Technische Hochschule Rosenheim, Fakultät für Informatik

Vorlesung: Verteilte Verarbeitung SS 2020

MSc. Thomas Mildner
Prof. Dr. Gerd Beneken

Praktikum 2 von 3 – Messaging Konzepte

Ziel der Aufgabe: Messaging, verteilte Systeme

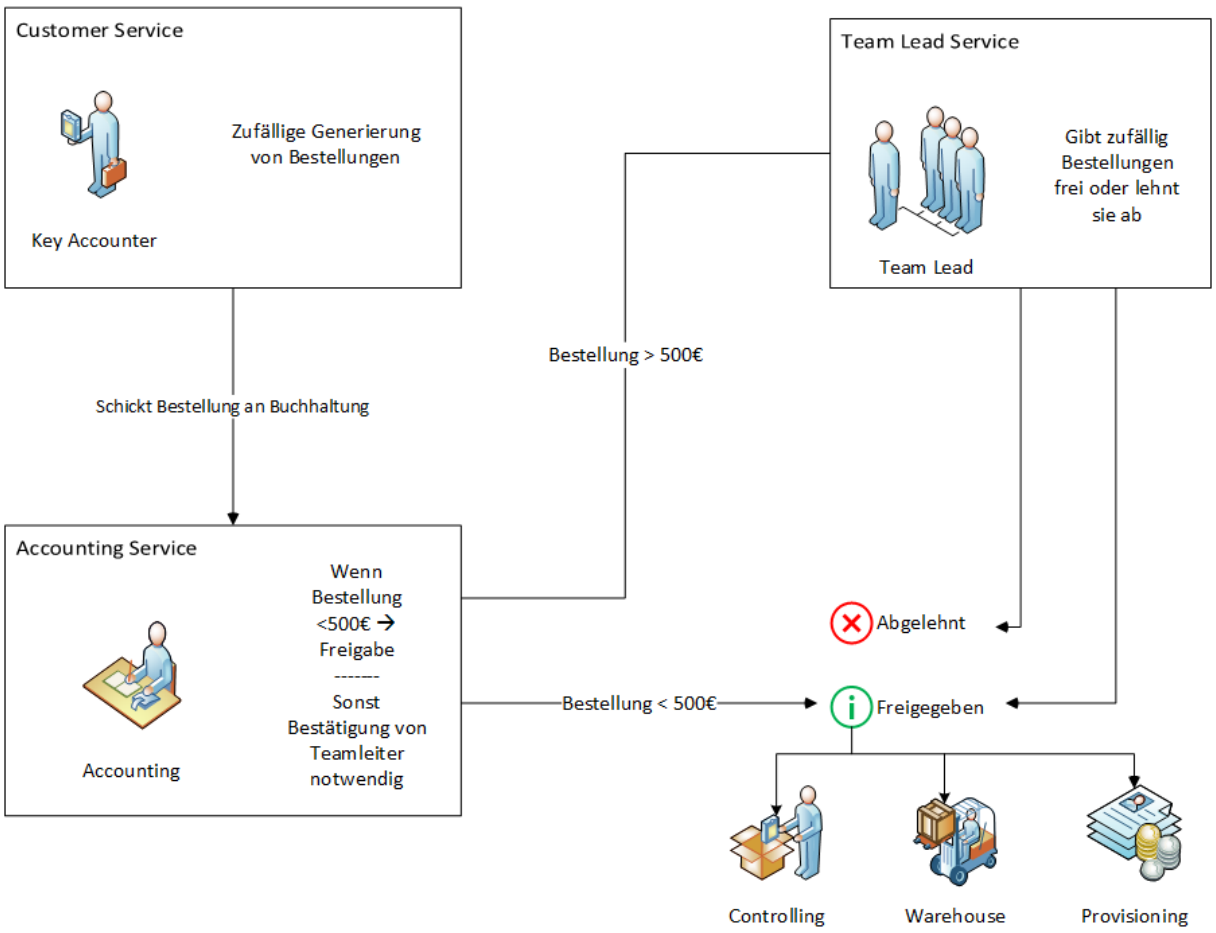
Wenn Sie diese Aufgabe erfolgreich abgeschlossen haben, können Sie

- Verteilte Systemarchitekturen entwerfen und planen
- Einen MessageBus für eine lose Kopplung der Systeme integrieren
- Das Publisher-Subscriber Pattern erklären und verwenden
- Das Prinzip hinter EDA (Event-driven Architecture) verstehen und erklären

Diese Aufgabe wird in Praktikumsaufgabe 3 erweitert werden. Achten Sie besonders auf eine saubere und gut dokumentierte Vorgehensweise.

Szenario: Stellen Sie sich folgendes, an die Praxis angelehnte Szenario vor.

- Ein Kunde gibt bei seinem Ansprechpartner (Key Accounter) eine Bestellung ein
- Diese Bestellung wird durch die Buchhaltung freigegeben oder an den Teamleiter weitergegeben
- Der Teamleiter entscheidet über die Freigabe oder die Ablehnung der hohen Bestellsumme



Services: Im verteilten System gibt es verschiedene Services mit unterschiedlichen, klar getrennten Aufgaben. Erstellen Sie für **jeden Service ein eigenes Projekt**. (Die Services laufen normalerweise nicht auf dem gleichen Rechner! Die Idee bei einer Microservice Architektur könnte sein, dass jeder Service in einer anderen Programmiersprache bzw. einem extra Git Repository liegt (verteilte, unabhängige Teams). Auf dieses Detail verzichten wir in dieser Übung und bauen stattdessen als Vorstufe eine Event-driven Architecture + alle Projekte in Ihrem, aus Aufgabe 1 bekanntem, TH Gitlab Repository.) Programmieren Sie auf Englisch, da wir uns vorstellen, dass ein englischsprachiges Team einen Service entwickelt, der ebenfalls mit dem System interagieren wird. Die Services kommunizieren untereinander mit Hilfe eines MessageBus. (Tipp: Verwenden Sie RabbitMQ)

Der MessageBus muss folgende Kommunikationskanäle bereitstellen:

- OpenOrders
- NeedsApproval
- ApprovedOrders
- DeclinedOrders

Machen Sie sich Gedanken ob sie eine Queue oder ein Topic nutzen wollen.

Tipp: Bedenken Sie bitte, dass teilweise Services nicht immer gleichzeitig online sein müssen (bspw. verschiedene Arbeitszeiten).

Schritt 0

Erstellen sie für diese Praktikumsaufgabe **Tickets in Gitlab** mit einer Beschreibung der hier dargestellten wichtigsten Schritte. Erzeugen Sie pro Service aus dem Ticket heraus einen **Feature-Branch** und checken diesen auf ihrem Rechner aus. Erstellen sie auf diesem Feature-Branch dann ihr jeweiliges Projekt unter der Verwendung von Gradle. Erstellen sie zusätzlich eine **Build-Pipeline in Gitlab**, welche ihren Code übersetzt, die Testtreiber ausführt und die **Testcoverage** berechnet.

Schritt 1

Erstellen Sie das Projekt „**CustomerService**“. Eine Bestellung („Order“) hat zwingend eine eindeutige (“OrderId“), eine Bestellsumme („Amount“), ein Erstelldatum („CreateDate“), ein Feld „ApprovedBy“ und einen zugeordneten Kunden. Das Feld „ApprovedBy“ wird durch den „AccountingService“ gefüllt werden, der CustomerService übermittelt hier keine Daten. Der Kunde hat mindestens eine eindeutige Kundennummer („CustomerId“ (Guid)), eine Anrede („Salutation“ bspw. Firma, Herr, Frau, Divers), einen Vor- und Nachnamen („First-LastName) sowie eine E-Mail-Adresse („Email“).

Die Bestellungen des Kunden werden durch einen „RandomOrderGenerator“ simuliert. Achten Sie auf plausible Daten und generieren Sie Bestellungen mit einer Bestellsumme zwischen 1 und 2000€.

Der Service besitzt eine Anbindung an den MessageBus und schreibt die generierten Orders in den Kanal „OpenOrders“.

Es muss möglich sein, dass viele „CustomerServices“ gleichzeitig gestartet werden. So simulieren wir ein Callcenter, das viele Bestellungen gleichzeitig aufnehmen kann.

Versuchen Sie diesen Service möglichst testgetrieben zu entwickeln.

Schritt 2

Erstellen Sie das Projekt „**AccountingService**“. Achten Sie darauf, dass Sie Unit-Tests für diesen Service schreiben müssen, daher sollte die main() Methode eher kurz sein.

- Der AccountingService überwacht den Kanal „OpenOrders“.
- Hat eine Bestellung einen Betrag unter 500€ wird diese freigegeben und in den Kanal „ApprovedOrders“ geschrieben. Bei einer freigegebenen Order wird dann das Feld „ApprovedBy“ mit dem Namen „Buchhaltung“ gefüllt.
- Bestellungen >= 500€ werden in den Kanal „NeedsApproval“ geschrieben.

- Zum Testen geben Sie die Informationen über Änderungen in ihrer Log-Datei aus, bzw. loggen auf die Konsole (ohne System.out, sondern mit Logger).

Schritt 3

Erstellen Sie das Projekt „**TeamLead**“. Dieses Programm wird vom Teamleiter des Callcenters bedient. Der Service kümmert sich um die Bestätigung von Bestellungen mit hohen Bestellsummen.

- Entwickeln Sie den Service testgetrieben.
- Der Service liest Bestellungen aus dem Kanal „NeedsApproval“ aus.
- Als „Freigabelogik“ wird ein Zufallsalgorithmus verwendet. Hier verzichten wir auf eine komplexere Entscheidungslogik. Der Teamleiter entscheidet per „Bauchgefühl“ ob die Bestellung freigegeben werden kann.
- Freigegebene Bestellungen werden in den Kanal „ApprovedOrders“ geschrieben. Zusätzlich wird das Feld „ApprovedBy“ gefüllt. Als Namen verwenden Sie „Teamleitung“.
- Abgelehnte Bestellungen werden in den Kanal „DeclinedOrders“ geschrieben.

Schritt 4

Ändern Sie die implementierten Klassen/Services wie folgt:

1. „RandomOrderGenerator“ wird in einem **eigenen Thread** ausgeführt. Das Generieren der Bestellungen wird kontinuierlich mit einem **konfigurierbaren Intervall** durchgeführt. Das Setzen des Intervalls kann beim Start als Kommandozeilen-Parameter mitgegeben werden.
2. Jeder Service braucht eine Anbindung an den MessageBus. Verwenden Sie für die Konfiguration der Verbindung zum MessageBus keine hardcodierten Werte. Schreiben Sie stattdessen eine kleine **JSON Konfigurationsdatei** inklusive einer Methode, die diese Werte beim Start des Services einlesen kann. Der Pfad zur JSON Datei wird über Kommandozeilen-Parameter beim Start mitgegeben. Falls kein Pfad angegeben wird, wird im aktuellen Verzeichnis nach einer JSON Datei gesucht. Die JSON Datei beinhaltet die MessageBus URL mit einer Port Angabe.
3. Stellen Sie sich vor, dass am Kommunikationskanal „ApprovedOrders“ weitere Services registriert wurden (bspw. die Controlling Abteilung, das Lager oder auch der Provisionsservice für den Callcenter Mitarbeiter). Diese Services müssen nicht implementiert werden! Bereiten Sie ihr System / Infrastruktur aber dementsprechend vor.

Schritt 5

Gehen Sie ihren Code noch mal durch. Und achten Sie besonders auf folgendes:

- Die Build Pipeline läuft durch.
- Ihre Testabdeckung ist über 80% und ihre Tests sind sinnvoll.
- Sie haben sich an alle Coding-Konventionen aus Java gehalten.
- Sie gehen sauber mit Fehlern / Exceptions um.
- Sie verwenden durchgehend Logging und nicht System.out
- Ihr Code ist auf einem Feature-Branch.
- Ihr Code + Kommentare sind in englischer Sprache

Wenn das alles erfolgt ist oder wenn sie sinnvolle Zwischenstände haben, dann stellen sie einen Merge-Request an ihren Betreuer.

Tipps:

- Achten Sie auch auf eine lose Kopplung innerhalb eines Services. Jede Klasse hat eine definierte Aufgabe und modulare, klar getrennte Methoden → falls die Namensgebung einer Klasse / Methode schwer fällt, ist die Funktionalität unklar bzw. zu groß.
- Verwenden Sie sinnvolle Methoden- und Klassennamen
- Achten Sie auf die richtigen Zugriffsmodifizierer (private, public usw.)
- Sie können den MessageBus als Docker Container laufen lassen. So müssen Sie keine extra Software auf Ihrem Rechner installieren. Bspw.

  ```
  docker run -d --hostname my-rabbit --name vvss20-exercise2rabbit -p 8080:15672 -p
  5672:5672 rabbitmq:3-management
  ```

  Die MessageBus Management UI ist anschließend unter `http://localhost:8080` erreichbar. Für die Verbindung zum MessageBus ist der Standard Port 5672 gewählt.
  Default User: `guest` Default Password: `guest`
- Jeder Service braucht eine Anbindung an den MessageBus. Vermeiden Sie möglichst „Duplicate Code“.