Technische Hochschule Rosenheim, Fakultät für Informatik

Vorlesung: Verteilte Verarbeitung SS 2020

Prof. Dr. Gerd Beneken
MSc. Thomas Mildner

Praktikum 3 von 3 – Schnittstellen-Konzepte

Ziel der Aufgabe: Rest API, Schnittstellendesign

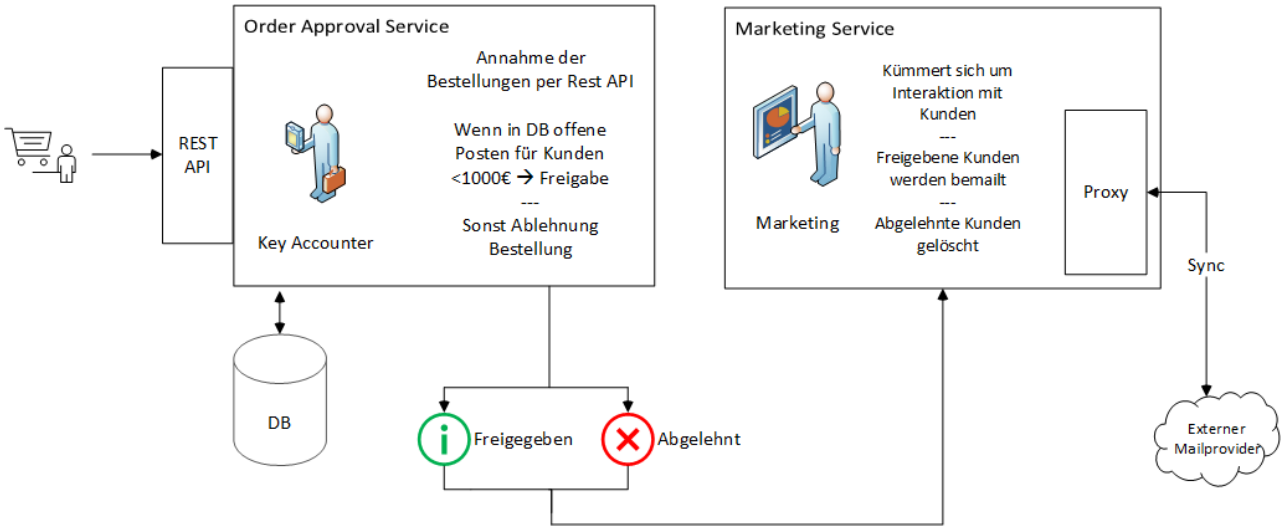
Wenn Sie diese Aufgabe erfolgreich abgeschlossen haben, können Sie

- Selbständig RESTful-Webservices entwerfen und dokumentieren. In diesem Beispiel mit Spring Boot mit OR-Mapper (Hibernate) und angebundener Datenbank
- Das Proxy-Pattern erklären und externe REST APIs anbinden
- Sichere Webservices programmieren und haben ein Grundverständnis für die Absicherung (TLS, Basic Auth, OAuth2, JWT) von RESTful Webservices

Diese Aufgabe ist eine Erweiterung der Praktikumsaufgabe 2. In dieser Aufgabe steht die Optimierung des Kundenservices im Vordergrund. Bekannte Prozesse sollen nun automatisiert und fehlertoleranter umgebaut werden.

Szenario: Das aus Aufgabe 2 bekannte Szenario soll nun im Kontext der Kundenzufriedenheit erweitert werden.

- Ein Kunde gibt automatisiert per REST API seine Bestellung in den Order Approval Service ein.
- Dieser hält in einer Datenbank alle notwendigen Informationen bereit und entscheidet selbständig über die Freigabe einer Bestellung. Hat ein Kunde offene Posten aus vergangenen Bestellungen über 1000€ wird seine Bestellung abgelehnt. Zusätzlich haben die Kunden die Möglichkeit direkt per API zu bezahlen und ihre Kontoinformationen abzurufen.
- Das Marketing Team hat mit einem externen Mailprovider die Möglichkeit freigebene Kunden zu bemailen. Abgelehnte Kunden werden nicht mit Angeboten beworben.



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 Services kommunizieren untereinander mit Hilfe eines MessageBus.

Um die Aufgabe zu vereinfachen muss der MessageBus nur noch folgende, aus Aufgabe 2 bekannte Kommunikationskanäle bereitstellen:

- ApprovedCustomers
- DeclinedCustomers

Normalerweise wären „Payment"-Aktionen weiterhin in einem Accounting-Service ausgelagert. Um eine komplexe Servicearchitektur zu vermeiden, kümmert sich der Approval-Service auch um Zahlungen und das Verwalten der Kundenkonten.

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

- Programmieren Sie den Service „**OrderApprovalService**". Eine Bestellung („**Order**") hat zwingend eine eindeutige OrderId, einen Amount („Bestellsumme"), ein Erstelldatum und einen zugeordneten Kunden. Der **Kunde** hat mindestens eine eindeutige CustomerId (Guid), eine Anrede („Firma, Herr, Frau, Divers"), einen Vor- und Nachnamen sowie eine **eindeutige** Email-Adresse. Die Klasse **Payment** hat eine eindeutige PaymentId, eine Referenz auf die OrderId, sowie die KundenId und einen bezahlten Betrag. Dieser kann von der Bestellsumme abweichen, darf sie aber nicht überschreiten (Teilzahlungen möglich).
- Der OrderApprovalService hat eine Anbindung an eine Datenbank. In dieser werden die Entitäten „Customer", „Order" und „Payment" gespeichert. Achten Sie beim Implementieren der Datenbank-Anbindung darauf, dass es zu Lost-Updates kommen kann und überlegen Sie sich bitte eine Strategie, wie Sie diese entweder vermeiden („Pessimistische Strategie" z.B. über Sperren) oder erkennen können („Optimistische Strategie" z.B. über Versionszähler oder Zeitstempel).
- Der Service besitzt eine Anbindung an den MessageBus. Der Service prüft bei eingehenden Bestellungen, ob ein Kunde noch offene Posten besitzt. Überschreiten diese den Betrag von 1000€ wird eine Bestellung nicht freigegeben und der Kunde in den MessageBus Kanal „DeclinedCustomers" geschrieben. Bei Freigabe einer Bestellung wird der Bestellbetrag auf den offenen Betrag eines Kunden addiert und der Kunde in den MessageBus Kanal „ApprovedCustomers" geschrieben.
- Der OrderApprovalService bietet eine REST API an. Diese muss mindestens folgende Endpunkte bereitstellen:
    - Customer:
        - GET:
            - Gibt alle Kunden zurück
            - Gibt Kunden anhand einer spezifischen KundenId zurück
        - PUT:
            - Ändere Kundeninformationen wie bspw. Mail, Vor- und Nachname. Die Kundennummer bleibt eindeutig und ist nicht veränderbar.
        - POST:
            - Legt einen neuen Kunden an. Die Kundennummer wird eindeutig generiert. Gleiche (= identische Email-Adresse) Kunden werden abgelehnt.
        - DELETE:
            - Lösche einen Kunden anhand einer gegebenen KundenId. Kunden können nur gelöscht werden, wenn keine offenen Posten mehr vorhanden sind.

    - Order:
        - POST:
            - Eine neue Bestellung wird aufgegeben. Die Bestellung wird auf Plausibilität geprüft. Überlegen Sie sich hier sinnvolle Validierungen.
    - Payment:
        - GET:
            - Gibt die Summe der offenen Posten für einen Kunden zurück.
        - POST:
            - Übermittelt eine neue Zahlung. Diese wird mit den offenen Posten des angegebenen Kunden verrechnet.

- Die API muss eine Validierung der eingehenden Requests vornehmen. Fehlerhafte Requests werden durch die API abgelehnt und nicht in den MessageBus bzw. an die Datenbank kommuniziert.
- Arbeiten Sie mit Versionierung Ihrer API. Diese API wird an einen imaginären Kunden gegeben. Stehen zukünftige Änderungen an der API an, werden die Kunden diese Änderungen realistischerweise nicht gleichzeitig implementieren. Durch eine Versionierung der API kann mit verschiedenen Versionen der API gearbeitet werden.
- Dokumentieren Sie Ihre API mit Swagger. Achten Sie auf eine umfangreiche und richtige Dokumentation. Bedenken Sie, dass Ihre API durch einen Kunden implementiert werden muss. Machen Sie sich besonders über die zurückgegebenen HTTP Return Codes Gedanken. Verwenden Sie hier standardisierte und sinnvolle Statuscodes.
- *(Optional)* Implementieren Sie den OrderApprovalService so, dass sich der Client authentisieren muss. Hierzu können Sie z.B. auf eine einfache Implementierung wie http-Basic-Auth setzen.

  (Beachten Sie bitte, dass Basic-Auth eine sehr einfache Implementierung ist und in der Praxis nicht verwendet werden sollte! Trotzdem gibt es keine Ausrede dass ein Webservice unsicher ins Netz gesetzt wird. Security ist in der Praxis kein optionaler Bestandteil!)

Schritt 2

Um die Kundenzufriedenheit zu erhöhen, wurde eine eigene Marketing Abteilung geschaffen. Der **MarketingService** besitzt eine Anbindung an den MessageBus um mit dem OrderApprovalService zu kommunizieren. Er liest aus den Kanalen „ApprovedCustomers“ und „DeclinedCustomers“ die Kunden aus.

Die Marketing Abteilung arbeitet mit einem externen Mail-Anbieter zusammen. In Schritt 1 haben Sie eine eigene Schnittstelle gebaut - ein ebenso häufiger Use Case ist aber die Anbindung einer externen Schnittstelle um die Funktionalität der eigenen Anwendung zu erweitern (bspw. Anbindung externer Payment-API wie PayPal usw.)

Das simulieren wir in dieser Praktikumsaufgabe mit der Anbindung eines externen VV-Demo-MailProvider. Dieser simuliert eine „Firma“, die eine API für einen Newsletter-Mailversand bereitstellt und ist unter https://vvdemomailserviceprovider.azurewebsites.net/index.html erreichbar. Beachten Sie, dass diese API deutlich umfangreicher ist, als wir in dieser Übung nutzen möchten. Die Kunst besteht darin zu erkennen, welche Endpunkte für die Use Cases notwendig sind und nur diese zu implementieren.

Das Marketing möchte folgende Use Cases abbilden:

- Die Kunden sollen an den Mail Provider übermittelt werden. Wenn ein neuer Kunde aus dem Kanal „ApprovedCustomers“ gelesen wird, wird er an den Mail Provider übermittelt.
- Das Marketing möchte eine Übersicht aller übermittelten Kunden, sowie nach speziellen Kunden im Mail Provider suchen können.
- Abgelehnte Kunden sollen auch im Mail Provider gelöscht werden. Wird ein abgelehnter Kunde aus dem Kanal „DeclinedCustomers“ gelesen, wird dieser beim Mail Provider gelöscht.
- Das Marketing möchte die Kunden per Mail bewerben können. Es soll genau definiert werden können, welche Kunden die Mail erhalten (Übergabe per Liste). Alle 5 Minuten soll eine Email über den Mail Provider versendet werden. Als Empfänger werden alle, in den Mail Provider hochgeladenen, Kunden ausgewählt.

- Der Status einer Mail soll durch das Marketing einsehbar sein. Nach Senden einer Mail wird der Status der Email überprüft. Diese Information wird in einer LogDatei im aktuellen Verzeichnis gespeichert.

Implementieren Sie das **Proxy Design-Pattern im MarketingService** um mit der externen API zu kommunizieren. Versuchen Sie die Logik für den Webservice Aufruf so zu kapseln, dass dieser leicht ausgetauscht werden könnte. (bspw. Wechsel des Marketing Mail Providers)

Hinweise:

- Die API des VV-Demo-MailProvider ist mit JSON Web Tokens (JWT) vor unbefugtem Zugriff geschützt (vgl. https://jwt.io/introduction/). Ohne einen gültigen JWT Token bekommen Sie nur den Return Code 401 – Unauthorized. Um einen gültigen Access-Token zu erhalten, müssen Sie den Endpunkt https://vvdemomailserviceprovider.azurewebsites.net/api/v1/authenticate aufrufen. Als Body verwenden Sie das Json-Format

  {

  "username": " **RepositoryName**",

  "password": " **vvSS20**"

  }

  mit Username = Ihr Repositoryname bspw. **ThomasMildner** und das Passwort ist **vvSS20**.

  Dieser JWT-Token muss als Bearer Token im Authorization Header bei jedem Request angefügt werden. (vgl. RFC6750 - https://tools.ietf.org/html/rfc6750)
- Ihre Requests werden auf dem VV-Demo-MailProvider protokolliert. Zur Beurteilung werden diese ausgewertet und auf Plausibilität überprüft. Die Anzahl der fehlgeschlagenen Versuche wird nicht abwertend beurteilt. Durch diese Maßnahme sollen Duplikate und Betrugsversuche unterbunden werden.
- Protokollieren Sie die API Abfragen als Ausgabe auf der Console. Verwenden Sie Logger!

Schritt 3

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

- Die Build Pipeline läuft durch.
- 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 definierte Funktionalitäten → 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.
- Jeder Service braucht eine Anbindung an den MessageBus. Vermeiden Sie möglichst „Duplicate Code“.
- Bauen Sie den RESTful-Webservice mit folgender Architektur (Controller – Services - Models). Vermeiden Sie umfangreiche Logik in den Controller Klassen.

- Testen Sie Ihre Schnittstellen mit Postman
- Achten Sie auf eine RESTful Implementierung Ihrer API. (vgl. Vorlesung)

VV DemoMailServiceProvider:

Die Implementierung auf https://vvdemomailserviceprovider.azurewebsites.net/index.html ist public erreichbar. Eingegebene Daten werden hier nicht persistiert.

Der DemoMailServiceProvider kann als Docker Container gestartet werden um lokal damit zu testen:

```
docker run -p 9210:80 -d --name VV_DemoMailProvider vvthromildner/vvdemomailserviceprovider:latest
```

Dieser Befehl startet den Service auf Port 9210.
Die Website ist damit lokal unter http://localhost:9210/index.html erreichbar.

Message Bus

Um einen vorkonfigurierten MessageBus für diese Aufgabe als Docker Container zu starten, kann der Container unter https://hub.docker.com/r/vvthromildner/vvss20_rabbitmq genutzt werden.