Entwicklung robuster Microservice-Architekturen mit Spring Boot und Spring Cloud

Domenic Cassisi
Fakultät Informatik
Hochschule Furtwangen
Furtwangen im Schwarzwald, Deutschland
domenic.cassisi@hs-furtwangen.de

***Zusammenfassung*—Microservices erfreuen sich immer größerer Beliebtheit, im Speziellen werden immer häufiger größere Systeme durch eine Microservice-Architektur umgesetzt. Diese Arbeit untersucht, wie sich solche Architekturen mithilfe des Spring Frameworks, insbesondere Spring Cloud und Spring Boot realisieren lassen. Die in der Arbeit vorgestellten Technologien wurden praktisch anhand eines Anwendungsbeispiels angewandt und evaluiert. Das Spring Framework ermöglicht einen sehr schnellen Aufbau von Microservice-Architekturen mit nur wenig Konfigurationsaufwand und bietet robuste Lösungen für charakteristische Probleme und Herausforderungen in verteilten Systemen.**

***Index Terms*—microservice, architecture, spring boot, spring cloud**

I. Einleitung

Monolithische Architekturen haben eine lange Tradition in der Softwareentwicklung. Dabei wird jegliche Funktionalität in einer einzigen großen Anwendung gekapselt. Überschaubare monolithische Anwendungen lassen sich verhältnismäßig leicht entwickeln, testen und in der Zielumgebung installieren [1]. So setzen beispielsweise die Entwickler von StackOverflow bis heute auf bestehende und etablierte monolithische Architekturen, womit Lasten von über 560 Millionen Seitenaufrufe im Monat bewältigt werden [2]. Ungeachtet dessen werden erfolgreiche Softwaresysteme mit der Zeit immer größer, komplexer und unüberschaubarer, was dazu führen kann, dass die Nachteile einer monolithischen Anwendung dessen Vorteile überwiegen. So sorgt beispielsweise eine überwältigende Codebasis für eine Verzögerung von Bugfixes und dem Einbau neuer Features, was den Entwicklungsprozess im Allgemeinen ausbremst [1].

In Microservice-Architekturen wird eine Anwendung aus einer Menge kleinerer Dienste (engl. Services) aufgebaut, die über gut definierte Schnittstellen miteinander kommunizieren [1]. Martin Fowler et al. beschreiben diesen Architekturstil als eine Ansammlung von Services, wobei eine wesentliche Besonderheit darin liegt, dass sich Services unabhängig voneinander ausliefern und skalieren lassen [3]. Microservices erfreuen sich gerade in der Industrie zunehmend großer Beliebtheit, nicht zuletzt aufgrund wichtiger Merkmale, wie besserer Wartbarkeit, Wiederverwendbarkeit, Robustheit, Skalierbarkeit und automatisiertem Ausliefern [1]. Viele große Internetfirmen, wie etwa Netflix, eBay und Amazon entwickelten ihre Anwendungen ursprünglich nach monolithischen Prinzipien, jedoch haben alle ihre Anwendungen mittlerweile in eine Microservice-Architektur überführt und in die Cloud migriert [1], [4]. Der Videostreaming-Dienst Netflix setzt über 600 verschiedene Microservices ein, wobei jeder Microservice üblicherweise mehrere Instanzen zur Laufzeit hat [5].

Jedoch sollten Microservice-Architekturen nicht als Patentlösung aller Architekturprobleme und Anwendungsfälle betrachtet werden, da diese Art von Architektur wieder eine ganze Reihe neuer Herausforderungen und Probleme mit sich bringt [1]. So bringen die fundamentalen Aufgaben, wie das Konfigurieren, Ausliefern, Skalieren und Überwachen eines jeden Microservices einen nicht zu unterschätzenden Aufwand mit sich. [5]. Das Überführen bestehender monolithischer Anwendungen in eine Microservice-Architektur ist ebenfalls ein nicht-trivialer Prozess, der bereits in zahlreichen anderen Arbeiten behandelt wurde [1], [6], [7].

Der Aufbau einer Microservice-Architektur kann von System zu System variieren. Jedoch müssen die meisten Systeme Lösungen für folgende Problemstellungen anbieten:

- Wie finden sich Microservices, um miteinander zu kommunizieren?
- Wie lässt sich die Last auf mehrere Instanzen eines Services verteilen?
- Wo wird die Konfiguration für einen Microservice abgelegt?
- Was passiert, wenn ein Service mal nicht erreicht werden kann?

Letztendlich müssen Lösungen für obige Problemstellungen von einer Softwarearchitektur angeboten werden. Bewährte Lösungen für bekannte und wiederkehrende Probleme finden sich häufig in Form einer Bibliothek oder eines Frameworks wieder. Das *Spring Framework* ist das meistgenutzte Framework im Java-Umfeld [8]. Mit dem Spring Framework werden eine ganze Reihe unterschiedlichster Probleme angegangen und entsprechende Lösungen angeboten, die jeweils in eigenen Unterprojekten entwickelt werden. In dieser Arbeit wird genauer auf zwei dieser Projekte eingegangen: Spring Boot und Spring Cloud. Folgend wird der relevante Bereich des Spring-Ökosystems, konkret Spring Boot und Spring Cloud tiefergehend betrachtet und anschließend schrittweise untersucht, wie sich Microservice-Architekturen mithilfe dieser beiden Projekte realisieren lassen.

II. Die Entwicklung von Microservices heute

Im Jahr 2020 führten Marek Gajda et al. in [9] eine Umfrage durch, um mehr über tatsächlich eingesetzte Microservice-Architekturen zu erfahren. Weltweit nahmen über 650 Softwareentwickler:innen an dieser Umfrage teil, davon die meisten in der Position eines CTO, Lead Developer oder Senior Developer. Microservice-Architekturen werden vor allem zur Verbesserung der Skalierbarkeit und Performance genutzt. Effizienteres Arbeiten sowie bessere Teamarbeit werden ebenfalls beobachtet. Schwierigkeiten und Probleme wurden überwiegend im Bereich der Wartung und des Debuggings genannt. Als eingesetzte Programmiersprache zum Bauen von Microservices wird JavaScript bzw. Typescript mit 65 % am häufigsten genannt, gefolgt von Java mit 26 %. Der Trend setzt sich fort, Microservice-Architekturen nicht nur für Enterprise-Anwendungen, sondern auch für mittelgroße Anwendungen zu bauen. Die Kommunikation zwischen den Microservices erfolgt mit fast 78 % über HTTP, wobei die Kommunikation über Events (44 %) ebenfalls eine wichtige Rolle spielt. Die Mehrheit der Befragten (63 %) setzt Message Broker, wie RabbitMQ (36 %) oder Kafka (24 %) ein. Im Bereich der Microservice-Entwicklung etabliert sich Continuous Integration als Standard, fast 87 % nutzen eine CI-Lösung, davon überwiegend GitLab CI oder Jenkins. Als bevorzugte Debugging-Lösungen werden Logs (86 %), Tracing (34 %) und Metriken (29 %) am häufigsten genannt. Für das Deployment der Microservices nutzen bereits die Hälfte der Befragten sog. Serverless-Technologien, wie AWS oder Azure, Tendenz steigend.

Container-Technologien ermöglichen eine isolierte Ausführungsumgebung und eignen sich gut für die Skalierbarkeit von Microservice-Architekturen, da sich Container bei Bedarf schnell instanziieren lassen. Jedoch sei angemerkt, dass dieser Trend, Software im Kontext von Container zu entwickeln, mehr als Zufälligkeit, als eine Konsequenz von Microservices anzusehen ist [10].

III. Spring Boot und Spring Cloud

Das Spring Framework gehört weltweit zu den beliebtesten Frameworks zur Entwicklung von Java-Anwendungen [8]. Das Spring Framework ist eine Sammlung unterschiedlichster Projekte. In dieser Arbeit werden zwei dieser Projekte genauer betrachtet: *Spring Boot* und *Spring Cloud*. Wie diese Projekte im Ökosystem von Spring einzuordnen sind, ist in Abbildung 1 dargestellt, welche an [11] angelehnt ist. Das *Spring Framework* bildet das Fundament einer jeden Spring-Anwendung. Das Spring Framework ist eine Ansammlung verschiedener Module, die teilweise aufeinander aufbauen. Für eine Spring-Applikation werden die benötigten Module ausgewählt und entsprechend für die Anwendung konfiguriert [12]. Spring Boot baut auf dem Spring Framework auf und soll das Arbeiten mit Spring erleichtern. Standardmäßig konfiguriert Spring Boot automatisch die Features vor, die im Classpath enthalten sind. Spring Cloud baut auf Spring Boot auf und besteht aus einer Menge von Tools und Frameworks zur Entwicklung von Microservice-Architekturen und verteilten Umgebungen [11], [12]. *Spring Integration* ist ein Framework zur Kommunikation mit externen Systemen, etwa Message Brokern, und dient als Grundlage für das später aufbauende Modul *Spring Cloud Stream* [11]. Analog dazu bildet *Spring Batch* die Grundlage für *Spring Cloud Task* [11].

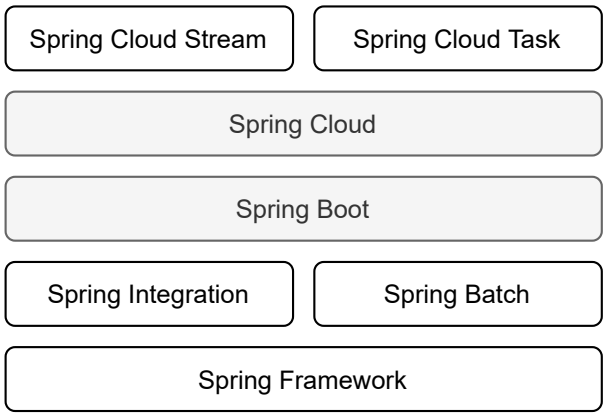

Abbildung 1. Ausschnitt des Spring-Ökosystems. Spring Cloud und Spring Boot als Basis zum Bauen verteilter Anwendungen und Microservice-Architekturen.

In den folgenden Kapiteln werden einige der Spring Cloud-Projekte genauer vorgestellt und gezeigt, wie sich eine Microservice-Architektur mithilfe dieser beiden Frameworks realisieren lässt, die insbesondere Lösungen für eingangs erwähnte Problemstellungen anbietet.

IV. Konfigurationsmanagement

A. Konfigurationsserver mit Spring Cloud Config Server

Spring Cloud Config ermöglicht sowohl serverseitige als auch clientseitige Unterstützung für ausgelagerte Konfigurationsmanagement in verteilten Umgebungen. Durch den Konfigurationsserver (Config Server) wird eine zentrale Instanz zur Verwaltung ausgelagerter Eigenschaften (Properties) für Services über mehrere Umgebungen zur Verfügung gestellt [11], [13]. In Abbildung 2 ist die Funktionsweise des Konfigurationsservers anschaulich dargestellt. Standardmäßig wird die Konfiguration für Services durch ein separates Git-Repository bereitgestellt. Unterstützt werden auch einige weitere Konfigurationsquellen, wie etwa Vault, Credhub, Zookeeper oder Config Maps von Kubernetes. Typischerweise fragen Services beim Starten den Konfigurationsserver nach der eigenen Konfiguration, entsprechend der aktuellen Umgebung [14].

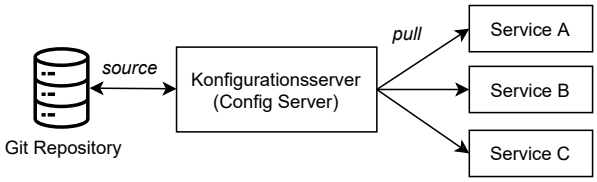

Abbildung 2. Zentrales Konfigurationsmanagement mittels Spring Cloud Config Server

Um einen Spring Cloud Config Server aufzusetzen, bedarf es lediglich einer einfachen Spring Boot-Anwendung mit der Annotation `@EnableConfigServer` [11]. Quellcode 1 zeigt eine Spring Boot-Anwendung mit dieser Annotation. Dadurch wird beim Starten der Anwendung der von Spring Cloud eingebettete Konfigurationsserver ebenfalls gestartet.

Quellcode 1. Applikation zur Erzeugung eines Konfigurationsservers

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(
            ConfigServerApplication.class);
    }
}
```

Die Konfiguration des Servers selbst wird in der `application.yaml` im *resources*-Verzeichnis der Anwendung abgelegt. Quellcode 2 zeigt eine mögliche Konfiguration. Die Anwendung läuft dabei auf Port 8081 und bezieht die Konfiguration aus dem angegebenen Git-Repository. Das Git-Repository stellt wiederum ein eigenes Projekt dar, in der die einzelnen Konfigurationen in Form von Konfigurationsdateien, beispielsweise im Yaml-Format vorliegen. Standardmäßig wird einem Client die Konfiguration entsprechend seines Namens zugewiesen, welcher durch die Eigenschaft `spring.application.name` definiert ist [14].

Quellcode 2. Konfiguration des Config Server mithilfe der application.yaml

```
server:
  port: 8081
spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/DomenicDev/
              KMT_Seminarprojekt_Config
```

B. Konfigurationsclients mit Spring Cloud Config Client

Spring Cloud Config Client bildet das clientseitige Gegenstück für den bereits vorgestellten Konfigurationsserver. Unterstützt werden unter anderem folgende Features (vgl. [11]):

- *Remote Properties*: Anwendungsumgebungen lassen sich durch ausgelagerte Properties mithilfe des Spring Cloud Config Server initialisieren.
- *Verschlüsselung*: Werte lassen sich verschlüsseln und entschlüsseln, sowohl durch ein symmetrisches als auch asymmetrisches Verfahren.
- *Auto-Refresh*: Neue Änderungen der Konfiguration lassen sich ohne Neustarten der Anwendung übernehmen.

Analog zum Server, gibt es auch für den Client eine Starter-Abhängigkeit: `spring-cloud-starter-config`. In der *application.yaml* muss nur noch die Adresse des Konfigurationsservers bekannt gemacht werden [13]. Eine bereits vollständige clientseitige Konfiguration ist in Quellcode 3 dargestellt. Der Service mit dem Namen *service-a* lädt seine Konfiguration beim Starten vom spezifizierten Konfigurationsserver, hier den in Abschnitt IV-A erstellten Server.

Quellcode 3. Konfiguration des Clients mittels Spring Cloud Client

```
spring:
  application:
    name: service-a
  config:
    import: optional:configserver:http://localhost
        :8081
```

V. Service Discovery

Damit die Kommunikation mit Microservices gelingt, muss die Adresse (IP-Adresse und Port) eines Microservices bekannt sein. Aufgrund der Dynamik von Microservice-Architekturen, besonders in Cloud-native-Umgebungen, ist das Hartkodieren der Adressinformation ungeeignet. Diese Information liegt meist erst zum Zeitpunkt der Instanziierung der Anwendung vor und kann sich zur Laufzeit auch ändern, beispielsweise aufgrund von Skalierungsmaßnahmen oder durch Containerorchestrierung [14], [15]. Eine Service Registry bzw. Service Discovery ist ein Mechanismus, der eine Lösung für dieses Problem darstellt, wobei Design Patterns sowohl für clientseitige als auch serverseitige Lösungen existieren [15].

In Anlehnung an [11] ist in Abbildung 3 das serverseitige Service Discovery Pattern dargestellt. Services registrieren sich beim Start bei der Service Registry. Ein Client kann bei der Service Registry Instanzen eines Services abfragen und mit diesen eine Verbindung herstellen [11], [15].

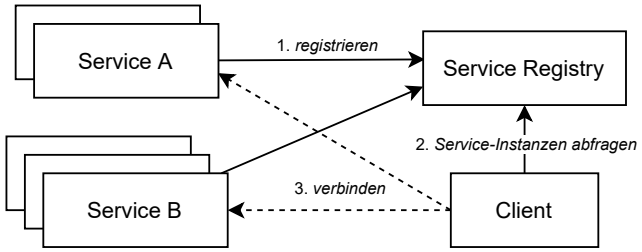

Abbildung 3. Service Discovery-Pattern

A. Service Discovery Server mit Netflix Eureka

Der *Eureka Server* ist die Implementierung des Service Discovery-Pattern von Netflix und Teil von Spring Cloud Netflix, eine Integration von Netflix Open Source Center in Spring Boot. Microservices können sich beim Eureka Server registrieren und von anderen Microservices abgefragt werden.

Spring Cloud bietet hierfür eine Starter-Abhängigkeit: `spring-cloud-starter-netflix-eureka-server`. Der Starter übernimmt die automatische Konfiguration des Servers und stellt Annotationen für ein unkompliziertes Aufsetzen des Servers zur Verfügung. Wie in Quellcode 4 dargestellt, reicht zum Starten des Servers die Annotation `@EnableEurekaServer` in einer Spring Boot-Anwendung aus.

Quellcode 4. Implementierung einer Service Discovery mit Netflix Eureka

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaDiscoveryServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(
            EurekaDiscoveryServerApplication.class);
    }
}
```

Damit die Anwendung sich nicht selbst beim Eureka-Server registriert, sollte die Eigenschaft `registerWithEureka` in der *application.yaml* angepasst werden [11], [16]. Die vollständige Konfiguration der Service Discovery-Anwendung ist in Quellcode 5 dargestellt.

Quellcode 5. Konfiguration des Eureka Servers

```
server:
  port: 8761
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

Nach dem Start der Anwendung lässt sich das Eureka-Dashboard unter *http://localhost:8761* aufrufen, welches neben allgemeinen Informationen zur Umgebung auch alle zurzeit registrierten Instanzen auflistet.

B. Service Discovery Client mit Eureka Client

Für Client-Anwendungen gibt es die Starter-Abhängigkeit `spring-cloud-starter-netflix-eureka-client`. Spring Boot übernimmt jegliche Konfiguration automatisch. Lediglich der Pfad des Eureka-Servers muss in der *application.yaml* angegeben werden, sofern er nicht der Standardadresse (http://localhost:8761) entspricht [11], [17]. Folgend ist in Quellcode 6 die Konfiguration eines Clients dargestellt, in der die Adresse des Eureka-Servers und der Aufbau der ID für Instanzen definiert ist. Letzteres ist nötig, da sich standardmäßig derselbe Service nur einmal pro Host registrieren lässt. Damit sich mehrere Instanzen dergleichen Anwendung auf dem lokalen Rechner registrieren lassen, muss der Aufbau der ID für Instanzen so angepasst werden, dass sich diese voneinander unterscheiden. In diesem Falle besteht die ID aus der Kombination des Anwendungsnamens und einer eindeutigen von Spring vergebenen ID [17].

Quellcode 6. Konfiguration eines Eureka-Clients

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    instanceId: ${spring.application.name}:${vcap.
        application.instance_id:${spring.application
        .instance_id:${random.value}}}
```

VI. Clientseitige Lastverteilung

Load Balancing ist ein Mechanismus, um Lasten auf mehrere Instanzen eines Services zu verteilen, beispielsweise über ein Round-Robin-Verfahren [15]. Ribbon ist ein clientseitiger Load Balancer von Netflix und wurde lange Zeit als Standardlösung von Spring Cloud verwendet. Mittlerweile wurde Ribbon in den Wartungsmodus versetzt und wurde folglich von Spring seit 2019 durch eine eigene Load Balancer-Implementierung ersetzt, namentlich *Spring Cloud LoadBalancer* [12], welche in der Starter-Abhängigkeit `spring-cloud-starter-loadbalancer` enthalten ist.

A. Konfigurieren des Load Balancer

Mit der `@LoadBalanced`-Annotation können Beans vom Typ `WebClient` bzw. `WebClient.Builder` sowie `RestTemplate` annotiert werden [12]. In Quellcode 7 ist der WebClient Builder mit dieser Annotation versehen, wodurch balancierte WebClient-Instanzen erzeugt werden [12]. Es sei darauf hingewiesen, dass auch spezifische Konfigurationen für bestimmte Services angelegt werden können [14].

Quellcode 7. Konfiguration des WebClient-Builder mit Load Balancing

```
@Configuration
public class LoadBalancerConfiguration {

    @Bean
    @LoadBalanced
    WebClient.Builder webClientBuilder() {
        return WebClient.builder();
    }
}
```

B. Lastverteilung über mehrere Instanzen

Quellcode 8 zeigt, wie der zuvor konfigurierte WebClient-Builder genutzt werden kann, um Anfragen über mehrere Instanzen zu verteilen. Der Service *service-b* nimmt Anfragen unter der Adresse *„/call“* entgegen. Durch den Builder wird ein WebClient erstellt, mithilfe dessen der Service *service-a* aufgerufen wird. Durch die Service Discovery sind alle verfügbaren Instanzen dieses Services bekannt, wovon eine Instanz ausgewählt und dieser die Anfrage geschickt wird. Standardmäßig wird Round-Robin zur Lastverteilung genutzt [12]. Der Spring Cloud Load Balancer nutzt zusätzlich einen Caching-Mechanismus, um die Anfragen nach Instanzen an die Service Discovery zu verringern [14].

Quellcode 8. Beispielanwendung zur Lastverteilung über mehrere Instanzen

```
@RestController
public class ServiceB {

    private final WebClient.Builder builder;

    public ServiceB(WebClient.Builder builder) {
        this.builder = builder;
    }

    @GetMapping("/call")
    public Mono<String> callMe() {
        return builder
                .build()
                .get()
                .uri("http://service-a/")
                .retrieve()
                .bodyToMono(String.class);
    }
}
```

VII. Circuit Breaker

In verteilten Systemen und damit auch in Microservice-Architekturen muss jederzeit mit Ausfällen einzelner Services gerechnet werden. Üblicherweise existieren für einen ausgefallenen Microservice andere Microservices, die von diesem abhängig sind. Wenn keine Vorsorgemaßnahmen für den Fehlerfall getroffen werden, können abhängige Microservices ebenfalls nicht mehr korrekt reagieren [18].

Das *Circuit Breaker Pattern* zielt darauf ab, dass sich ein Ausfall einer einzelnen Komponente nicht über andere Komponenten ausbreitet und somit im schlimmsten Fall das ganze System zum Erliegen bringt [19]. Das Pattern lässt sich als endlicher Zustandsautomat implementieren, wie in Abbildung 4 dargestellt. Nach [19] lassen sich die drei Zustände wie folgt beschreiben:

Closed: Anfragen werden an den Zielservice weitergeleitet. Im Fehlerfall, verursacht etwa durch Exceptions oder Timeouts, erhöhen den Fehler- und Timeout-Zähler. Bei zu häufigen Fehlschlägen greift der Unterbrecher und es wird in den Zustand *Open* gewechselt.

Open: Anfragen werden nicht weitergeleitet. Stattdessen wird sofort mit einem Fehler geantwortet. Es wird in den Zustand *Half-Open* gewechselt, wenn der Service wieder erreichbar ist. Die Überprüfung auf Erreichbarkeit erfolgt beispielsweise nach einer gewissen Zeit oder durch periodisches Pingen an des ausgefallenen Services.

Half-Open: In diesem Zustand werden eine begrenzte Anzahl an Anfragen an den Service weitergeleitet. Ist die Verarbeitung der Anfragen erfolgreich, wird wieder in den Zustand *Closed* gewechselt, ansonsten wird im Fehlerfall sofort wieder zurück in den *Open*-Zustand gewechselt.

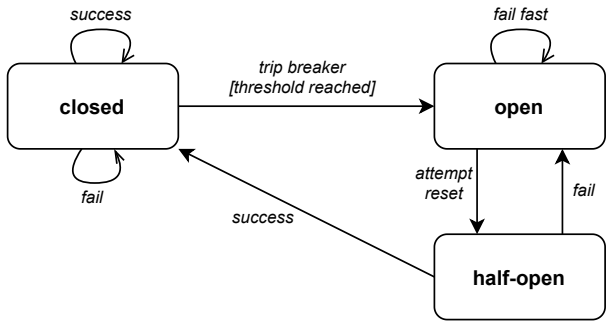

Abbildung 4. Das Circuit Breaker Pattern zur Erhöhung der Robustheit von verteilten Systemen [19]

Spring bietet mit *Spring Cloud Circuit Breaker* eine Abstraktion für verschiedene Implementierungen des Circuit Breaker Patterns, darunter Resilience4J, Netfix Hystrix und Spring Retry. Zwar sind die Circuit Breaker in Starter-Abhängigkeiten bereits vorkonfiguriert, jedoch lassen sich auch eigene Konfigurationen anlegen [20]. In Quellcode 9 ist exemplarisch die Verwendung des Circuit Breaker dargestellt. Bei jedem Aufruf von `runWithCircuit()` wird mithilfe der `CircuitBreakerFactory` ein neuer Circuit Breaker erzeugt. Der auszuführende Task wird als erstes Argument der Methode `run()` übergeben, in diesem Fall wird versucht, die Adresse *http://service-a/*, also Service A, zu erreichen und das Ergebnis dieser Anfrage zurückzugeben. Als (optionales) zweites Argument der Methode wird der Fallback-Mechanismus spezifiziert, wobei in diesem einfachen Fall ein definierter String zurückgegeben wird.

Quellcode 9. Anwendungsbeispiel des Circuit Breaker von Spring Cloud

```
@RestController
public class ServiceB {

    private final CircuitBreakerFactory factory;
    private final WebClient.Builder builder;

    public ServiceB(CircuitBreakerFactory factory,
        WebClient.Builder builder) {
        this.factory = factory;
        this.builder = builder;
    }

    @GetMapping("/run-with-circuit")
    public String runWithCircuit() {
        return factory.create("service-a").run(
                () -> builder.build()
                        .get()
                        .uri("http://service-a/")
                        .retrieve()
                        .bodyToMono(String.class)
                        .block(),
                throwable -> "This is the fallback
                    value!"
        );
    }
}
```

VIII. API Gateway

Microservice-Architekturen können Dienste für verschiedene Arten von Clients und Benutzerschnittstellen bereitstellen, etwa durch Webbrowser, Desktopanwendungen, oder Smartphones. Der API Gateway geht das Problem an, Clienten aus mehreren Umgebungen zu bedienen und stellt als eigenständiger Service einen zentralen Einstiegspunkt für das System dar [18]. In Microservice-Architekturen agiert der API Gateway häufig als Proxy, nimmt Anfragen entgegen und leitet diese an den entsprechenden Microservice weiter [19]. Das Konzept des API Gateways ist in Abbildung 5 dargestellt. Damit sich die einzelnen Microservices auf ihre Kernfunktionalität konzentrieren können, werden vom API Gateway neben dem Routing auch eine Reihe von sog. übergreifenden Features übernommen, wie etwa Security, Monitoring und Ausfallsicherheit [18], [21].

Spring Cloud Gateway stellt eine Möglichkeit zur Realisierung solcher API Gateways dar. Das Framework zielt darauf ab, Anfragen an APIs weiterzuleiten und oben genannte nichtfunktionale Anforderungen, wie Sicherheit, Monitoring und Ausfallsicherheit zu gewährleisten [21].

Spring bietet die Möglichkeit, die Konfiguration des Gateways sowohl über die *application.yml* als auch programmatisch zu definieren. Letztere Möglichkeit erlaubt es, komplexere boolesche Bedingungen zu programmieren, wohingegen

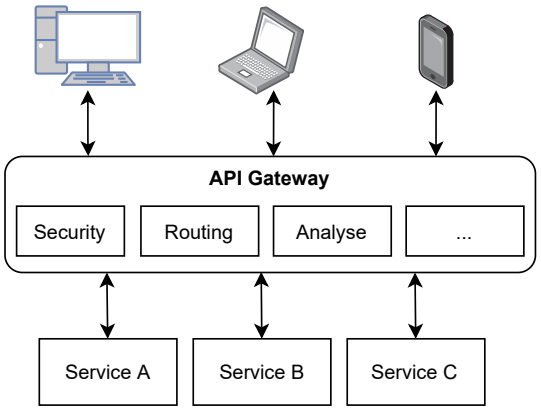

Abbildung 5. Der API Gateway als Einstiegspunkt für Clients in die Microservice-Architektur

erstere Möglichkeit den Vorteil hat, dass die Konfiguration durch den Konfigurationsserver bereitgestellt und bei Änderungen neu geladen werden kann [14].

In Quellcode 10 ist eine einfache Konfiguration des Spring Cloud Gateways dargestellt. Anfragen mit dem Pfad *service-a* werden an Instanzen des Service *service-a* weitergeleitet. Der Prefix `lb` sorgt dafür, dass die Anfragen auf die Instanzen verteilt werden. Im Fehlerfall wird die Anfrage höchstens dreimal erneut gesendet [14], [21].

Quellcode 10. Konfiguration des API Gateway als Proxy

```
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      enabled: true
      routes:
        - id: service-a
          uri: lb://service-a
          predicates:
            - Path=/service-a/**
          filters:
            - name: Retry
              args:
                retries: 3
```

IX. Verhaltensanalyse durch Tracing

Die Verhaltensanalyse und damit verbunden auch das Finden von Fehlern in verteilten Anwendungen kann schnell komplex und unübersichtlich werden. Anfragen durchlaufen eine Reihe von Komponenten, wobei der Ablauf je nach Kontext variieren kann. Um die Nachvollziehbarkeit einer Anfrage zu verbessern, hilft es, diese in einen Kontext einzubinden [22].

Die Idee von *Distributed Tracing* besteht darin, den Verlauf einer Anfrage über mehrere Komponenten durch Instrumentalisierung des Quellcodes und mithilfe einer eindeutigen *Trace ID* nachvollziehbar zu machen. Wenn eine Anfrage an einer Komponente eintrifft, wird ihr eine neue *Span ID* zugewiesen und diese zum Trace hinzugefügt. Der Trace selbst repräsentiert den ganzen Verlauf eine Anfrage [22]. Durch Tools, wie Zipkin, lassen sich Traces grafisch aufbereiten und ermöglichen so eine einfachere Verhaltens- und Fehleranalyse [22], [23].

Spring Cloud Sleuth instrumentalisiert die meisten Kommunikationskanäle und erweitert Logs um Trace und Span IDs [22]. Für die Instrumentalisierung muss lediglich der Starter `spring-cloud-starter-sleuth` eingebunden werden. In Kombination mit `cloud:spring-cloud-sleuth-zipkin` werden die Traces automatisch an den lokalen Zipkin-Server gesendet.

X. Evaluation anhand einer Beispielanwendung

In diesem Abschnitt wird eine Beispielapplikation vorgestellt, um anschließend die verwendeten Konzepte und Technologien bewerten zu können.

A. Architektur der Anwendung

Die vorgestellten Konzepte und Technologien der vorherigen Kapitel wurden anhand einer kleinen Applikation kombiniert praktisch angewandt. Bei dieser Applikation handelt es sich um ein minimalistisches Planetarium, welches für einen bestimmten Ort auf der Erde die heute sichtbaren astronomischen Objekte am Nachthimmel berechnet. Hierfür wurde die in Abbildung 6 dargestellte Microservice-Architektur mit drei Microservices entworfen, die im Folgenden näher beschrieben werden:

LocationService:
Spring Boot-Anwendung, die Orte unter Angabe eines Namens, sowie Längen- und Breitengrad in einer MySQL-Datenbank speichert. Die Ortsinformationen eines Ortes lassen sich unter Angabe des Ortsnamens abfragen.

AstronomicalObjectService:
Spring Boot-Anwendung, die astronomische Objekte (Sterne, Galaxien, etc.) zusammen mit Koordination ebenfalls in einer MySQL-Datenbank speichert. Der Microservice liefert die Objekte für einen bestimmten Koordinatenbereich.

ForecastService:
Spring Boot-Anwendung, die Vorhersage-Anfragen für einen bestimmten Ortsnamen entgegennimmt. Für jede Anfrage werden die Koordinaten des Ortes vom *LocationService* abgefragt, anschließend die minimalen und maximalen Koordinaten der sichtbaren astronomischen Objekte berechnet, für diese die konkreten Objekte beim *AstronomicalObjectService* angefragt werden.

Alle Microservices melden sich beim Start beim Service Discovery Server an und sind so für andere Microservices auffindbar. Die Konfiguration eines jeden Microservices wird vom zentralen Konfigurationsserver bereitgestellt. Der API Gateway dient als Proxy und leitet Anfragen an die entsprechenden Microservices weiter, wobei die Last über mehrere Instanzen mittels Round Robin verteilt wird. Die Zipkin-Instanz sorgt für die visuelle Aufbereitung der Traces von den Microservices.

Der Umfang des Projekts beläuft sich auf ca. 600 Zeilen Java-Quellcode und rund 150 Zeilen Konfiguration. Das

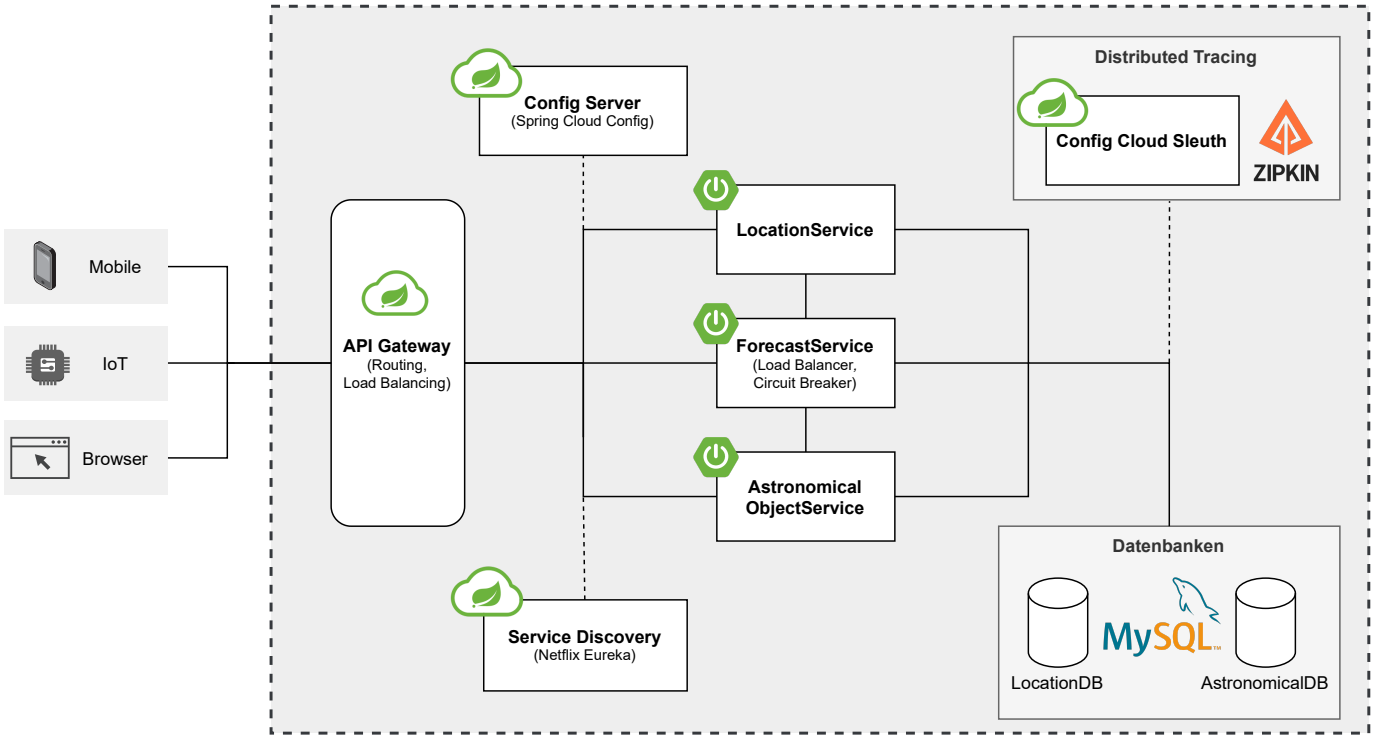

Abbildung 6. Die Planetariumsanwendung realisiert durch eine Microservice-Architektur mit Spring Boot und Spring Cloud. Die drei Microservices sind in der Mitte dargestellt. Die anderen Komponenten des Systems bilden die grundlegende Infrastruktur der Anwendung ab.

Projekt ist quelloffen und unter folgender Adresse einsehbar: *https://github.com/DomenicDev/KMT_Seminarprojekt*

B. Bewertung der Konzepte

In Tabelle I ist die Evaluation der einzelnen Konzepte mit ihrer primären Funktionalität dargestellt. Die Bewertung basiert auf den Ergebnissen der Entwicklung der Beispielanwendung, die im vorherigen Abschnitt vorgestellt wurde. Zusammenfassend lässt sich sagen, dass alle vorgestellten Konzepte aufgrund der Starter-Abhängigkeiten sehr leicht in eine Microservice-Architektur eingebunden werden können. Die Autokonfiguration von Spring Boot senkt den Konfigurationsaufwand erheblich und ermöglicht das Fokussieren auf die eigentliche Anwendung. Kleinere Anpassungen der Konfiguration sind dennoch nötig, um das korrekte Zusammenspiel der einzelnen Komponenten zu gewährleisten. Die Breite der von Spring zur Verfügung gestellten Lösungen ermöglicht das Aufsetzen ganzer Microservice-Infrastrukturen mit sehr überschaubarem Aufwand. Die fundamentalen Komponenten werden durch den Spring Cloud Config Server und der Service Discovery gebildet, da sie von jedem Microservice verwendet werden. Konzepte wie der Spring Cloud Circuit Breaker und der Spring Cloud Load Balancer sind Maßnahmen zur Erhöhung der Robustheit des Gesamtsystems sowie der einzelnen Microservices. Der API Gateway als Proxy und Einstiegspunkt ist leicht zu konfigurieren und verfügt über weitere Funktionen, wie etwa Lastverteilung. Verhaltensanalysen durch Tracing sind mit Spring Cloud Sleuth problemlos möglich.

XI. Zusammenfassung

In dieser Arbeit wurden Technologien des Spring Frameworks zur Lösung ausgewählter Herausforderungen in verteilten Systemen, insbesondere in Microservice-Architekturen vorgestellt. Durch den Spring Cloud Config Server lassen sich Konfigurationen für verschiedene Clients und deren Umgebungen zentral ablegen. Microservices registrieren sich bei einer Service Discovery, wodurch sie sich von anderen Microservices auffinden lassen. Als Einstiegspunkt in das System und zur Lastverteilung der Anfragen auf Instanzen eines Microservices erweist sich der Spring Cloud Gateway als geeignet. Unterstützt wird auch die clientseitige Lastverteilung durch den Spring Cloud LoadBalancer, welcher sich mit wenig Programmieraufwand einbinden lässt. Ein weiterer Faktor zur Erhöhung der Robustheit des Systems bietet Spring Cloud Circuit Breaker, um die Auswirkungen von Ausfällen einzelner Komponenten einzudämmen. Verteiltes Tracing wird durch Spring Cloud Sleuth bereitgestellt und bildet das Fundament zur verbesserten Verhaltensanalyse von verteilten Systemen. Für alle diese Technologien und Lösungen werden Starter-Abhängigkeiten angeboten, die ein schnelles Einbinden in eine Anwendung ermöglichen. In manchen Fällen reicht die Standardkonfiguration aus und es bedarf keiner weiteren Konfiguration, weder programmatisch noch über die Umgebungsvariablen. Fernab der Entwicklung am eigenen Rechner, beispielsweise in einer Produktionsumgebung, ist die Standardkonfiguration nicht mehr ausreichend und es müssen kleinere Anpassungen vorgenommen werden.

Tabelle I
EVALUATION DER BETRACHTETEN SPRING CLOUD-KONZEPTE

Konzept	Funktionalität	Bewertung	Anmerkung
Config Server	Zentraler Dienst zur Bereitstellung von Konfigurationen für Services für unterschiedliche Umgebungen.	✓ Sehr einfache Umsetzung durch Starter-Abhängigkeit.	Möglichkeit zur Anbindung an bereits existierende Konfigurationsdienste, wie etwa Kubernetes Config Maps.
Config Client	Der Config Client verbindet sich mit dem Config Server, um seine Konfiguration abzufragen.	✓ Sehr einfache Umsetzung durch Starter-Abhängigkeit.	Adresse des Config Server muss an eigene Umgebung angepasst werden.
Service Discovery	Zentraler Dienst zum Registrieren und Abfragen von Microservices. Standardimplementierung verwendet Netflix Eureka.	✓ Bequeme und schnelle Umsetzung durch Starter-Abhängigkeit ✓ Dashboard mit aufbereiteten Informationen	Bereits bei Entwicklungsumgebungen zusätzlicher Konfigurationsbedarf nötig, etwa um mehrere Instanzen des gleichen Services zu starten.
Load Balancing	Einbindung von client-seitiger Lastverteilung anhand vordefinierter oder eigener Lastverteilungsmethoden.	✓ Nur wenig Konfiguration nötig. ✓ Ausreichende Lastverteilungsmechanismen eingebaut für die Klassen *RestTemplate* und *WebClient.Builder*.	Möglichkeit zur Implementierung eigener Lastverteilungsmethoden. Typischerweise Einsatz zusammen mit Service Discovery, um Instanzen eines Services abzufragen.
Circuit Breaker	Erhöhung der Robustheit und Fehlertoleranz des Systems. Bei Störungen wird ein vordefiniertes Verhalten ausgeführt, wodurch der Nutzer weiterhin sinnvolle Rückmeldungen bekommt.	✓ Spring Cloud Circuit Breaker bietet eine Abstraktion für verschiedene Implementierungen, wodurch eine modulare und einfache Entwicklung ermöglicht wird. ✓ Standard-Konfiguration reicht in Beispielanwendung völlig aus.	Umfangreiche Konfigurationsmöglichkeiten, die in der Dokumentation gut beschrieben sind.
API Gateway	Stellt Einstiegspunkt in das System dar und kümmert sich um weitere übergreifende Anforderungen wie Lastverteilung und Sicherheit.	✓ Der Spring Cloud Load Balancer lässt sich bequem über die *application.yml* oder programmatisch konfigurieren. ✓ Sehr einfache Einbindung von Lastverteilungsmechanismen.	Gutes Zusammenspiel mit Service Discovery und Config Server.
Tracing	Ermöglicht Verhaltensanalyse und Debugging in verteilten Anwendungen. Logs werden um Trace ID und Spand ID erweitert.	✓ Völlig transparente Einbindung durch Abhängigkeit. ✓ Es muss kein Quellcode angepasst werden.	Es existieren weitere Pakete, um Logs direkt an Dienste, wie Zipkin, zur Aufbereitung zu schicken.

LITERATUR

[1] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017, pp. 466–475.

[2] T. Hoff, "Stackoverflow update: 560m pageviews a month, 25 servers, and it's all about performance," 07 2021. [Online]. Available: http://highscalability.com/blog/2014/7/21/stackoverflow-update-560m-pageviews-a-month-25-servers-and-i.html

[3] M. Fowler and J. Lewis, "Microservices," Mar 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[4] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, 2018, pp. 149–154.

[5] C. Richardson, "Introduction to microservices," May 2019. [Online]. Available: https://www.nginx.com/blog/introduction-to-microservices

[6] T. Prasandy, Titan, D. F. Murad, and T. Darwis, "Migrating application from monolith to microservices," in *2020 International Conference on Information Management and Technology (ICIMTech)*, 2020, pp. 726–731.

[7] L. De Lauretis, "From monolithic architecture to microservices architecture," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 93–96.

[8] S. Maple and A. Binstock, "Jvm ecosystem report 2018 – about your platform and application," October 2018. [Online]. Available: https://snyk.io/blog/jvm-ecosystem-report-2018-platform-application/

[9] M. Gajda, P. Cooper, L. Mezzalira, R. Rodger, and S. Banskota, "State of microservices 2020," The Software House, Tech. Rep., 2020. [Online]. Available: https://tsh.io/state-of-microservices/

[10] A. Sill, "The design and architecture of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 76–80, 2016.

[11] F. Gutierrez, *Spring Cloud*. Berkeley, CA: Apress, 2021, pp. 89–127. [Online]. Available: https://doi.org/10.1007/978-1-4842-1239-4_5

[12] M. Deinum and I. Cosmina, *Spring Applications in the Cloud*. Berkeley, CA: Apress, 2021, pp. 521–561. [Online]. Available: https://doi.org/10.1007/978-1-4842-5666-4_13

[13] Spring. (2021) Spring cloud config. Version: 3.1.0. [Online]. Available: https://docs.spring.io/spring-cloud-config/docs/current/reference/html/

[14] O. Maciaszek-Sharma and S. Gibb. (2019) Introduction to spring cloud. Spring Developer. [Online]. Available: https://youtu.be/PpW5aPfw06I

[15] S. R. Goniwada, *Microservices Architecture and Design*. Berkeley, CA: Apress, 2022, pp. 191–240. [Online]. Available: https://doi.org/10.1007/978-1-4842-7226-8_5

[16] Spring. (2021) Service registration and discovery. Version: 3.1.0. [Online]. Available: https://spring.io/guides/gs/service-registration-and-discovery/

[17] ——. (2021) Spring cloud netflix. Version: 3.1.0. [Online]. Available: https://docs.spring.io/spring-cloud-netflix/docs/current/reference/html/

[18] F. Montesi and J. Weber, "Circuit breakers, discovery, and api gateways in microservices," 2016.

[19] B. Christudas, *Spring Cloud*. Berkeley, CA: Apress, 2019, pp. 183–244. [Online]. Available: https://doi.org/10.1007/978-1-4842-4501-9_8

[20] Spring. (2021) Spring cloud circuit breaker. Version: 2.1.0. [Online]. Available: https://docs.spring.io/spring-cloud-circuitbreaker/docs/current/reference/html/

[21] ——. (2021) Spring cloud gateway. Version: 3.1.0. [Online]. Available: https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/

[22] K. Bastani and J. Long, *Cloud Native Java: Designing Resilient Systems with Spring Boot, Spring Cloud and Cloud Foundry*. O'Reilly Media, Inc., August 2017.

[23] Zipkin, "Zipkin," letzter Zugriff: 06.01.2022. [Online]. Available: https://zipkin.io/