Verteilte Softwaresysteme

Blatt 2 — Teil 1 (Abgabe)

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung 28.11.2021 21:13

Abgabe spätestens am 25.01.2022, 08:00 Uhr.

Teilaufgabe 1: Abgabe im Wiki ihres Repositories auf dem LRZ GitLab.

Teilaufgabe 2: Abgabe durch Merge-Request auf GitLab.

Aufgabe — Online-Shop

Sie können dieses Blatt zu zweit oder alleine bearbeiten.

Wenn Sie das Blatt zu zweit bearbeiten, müssen Sie am Ende für Teil 2 nachweisen, wie Sie die Arbeit aufgeteilt haben. Schreiben Sie dazu die Aufteilung in den Merge-Request oder in die Datei `README.md`.

Das Repository wird für Ihre Gruppe auf GitLab generiert.

Teilaufgabe 1 — Planung

Die Planung geben Sie ab als Wiki-Seite(n), angereichert mit Grafiken etc. in Ihrem Repository. Um Grafiken hinzufügen, können Sie das Wiki als Git-Repository clonen, die Grafiken pushen und dann auf der Wikiseite referenzieren. Mehr Infos finden Sie z.B. unter `https://gitlab.lrz.de/help/user/markdown#wiki-specific-markdown`. Nutzen Sie insgesamt im Wiki eine vernünftige Formatierung per Markdown.

Ihr Chef beauftragt Sie mit der Planung eines Online-Shops, der auf einer Microservice-Architektur basieren soll. Gemäß des Domain-Driven-Designs wurden in einem Kickoff-Meeting bereits sieben Microservices identifiziert:

customer Verwaltung der Kundendaten
catalog Verwaltung der Artikeldaten
stock Verwaltung des Artikelbestands
order Verwaltung der Bestellungen
payment Verwaltung der Zahlung, Anbindung an Zahlungsdienstleister
shipment Verwaltung des Versands, Anbindung an Versanddienstleister
supplier Verwaltung des Wareneingangs, Anbindung an die Zulieferer

Verschiedene Randbedingungen sind bereits andiskutiert worden. Vieles davon ist relativ klar. Für den, sehr korrekten und sehr vorsichtigen, Chef sind folgende Anforderungen sehr wichtig:

- Eine Bestellung wird nur komplett verschickt. Wenn ein bestelltes Produkt gerade nicht auf Lager ist, wird die Bestellung trotzdem angenommen. Wenn das Produkt dann wieder auf Lager ist (Wareneingang durch Zulieferer), kann die Bestellung verschickt werden.
- Bestellte Artikel, die solange noch auf Lager sind, werden als reserviert markiert.
- Eine Bestellung darf erst verschickt werden, wenn sie bezahlt ist.

Erstellen Sie eine Planung für Ihren Online-Shop. Gehen Sie dabei insbesondere auf folgende Fragen ein:

- Welcher Service ist Besitzer welcher Daten?
- Welcher Service muss welchen anderen Service kennen?
- Wie kommuniziert welcher Service mit welchem anderen (synchron, asynchron)?

Strukturieren Sie Ihre Dokumentation sinnvoll mit Grafiken, Tabellen, etc.

Beachten Sie dabei die insbesondere die Grundsätze einer Microservices-Architektur: *loosly coupled* und *bounded context*. Vermeiden Sie unbedingt zyklische Abhängigkeiten.

Entwerfen Sie die Services so, dass die Services durch einen Client direkt angesprochen werden können und antworten. Für die Antwort notwendige Infos von anderen Services müssen die Services selbst einholen.

Beispiel: Der Client fragt den `catalog`-Service nach Infos zu allen Artikeln. Der `catalog`-Service fragt den `stock`-Service welche Artikel gerade auf Lager sind und kann in der Antwort die Artikel entsprechend markieren. Performanter wäre die Anfrage natürlich, wenn der `catalog`-Service gecached hat, was gerade verfügbar ist. Das hätte aber wieder den Nachteil, dass der `catalog`-Service bei jeder `stock`-Änderung seinen Cache updaten müsste. Entscheiden Sie selbst was für Sie mehr Sinn macht und begründen Sie so etwas.

Ist für die Umsetzung einer Anfrage eine komplexere Anfrage an einen anderen Service notwendig, muss der Client diese selbst machen.

Beispiel: Der Client möchte beim `order`-Service eine Bestellung aufgeben, es ist aber dazu noch kein Kunde im `customer`-Service angelegt. Dann muss der Client erst den Kunden im `customer`-Service anlegen und kann dann mit der Kunden-ID die Bestellung

aufgeben. Sinnvollerweise versichert sich der `order`-Service aber trotzdem noch, dass es wirklich einen Kunden mit der übergebenen ID gibt.

Treffen Sie selbst solche Designentscheidungen und begründen Sie sie entsprechend. Überlegen Sie sich gut wo synchrone Kommunikation und wo asynchrone sinnvoll ist.

Teilaufgabe 2 — Prototypische Implementierung

Wenn Sie dieses Blatt alleine bearbeiten, können Sie den Customer-Service sowie die Umsetzung der Szenarien 4 und 5 weg lassen. Bei Szenario 1 nutzen Sie einfach eine beliebige CustomerID und legen Sie keinen neuen Kunden an.

Implementieren Sie die Services gemäß Ihrer Planung in Blatt 2, Teil 1 prototypisch so, dass folgende Szenarien funktionieren:

1. Bestellung lagernder Produkte durch einen Neukunden ([*Nur für Zweierteams*] Kunde muss erst angelegt werden) bis zum Verschicken.
2. Bestellung von drei Produkten, von denen nur eines auf Lager ist, bis zum Verschicken. Die beiden nicht lagernden Produkte werden zu **verschiedenen** Zeitpunkten, durch verschiedene Zulieferer, geliefert.
3. Stornieren einer Bestellung, die noch nicht verschickt wurde.
4. [*Nur für Zweierteams*] Empfangen eines defekten Artikels aus einer Bestellung mit mehreren Artikeln als Retoure mit sofortiger Ersatzlieferung.
5. [*Nur für Zweierteams*] Empfangen eines defekten Artikels aus einer Bestellung mit mehreren Artikeln als Retoure mit Rückbuchung des entsprechenden Teilbetrages.

Loggen Sie in Ihren Services viel und sinnvoll, damit der Ablauf von mir nachvollzogen werden kann. Verwenden Sie zur Umsetzung Go, für synchrone Kommunikation gRPC und für asynchrone Kommunikation NATS. Verwenden Sie als Registry und als Key/Value-Store Redis.

Unter `https://gitlab.lrz.de/vss/misc/example-microservices` finden Sie ein Beispielprojekt, in dem gRPC, NATS und Redis verwendet wird. Persistierung von Daten ist nicht notwendig. Sie können Beispieldaten z.B. beim Start aus einer JSON-Datei einlesen. Sie brauchen keine Funktionalität zu implementieren um die ganzen Daten von außen, z.B. per gRPC, einzupflegen.

Implementieren Sie einen Testclient in Go für jedes Szenario. Der Client soll dabei alle Anfragen von außen an das System durchführen. Beispiel für Szenario 1: Anlegen Kunde, Aufgeben Bestellung, … aber auch Bestätigung durch externen Zahlungsdienstleister, dass die Zahlung eingegangen ist. Die Services `supplier`, `shipment` und `payment` würden in einem realen System externe Dienste anbinden, mit denen Ihr System kommuniziert. Triggern Sie das einfach durch den Client an, z.B. `Lieferung von 10 mal Produkt X` oder `Zahlung erhalten für Bestellnummer 1287` oder `Empfangen einer Retoure mit Ersatzlieferungswunsch`. Der Client ist also ein Testdriver für das gesamte System, kein Client im Sinne eines Kunden.

Erstellen Sie für jedes Szenario eine `docker-compose`-Datei so, dass zum Starten aller Services und des Clients ein Aufruf `docker-compose -f <file> up` reicht. Die `docker-compose`-Dateien müssen dabei Docker-Images aus Ihrem GitLab-Repository nutzen, die sie durch das GitLab CI erzeugen lassen. Verwenden Sie bitte auf keinen Fall nur eine `docker-compose`-Datei und starten Sie dort die verschiedenen Szenarios durch Verknüpfung mit `depends_on`. Hintergrund ist, dass ich beim Start jedes Szenarios alle Log-Ausgaben von allen Services sehen will. Loggen Sie daher lieber etwas zuviel als zuwenig! Achten Sie dabei aber darauf, dass es übersichtlich und sinnvoll bleibt.

Z.B. könnte das dann so oder ähnlich aussehen:

```
order: req info for customer id=7
customer: req for customer id=7
customer: customer id=7 does not exist
order: customer id=7 not found
```

Weiteres (eventuelle Abnahme etc.) wird per Moodle geregelt.

Anmerkung: Sie müssen **keine** (Unit-)Tests schreiben. Sie dürfen natürlich gerne. Bei dem abzugebenden System handelt es sich um einen Proof-of-Concept-Prototyp und nicht um ein qualitätsgesichertes System.