscheiden kann, an wen diese Nachricht ausgeliefert werden muss – ähnlich wie Verteilerlisten.

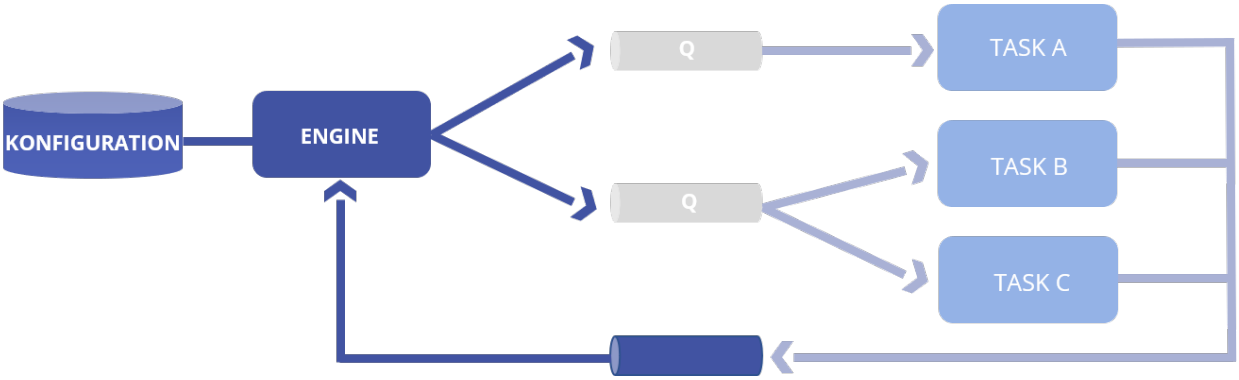


Indem jeder Microservice nun nur noch an einen Empfänger – die zentrale Workflow Steuerung – seine Nachrichten absetzt, kann der Prozessablauf als Regelwerk in einer Datenbank festgehalten werden.

Hier werden für jeden Workflow die Events mit den jeweiligen Folgeevents definiert, so dass komplexe Event-Ketten aus kleinen Bausteinen aufgebaut werden können, die letztlich den Geschäftsprozess abbilden.

Da die Prozessdefinition innerhalb der Datenbank festgehalten ist, kann diese dynamisch (sogar zur Laufzeit) geändert werden.

Die unterschiedlichen Message Broker Topologien erlauben dann auch die dedizierte Skalierung, da gesteuert werden kann, ob mehrere Nachrichten in dieselbe Warteschlange gehen, oder eine eigene Warteschlange besitzen. Pro Warteschlange kann dann eine eigene Skalierung definiert werden.



5. Anforderungen an die Implementierung

Bei der Implementierung einer solchen Workflow Steuerung ergeben sich einige Anforderungen, die von Anfang an bedacht werden sollten:

Idealerweise ein identisches Call-Interface der Microservices:
- JSON als Parameter, als hierarchischer Key:Value Store
- Nicht-relevante Felder im JSON sollten ignoriert werden
- Durchgängig identische Namenskonvention für Felder

Allgemeine Microservice Anforderungen:
- Self-contained
- Singuläre Aufgabe
- Stateless

Auf diese Art ist sichergestellt, dass die Workflow Steuerung alle Microservices über das identische Interface ansteuern und Parameter auf die gleiche Weise ausliefern kann.

6. Transaktionen

Um Transaktionen abzubilden sollte jedem Event eine Transaktions-ID mitgegeben werden, die für das Erste Event eines Workflows erstellt und an alle folgenden Events damit weitergereicht werden kann. Da es ein Rollback im herkömmlichen Sinne nicht gibt, müssen die Services in der Lage sein, aus den gleichen Parametern ihre jeweiligen Änderungen selbstständig rückgängig zu machen.

Das korrekte Abbilden und der Rollback von Transaktionen ist einer der komplexen Aspekte, die durch die Verteilung der Speicherung einen Mehraufwand in einer solchen Lösung darstellen.

7. Asynchrone Programmierung und Prozesse

Aufgrund der asynchronen Natur workflow-basierter Systemen ergeben sich eine Reihe von Anforderungen und Möglichkeiten bei der Umsetzung. Diese betreffen alle Bereiche, von der eigentlichen Programmierung, über das Userinterface (UI) bis hin zu den zugrundeliegenden Geschäftsprozessen.

Wo früher beispielsweise ein Auftrag im UI erstellt wurde, und der Benutzer beim Speichern solange warten musste, bis die Daten vollständig verarbeitet wurden, wäre heute in einer asynchronen Umgebung das UI sofort für die nächste Eingabe bereit, da die eigentliche Verarbeitung asynchron und somit „im Hintergrund“ abläuft.

Während das UI damit deutlich beschleunigt werden kann, muss an anderer Stelle dafür Sorge getragen werden, dass Folgeaktionen im UI erst nach Verarbeitung der Daten ausgeführt werden können, dass das UI also ebenfalls entsprechende Events erhält und verarbeitet.

Damit ist es möglich – und manchmal auch erforderlich – die Arbeitsabläufe anzupassen. Im genannten Beispiel wäre es jetzt möglich, von einer Person alle eingehenden Aufträge erfassen zu lassen, während die Sachbearbeiter im Hintergrund automatisch informiert werden, wenn ein Auftrag zur Weiterverarbeitung im System angelegt wurde.