

Cloud Grundlagen und Programmierung

Wahlmodul Sommersemester 2020

Professor: Prof. Dr. Raphael Herding

Abschlussdokumentation

Christian Szablewski

Marek Linnenbaum

Inhalt

Frontend.....	3
Produktübersicht	4
Warenkorb	4
Auftragsbestätigung	5
Backend	7
Architektur.....	7
Kommunikation/ Authentifizierung	7
Kubernetes.....	9
MiniKube/ Helm	9
API-Dokumentation	11
Resilience-Pattern.....	12
Künstliche Anfrageverzögerung der Services.....	12
Timeout	13
Testfall	13
Retry	13
Testfall	14
Circuit Breaker.....	15
Testfall	16
Bulkhead	16
Rate Limiter.....	17
Probleme.....	18
Weitere Fragestellungen	19
Basic Authentication.....	19
Kubernetes Network Policies	19
Service-Mesh.....	20

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.

Außerdem gibt es zu jedem Backend-Service eine TypeScript-Datei im Frontend, welche Funktionen des Backends über http-Befehle implementiert. Diese Funktionen werden dann in den einzelnen Oberflächen verwendet.

Ein Beispiel hierfür wäre die Methode *addItemToCart* in der *cart.service.ts*, welche über einen http-PUT Befehl die entsprechende API des Backend-Services anspricht:

```
63 addItemToCart(itemID: string): Observable<any>{
64   const headers = new HttpHeaders({
65     Authorization: 'Basic ' + btoa(this.cartApiUser + ':' + this.cartApiPW)
66   });
67   return this.http.put<any>(this.cartApi + this.currentCardID + '?newItem=' + itemID, {}, {headers});
68 }
```

Abbildung 1 - addItemToCart Beispiel

Produktübersicht

Die Produktübersicht zeigt alle verfügbaren Produkte. Diese werden aus dem Catalog-Service gezogen. Innerhalb der HTML-Datei für die Produktübersicht werden über Angular-Funktionen die Produkte in einer for-Schleife angezeigt. Hierfür wurde **ngFor* verwendet. Mit diesem Befehl lassen sich dynamisch Objekte in die Webseite einbinden. In der folgenden Abbildung sieht man die Produktübersicht.

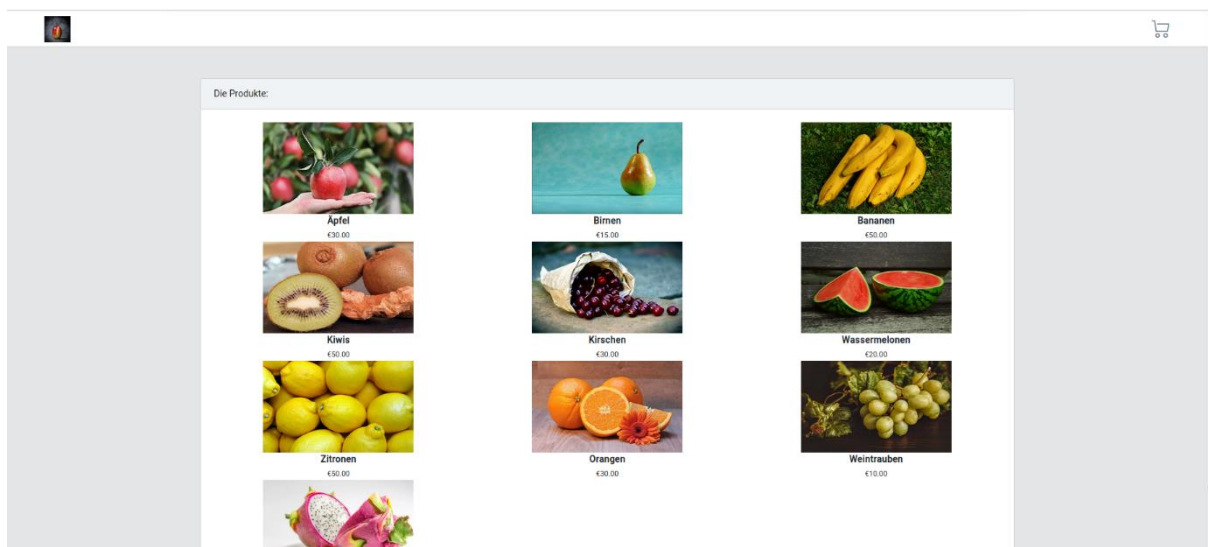


Abbildung 2 - Produktübersicht

Warenkorb

In dem Warenkorb werden die Produkte angezeigt, welche man zuvor in der Produktübersicht angeklickt hat. Außerdem werden die Kosten der Produkte, die Versandkosten, sowie die Gesamtkosten angezeigt. Diese Informationen werden aus dem Cart-Service im Backend gezogen. Genau wie in der Produktübersicht werden

die Produkte mithilfe des Befehls `*ngFor` angezeigt. Allerdings erst nachdem vorher mit `*ngIf` überprüft wurde, ob der Warenkorb überhaupt Elemente enthält. Außerdem wurde in die `ngOnInit()` des Warenkorbs implementiert, dass dieser mit Beispielswerten vorgefüllt werden soll. Dies sieht man in folgender Abbildung.

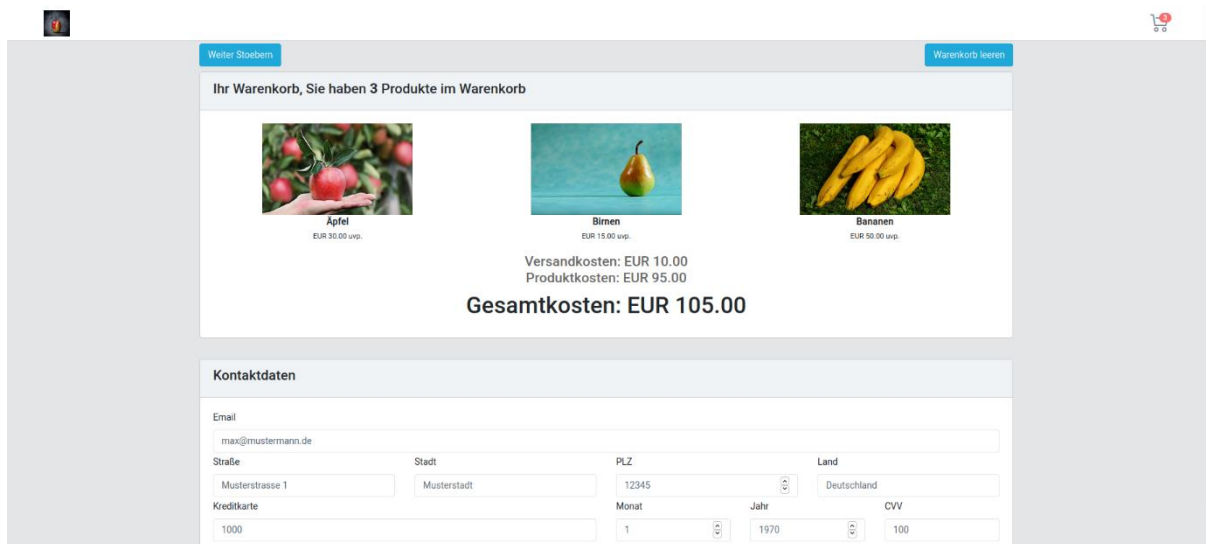


Abbildung 3 - Warenkorb

Auftragsbestätigung

In der Auftragsbestätigung werden die Informationen der Bestellung zusammengefasst und dargestellt. Diese Informationen kommen aus dem Shipping-Service im Backend. Beispielsweise wird die automatisch generierte Tracking Nummer.

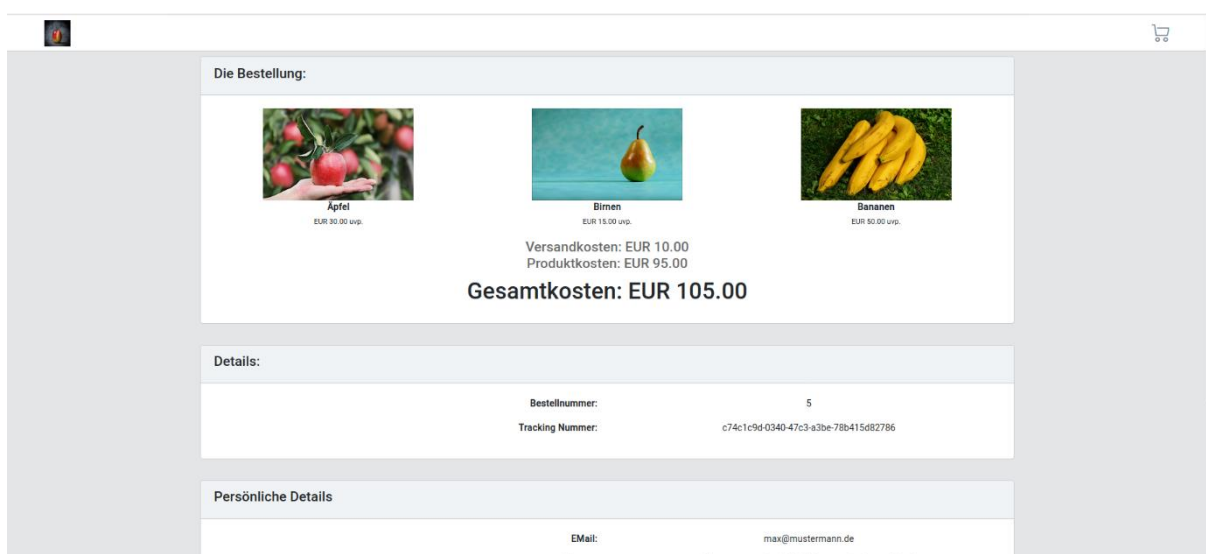




Abbildung 4 - Auftragsbestätigung 1



Produktkosten: EUR 95.00

Gesamtkosten: EUR 105.00

Details:

Bestellnummer:5

Tracking Nummer:c74c1c9d-0340-47c3-a3be-78b415d82786

Persönliche Details

EMail:max@mustermann.de

Adresse:Musterstrasse 1, 12345 - Musterstadt, Deutschland

Bezahl Details

Kreditkarte:1000

Läuft ab:1 / 1970

CVV:100

Abbildung 5 - Auftragsbestätigung 2

Backend

Die Backend-Services wurden jeweils mit Java Spring Boot erstellt. Das Framework ist ebenfalls bereits aus den Labs bekannt. Das Verwalten der Abhängigkeiten wurde dabei mithilfe von Maven umgesetzt. Es wurden Funktionen des Spring Boot Maven Plugins genutzt, um das Erstellen der Images zu vereinfachen.

Architektur

Die einzelnen Services besitzen neben einem Controller eine grundlegende Konfiguration, eine CORS-Erweiterung, eine Swagger 2-Konfiguration und eine Konfiguration der Authentifizierung.

Im Controller befinden sich dabei alle Routen, die als Endpunkte des Service definiert sind und die alle mithilfe des Swagger 2 Frameworks dokumentiert werden. Diese Dokumentation ist für jeden Service über die swagger-ui einsehbar.

In jedem Controller gibt einen Endpunkt, der genutzt wird, um die liveness-Probe des Service durchzuführen. Dieser Endpunkt gibt nur einen Status 200 OK zurück und wird zur Vereinfachung der liveness-Probe nicht authentifiziert.

Die Services, die eine Datenbank Verbindung besitzen, nutzen ebenfalls das Spring Boot JPA Framework, um diese Verbindung aufbauen zu können und automatisch Repositories nutzen zu können, um Daten in der Datenbank zu speichern. Dadurch eine Definition der Datenbankstruktur nicht nötig und es müssen keine Queries geschrieben werden, um auf die Daten zu zugreifen oder sie zu verändern.

Kommunikation/ Authentifizierung

Zur Kommunikation zwischen den Services haben wir Basic Authentication genutzt. Realisiert wird die Kommunikation durch http-Requests an die REST Endpunkte der einzelnen Services. Um diese Endpunkte anzusprechen muss der Service die IPs der anderen Services kennen. Das wird mithilfe des Cluster Addons „kube-dns“ erreicht. Der DNS ermöglicht es innerhalb des Clusters einen Service über den in Kubernetes gesetzten Namen zu erreichen, da dieser Name durch den DNS automatisch in die entsprechende IP übersetzt wird. Um den DNS-Namen, der Anwendung zur Verfügung zu stellen wurden Umgebungsvariablen genutzt. Dies ermöglicht zum einen eine Konfiguration der Service Kommunikation, ohne dass ein

Service neu erstellt werden muss. Zum anderen vereinfacht es die Entwicklung und das Testen, da die Umgebungsvariable in der Entwicklungsumgebung beispielsweise auf die lokale Adresse des jeweiligen Service gesetzt werden kann. Für die Basic Authentication wird nun jedem Service ein Username und Passwort über Umgebungsvariablen bereitgestellt. Diese werden ebenfalls über Kubernetes definiert und werden als Secret gespeichert. Den aufrufenden Services werden so ebenfalls die Authentication Daten der Services übergeben, welche sie aufrufen wollen.

Kubernetes

Es gibt für jeden Service, sowie das Frontend, ein Deployment in dem das genutzte Image, die liveness probe und die Umgebungsvariablen definiert werden.

Außerdem wird für jedes Deployment ein Kubernetes Service erstellt, der die Ports definiert, über welche man die Anwendungen erreichen kann. Dabei bekommt jeder Service nur einen Port zugewiesen, da nur einen Port pro Deployment genutzt wird, der einen Namen erhält. Über diesen Namen kann der Port von der liveness probe und dem Ingress genutzt werden.

Der Ingress bekommt für jede Anwendung einen Pfad, der dem entsprechenden Service und seinem Port zugewiesen wird. Da der bei Minikube als Addon verfügbare Ingress Controller genutzt wird, wird geben per Annotation angegeben, dass der Ingress keinen ssl-redirect durchführen soll. Andernfalls könnte der Ingress nicht korrekt ausgeführt werden, da keine zertifizierte Domain genutzt wird.

Zuletzt wurde noch ein Kubernetes Secret erstellt, um die Services korrekt absichern zu können. Dafür wurde jeweils Benutzername und Passwort mit einem entsprechenden Key als *stringData* im Secret gespeichert.

MiniKube/ Helm

Es wurde Helm genutzt um damit auf ein Minikube Cluster zu deployen. Bei dem Minikube Cluster wurde zuvor das *Ingress* Addon aktiviert, das einen Ingress Controller erstellt.

Um die Chart zu erstellen wurde jeweils ein Template für jeden zuvor beschriebenen Aspekt der Kubernetes Einstellungen erstellt. Demnach gibt es ein Template für die Deployments, die Services, den Ingress und das Secret.

In der *values.yaml*, werden Defaultwerte für die Templates gesetzt. Dabei wurden die Templates so erstellt, dass in der *values.yaml* ein Set aus Definitionen für Anwendungen genutzt wird. Das heißt, um eine Anwendung hinzuzufügen wird nur ein weiterer Eintrag in der dieser benötigt. Die Templates gehen dann alle

Definitionen durch und erstellen den Service, das Deployment und fügen einen Pfad zum Ingress hinzu.

```
deployments:
  catalog:
    port: 8080
    targetPort: 8080
    portName: catalog-http
    path: "/catalog"
    replicaCount: 1
    name: catalog
    image: "cszab/gca-catalog:0.0.9"
    livenessUrl: "catalog/api/health"
    env:
      - name: DOCKER_USER
        valueFrom:
          secretKeyRef:
            name: gca-secrets
            key: catalog_user
      - name: DOCKER_PW
        valueFrom:
          secretKeyRef:
            name: gca-secrets
            key: catalog pw
```

Abbildung 6 - Definition einer Anwendung

Um die Datenbank zu deployen wurde eine bereits existierende Chart genutzt, die als Subchart eingebunden wurde. Diese Subchart wurde als Dependency in der *Chart.yaml* definiert.

Um die Anwendung nun zu deployen werden die Helm Kommandos „install“ für ein erstes Rollout und „upgrade“ für jegliche weiteren Revisionen genutzt.

API-Dokumentation

Die API-Dokumentation wurde, wie zuvor erklärt mithilfe von Swagger 2 umgesetzt. Dabei wird jeder Endpunkt der Services über die `@ApiOperation` Annotation dokumentiert.

The image shows a Swagger 2 API documentation interface for a GET request. At the top, a blue button labeled 'GET' is next to the endpoint `/api/{id}` and the description 'Get single item from catalog'. Below this is a 'Parameters' section with a 'Try it out' button. The parameters table has two columns: 'Name' and 'Description'. It lists a required integer parameter 'id' with the type `integer($int64)` and the location `(path)`. The 'Responses' section has a 'Response content type' dropdown set to `*/*`. It lists four response codes: 200 (OK), 401 (Unauthorized), 403 (Forbidden), and 404 (Not Found). An 'Example Value' is shown for the 200 response, displaying a JSON object with fields 'id', 'imageUrl', 'name', and 'price'.

Name	Description
id * required <code>integer(\$int64)</code> (path)	id

Code	Description
200	OK Example Value Model <pre>{ "id": 0, "imageUrl": "string", "name": "string", "price": 0 }</pre>
401	Unauthorized
403	Forbidden
404	Not Found

Abbildung 7 - Swagger 2 Dokumentation für GET Request auf den Catalog Service

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 vorimplementiert 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:

```
public Collection<ShopItem> GetCatalog() {  
    try {  
        Thread.sleep(DOCKER_TIMEOUT);  
    } catch (Exception e) {}  
    return catalog;  
}
```

Abbildung 8 - Künstliche Anfrage Verzögerung

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:

```
@Bean
public RestTemplate getRestTemplate(RestTemplateBuilder restTemplateBuilder){
    return restTemplateBuilder
        .setConnectTimeout(Duration.ofSeconds(3))
        .setReadTimeout(Duration.ofSeconds(3))
        .build();
}
```

Abbildung 9 - Timeout-Pattern Implementierung

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 *@EnableRetry* 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:

```
236 @Retryable(  
237     value = {SocketTimeoutException.class},  
238     maxAttempts = 2,  
239     backoff = @Backoff(delay = 1000)  
240 )  
241 private Shipment getShipment(Long cartId) {  
242     HttpHeaders headers = new HttpHeaders();  
243     headers.setBasicAuth(new String(Base64.getEncoder().encode((DOCKER_SHIPPING_USER + ":" + DOCKER_SHIPPING_PW).getBytes())));  
244     HttpEntity<String> entity = new HttpEntity<String>(headers);  
245     return restTemplate.exchange("http://" + shipping_url + ":8082/shipping/api/?cartId=" + cartId, HttpMethod.POST, entity, Shipment.class).getBody();  
246 }  
247  
248 @Recover  
249 private Shipment getRecoveryShipment(SocketTimeoutException e) {  
250     return new Shipment();  
251 }
```

Abbildung 10 - Retry Resilience-Pattern Implementierung

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, wird 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:

```

13 resilience4j.circuitbreaker:
14   configs:
15     default:
16       slidingWindowType: COUNT_BASED
17       slidingWindowSize: 100
18       permittedNumberOfCallsInHalfOpenState: 10
19       waitDurationInOpenState: 10
20       failureRateThreshold: 60
21       registerHealthIndicator: true
22   instances:
23     catalog:
24       baseConfig: default

```

Abbildung 111 – Circuit Breaker Beschreibung in application.yaml

```

203 @CircuitBreaker(name = "catalog", fallbackMethod = "getRecoveryCatalog")
204 private Item[] getCatalog() {
205     Item[] items;
206     HttpHeaders headers = new HttpHeaders();
207     headers.setBasicAuth(new String(Base64.getEncoder().encode((DOCKER_CATALOG_USER + ":" + DOCKER_CATALOG_PW).getBytes())));
208     HttpEntity<String> entity = new HttpEntity<String>(headers);
209     ResponseEntity<Item[]> response = restTemplate.exchange("http://" + catalog_Url + ":8080/catalog/api/", HttpMethod.GET, entity, Item[].class);
210     items = response.getBody();
211     if (items == null || items.length == 0) {
212         items = new Item[0];
213     }
214     return items;
215 }
216
217 @Recover
218 private Item[] getRecoveryCatalog(SocketTimeoutException e) {
219     return new Item[0];
220 }

```

Abbildung 12 - Circuit Breaker Implementierung

Testfall

Um die Resilience-Pattern Circuit Breaker, Bulkhead und Rate Limiter zu testen, haben wir die Open Source Software JMeter verwendet. Mit diesem Tool konnten wir mehrere simultane Nutzeranfragen simulieren.

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:


```

25 resilience4j.bulkhead:
26   instances:
27     cart:
28       maxConcurrentCalls: 10
29       maxWaitDuration: 10ms

```

Abbildung 13 - Beschreibung Bulkhead in application.yaml

```

187 @Bulkhead(name = "cart", fallbackMethod = "getRecoveryCart", type = Bulkhead.Type.SEMAPHORE)
188 private ShopCart getCart(Long id) {
189     ShopCart cart;
190     HttpHeaders headers = new HttpHeaders();
191     headers.setBasicAuth(new String(Base64.getEncoder().encode((DOCKER_CART_USER + ":" + DOCKER_CART_PW).getBytes())));
192     HttpEntity<String> entity = new HttpEntity<String>(headers);
193     cart = restTemplate.exchange("http://" + cart_Url + ":8081/cart/api/{id}", HttpMethod.GET, entity, ShopCart.class, id).getBody();
194     if (cart == null) cart = new ShopCart();
195     return cart;
196 }
197
198 @Recover
199 private ShopCart getRecoveryCart(SocketTimeoutException e) {
200     return new ShopCart();
201 }

```

Abbildung 12 - Bulkhead Implementierung

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 gecachte Antwort einer älteren Anfrage verwendet werden.

So wird nur alle x Sekunden eine Anfrage geschickt, aller anderen Requests innerhalb dieser Zeit, erhalten die gecachte 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:

```

30 resilience4j.ratelimiter:
31   instances:
32     shippingCost:
33       limitForPeriod: 1
34       limitRefreshPeriod: 5s
35       timeoutDuration: 0s

```

Abbildung 15- Beschreibung Rate Limiter in application.yaml

```

222 @RateLimiter(name="stockService", fallbackMethod = "getRecoveryCost")
223 private Long getCost(Long cartPrice) {
224     HttpHeaders headers = new HttpHeaders();
225     headers.setBasicAuth(new String(Base64.getEncoder().encode((DOCKER_SHIPPING_USER + ":" + DOCKER_SHIPPING_PW).getBytes())));
226     HttpEntity<String> entity = new HttpEntity<String>(headers);
227     this.cachedCost = restTemplate.exchange("http://" + shipping_Url + ":8082/shipping/api/cost/?cost={cost}", HttpMethod.GET, entity, Long.class, cartPrice).getBody();
228     return this.cachedCost;
229 }
230
231 @Recover
232 private Long getRecoveryCost(SocketTimeoutException e) {
233     return this.cachedCost;
234 }

```

Abbildung 16 - Rate Limiter Implementierung

Probleme

Wir hatten Probleme dabei, die Testergebnisse von JMeter festzuhalten, da uns die Darstellung der Ergebnisse leider nicht gelungen ist. Die Anwendung hat immer wieder Fehler geworfen, welche wir nicht beheben konnten.

Weitere Fragestellungen

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

Basic Authentication

Generell kann man sagen, dass nach der Authentifizierung mithilfe von Basic Authentication der Server weiß, dass der User berechtigt ist die Webseite zu nutzen, allerdings weiß der User nicht, ob der Server wirklich der ist, den er ansprechen wollte oder nicht. Die Authentifizierung funktioniert also nur in eine Richtung. Außerdem werden die Passwörter durchs Netz geschickt und Dritte könnten die Passwörter abfangen und so Zugriff auf den Server erhalten. Selbst bei verschlüsselten Informationen kann einfach diese Information an die Server weitergeschickt werden, um Zugriff zu erhalten.

In unserem Fall könnten Dritte auf die Zugangsdaten Zugriff erhalten (vorausgesetzt wird deployen die Anwendung ins Internet) und so Daten aus zum Beispiel dem Warenkorb manipulieren.

Kubernetes Network Policies

Durch Network Policies kann innerhalb eines Kubernetes-Clusters für einzelne Services angegeben werden, über welche Endpunkte Anfragen und Antworten angenommen werden dürfen. So kann man zum Beispiel Datenbank Anfragen nur von bestimmten Services gesendet werden können. So kann man sicherstellen, dass Daten auch nur von diesen bestimmten Anwendungen verändert werden können

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-deny-ingress
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: postgresql
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
      - podSelector:
          matchLabels:
            role: repo
  egress:
    - to:
      - podSelector:
          matchLabels:
            role: repo
```

Abbildung 13 - Network Policy um Datenbank zu schützen

In der vorherigen Abbildung war zu erkennen, wie eine Network Policy aussehen könnte, die die Datenbank vor ungewollten Zugriffen schützt. Durch den Pod Selector bei Ingress und Egress wird sichergestellt, dass nur Pods mit dem Label „role: repo“ auf die Datenbank mit dem entsprechenden Label zugreifen können. Weitere Möglichkeiten sind noch das Filtern nach IPs und namespaces.

Service-Mesh

Durch einen Service-Mesh lässt sich, direkt in die Anwendung integriert, steuern, wie Daten innerhalb der Anwendung geteilt werden. So kann dieser Austausch optimiert und ausfallssicherer gemacht werden.

In einem Service-Mesh werden Anfragen zwischen den Services über Proxies ausgeführt. Diese Proxies arbeiten neben den Services und werden auch Sidecars genannt. Die Kommunikation wird über diese Sidecars gesteuert und so muss die Kommunikation nicht mehr von den Entwicklern einzeln und für jeden Service implementiert werden.

Eine Implementierung eines Service Meshes ist „istio“. Istio stellt viele Funktionen wie Traffic Management, Sicherheitsfunktionen und Überwachungsfunktionen für das Mesh zur Verfügung. Man kann istio über eine Kommandozeilenanwendung namens `istioctl` auf einem Cluster installieren. Wenn man die Kommandozeilenanwendung installiert hat muss man nur `istioctl install` ausführen und schon hat man eine sogenannte default Konfiguration installiert. Es gibt verschiedene Konfigurationen, deren Umfang verschiedene Ausmaße annimmt.