Cloud Grundlagen und Programmierung

Wahlmodul Sommersemester 2020

Professor: Raphael Herding

Abschlussdokumentation

Christian Szablewski

Marek Linnenbaum

Inhalt

[Einleitung 4](#_Toc46073582)

[Frontend 5](#_Toc46073583)

[Produktübersicht 5](#_Toc46073584)

[Warenkorb 5](#_Toc46073585)

[Auftragsbestätigung 5](#_Toc46073586)

[Backend 6](#_Toc46073587)

[Architektur 6](#_Toc46073588)

[Kommunikation/ Authentifizierung 6](#_Toc46073589)

[Kubernetes 7](#_Toc46073590)

[MiniKube/ Helm 7](#_Toc46073591)

[Horizontal Pod Autoscaler 7](#_Toc46073592)

[API-Dokumentation 7](#_Toc46073593)

[Resilience-Pattern 8](#_Toc46073594)

[Künstliche Anfrageverzögerung der Services 8](#_Toc46073595)

[Timeout 9](#_Toc46073596)

[Testfall 9](#_Toc46073597)

[Retry 9](#_Toc46073598)

[Testfall 10](#_Toc46073599)

[Circuit Breaker 10](#_Toc46073600)

[Testfall 10](#_Toc46073601)

[Bulkhead 11](#_Toc46073602)

[Testfall 11](#_Toc46073603)

[Rate Limiter 11](#_Toc46073604)

[Testfall 11](#_Toc46073605)

[Weitere Fragestellungen 12](#_Toc46073606)

[Basic Authentication 12](#_Toc46073607)

[Kubernetes Network Policies 12](#_Toc46073608)

[Service-Mesh 12](#_Toc46073609)

# Einleitung

Im Rahmen des Abschlussprojekts des Moduls Cloud Grundlagen und Programmierung wurde diese Dokumentation erstellt. Im Folgenden werden die funktionalen und nicht-funktionalen Eigenschaften dieses Abschlussprojekts beschrieben.

Es sollte ein einfaches Shopsystem entwickelt werden, welches aus drei verschiedenen Oberflächentypen, nämlich der Produktübersicht, dem Warenkorb und der Auftragsbestätigung, besteht. Im Backend laufen vier Microservices:

1. Cart-Service: Vorhalten der Einträge aus dem Warenkorb und Validation
2. Checkout-Service: Abwicklung des Bestellvorgangs und Validation
3. Shipping-Service: Abwicklung der Versandlogik
4. Catalog-Service: stellt alle angebotenen Produkte bereit

Zu Nutzen waren Angular JS für das Frontend und die Backend-Services sollten mithilfe von Spring-Boot umgesetzt werden. Außerdem sollte die REST-API in Kombination mit JSON benutzt werden.

# Frontend

Das Frontend wurde mithilfe von Angular CLI erstellt. Dieses Programm wurde bereits bei einem vorherigen Lab verwendet und somit ist man bereits mit den grundlegenden Funktionen vertraut. Es wurde für jede der drei verschiedenen Übersichten eine Komponente erstellt, in der dann jeweils eine HTML-, eine CSS und TypeScript-Datei erstellt worden sind. Dies wurde durch den Befehl

*„ng g c {KomponentenName}“* realisiert. Zusätzlich zu den einzelnen Seiten wurde eine *Default*-Seite erstellt, welche den Grundriss des Shopsystems beschreibt. In dieser wurde der Header des Shopsystems implementiert. Diese Default-Seite wird in jeder einzelnen Übersicht verwendet, sodass eine *Single-Page-Application* (SPA) entsteht.

Inhalt des Headers ist unter anderem der Warenkorb oben rechts, welcher per Klick an die Warenkorb-Seite weiterleitet und auch die aktuelle Anzahl der Produkte im Warenkorb anzeigt.

## Produktübersicht

## Warenkorb

## Auftragsbestätigung

# Backend

## Architektur

## Kommunikation/ Authentifizierung

Zur Kommunikation zwischen den Services haben wir Basic Authentication genutzt.

## Kubernetes

### MiniKube/ Helm

### Horizontal Pod Autoscaler

## API-Dokumentation

# Resilience-Pattern

Eine weitere Aufgabenstellung für das Abschlussprojekt war die Implementierung von fünf verschiedenen Resilience-Pattern. Resilience (übersetzt: Belastbarkeit) bedeutet, dass ein System nach Auslastungen und Belastungen verschiedener Art, wieder in seinen ursprünglichen Zustand wiederherstellen kann. Resilience-Pattern helfen dabei, diese Belastungen Stand zu halten. Es gibt viele verschiedene Resilience-Pattern, welche auf verschiedenste Belastungen reagieren.

Durch die Verwendung von Java Spring-Boot sind einige Pattern bereits vor implementiert und somit einfach umzusetzen.

In unserem Shopsystem haben wir die folgenden fünf Pattern implementiert.

### Künstliche Anfrageverzögerung der Services

Um die Resilience-Pattern vernünftig testen zu können, soll die Möglichkeit eingebaut werden, Anfragen von Services zwischen 100 Millisekunden und 10 Sekunden zu verzögern. Hierfür wurde einfach eine *SleepTime* im Service festgelegt. Danach wurde mit dem Befehl *Thread.Sleep(SleepTime)* die Anfrage künstlich verlängert. Diese Veränderung wurde in jedem Backend-Service implementiert. Zur Veranschaulichung hier ein Code-Ausschnitt aus dem Catalog-Service:

*//EINFÜGEN VON BEISPIELCODE*

## Timeout

Das Resilience-Pattern Timeout wird eingesetzt, um zu verhindern, dass Services unendlich lange aufeinander warten. Entsteht also bei der Kommunikation zweier Services ein Netzwerkproblem, welches die Antwort des einen Services um unbestimmte Zeit verzögern würde, so wird auch der wartende Service diese unbestimmte Zeit abwarten. Um also die potenziell unendlich lange Wartezeit zu verhindern, wird das Timeout-Pattern implementiert.

In der durch Spring-Boot zur Verfügung gestellten Klasse *RestTemplateBuilder* kann über die Methoden *setConnectTimeout* und *setReadTimeOut* eine Zeit angegeben werden, die der Service höchstens auf eine Antwort wartet. Wenn diese Zeit überschritten wird, so wird ein Timeout geworfen, welches an anderer Stelle verarbeitet werden kann. Eine Option wäre die Rückgabe einer leeren Antwort. In unserem Shopsystem haben wir das Pattern wie folgt implementiert:

*//EINFÜGEN VON BEISPIELCODE*

### Testfall

Zum Testen wurde die künstliche Verzögerung, welche implementiert wurde, auf 10 Sekunden gesetzt. Da das Timeout auf 3 Sekunden gesetzt wurde, ist ein Timeout geworfen worden, welcher dann abgefangen wurde. Die Auswirkung war, dass nach drei Sekunden eine leere Antwort zurückkam. Als Maßnahme reicht das Timeout-Pattern alleine also nicht aus, da auch mit einer leeren Antwort nicht weitergearbeitet werden konnte. Hierfür ist eine Kombination mit weiteren Resilience-Pattern notwendig.

## Retry

Das Retry-Pattern wird häufig in Kombination mit dem Timeout-Pattern verwendet. Das Timeout-Pattern bestimmt einfach nur, dass wenn innerhalb der Timeout-Zeit keine Antwort erhalten wurde, dann wird es auch keine mehr geben. Durch Implementierung des Retry-Patterns wird allerdings nach dem ersten erfolglosen Versuch eine Antwort zu erhalten, eine weiterer Request gesendet und diesmal auf eine Antwort gehofft. So besteht die Möglichkeit, dass die Antwort noch nachgereicht wird. In größeren Anwendungen kann beispielsweise der Load-Balancer den Request an eine funktionierende Instanz des Services weiterleiten, sodass dann eine Antwort erhalten werden kann.

In der Implementierung hilft Spring-Boot weiter, denn durch Einbinden der Dependency *org.springframework.retry* kann das Pattern eingebunden werden. Mit der Annotation *@EnablyRetry* wird in der Main-Klasse des Services festgelegt, dass das Retry-Pattern verwendet werden soll.

Anschließend wird durch die Annotation *@Retryable* festgelegt welche Methode erneut und mit welchen Optionen erneut ausgeführt werden soll.

Die durch *@Recover* markierte Methode legt zusätzlich fest, was passieren soll, wenn wieder keine Antwort erhalten wird.

Im Folgenden sieht man die Implementierung in unserem Projekt:

//BEISPIELCODE EINFÜGEN

### Testfall

Um das Retry-Pattern zu testen, wurde statt der festen Verzögerung von 10 Sekunden, eine Zufallszahl zwischen 10 Millisekunden und 5 Sekunden erzeugt, sodass Schwankungen im Netzwerk simuliert werden. Bei den darauffolgenden Testanfragen konnte man teilweise erkennen, dass erst beim zweiten Versuch eine Antwort von dem Service erhalten wurde.

Allerdings ist auch wie beim Timeout-Pattern eine Maßnahme allein nicht ausreichend, um die Anwendung stabil laufen zu lassen. Hier ist auch eine Kombination aus mehreren Resilience-Pattern unabdingbar.

## Circuit Breaker

Das Resilience-Pattern Circuit Breaker ist eine Art Hybrid aus Timeout- und Retry-Pattern. Die Idee hinter dem Circuit Breaker ist es, zu erkennen, wann ein Service nicht erreichbar ist und ihm dann auch keine Requests mehr entgegenzuschicken. So wird nicht jedes Mal die Zeit des Timeouts abgewartet und es werden keine Requests geschickt, wenn der Service nicht erreichbar ist.

Der Circuit Breaker arbeitet grundlegend mit drei verschiedenen Zuständen:

* CLOSED: der Service funktioniert
* OPEN: der Service ist nicht erreichbar. Dieser Zustand wird erreicht, wenn die Fehlschlagsrate einen vorher festgelegten Wert erreicht.
* HALF\_OPEN: In diesem Zustand werden Requests an den Service geschickt. Wenn die Fehlschlagsrate über einem bestimmten Wert liegen, wir auf OPEN gewechselt, liegt sie darunter, wird auf den Zustand CLOSED gewechselt.

Für die Implementierung wird eine externe Bibliothek benötigt, das Spring-Boot nicht alle nötigen Funktionen beinhaltet. Es wird *Resilience4J* verwendet. Verschiedene Eigenschaften werden festgelegt, wie z.B. die Wartezeit, die in dem Zustand OPEN verbracht werden soll.

Durch Hinzufügen der Annotation *@CircuitBreaker* markiert man die betroffene Methode.

Die Implementierung in unserem Projekt sieht man im Folgenden:

//BEISPIELCODE EINFÜGEN

### Testfall

## Bulkhead

Das Bulkhead-Pattern wird verwendet, wenn Teile von der API eines Services von anderen Services abhängen, Teile aber auch eigenständig stehen können. Wird eine Anfrage an den Service geschickt, welcher von einem anderen Service abhängt, dieser allerdings gerade nicht erreichbar ist, so wird die Anfrage nicht ausgeführt. Obwohl es sein kann, dass dieser Teil nicht von einem anderen Service abhängt. So werden Anfragen abgelehnt, die eigentlich hätten beantwortet werden können.

Um das zu umgehen, werden die Anfragen kategorisiert, sodass entschieden werden kann, ob diese Anfrage eventuell doch ausgeführt werden kann.

Die Implementierung findet wieder über eine externe Bibliothek statt und sieht an unserem Beispiel wie folgt aus:

//BEISPIELCODE EINFÜGEN

### Testfall

## Rate Limiter

Das Rate Limiter-Pattern wird eingesetzt, wenn ein zeitintensiver Request an einen externen Webservice geschickt werden muss. Wird diese Anfrage mehrfach in der Sekunde aufgerufen, es allerdings schon reicht, wenn man Antworten bekommt, die bis zu *x* Sekunden alt sind, so kann die gecachete Antwort einer älteren Anfrage verwendet werden.

So wird nur alle *x* Sekunden eine Anfrage geschickt, aller anderen Requests innerhalb dieser Zeit, erhalten die gecachete Antwort einer vorherigen Anfrage. So wird die Last auf den externen Server verringert und die Antwortzeit verkürzt.

Zur Implementierung wird wieder die externe Bibliothek verwendet und einige Parameter können eingestellt werden, wie beispielsweise die Wartezeit zwischen den Requests.

In unserem System sieht die Implementierung wie folgt aus:

//BEISPIELCODE EINFÜGEN

### Testfall

# Weitere Fragestellungen

Neben der Dokumentation der Implementierung des Shopsystems waren auch weitere Fragestellungen zu beantworten. Diese werden im Folgenden nacheinander beantwortet.

## Basic Authentication

## Kubernetes Network Policies

## Service-Mesh