

Resilience4j bei Microservices – Gegenüberstellung Hystrix anhand einer Beispielanwendung

Studienarbeit

im Studiengang
Softwaretechnik und Medieninformatik

vorgelegt von

Heiko Fischer

Matr.-Nr.: 751209

Patrick Auer

Matr.-Nr.: 755350

am 15.01.2020
an der Hochschule Esslingen

Prüfer:

Matthias Häussler

Kurzfassung

Inhalt der hier vorgestellten Studienarbeit ist eine Gegenüberstellung zwischen den Open Source Frameworks Resilience4j und Hystrix anhand einer Beispielanwendung. Es handelt sich hierbei um Bestandteile der sogenannten Spring Cloud Netflix, die es ermöglicht moderne Microservice-Architekturen aufzusetzen. Hystrix und Resilience4j sind dabei die Implementierungen des sogenannten Circuit Breaker Patterns, wiederum ist Resilience4j die Ablösung von Hystrix. Zu diesem Zweck wurden zwei unterschiedliche Backendanwendungen implementiert, eine für den Umgang des Circuit Breakers mit Hystrix, die andere mit Resilience4j.

Mit Hilfe von Circuit Breaker wird dafür gesorgt, dass Fehler auf unteren Ebenen des Callstacks nicht nach oben kaskadieren. So wird beispielsweise nach bestimmter Anzahl von Fehlern in einer bestimmten Zeit im Sekundenbereich die Verbindung unterbrochen. Dabei gibt es einen vordefinierten Default Wert, welcher nach eigenen Anforderungen angepasst werden kann und auf eine Fallback-Implementierung umleitet.

Für die Gegenüberstellung wurde eine Implementierung eines solchen widerstandsfähigen Systems durchgeführt. Die Backendanwendung wurde in Java und Maven umgesetzt. Eine Frontendanwendung wurde mit React JS implementiert, um die Funktionalität der beiden Backendanwendungen zu prüfen. Mittels einer API (REST-Schnittstelle) werden die Daten übermittelt. Grundsätzlich wird hierbei der Nutzen von Resilience4j und Hystrix aufgezeigt, welche dabei helfen ein System performanter zu gestalten.

Inhaltsverzeichnis

Kurzfassung	2
Inhaltsverzeichnis	3
1 Wahl der Programmiersprache	4
2 Bibliothek Resilience4j.....	5
2.1 Umsetzung.....	6
2.2 Circuit Breaker	6
2.2.1 Die Problematik	6
2.2.2 Lösungsansätze durch Circuit Breaker	7
2.2.3 Funktionsweise von Circuit Breaker	7
2.2.4 Code Beispiel aus Anwendung	8
2.2.5 Weitere Funktionen von Resilience4j.....	8
2.3 Die REST-API	9
3 Bibliothek Hystrix	10
4 Vergleich Hystrix mit Resilience4j	11
4.1 API Beispiele	13
5 Der React Client.....	15
5.1 Vorstellung des Frameworks	15
5.2 Erstellen eines React Projekts	16
5.3 Aufbau der Frontend Anwendung.....	17
Literaturverzeichnis	18
Abbildungsverzeichnis.....	19
Quellcode	20
Ehrenwörtliche Erklärung.....	21
Ehrenwörtliche Erklärung.....	22

1 Wahl der Programmiersprache

Für die Umsetzung der Studienarbeit wurde die Programmiersprache Java für die Backendentwicklungen von Hystrix und Resilience4J gewählt, sowie für das Frontend die Sprachen JavaScript, HTML und CSS mit der Bibliothek React JS.

Auch andere Programmiersprachen oder Frameworks sind an dieser Stelle möglich. Aufgrund der Erfahrungen mit Java im Studium und die damit erlernten Vor- und Nachteile dieser Sprache, war dies die begründete Wahl zur Programmiersprache, sowie React JS aufgrund von Umsetzungen anderer Studienprojekten. Des Weiteren wurde Java gewählt, aufgrund der Verbindung von einzelnen Systemen über Schnittstellen. Weiter spielte auch die von Java angebotene Flexibilität eine große Rolle. Da wir mit Windows sowie Apple Geräten arbeiten wurde Wert auf Plattform unabhängiges Programmieren gelegt. Um eine gute Grundlage für das Testen der Circuit Breaker zu ermöglichen, empfiehlt sich Java, da hier Multithreading möglich ist.

Java liefert dabei eine etwas schlechtere Performance, als das beispielsweise bei C# der Fall ist, da Java mit einem Interpreter arbeiten muss. Dies spielt in unserem Fall allerdings keine große Rolle, da die Beispielanwendung sehr klein ist und nur die nötigsten Funktionalitäten zur Darstellung genutzt werden.

2 Bibliothek Resilience4j

Resilience4j dient dazu, bei der Implementierung widerstandsfähiger Systeme anhand von Fernkommunikation die Fehlertoleranz zu verwalten. Die Bibliothek wurde auf Basis von Hystrix vorangetrieben und stammt aus dem Hause Netflix. Resilience4j bietet allerdings eine bessere API und eine Reihe weiterer Funktionen wie Rate Limiter (zu häufige Anfragen blockieren). Resilience4j ist im Bereich von Microservices daher kaum noch wegzudenken (vgl. codeflow.site, 2019).

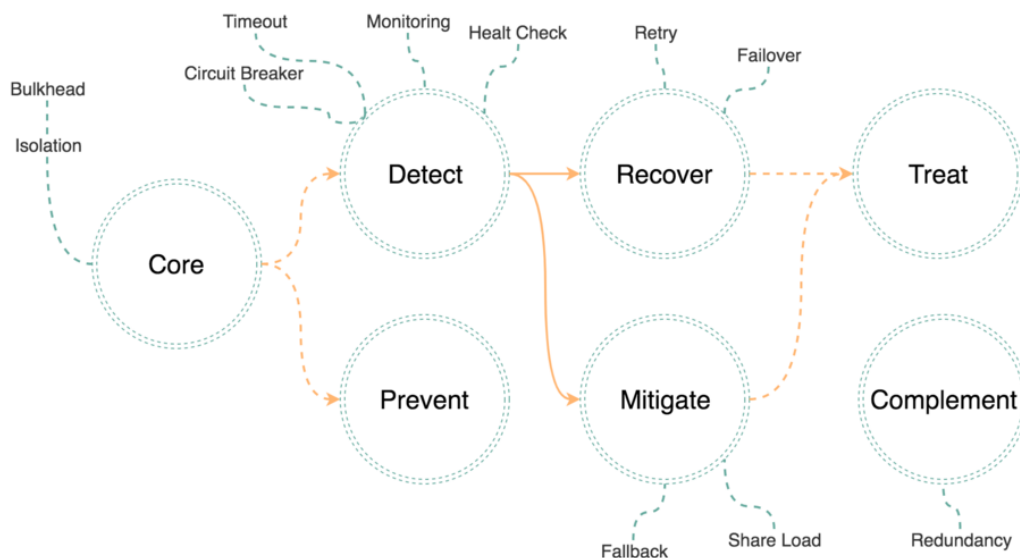


Abbildung 1: Resilience Pattern by Benjamin Wilms

2.1 Umsetzung

Für die Umsetzung wurde mit Hilfe des Maven-Setups die Zielmodule in eine pom.xml Datei hinzugefügt. Sowie der bereits beschriebene Circuit Breaker. Im aufgeführten Beispiel wird das Circuit Breaker Modul eingebunden. Eine Übersicht aller verfügbaren Module, sowie deren Versionen, sind auf den Seiten von Maven Central zu finden.

```
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-circuitbreaker</artifactId>
  <version>1.0.0</version>
</dependency>
```

Code 1: Dependencies Resilience4j

2.2 Circuit Breaker

2.2.1 Die Problematik

Innerhalb eines Systems, beziehungsweise einer Software kann es immer zu Fehlern kommen. Neben den „normalen“ Fehlern, die mit Hilfe von Fehlerbehandlungen (Exception Handling), gibt es noch zahlreiche andere Fehler, die auftreten können.

Hierzu zählen beispielsweise Fehler durch:

- Schlechte Netzwerkverbindung -> Connection Timeout
- Verlust der Netzwerkverbindung -> Connection Timeout
- Übertragungsfehler
- Falsche Datenstände durch Commits
- Nicht verfügbarer Ressourcen
- Defekte Hardware

2.2.2 Lösungsansätze durch Circuit Breaker

Mit Circuit Breaker kann man oben genannte Fehler behandeln. So kann es beispielsweise sein, dass ein gewünschter Service für längere Zeit ausfällt. Hier würde es keinen Sinn machen, immer wieder einen neuen Verbindungsaufbau zu starten. Mit Hilfe von Circuit Breaker kann der erneute Verbindungsaufbau gesteuert und somit untersagt werden.

Ein weiteres Beispiel ist der Umgang mit Services, die aufgrund hoher Anfragen überlastet sind. Hier kann es dazu kommen, dass das Warten auf den Service kritische Systemressourcen blockiert und somit das gesamte System zum Abstürzen bringen kann. Mit Circuit Breaker kann eine Regelung getroffen werden, die die Ressourcen in so einem Fall dann rechtzeitig wieder frei gibt. Daher werden sich konkurrierende Prozesse gemanagt. So nutzen dann nur noch Services die Ressourcen, die auch tatsächlich verfügbar sind. Services, die nicht verfügbar sind, werden automatisch geblockt und die notwendigen Ressourcen wieder dem System zur Verfügung gestellt.

2.2.3 Funktionsweise von Circuit Breaker

Grundsätzlich dienen Circuit Breaker dazu, dass man die Anfragen von Usern reduziert. Wenn ein Client beispielsweise eine Ressource anfordert, die zum Zeitpunkt nicht zur Verfügung steht, so fragt der Client ständig erneut den Server nach dieser Ressource an. Dies erzeugt erheblichen Traffic und kann im schlimmsten Fall zum Serverausfall führen. Das ist kein großes Problem, so fern wenige User vorliegen. Allerdings kommt es vor Allem bei Streaming Portalen dazu, dass zu einem Zeitpunkt sehr viele User ein und dieselbe Ressource anfordern. Hier kommen die sogenannten Circuit Breaker zum Einsatz. Diese unterbrechen den „normalen“ Kreislauf. Dies kann zum Beispiel ein Timeout sein, der dem User erst wieder nach einer gewissen Zeit erlaubt, die gewünschte Ressource anzufragen.

2.2.4 Code Beispiel aus Anwendung

```
public ResilienceShoppingService() {

    BulkheadConfig config = BulkheadConfig.custom()
        .maxConcurrentCalls(150)
        .maxWaitDuration(Duration.ofMillis(500))
        .build();
    BulkheadRegistry registry = BulkheadRegistry.of(config);
    bulkhead = Bulkhead.of("apiCall", BulkheadConfig.ofDefaults());
    /** chainedCallable = Bulkhead.decorateFunction(bulkhead,
    this::restrictedCall);**/

    circuitBreaker = CircuitBreaker.of("apiCall", CircuitBreakerConfig.custom()
        .failureRateThreshold(50)
        .waitDurationInOpenState(Duration.ofMillis(1000))
        .ringBufferSizeInHalfOpenState(2)
        .ringBufferSizeInClosedState(2)
        .build());
    chainedCallable = CircuitBreaker.decorateFunction(circuitBreaker,
    this::restrictedCall);

    ratelimiter = RateLimiter.of("apiCall", RateLimiterConfig.custom()
        .limitRefreshPeriod(Duration.ofSeconds(1))
        .limitForPeriod(10)
        .timeoutDuration(Duration.ofSeconds(2))
        .build());
    /**chainedCallable = RateLimiter.decorateFunction(ratelimiter,
    this::restrictedCall);**/
}
```

Code 2: Circuit Breaker Resilience4j

2.2.5 Weitere Funktionen von Resilience4j

Die hier aufgeführten Funktionen werden im Einzelnen kurz erläutert und haben im Projekt teilweise Verwendung gefunden.

- Ratelimiter
- Bulkhead
- Retry
- Cache
- Timelimiter

Ein weiterer Anwendungsbereich bei Resilience4j ist der sogenannte Ratelimiter, im deutschen als Ratenbegrenzer bezeichnet. Diese Funktion ermöglicht, dass Zugriffe auf bestimmte Dienste eingeschränkt werden. Der Ratelimiter ist sehr ähnlich zum Circuit Breaker aufgebaut, wie man es im vorangegangenen Beispiel bereits erkennen konnte. Es können dabei der Zeitraum der Limitaktualisierungen, die Berechtigungsgrenze für

den Aktualisierungszeitraum und die Standardwartezeit für Genehmigungen konfiguriert werden.

Des Weiteren können mit dem Bulkhead die Anzahl gleichzeitiger Aufrufe zu einem bestimmten Dienst begrenzt werden. Wird die Abhängigkeit von Bulkhead benötigt und implementiert, können die maximale Anzahl paralleler Ausführungen angegeben werden, sowie die maximale Zeitdauer, die ein Thread bei der Ausführung einzutreten wartet.

Die Funktion Retry, erlaubt es fehlgeschlagene Aufrufe mittels der API automatisch zu wiederholen. Um damit den vollen Umfang nutzen zu können, ist es erlaubt, die Anzahl der Versuche, die Wartezeit von neuen Versuchen, sowie ein Warteintervall nach einem Fehler zu konfigurieren.

Mit dem Cache, dem sogenannten Zwischenspeicher, wird eine etwas andere Implementierung durchgeführt, als es bisher bei den vorangegangenen Modulen der Fall ist. Es wird eine Zwischenspeicherung durch die Implementierung vom sogenannten JSR-107 Cache durchgeführt. Resilience4j bietet hier einfach die Möglichkeit dieses anzuwenden. Die API unterstützt hier nur die Typen Supplier und Callable.

Wird ein Timelimiter hinzugefügt, bedeutet dies, dass es ermöglicht wird, die Zeit einzuschränken, die ein Remote Service benötigt. Ein Timelimiter kann ein konfiguriertes Timeout besitzen. Resilience4j kann dann prüfen, ob ein erwartetes Timeout aufgerufen wird. Des Weiteren ist der Timelimiter mit dem Circuit Breaker kombinierbar. (vgl. codeflow.site, 2019).

2.3 Die REST-API

Das Projekt beinhaltet eine implementierte REST API. Hierbei wurde ein Service implementiert, der dazu dient, REST-Requests durchzuführen. Dies dient der Kommunikation zwischen Client und Server. In der Regel werden hier GET-Requests ausgeführt die der Client an den Server sendet. Dieser antwortet dann auf den Request und sendet, sofern eine gültige Anfrage vorliegt, eine entsprechende Antwort an den Client. Andernfalls wird eine entsprechende Fehlermeldung gesendet.

3 Bibliothek Hystrix

Auch Hystrix ist eine Entwicklung von Netflix. Hystrix kann mit Java, Java EE und Spring eingesetzt werden. Dabei wird es als Dependency im Projekt eingebunden und hat immer die Möglichkeiten, Hystrix in seinem Code zu verwenden. Das Command Patterns spielt bei Hystrix eine wichtige Rolle, allerdings muss für jeden externen Service Aufruf ein eigener Fallback bereitgestellt werden. Mit Hystrix werden ebenfalls größere Mengen an Konfigurationen ausgeliefert, die es ermöglicht mit Default Parametern weitgehendste Entwicklungen durchzuführen (vgl. Nicolas Fränkel).

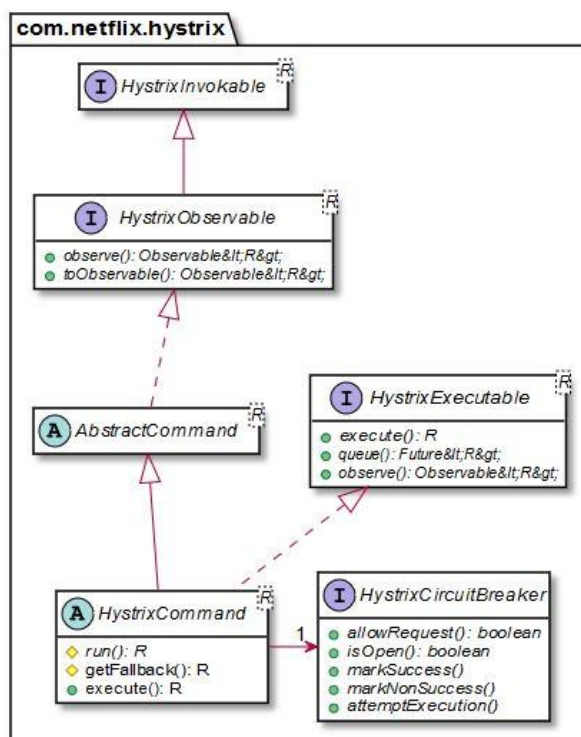


Abbildung 2: Klassendiagramm Hystrix's Model by Nicolas Fränkel

Features von Hystrix:

- Schutz vor Latenz und Ausfall von Abhängigkeiten, auf Third-Party Bibliotheken
- Verhindern von Kaskadenfehlern in einem komplexen verteilten System
- Schnelles versagen und schnelles erholen möglich
- Fallback und gracefully degrade, wenn möglich
- Echtzeit Überwachung, Alarmierung und Betriebskontrollen

4 Vergleich Hystrix mit Resilience4j

Im Folgenden wird eine Gegenüberstellung von Hystrix und Resilience4j dargestellt, um die wesentlichen Unterschiede darzustellen.

Hystrix calls an externe Systeme werden in einen Hystrix-Befehl eingeschlossen werden.	Resilience4j bietet Funktionen höherer Ordnung, um jede funktionale Schnittstelle oder jede Methodenreferenz zu verbessern. Es ist möglich, mehrere davon zu stapeln.
Hystrix unterstützt keine Java 8 Funktionen.	Jede dekorierte Funktion kann mithilfe von CompletableFuture oder RxJava synchron oder asynchron ausgeführt werden.
Hystrix führt im half-open State eine einzige Ausführung durch, um zu bestimmen, ob ein CircuitBreaker geschlossen werden soll.	Resilience4j ermöglicht die Ausführung einer konfigurierbaren Anzahl von Ausführungen und vergleicht das Ergebnis mit einem konfigurierbaren Schwellenwert, um zu bestimmen, ob ein Leistungsschalter geschlossen werden muss.
Hystrix sendet einen Stream von Events aus, die für Systembetreiber nützlich sind, um Metriken über Ausführungsergebnisse und Latenzzeiten zu überwachen.	Resilience4j bietet benutzerdefinierte Java Operatoren zum Ausführen eines CircuitBreaker, Bulkhead oder Ratelimiter.

Hystrix's Aufgabe ist es verteilte Systeme stabiler zu machen, was mit Circuit Breakern erreicht wird. Hystrix ist dabei Objektorientiert designed. Das heißt, dass jeder Aufruf in ein Hystrixcommand gewrapped werden muss.

Im Gegensatz dazu ist Resilience zwar nach dem Vorbild von Hystrix entstanden, allerdings handelt es sich hierbei um eine Standalone Bibliothek. Resilience ist auf den Prinzipien von funktionalem Programmieren aufgebaut. So nutzt man eine Komposition aus Decorators, je nachdem was alles behandelt werden, damit kann viel mehr konfiguriert und behandelt werden. Auch die Art der Behandlung kann dabei individualisiert werden. Decorators arbeiten häufig mit Lambda Funktionen und können asynchron aber auch synchron implementiert werden.

Letztendlich kann man also sagen, das mit Resilience mehr Flexibilität geschaffen wird. Es gibt wesentlich nicht mehr Konfigurationsmöglichkeiten sowie Funktionalität. Außerdem wurde die Bibliothek etwas aufgelockert indem sie nun wesentlich weniger Abhängigkeiten aufweist.

4.1 API Beispiele

Vergleich der APIs von Hystrix und Resilience4j werden im Folgenden näher aufgezeigt.

Um einen CircuitBreaker mit Hystrix zu erstellen, muss die Klasse HystrixCommand

erweitert und die Methoden run und getFallback implementiert werden. Im

Konstruktor wird eine Vielzahl von Parametern festgelegt, einschließlich Timeout und Threshold.

```
public class HystrixWrapper extends HystrixCommand<Integer> {  
    private final String param;  
    public HystrixWrapper(String param) {  
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey(HystrixWrapper.class.getName()))  
        .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()  
            .withExecutionTimeoutInMilliseconds(500)  
            .withCircuitBreakerErrorThresholdPercentage(20)));  
        this.param = param;  
    }  
    protected Integer run() throws Exception {  
        return UnstableApi.call(param);  
    }  
    protected Integer getFallback() {  
        System.out.println(this.circuitBreaker.isOpen() ? "OPEN" : "CLOSED");  
        return 1;  
    }  
}
```

Code 3: HystrixWrapper Code Ausschnitt

Im Vergleich dazu scheint der Resilience4jWrapper zunächst etwas komplizierter zu

sein. Es gibt keine zu erweiternde Kommandoklasse, aber es gibt Decorator-

Funktionen, die den Service-Aufruf abschließen. Um die gleiche Funktionalität wie im

Hystrix-Beispiel zu erhalten, müssen zwei Muster angewendet werden: CircuitBreaker

und TimeLimiter. Es gibt auch keinen eingebauten Fallback-Mechanismus, dieser muss selbst implementiert werden.

```
public class Resilience4jWrapper {  
  
    private final Callable<Integer> callable;  
    private final CircuitBreaker circuitBreaker;  
    private final TimeLimiter timeLimiter;  
  
    public Resilience4jWrapper(String param) {  
  
        ExecutorService executorService = Executors.newSingleThreadExecutor();  
        CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()  
            .failureRateThreshold(20)  
            .ringBufferSizeInClosedState(20)  
            .waitDurationInOpenState(Duration.ofSeconds(10)).build();  
        timeLimiter = TimeLimiter.of(Duration.ofMillis(500));  
        Callable<Integer> timeRestricted =  
            TimeLimiter.decorateFutureSupplier(timeLimiter, () ->  
executorService.submit(() -> UnstableApi.call(param)));  
        circuitBreaker = CircuitBreaker.of("test", circuitBreakerConfig);  
        callable = CircuitBreaker.decorateCallable(circuitBreaker,  
timeRestricted);  
    }  
  
    public Integer run() {  
        return Try.ofCallable(callable).getOrElse(-1);  
    }  
}
```

Code 4: Resilience4jWrapper Code Ausschnitt

5 Der React Client

5.1 Vorstellung des Frameworks

React ist eine deklarative, effiziente und flexible JavaScript-Bibliothek für den Aufbau von Benutzeroberflächen. Es ermöglicht das Zusammenstellen komplexer UIs aus kleinen und isolierten Codestücken, die "Komponenten" genannt werden. Facebook veröffentlichte 2013 React als OpenSource Projekt und seitdem beeinflusst es die gesamte JavaScript Frontend Landschaft. Facebook, Instagram, AirBnB und weitere andere bekannte Anwendungen nutzen React.

React arbeitet mit einem virtuellen DOM, was das System sehr performant macht. Es ist modular aufgebaut, was React zu einem leicht zu lesenden und übersichtlichen Framework macht. Außerdem fordert dies die Flexibilität. Außerdem ist React beliebt, da es wenige Vorschriften und Bedingungen gibt. Das macht es bei Entwicklern beliebt, da es einfach ist bestehenden Code zu integrieren, ohne ein bestimmtes Muster bzw. Format zu benutzen.

Zusammenfassend hat React folgende Bestandteile:

- Komponenten basierend
- Virtueller DOM
- Browserkompatibilität

5.2 Erstellen eines React Projekts

Um einen React Client zu starten wird NodeJS auf der lokalen Maschine benötigt. Über die Konsole der lokalen Maschine werden entsprechende Ausführungen durchgeführt:

- Als erstes installiert man global mit dem Node Package Manager die Anwendung mit `npm install -g create-react-app`
- Im nächsten Schritt wird der Generator im ausgewählten Verzeichnis ausgeführt: `create-react-app my-app`
- In dem neu erstellten Verzeichnis wird nun das Startscript ausgeführt: `npm start`

Im Folgenden wird die Beispielanwendung des Studienprojekts dargestellt. Anstelle von „my-app“ wurde das React Projekt als „list“ erstellt. Hierzu im nächsten Abschnitt mehr.

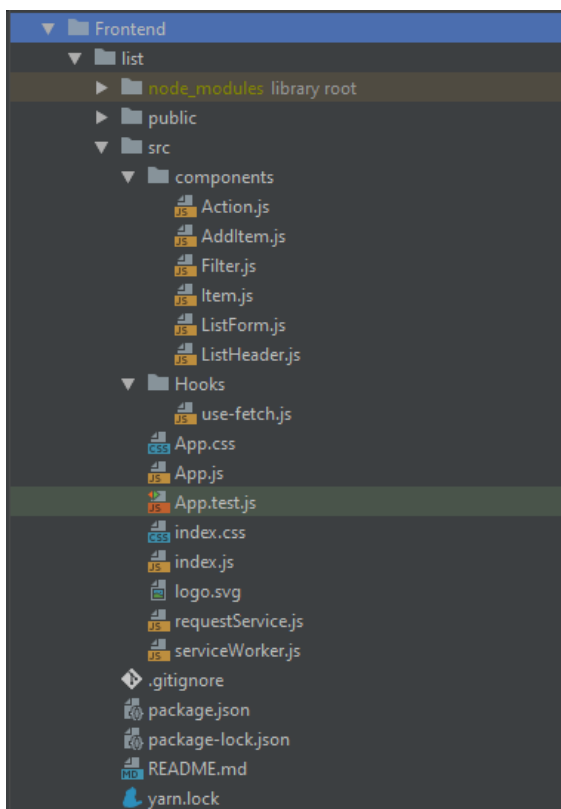


Abbildung 3: Projekt Struktur Frontend

5.3 Aufbau der Frontend Anwendung

Für die Umsetzung der Anwendung haben wir uns eine Einkaufsliste ausgedacht. In der Anwendung wird eine Liste angezeigt, hier können Items hinzugefügt und wieder gelöscht werden. Die Anwendung erhält dabei die Daten aus der Hystrix oder Resilience4J Backendanwendung, hierzu müssen entsprechend die API Calls im Frontend angepasst werden. Hierzu wurde eine Funktion umgesetzt, mit der die Daten für das Frontend gefetched werden. In der folgenden Darstellung wird eine Mock API eingebunden, alternativ ist der localhost auskommentiert dargestellt.

```
const itemApi = useFetch(  
  endpoint: "https://5cfabdcbf26e8c00146d0b0e.mockapi.io/tasks"  
/* const itemApi = useFetch(  
  "http://localhost:8080/" */  
);
```

Abbildung 4: Code Ausschnitt Frontend

Literaturverzeichnis

codeflow.site. 2019. [Online] 2019. [Zitat vom: 02. 11 2019.]

<https://www.codeflow.site/de/article/resilience4j>.

Fränkel, Nicolas. 2018. Exoscale. [Online] 2018. [Zitat vom: 15. 11 2019.]

<https://www.exoscale.com/syslog/istio-vs-hystrix-circuit-breaker/>.

Abbildungsverzeichnis

Abbildung 1: Resilience Pattern by Benjamin Wilms	5
Abbildung 2: Klassendiagramm Hystrix's Model by Nicolas Fränkel.....	10
Abbildung 3: Projekt Struktur Frontend	16
Abbildung 4: Code Ausschnitt Frontend	17

Quellcode

Code 1: Dependcies Resilience4j	6
Code 2: Cicruit Breaker Resilience4j	8
Code 3: HystrixWrapper Code Ausschnitt	13
Code 4: Resilience4jWrapper Code Ausschnitt.....	14

Ehrenwörtliche Erklärung

Name:	Fischer	Vorname:	Heiko
Matrikel-Nr.:	751209	Studiengang:	SWB

Hiermit versichere ich, Heiko Fischer, dass ich die vorliegenden Studienarbeit mit dem Titel „Resilience4j bei Microservices – Gegenüberstellung Hystrix anhand einer Beispielanwendung“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebene Literatur und Hilfsmittel verwendet habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ort, Datum

Unterschrift

Ehrenwörtliche Erklärung

Name:	Auer	Vorname:	Patrick
Matrikel-Nr.:	755350	Studiengang:	SWB

Hiermit versichere ich, Patrick Auer, dass ich die vorliegenden Studienarbeit mit dem Titel „Resilience4j bei Microservices – Gegenüberstellung Hystrix anhand einer Beispielanwendung“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebene Literatur und Hilfsmittel verwendet habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ort, Datum

Unterschrift