HAW HAMBURG

BACHELOR THESIS
Wassim Berraqui

Vergleich der Java-Frameworks Quarkus und Spring anhand der Entwicklung und Performance der PetClinic-Microservices-Anwendung

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Wassim Berraqui

Vergleich der Java-Frameworks Quarkus und Spring anhand der Entwicklung und Performance der PetClinic-Microservices-Anwendung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 22.10.2024

Wassim Berraqui

Thema der Arbeit

Vergleich der Java-Frameworks Quarkus und Spring anhand der Entwicklung und Performance der PetClinic-Microservices-Anwendung

Stichworte

Spring Boot, Quarkus, Microservices, Java Framework, Performance, Skalierbarkeit, Ressourceneffizienz

Kurzzusammenfassung

Diese Bachelorarbeit analysiert die Java-Frameworks Quarkus und Spring im Kontext der Entwicklung und des Betriebs von Microservices. Die Untersuchung erfolgt anhand der Implementierung der PetClinic-Microservices-Anwendung, wobei Schlüsselkriterien wie Anwendungsentwicklung, Startzeit, Skalierbarkeit und Ressourcenverbrauch verglichen werden. Spring, als etabliertes Framework, zeichnet sich durch eine breite Funktionalität und Stabilität aus, während Quarkus besonders durch seine optimierte Ressourcennutzung und schnelle Startzeiten für moderne Cloud-Umgebungen überzeugt. Die Arbeit zeigt, dass Quarkus vor allem bei der Performance und Ressourceneffizienz Vorteile bietet, während Spring aufgrund seiner ausgereiften Architektur und umfassenden Tools in größeren, komplexeren Umgebungen punktet. Die Ergebnisse liefern eine fundierte Grundlage zur Auswahl des geeigneten Frameworks für Microservices-Architekturen je nach spezifischen Anforderungen.

Wassim Berraqui

Title of Thesis

Comparison of the Java frameworks Quarkus and Spring based on the development and performance of the PetClinic microservices application

Keywords

Spring Boot, Quarkus, Microservices, Java Framework, Performance, Scalability, Resource Efficiency

Abstract

This bachelor thesis analyzes the Java frameworks Quarkus and Spring in the context of developing and operating microservices. The investigation is based on the implementation of the PetClinic microservices application, comparing key criteria such as application development, startup time, scalability, and resource consumption. Spring, as an established framework, stands out for its extensive functionality and stability, while Quarkus excels with its optimized resource usage and fast startup times, making it ideal for modern cloud environments. The study demonstrates that Quarkus offers advantages in terms of performance and resource efficiency, while Spring scores in larger, more complex environments due to its mature architecture and comprehensive tools. The results provide a solid basis for selecting the appropriate framework for microservices architectures based on specific requirements.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Hintergrund und Relevanz der Frameworks Quarkus und Spring

Java-Frameworks wie Quarkus und Spring spielen eine zentrale Rolle in der modernen Softwareentwicklung. Sie bieten strukturierte Lösungen für die Entwicklung komplexer Anwendungen und sind insbesondere im Bereich der Microservices-Architekturen unverzichtbar geworden.

Spring ist eines der am weitesten verbreiteten Frameworks für die Java-Entwicklung, besonders im Bereich der Webanwendungen [13]. Es hat sich seit über einem Jahrzehnt als das Framework etabliert, nach dem die Java-Community lange gesucht hat. Mit seinen benutzerfreundlichen Entwicklungsfunktionen und der sofort einsatzbereiten Operationalisierung macht es die Java-Entwicklung wieder attraktiv [36]. Besonders hervorzuheben ist Spring Boot, ein Tool, das die Entwicklung von Webanwendungen und Microservices vereinfacht und beschleunigt [12].

Quarkus hingegen, zielt es darauf ab, die Leistung von Java-Anwendungen zu optimieren und gleichzeitig ihren Platzbedarf und ihre Startzeit zu reduzieren. Quarkus nutzt eine Vielzahl von Technologien, darunter GraalVM, Kubernetes und reaktive Programmierung, um eine effiziente und leichtgewichtige Plattform für die Entwicklung und Bereitstellung von Java-Anwendungen zu bieten[29]. Es wurde entwickelt, um die bestmöglichen Ergebnisse in Kategorien wie Speicherverbrauch, Laufzeit und Größe des Docker-Images. Diese Eigenschaften haben es ermöglicht eine effektive Plattform für serverlose, Cloud- und Kubernetes-Umgebungen zu werden [31]. Es hat in den letzten Jahren an Popularität gewonnen, da es verspricht, sowohl in Bezug auf Startzeit als auch Ressourcennutzung effizienter zu sein [8].

Die Wahl des richtigen Frameworks ist für viele Entwickler und Unternehmen von großer Bedeutung, da sie direkten Einfluss auf die Effizienz und Wartbarkeit der Anwendungen

hat. Spring bietet Stabilität und eine breite Palette an Funktionen, die durch jahrelange Weiterentwicklung und eine große Community entstanden sind. Quarkus hingegen bringt innovative Ansätze und Technologien mit, die auf die spezifischen Bedürfnisse moderner Cloud-Umgebungen zugeschnitten sind. Diese beiden Frameworks bieten Entwicklern unterschiedliche Ansätze und Werkzeuge zur Bewältigung der Herausforderungen in der Softwareentwicklung, weshalb es von großer Bedeutung ist, ihre jeweiligen Stärken und Schwächen zu untersuchen [6].

1.2 Zielsetzung der Arbeit und Fragestellung

Das Ziel dieser Arbeit ist es, die Java-Frameworks Quarkus und Spring im Hinblick auf ihre Eignung für die lokale Entwicklung und den Betrieb von Microservices zu vergleichen. Im Fokus steht die Implementierung der PetClinic-Microservices-Anwendung, um praxisnahe Erkenntnisse über die Unterschiede der Frameworks in Bezug auf Anwendungsentwicklung, Performance, Skalierbarkeit und Ressourceneffizienz zu gewinnen. Durch diesen Vergleich sollen fundierte Erkenntnisse darüber gewonnen werden, welches Framework für bestimmte lokale Einsatzszenarien besser geeignet ist. Folgende Forschungsfragen stehen im Mittelpunkt: 1. Wie unterscheiden sich Quarkus und Spring bei der Anwendungsentwicklung in Bezug auf Architektur, Datenmodellierung und Benutzerfreundlichkeit? 2.Welches Framework bietet bei lokalen Tests eine höhere Performance hinsichtlich Startzeit, Antwortzeit und Stabilität? 3. Welches Framework ist besser in Bezug auf horizontale und vertikale Skalierbarkeit sowie effiziente Ressourcennutzung?.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist in fünf Kapitel unterteilt. In Kapitel 1 werden die Relevanz der beiden Frameworks Quarkus und Spring sowie die Zielsetzung und die Forschungsfragen der Arbeit erläutert. Kapitel 2 stellt die theoretischen Grundlagen vor, einschließlich eines Überblicks über Java-Frameworks, Microservices-Architekturen und die spezifischen Eigenschaften von Spring und Quarkus. Kapitel 3 beschreibt das methodische Vorgehen der Arbeit, insbesondere das Forschungsdesign, die Implementierung der PetClinic-Microservices-Anwendung in Quarkus und die Vergleichsstrategien zu Spring. In Kapitel 4 erfolgt eine komparative Analyse der beiden Frameworks, die sich auf die Anwendungsentwicklung, Performance, Skalierbarkeit und Ressourcennutzung konzentriert.

Abschließend fasst Kapitel 5 die wichtigsten Ergebnisse zusammen und gibt einen Ausblick auf mögliche zukünftige Entwicklungen in der Nutzung von Quarkus und Spring in der Microservices-Architektur.

2 Theoretische Grundlagen

In diesem Kapitel werden die Grundlagen behandelt, also verschiedene Begrifflichkeiten und Konzepte, die für den Rest der Arbeit von Bedeutung sind. Dazu gehören die hier verwendeten Technologien und Architekturprinzipien, die den Kern der Java-Frameworks bilden.

2.1 Überblick über Java-Frameworks

Java Frameworks sind wiederverwendbare vorformulierte Codeabschnitte, die als Vorlagen für Entwickler dienen. Sie können damit Anwendungen erstellen, indem sie nach Bedarf benutzerdefinierten Code eingeben [26]. Diese Frameworks sind vielfältige und spezialisierte Entwicklungswerkzeuge, die speziell für die Java-Plattform entwickelt wurden. Wie in der Abbildung 2.1 dargestellt, gibt es eine Vielzahl von Frameworks, die jeweils auf unterschiedliche Einsatzzwecke und Anforderungen zugeschnitten sind.

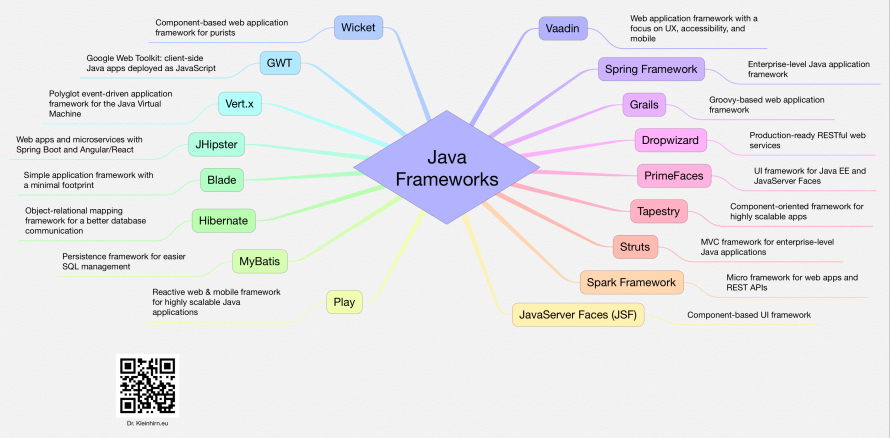


Abbildung 2.1: Java Frameworks. Quelle: [7]

2.2 Microservices-Architekturen

Microservice Architecturen ist Im Gegensatz zu monolithischen Architekturen, bei denen die Anwendungen als eine einzige, zusammenhängende Einheit gebaut werden, fördert die Microservice-Architektur einen Ansatz. Sie zerlegt Anwendungen in kleinere, unabhängig voneinander einsetzbare Dienste, die eher nach Geschäftsfunktionen als nach technischen Aspekten[19]. Ein Kernprinzip der Microservice-Architektur ist die Granularität. Jeder Microservice konzentriert sich auf eine einzige, isolierte Funktion, was die Wartbarkeit und Skalierbarkeit verbessert[14].

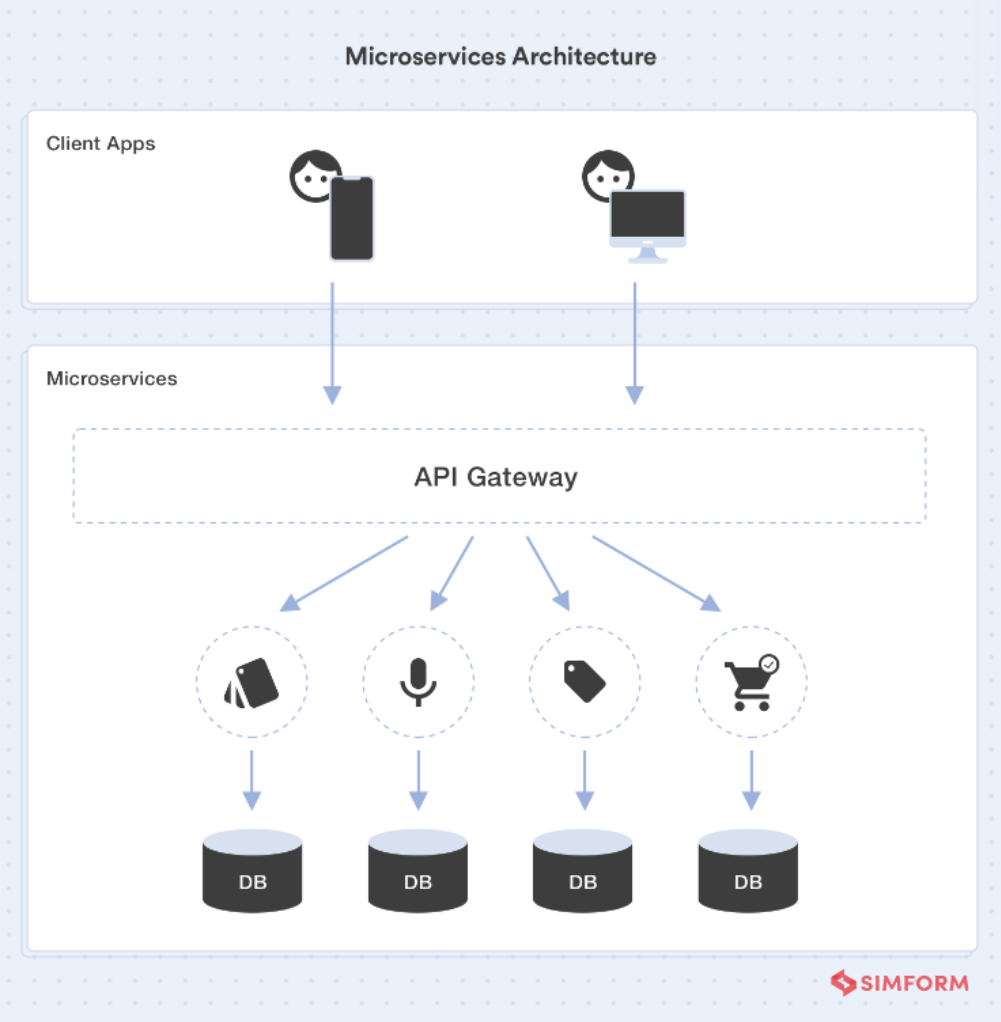


Abbildung 2.2: Microservices Architektur. Quelle: [7]

2.2.1 Merkmale von Microservices [7]

- Jeder Dienst verfügt über eine eigene Datenbankschicht, eine unabhängige Codebasis und CI/CD-Tooling-Sets.
- Die Dienste sind für die Bewahrung ihrer Daten oder ihres externen Zustands verantwortlich.
- Jeder Dienst ist unabhängig einsatzfähig und kann isoliert getestet werden, ohne von anderen Diensten abhängig zu sein.
- Die interne Kommunikation zwischen den Diensten erfolgt über genau definierte APIs oder ein leichtgewichtiges Kommunikationsprotokoll.
- Jeder Dienst kann den Technologie-Stack, die Bibliotheken und Frameworks auswählen, die für seine Anwendungsfälle am besten geeignet sind.
- Dienste sollten eine Wiederholungsfunktion für den Fall eines Netz- oder Systemausfalls implementieren.

2.2.2 Kommunikation zwischen Microservices

Die Kommunikation zwischen Microservices bezieht sich auf den Datenaustausch und die Interaktionen zwischen unabhängigen Diensten innerhalb einer Microservices-Architektur. Es gibt verschiedene Kommunikationsmethoden, die von Clients und Diensten genutzt werden können, wobei jede Methode für spezifische Szenarien und Ziele ausgelegt ist. Diese Kommunikationsmethoden lassen sich grundlegend in zwei Hauptkategorien unterteilen :

- **Asynchrone Kommunikation** hingegen ermöglicht es einem Service, eine Nachricht zu senden, ohne auf eine sofortige Antwort zu warten. Der empfangende Service verarbeitet die Nachricht zu einem späteren Zeitpunkt. Typische Beispiele sind Message Queues wie RabbitMQ oder Event-Driven Architecture. Diese Methode entkoppelt die Services, verbessert die Skalierbarkeit und macht das System fehlertoleranter, aber sie kann komplexer zu implementieren und zu überwachen sein [14].

- **Synchrone Kommunikation** bedeutet, dass ein Service eine Anfrage an einen anderen Service sendet und darauf wartet, bis eine Antwort zurückkommt, bevor er weiterarbeitet. Ein typisches Beispiel dafür ist die Kommunikation über RESTful APIs. Dies ist einfach zu implementieren, aber die Services sind stärker voneinander abhängig, was zu Problemen führen kann, wenn einer der Services nicht verfügbar ist [19].

Da Microservices isoliert sind und über HTTP-Anfragen oder Hilfsdienste kommunizieren, können Teams ihre Dienste mit verschiedenen Programmiersprachen und Technologie-Stacks entwickeln. Diese Autonomie ermöglicht es jedem Team, die Tools und Frameworks zu wählen, die am besten zu den Anforderungen ihres Services passen, ohne durch die Entscheidungen anderer Teams eingeschränkt zu sein. Diese Flexibilität ist ein entscheidender Vorteil von Microservices gegenüber monolithischen Architekturen, da sie unabhängige Entwicklungs- und Bereitstellungszyklen ermöglicht [14].

Kommunikation	**Typ**	**Vorteile**	**Nachteile**	**Beispiele**
RESTful API	Synchron	Einfach zu implementieren, weit verbreitet	Eng gekoppelt, Latenz bei Dienstfehlern	HTTP, JSON
gRPC	Synchron	Hochperformant, Multilingual	Komplexer, erfordert Protokollpuffer	HTTP/2, Protocol Buffers
Message Queues	Asynchron	Entkoppelt, Fehlertoleranz, Skalierbarkeit	Erhöhte Komplexität bei der Nachrichtenverwaltung	RabbitMQ, Apache Kafka
Event-Driven Communication	Asynchron	Entkoppelt, gut für reaktive Systeme	Schwieriger zu debuggen und zu überwachen	Apache Kafka, AWS SNS

Tabelle 2.1: Kommunikationsmethoden im Vergleich

2.2.3 Design Patterns für die Kommunikation

In einer Microservices-Architektur sind effiziente Kommunikation und Koordination entscheidend. Das Service-Registry-Muster (mit Eureka oder Consul) und das API-Gateway-Muster sind zentrale Komponenten, die Service-Erkennung, Routing und Aggregation von Anfragen ermöglichen. Dieser Abschnitt untersucht, wie Microservices diese Tools nutzen, um skalierbare und belastbare Architekturen zu unterstützen. Durch den Einsatz von Service-Registries und API Gateways können Unternehmen anpassungsfähige Microservices-Architekturen schaffen, die den Anforderungen moderner Softwaresysteme gerecht werden.

- **Service Discovery** In einer dynamischen Umgebung, in der Microservices instanziiert, skaliert und terminiert werden können, müssen Dienste in der Lage sein, die Adressen der benötigten Dienste zur Laufzeit zu ermitteln. Service Discovery ermöglicht es, dass Microservices ihre Standorte beim Start registrieren und andere Dienste diese über ein zentrales Verzeichnis finden können [27].
- **API Gateway** Ein API Gateway dient als zentraler Punkt für alle externen Anfragen an die Microservices. Es kann Anfragen weiterleiten, aggregieren, Authentifizierungen vornehmen und als Load Balancer fungieren. Dies vereinfacht die Client-Interaktion und zentralisiert die Zugangskontrolle und Überwachung [18].

2.2.4 Schwachstellen von Microservices

Microservices können die Zuverlässigkeit und Skalierbarkeit einer Anwendung verbessern. Die Codebasen sind modularer als bei einem Monolithen, was einige Aspekte der Microservices-Entwicklung vereinfachen kann, und der Ausfall eines einzelnen Microservices bringt nicht die gesamte Anwendung zum Absturz. Aber Microservices können jedoch einige echte Herausforderungen und Nachteile mit sich bringen, darunter eine erhebliche Zunahme der Anwendungskomplexität und mögliche Leistungseinbußen.

Nachteil	**Beschreibung**
Infrastruktur-Overhead	Höherer Ressourcenverbrauch durch redundante Logik und den Einsatz zusätzlicher Tools wie Container-Orchestratoren. Monitoring jedes Microservices erhöht ebenfalls den Overhead.
Entwicklungskomplexität	Modularität erhöht Abhängigkeiten und führt zu komplexeren Build- und Testprozessen, da jeder Microservice separat kompiliert und getestet werden muss.
Betriebskomplexität	Mehr bewegliche Teile und potenzielle Fehlerquellen. Zusätzliche Tools wie Orchestratoren und Service Meshes erhöhen die Komplexität.
Leistungsrisiken	Komplexität kann zu geringerer Leistung führen. Einzelne Services können Ressourcen erschöpfen, was Leistungsprobleme verursacht. Fehlerhafter Code kann leichter unbemerkt bleiben.

Tabelle 2.2: Nachteile von Microservices [35]

2.3 Spring Boot

Java Spring Boot ist ein Open-Source-Tool, das die Verwendung von Java-basierten Frameworks zum Erstellen von Microservices und Web-Apps vereinfacht. Bei der Definition von Spring Boot muss zunächst Java erwähnt werden – eine der beliebtesten und meistverbreiteten Entwicklungssprachen und Computingplattformen für die App-Entwicklung. Es ist bekannt für seine Betonung auf Wiederverwendbarkeit und Modularität von Komponenten, was eine schnelle und flexible Entwicklung ermöglicht.
Spring und die verschiedenen darauf aufbauenden Frameworks wie Spring Boot, die 2002 auf den Markt kamen, haben die Art und Weise, wie Java-Entwickler Code schreiben, dominiert. Der Entwickler Rod Johnson hatte die Idee zu Spring im Jahr 2002. Im folgenden Jahr entwickelte Johnson zusammen mit Jürgen Hoeller und Yann Caroff Spring als Open-Source-Framework. Sie entwickelten Spring, um die serverseitige Java-Entwicklung zu vereinfachen und Entwicklerteams die Möglichkeit zu geben, ihre Anwendungen schneller zu erstellen [4].
Spring integriert eine Vielzahl wichtiger Java-Bibliotheken für die Webentwicklung, einschließlich Datenbankzugriff, Sicherheit, Cloud-Bereitstellung und Webservices. Seit seiner Einführung hat sich Spring schnell weiterentwickelt, wobei das Kernmodul wesentliche Funktionen wie Inversion of Control, Dependency Injection und gängige Annotations bietet. Diese bilden die Grundlage für verschiedene andere Frameworks wie Thymeleaf, MyBatis und Spring MVC.
Im April 2014 wurde Spring Boot als Teil des Spring-Ökosystems eingeführt und revolutionierte die schnelle Anwendungsentwicklung. Im Gegensatz zum traditionellen Spring reduziert Spring Boot den Konfigurationsaufwand erheblich, indem es automatische Konfigurationsfunktionen bereitstellt. Diese Funktion ermöglicht es Entwicklern, Projekte mit minimalem Setup schnell zu starten, wodurch sowohl für eigenständige als auch für unternehmensweite Webanwendungen gesorgt wird.
Spring Boot automatisiert die Konfiguration von Spring und Drittanbieter-Bibliotheken, was den Abhängigkeitsverwaltungsprozess vereinfacht. Durch die nahtlose Integration verschiedener Spring-Projekte steigert es die Produktivität der Entwickler durch zahlreiche produktionsreife Integrationen. Dabei legt Spring Boot Wert auf Einfachheit und Effizienz, minimiert den Konfigurationsaufwand und Boilerplate-Code, um den Entwicklungsprozess zu beschleunigen und die Markteinführungszeit von Java-Anwendungen zu verkürzen [3].
Spring Boot bietet ein neues Paradigma für die Entwicklung von Spring-Anwendungen mit minimalen Reibungsverlusten. Mit Spring Boot können Sie Spring-Anwendungen fle-

xibler entwickeln und sich auf die Anforderungen an die Funktionalität Ihrer Anwendung konzentrieren, ohne sich Gedanken über die Konfiguration von Spring selbst machen zu müssen (oder möglicherweise gar nicht) [36].

Schlüsselmerkmale von Spring Boot [16]:

- **Auto-configuration**: Automatische Konfiguration der Anwendung basierend auf den hinzugefügten Abhängigkeiten, wodurch explizite Konfigurationen minimiert werden.
- **Standalone Applications**: Ermöglicht das eigenständige Ausführen von Anwendungen ohne externen Webserver, dank eingebetteter Server wie Tomcat, Jetty oder Undertow.
- **Opinionated Defaults**: Vordefinierte Standardkonfigurationen für den schnellen Projektstart, mit der Möglichkeit zur individuellen Anpassung.
- **Actuator**: Bietet integrierte Endpunkte zur Überwachung und Verwaltung der Anwendungszustände, Metriken und mehr, ohne zusätzlichen Konfigurationsaufwand.

Spring Boot ist der perfekte Partner für jeden Entwickler, der seinen Entwicklungsprozess rationalisieren und etwas wirklich Außergewöhnliches entwickeln möchte. Es ist ein One-Stop-Shop für alle Ihre Entwicklungsbedürfnisse, und die Möglichkeiten sind wirklich endlos. Machen Sie sich die Magie von Spring Boot zunutze und heben Sie Ihr Entwicklungsspiel auf neue Höhen [28].

2.3.1 Vorteile und Nachteile von Spring Boot

Spring Boot Pros	**Spring Boot Cons**
Bietet viele Lösungen zur Automatisierung der Konfiguration von Abhängigkeiten, Objekten und Containern sowie sofort einsatzbereite Konfigurationslösungen.	Mit einer Vielzahl eigener Lösungen, weniger Kontrolle und Optionen für Integrationen.
Performt besser mit JVM, wo JIT-Kompilierung vorteilhafter ist.	Höherer Ressourcenbedarf beim Ausführen oder Erstellen von Anwendungen.
Bietet die Möglichkeit zur aspektorientierten Programmierung.	Viele interne Abhängigkeiten zwischen Modulen können ebenfalls Komplexität erzeugen und erfordern mehr Ressourcen zur Verwaltung.
Setzt auf JIT-Kompilierung und kann unter hoher Last bessere Ergebnisse erzielen.	
Ein bewährtes, bei Entwicklern und in der gesamten Unternehmensbranche beliebtes und respektiertes Framework.	
Eine riesige Community, die auf fast jede Frage eine Antwort weiß und bereit ist, diese zu teilen.	
Entwicklung kann in den meisten Fällen viel schneller und günstiger sein.	

Tabelle 2.3: Vor- und Nachteile von Spring Boot [17]

2.4 Quarkus

Quarkus ist von Red Hat entwickeltes Java-Framework, das speziell auf die Anforderungen der heutigen schnelllebigen Industrie ausgerichtet ist. Quarkus ist für Kubernetes-Umgebungen konzipiert und bietet native Unterstützung für Microservices mit minimalem Speicherverbrauch und schneller Ausführung.
Der Fokus von Quarkus bleibt auf moderne Architekturen und kompatibel mit vertrauten Java-Bibliotheken wie Hibernate und REST-Services. Diese Komponenten, zusammen mit innovativen APIs wie MicroProfile und reaktiven Programmiermodellen wie Vert.x, bilden das Fundament von Quarkus ausführliche Toolkit.
Im Bereich des serverlosen Computings, der Microservices-Architektur, der Containerisierung, der Kubernetes-Orchestrierung, von Function as a Service und der Cloud-Bereitstellung sticht Quarkus als Lösung hervor, die speziell für diese Umgebungen entwickelt wurde. Der containerzentrierte Ansatz von Quarkus passt sich nahtlos an Cloud-native Java-Anwendungen an und überbrückt die Kluft zwischen imperativen und reaktiven Programmierparadigmen. Mit einer umfangreichen Sammlung von unternehmensgerechten Java-Bibliotheken und Frameworks setzt Quarkus auf Entwicklerproduktivität und verspricht, das Java-Entwicklungserlebnis neu zu definieren.
Quarkus wird immer beliebter, insbesondere in Verbindung mit Kubernetes, da es einen wesentlich geringeren Ressourcenverbrauch bietet. All dies ist auch für Unternehmen wichtig, denn der Ressourcenverbrauch muss bezahlt werden. Natürlich wollen die Eigentümer ihn reduzieren, was Quarkus sehr gut kann. Außerdem beginnen viele Entwickler und Unternehmen, Quarkus als die Zukunft zu sehen, weil es die Entwicklung viel einfacher macht und auf Industriestandards basiert, im Gegensatz zu Spring Boot, das alle Standards neu gestaltet hat [17].

Schlüsselmerkmale von Quarkus [17]:

- **Container-Fokus:** Quarkus ist speziell auf die Arbeit mit Containern, insbesondere Kubernetes, ausgelegt. Es setzt auf maximale Integration mit bestehenden Frameworks und Diensten, anstatt eigene Lösungen zu entwickeln.
- **Imperative und reaktive Programmierung:** Quarkus unterstützt sowohl imperative als auch reaktive Programmieransätze, wodurch Entwickler flexibel arbeiten können.

- **Native Kompilierung:** Dank der Unterstützung von OpenJDK HotSpot und GraalVM kann Quarkus Code nativ für Container kompilieren, was den Ressourcenverbrauch erheblich reduziert. In Kubernetes-Umgebungen läuft Quarkus als natives Linux-Executable.
- **Live Coding:** Quarkus ermöglicht automatisches Deployment von Code innerhalb eines Containers, wodurch traditionelle Entwicklungsprozesse verkürzt werden. Der Workflow beschränkt sich auf "Write Code/Refresh Browser/Repeat".
- **Zero Configuration mit Dev Services:** Quarkus startet und konfiguriert automatisch Datenbanken während der Entwicklung oder des Testens von Anwendungen.
- **DEV UI:** Eine interaktive Entwickleroberfläche, die Dependency-Updates und Konfigurationen schnell und einfach ermöglicht.
- **Continuous Testing:** Integrierte kontinuierliche Tests direkt während der Entwicklung, ohne die Notwendigkeit zusätzlicher Tools.
- **Bewährte Standards:** Quarkus basiert auf etablierten Standards wie JAX-RS, JPA, JTA, Apache Camel und Hibernate.

Im Kern konzentriert sich Quarkus auf die Entwickler, mit dem Ziel, die Produktivität zu steigern und den Entwicklungsprozess zu vereinfachen. Durch das Angebot einer breiten Palette von Tools, Bibliotheken und Erweiterungen bietet Quarkus ein reibungsloses Entwicklungserlebnis, insbesondere im Entwicklungsmodus, in dem Entwickler schnell und effizient iterieren können. Darüber hinaus glänzt Quarkus durch seine Integration mit Kubernetes, die den Bereitstellungsprozess vereinfacht, ohne tiefgehende Kenntnisse der zugrunde liegenden Komplexitäten zu erfordern. Mit Funktionen zur automatischen Generierung von Kubernetes-Ressourcen und zum Bereitstellen von Container-Images ermöglicht Quarkus Entwicklern, sich auf die Anwendungslogik zu konzentrieren, anstatt sich mit Infrastrukturdetails auseinanderzusetzen.
Quarkus unterstützt verschiedene Entwicklungsstile, einschließlich imperativer und reaktiver Programmierung. Diese Flexibilität ermöglicht es Entwicklern, ihren bevorzugten Ansatz zu verwenden und gleichzeitig nahtlos zu Cloud-native und reaktiven Architekturen überzugehen, wenn dies erforderlich ist. Insgesamt stellt Quarkus einen bedeutenden Fortschritt in der Java-Entwicklung dar, indem es eine vielseitige Plattform bietet, die

auf moderne Cloud-native Umgebungen zugeschnitten ist und gleichzeitig Entwicklerproduktivität und Effizienz in den Vordergrund stellt [20]. Das Framework erfreut sich großer Beliebtheit und wird regelmäßig aktualisiert.

2.4.1 Vorteile und Nachteile von Quarkus

Quarkus Pros	**Quarkus Cons**
Vereinfacht und fördert die Arbeit mit Containern, insbesondere Kubernetes, und ermöglicht es Entwicklern, redundante Prozesse in der Konfiguration und Bereitstellung zu vermeiden.	Die anfängliche Installation von Graal VM kann für einige Entwickler komplex sein.
Reduziert den Ressourcenverbrauch durch die Verwendung von Graal oder Substrate VM für autonome native Builds.	Weniger große Basis spezialisierter Entwickler, weniger Auswahl und möglicherweise längere Suche nach geeigneten Kandidaten.
Bietet sowohl imperative als auch reaktive Programmiermöglichkeiten.	Die Entwicklung kann teurer und langsamer sein im Vergleich zu Spring Boot.
Setzt auf AOT-Kompilierung und kann schneller laufen und arbeiten, wobei weniger Ressourcen verbraucht werden.	
Geringerer Speicherbedarf für native Images.	
Hervorragende, stets aktuelle und umfassende Dokumentation sowie wachsender Support durch die Community und führende Technologieunternehmen.	
Fokussiert sich auf qualitativ hochwertige Unterstützung für Drittanbieter-Integrationen.	

Tabelle 2.4: Vor- und Nachteile von Quarkus [17]

3 Methodik

In diesem Kapitel wird das methodische Vorgehen zur Analyse der Java-Frameworks Quarkus und Spring im Kontext der Microservices-Entwicklung beschrieben. Zunächst wird das Forschungsdesign erläutert, gefolgt von der Auswahl und Begründung der Fallstudie. Anschließend werden die Implementierungsansätze und die Testverfahren sowie Bewertungskriterien vorgestellt, die zur Analyse der Ergebnisse verwendet werden.

3.1 Forschungsdesign und Vorgehensweise

Das Forschungsdesign dieser Arbeit folgt einen Ansatz, der sowohl quantitative als auch qualitative Methoden kombiniert. Es geht darum, dass die Unterschiede zwischen den Java-Frameworks Quarkus und Spring in Bezug auf Performance, Skalierbarkeit und Entwicklererfahrung untersuchen. Diese Kombination von Methoden ermöglicht es, ein vollständiges Bild der Stärken und Schwächen der beiden Frameworks zu darstellen.

3.1.1 Forschungsansatz

- **Quantitative Analyse:** Die quantitative Analyse konzentriert sich auf die Sammlung messbarer Daten, um die Leistung der beiden Frameworks zu bewerten. Zu den wichtigsten Metriken gehören Antwortzeiten, CPU-Auslastung und Speicherverbrauch. Diese Metriken sind entscheidend, um objektive Vergleiche zwischen den beiden Frameworks zu nutzen. Hierfür werden spezielle Tools wie Locust für Performance-Tests und Lasttests verwendet. Die Ergebnisse dieser Tests liefern konkrete Zahlen, die zeigen, welches Framework unter bestimmten Bedingungen effizienter arbeitet.

- **Qualitative Analyse:** Zusätzlich zur quantitativen Analyse wird auch die qualitative Seite betrachtet. Hier geht es darum, wie angenehm und effektiv es ist, mit den Frameworks zu arbeiten. Dabei wird untersucht, wie einfach es ist, eine Anwendung zu implementieren, den Code zu warten und Anpassungen vorzunehmen. Die Entwicklererfahrung spielt eine wichtige Rolle, da sie direkt beeinflusst, wie schnell und effektiv Projekte umgesetzt werden können.

3.1.2 Vorgehensweise

Die Vorgehensweise in dieser Arbeit ist sorgfältig strukturiert, um sicherzustellen, dass alle wichtigen Aspekte der Untersuchung abgedeckt werden. Der Forschungsprozess besteht aus mehreren Schritten, die logisch aufeinander aufbauen. Diese Schritte werden in den nachfolgenden Unterkapiteln detailliert beschrieben, wobei jeder Schritt auf den vorherigen aufbaut, um ein vollständiges Bild zu erhalten.

- **Implementierung der PetClinic-Microservices-Anwendung:** Einer der zentralen Schritte dieser Arbeit ist die Implementierung der PetClinic-Microservices-Anwendung in Quarkus. Diese Implementierung wird in Abschnitt 3.2 ausführlich beschrieben. Ziel dieser Implementierung ist es, die Besonderheiten und Herausforderungen von Quarkus im Vergleich zu Spring Boot zu analysieren. Die bereits bestehende Spring-Version der PetClinic-Microservices-Anwendung dient dabei als Referenz, um direkte Vergleiche zwischen den beiden Frameworks ziehen zu können.
- **Verwendung von Tools und Techniken:**
  - **Entwicklungsumgebung:** Für die Entwicklung der Anwendung wurde IntelliJ IDEA als integrierte Entwicklungsumgebung (IDE) verwendet. Diese IDE ist besonders geeignet, da sie eine breite Unterstützung für beide Frameworks bietet und Entwicklern viele nützliche Funktionen zur Verfügung stellt, die die Arbeit erleichtern.
  - **Testing:** Zur Überprüfung der Funktionalität der REST-APIs wurde Postman eingesetzt, ein weit verbreitetes Tool für API-Tests. Für Unit-Tests wurde JUnit verwendet, um sicherzustellen, dass einzelne Komponenten der Anwendung korrekt funktionieren. Darüber hinaus kam Locust zum Einsatz, um die

Performance und Belastbarkeit der Anwendung unter verschiedenen Bedingungen zu testen. Diese Tests sind entscheidend, um die Leistungsfähigkeit der Frameworks unter realistischen Bedingungen zu bewerten.

- **Testverfahren:** Nach der Implementierung wurden vollständige Tests durchgeführt, um die Performance und Skalierbarkeit der Anwendungen zu bewerten. Diese Tests sind ein wichtiger Teil der Forschung, da sie konkrete Daten liefern, die zeigen, wie die beiden Frameworks in unterschiedlichen Szenarien abschneiden. Die Tests konzentrierten sich auf:
  - **Startzeit der Microservices:** Die Zeit, die die Microservices benötigen, um nach dem Start betriebsbereit zu sein, wurde ebenfalls gemessen. Dies ist besonders relevant für Szenarien, in denen Microservices dynamisch hoch- oder heruntergefahren werden, da eine kürzere Startzeit zu einer schnelleren Reaktion auf veränderte Lastanforderungen führt.
  - **Antwortzeiten:** Die Zeit, die die Anwendung benötigt, um auf Anfragen zu reagieren, wurde gemessen. Diese Messungen sind wichtig, um die Effizienz der Frameworks zu vergleichen und festzustellen, welches Framework schneller auf Benutzeranfragen reagiert.
  - **Skalierbarkeit** Zusätzlich wurde untersucht, wie gut die Anwendungen auf unterschiedliche Lasten reagieren können. Dieser Aspekt ist besonders wichtig, da ‚sie die Eignung der Frameworks für den Einsatz in großen Umgebungen bewerten.
  - **Ressourcennutzung:** Es wurde untersucht, wie viel CPU und Speicher während der Ausführung der Anwendung verbraucht werden. Diese Daten zeigen, wie gut die Frameworks mit den verfügbaren Ressourcen umgehen und welche Anforderungen sie an die Systemhardware stellen.

Mit dieser methodischen Vorgehensweise wird sichergestellt, dass die Untersuchung der beiden Frameworks vollständig und zuverlässig erfolgt. Die in den folgenden Abschnitten beschriebenen Schritte zeigen eine klare Struktur und ermöglichen es, die Ergebnisse der Forschung systematisch zu analysieren und zu bewerten. Dies bildet die Grundlage für gründliche Schlussfolgerungen und Empfehlungen, die in den abschließenden Kapiteln der Arbeit präsentiert werden.

3.2 Auswahl der Fallstudie: PetClinic-Microservices-Anwendung

Für diese Arbeit wurde die PetClinic-Microservices-Anwendung als Fallstudie ausgewählt. Die Wahl dieser speziellen Anwendung ist strategisch und methodisch begründet, da sie eine Vielzahl von Aspekten abdeckt, die für den Vergleich der Java-Frameworks Quarkus und Spring relevant sind.

3.2.1 Hintergrund und Überblick

Die Spring PetClinic ist eine Beispielanwendung, die ursprünglich entwickelt wurde, um die Leistungsfähigkeit des Spring Frameworks zu demonstrieren. Sie simuliert die Abläufe einer Tierklinik und deckt Funktionen wie die Verwaltung von Kunden, Haustieren und Tierärzten ab. Ursprünglich als monolithische Anwendung gestartet, wurde sie in eine Microservices-Architektur umgeformt, um die Vorteile von Spring Boot und Spring Cloud in verteilten Systemen zu veranschaulichen [32].

Die Microservices-Version nutzt verschiedene Spring-Komponenten, darunter Spring Cloud Config Server und Eureka Service Discovery, um dynamische Skalierbarkeit und flexible Anpassung zu ermöglichen. Ein API Gateway dient als zentraler Zugangspunkt, der Anfragen an die jeweiligen Microservices weiterleitet. Monitoring-Tools wie der Spring Boot Admin Server und Zipkin überwachen die verteilten Systeme [1].

Die Wahl dieser Anwendung als Fallstudie ist besonders geeignet, da sie ein praxisnahes Beispiel bietet, um die Performance und Flexibilität der Java-Frameworks Quarkus und Spring zu vergleichen

3.2.2 Warum PetClinic-Microservices?

Die PetClinic-Microservices-Anwendung wurde aus mehreren Gründen als Fallstudie ausgewählt:

- **Repräsentative Beispielanwendung:** Die PetClinic-Anwendung ist ein weit verbreitetes und anerkanntes Beispiel, das in der Softwareentwicklungs-Community

häufig verwendet wird, um die Funktionalitäten des Spring Frameworks zu demonstrieren. Sie bietet eine standardisierte Grundlage, die sowohl für Bildungszwecke als auch für den praktischen Einsatz geeignet ist. Ihre lange Geschichte und breite Akzeptanz machen sie zu einer zuverlässigen Wahl für eine Fallstudie, die verschiedene Java-Frameworks vergleicht.

- **Vollständige Nutzung von Spring:** Die PetClinic-Microservices-Version nutzt das volle Potenzial des Spring-Ökosystems, einschließlich Spring Boot, Spring Cloud Config Server und Eureka Service Discovery. Diese Komponenten sind wesentliche Bestandteile moderner Microservices-Architekturen, die es ermöglichen, Anwendungen dynamisch zu skalieren und flexibel auf sich ändernde Anforderungen zu reagieren. Die Implementierung dieser Technologien in der PetClinic-Anwendung zeigt, wie eine realistische Unternehmensanwendung aufgebaut werden kann.
- **Leichte Anpassbarkeit und öffentliche Verfügbarkeit:** Ein weiterer wichtiger Grund für die Wahl dieser Anwendung ist ihre öffentliche Verfügbarkeit auf GitHub. Der Quellcode ist offen zugänglich und gut dokumentiert, was die Anpassung und Erweiterung für spezifische Testzwecke erleichtert. Entwickler können die Anwendung leicht an ihre Bedürfnisse anpassen und die Ergebnisse in verschiedenen Szenarien vergleichen.
- **Dokumentation und Community-Unterstützung:** Die PetClinic-Microservices-Anwendung wird von einer großen Entwicklergemeinschaft unterstützt, die kontinuierlich zur Verbesserung und Aktualisierung der Anwendung beiträgt. Diese Unterstützung stellt sicher, dass die Anwendung immer auf dem neuesten Stand der Technik ist und erleichtert es Entwicklern, Probleme zu lösen und neue Funktionen zu implementieren. Mit umfangreicher Dokumentation ist perfekt für alle, die die Anwendung kennenlernen oder anpassen wollen. Sie bietet viele Beispiele und Erklärungen.

3.2.3 Ziel der Fallstudie

Das Hauptziel dieser Fallstudie ist es, dass eine entsprechende Vergleichbasis zu schaffen, auf der die Unterschied zwischen Quarkus und Spring Boot in einem praktischen Umfeld getestet werden können. Mittels die Implementierung der Petclinic-Microservices-Anwendung in Quarkus und den Vergleich mit der vorhandenen Spring Boot Version können konkrete Erkenntnisse über die Stärken und Schwächen beider Frameworks gewon-

nen werden. Besonders wird untersucht, wie gut die Frameworks in einer Microservices-Architektur performen, wie einfach sie zu implementieren und zu warten sind und welche Skalierungsmöglichkeiten sie bieten.

3.3 Implementierungsansatz

In diesem Abschnitt wird detailliert beschrieben, wie die PetClinic-Microservices-Anwendung in Quarkus implementiert wurde. Der Fokus liegt auf der Entwicklung der einzelnen Services, der Nutzung von Consul für die Service-Registrierung und den Unterschieden zur ursprünglichen Spring-Implementierung.

3.3.1 Implementierung der Microservices in Quarkus

Die Implementierung der PetClinic-Microservices-Anwendung begann mit der schrittweisen Entwicklung der einzelnen Microservices, und jeder Service wurde sorgfältig in Quarkus umgesetzt.

Customers-Service
Dieser Service speichert alle Daten zu den Tierbesitzern und ihren Haustieren. Zuerst wurden die Datenstrukturen, wie Owner, Pet und PetType, erstellt. Danach wurden REST-APIs entwickelt, die es ermöglichen, diese Daten zu erstellen, abzurufen, zu ändern und zu löschen (CRUD-Operationen). Die Verbindung zur Datenbank wurde mit JPA/Hibernate umgesetzt, das durch das Panache-Framework von Quarkus unterstützt wird. Für die Entwicklung und das Testen wurde eine In-Memory-Datenbank H2 verwendet, die es erlaubt, Daten schnell zu speichern und abzurufen. In der produktiven Umgebung wird eine MySQL-Datenbank verwendet, um die Daten dauerhaft zu speichern.

Vets-Service
Dieser Service verwaltet die Daten der Tierärzte und ihrer Fachgebiete. Die Datenstrukturen, wie Vet und Specialty, wurden zuerst erstellt. Anschließend wurden REST-APIs entwickelt, die CRUD-Operationen ermöglichen. Die Datenbankanbindung erfolgte ebenfalls mit JPA/Hibernate und dem Panache-Framework. H2 wurde für die Entwicklung und Tests genutzt, während MySQL in der Produktivumgebung eingesetzt wird.

Visits-Service
Dieser Service kümmert sich um die Verwaltung von Besuchen und Terminen der Haustiere bei den Tierärzten. Die Implementierung begann mit der Erstellung der Visit-Datenstrukturen, gefolgt von der Entwicklung der REST-APIs für CRUD-Operationen. Auch hier kam JPA/Hibernate zusammen mit Panache zum Einsatz. Während der Entwicklung und für Tests wurde H2 verwendet, aber für den Betrieb wird MySQL genutzt, um die Daten dauerhaft zu speichern.

Jeder dieser Services ist so gestaltet, dass er unabhängig arbeiten kann, was eine zentrale Eigenschaft der Microservices-Architektur ist. Diese Unabhängigkeit erleichtert die Wartung und Weiterentwicklung der Anwendung, da Änderungen an einem Service keine direkten Auswirkungen auf die anderen haben.

3.3.2 Nutzung von Consul zur Service-Registrierung

Ein wichtiger Teil der Implementierung in Quarkus war die Integration von Consul zur Registrierung und Erkennung der Microservices. Da Quarkus keine native Unterstützung für Eureka bietet, wurde Consul als Alternative eingesetzt.

Service-Registrierung mit Consul
Für jeden der erstellten Services (Customers, Vets und Visits) wurde eine Consul-Integration hinzugefügt. Dies bedeutete, dass die Services sich automatisch bei Consul registrieren, sobald sie gestartet werden. Diese Registrierung ermöglicht es anderen Services und dem API Gateway, die registrierten Services zu finden und mit ihnen zu kommunizieren. Im Gegensatz zur ursprünglichen Spring-Version, die Eureka für die Service-Erkennung verwendet, erforderte die Nutzung von Consul einige Anpassungen in der Konfiguration und im Management der Services.

API Gateway mit Consul-Registrierung
Zusätzlich zu den einzelnen Services wurde auch ein API Gateway in Quarkus entwickelt. Dieses Gateway dient als zentraler Zugangspunkt für alle Anfragen, die an die Microservices weitergeleitet werden. Auch das API Gateway wurde bei Consul registriert, um sicherzustellen, dass es korrekt in die Microservices-Architektur integriert ist.

Durch die Verwendung von Consul konnte die Service-Registrierung und -Erkennung in Quarkus effektiv umgesetzt werden. Diese Implementierung ermöglichte eine dynamische

Kommunikation zwischen den Services und stellte sicher, dass die Anwendung in einer Microservices-Architektur stabil läuft.

3.3.3 Unterschiede zur Spring-Implementierung

Bei der Implementierung in Quarkus traten einige wesentliche Unterschiede zur ursprünglichen Spring-Version auf, die die Architektur und Funktionalität beeinflussten:

Discovery-Service
In der ursprünglichen Spring-Version wird Eureka als Discovery-Service verwendet. Da Quarkus Eureka nicht unterstützt, wurde Consul als Alternative genutzt. Diese Änderung erforderte Anpassungen in der Art und Weise, wie die Services registriert und entdeckt werden. Trotz dieser Anpassungen funktionierten die Microservices in Quarkus genauso zuverlässig wie in der ursprünglichen Spring-Version.

Admin-Server
In der Spring Boot Version wird ein Spring Boot Admin Server zur Überwachung der Microservices verwendet. In Quarkus wurde ein solcher Admin-Server nicht implementiert, da Quarkus keine direkte Unterstützung für ein vergleichbares Überwachungstool bietet. Dies führte dazu, dass alternative Methoden zur Überwachung in Betracht gezogen oder auf bestimmte Überwachungsfunktionen verzichtet werden musste.

Config-Server
In der Spring Boot Version wird ein Spring Cloud Config Server verwendet, um zentrale Konfigurationen für die Microservices zu verwalten. Diese zentrale Verwaltung ermöglicht es, Konfigurationen an einem Ort zu speichern und bei Bedarf in allen Services zu aktualisieren. Da Quarkus diese Funktionalität nicht direkt unterstützt, wurde in der Quarkus-Implementierung darauf verzichtet. Dies führte zu einer vereinfachten Architektur, bedeutete jedoch auch, dass die Konfigurationen direkt in den Services verwaltet werden mussten.

Diese Unterschiede beeinflussten die Implementierung und den Betrieb der Microservices in Quarkus. Es war notwendig, auf bestimmte Funktionen zu verzichten oder alternative Lösungen wie Consul zu verwenden, um die Anwendung in Quarkus erfolgreich umzusetzen.

3.3.4 Zusammenfassung der Implementierung

Die Umsetzung der PetClinic-Microservices-Anwendung in Quarkus konzentrierte sich auf die Kernfunktionen der einzelnen Services und deren Integration in eine Microservices-Architektur mit Consul als zentralem Discovery-Service. Trotz der Unterschiede zur ursprünglichen Spring-Implementierung konnten die wichtigsten Funktionen beibehalten werden, was den Vergleich der beiden Frameworks ermöglicht. Die verschiedenen Unterstützungen von Tools und Funktionen zwischen Quarkus und Spring erforderten jedoch spezifische Anpassungen und alternative Ansätze, um die Anwendung erfolgreich in Quarkus zu betreiben.

4 Komparative Analyse

In diesem Kapitel werden die beiden Frameworks Spring und Quarkus für das PetClinic-Microservices-Projekt verglichen. Es wird untersucht, wie die Anwendungsarchitektur, die Datenmodellierung, die Repository-Implementierung, die Datenbankkonfiguration, die REST-APIs, die Service Discovery und das API-Gateway in beiden Frameworks umgesetzt wurden. Anhand von Beispielen aus der Implementierung wird auch die Performance und Skalierbarkeit der Frameworks betrachtet.

4.1 Vergleich der Anwendungsentwicklung

In diesem Abschnitt wird die Entwicklung der PetClinic-Anwendung in Spring und Quarkus verglichen. Zunächst betrachten wir die Architektur beider Frameworks, gefolgt von der Datenmodellierung und der Repository-Implementierung. Anschließend vergleichen wir die Datenbankkonfigurationen und die Umsetzung der REST-APIs. Zum Schluss untersuchen wir die Service Discovery-Lösungen und das API Gateway in beiden Frameworks.

4.1.1 Vergleich der Architektur

Die folgenden Diagramme zeigen die Architektur der PetClinic-Microservices-Anwendung, einmal mit Spring und einmal mit Quarkus implementiert. Beide Architekturen bestehen aus mehreren Microservices, die über ein API Gateway miteinander verbunden sind. Dabei spielt die Service Discovery eine wichtige Rolle: Bei Spring wird Eureka verwendet, während bei Quarkus Consul zum Einsatz kommt.

Spring Boot Architektur

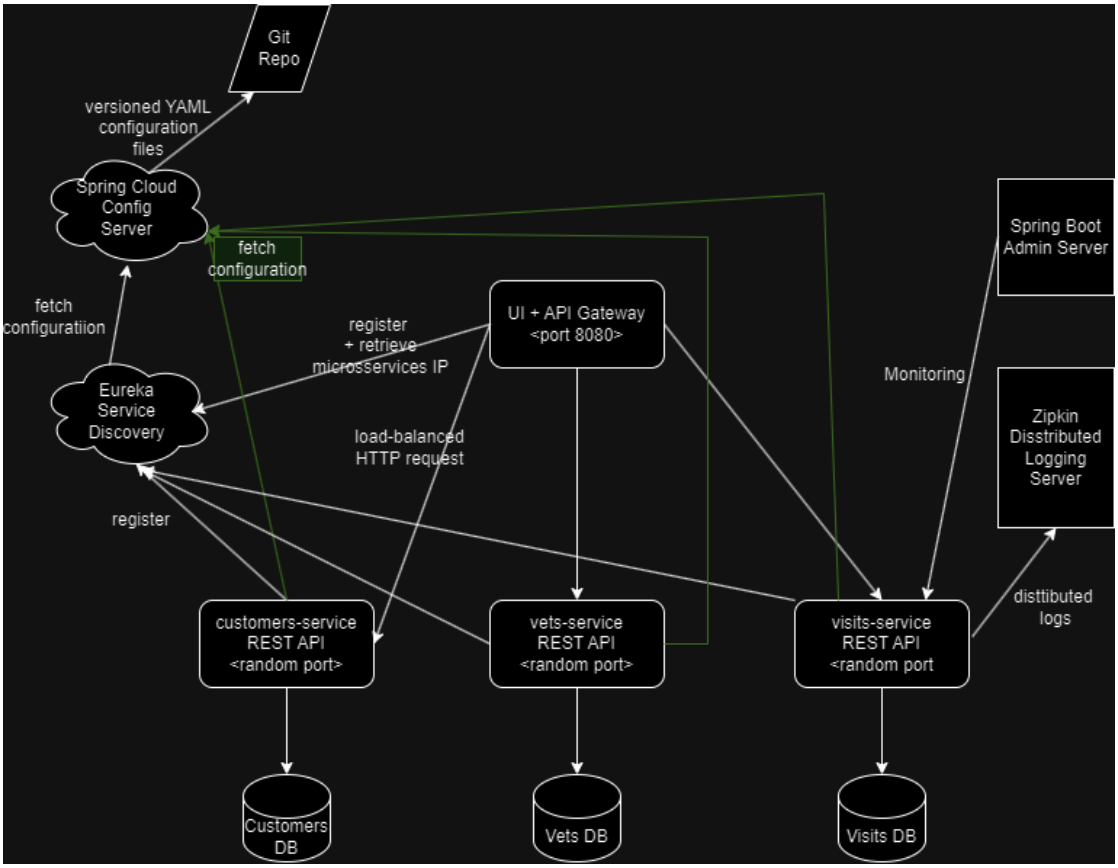


Abbildung 4.1: Spring Petclinic-Microservices-Anwendung Architektur. Quelle: [32]

wie in Abbildung 4.1 die Service Discovery in Spring Boot Implementierung ist mit Eureka vrwendet. Alle Microservices (Customers-Service, Vets-Service, Visits-Service) registrieren sich bei Eureka, das die Service-Informationen speichert und diese Informationen dem API Gateway zur Verfügung stellt. Das API Gateway nutzt diese Informationen, um die Anfragen an die richtigen Microservices weiterzuleiten. Diese Kommunikation erfolgt über HTTP-Anfragen, die von Eureka load-balanced werden.
Spring Cloud Config Server ist ein wichtiges Element in Spring Architektur, der zentrale Konfigurationsdateien (YAML) aus einem Git-Repository bezieht und diese Konfigurationen an die Microservices verteilt. Dadurch können Konfigurationen zentral verwaltet und bei Bedarf dynamisch aktualisiert werden, was die Wartbarkeit der Anwendung verbessert.
Die Überwachung und das Monitoring der Microservices erfolgt über den Spring Boot

Admin Server und Zipkin. Der Admin Server überwacht den Zustand der Microservices, während Zipkin für das verteilte Tracing und Logging verantwortlich ist, was es einfacher macht, Anfragen zu verfolgen und Fehler zu diagnostizieren.

Quarkus Architektur

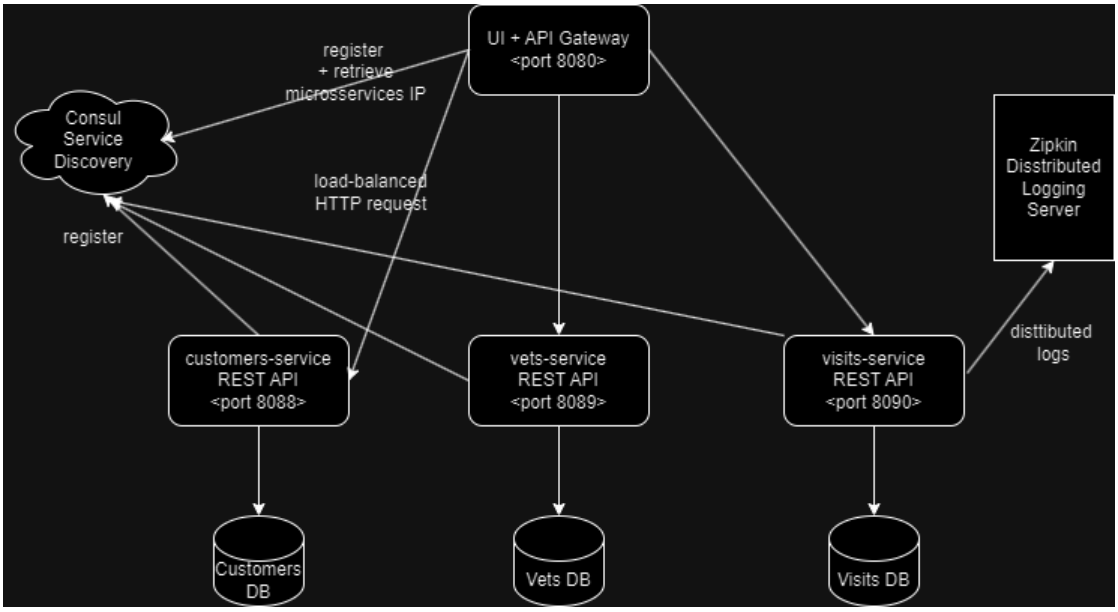


Abbildung 4.2: Quarkus Petclinic-Microservices-Anwendung Architektur.

Da in der Abbildung 4.2 die Service Dicovery in Quarkus-Implementierung mit Consul übernimmt. Ähnlich wie bei Eureka registrieren sich alle Microservices bei Consul. Das API Gateway nutzt Consul, um die Service-Informationen abzurufen und Anfragen entsprechend zu verteilen. Auch hier wird das Load-Balancing über Quarkus Stork verwaltet, allerdings ohne die Notwendigkeit eines separaten Konfigurationsservers wie in der Spring-Architektur.

Im Gegensatz zur Spring Boot fehlen 2 Komponenten in der Quarkus-Architektur der Config Server und der Admin Server.

Vergleich der beiden Architekturen

Die Spring-Architektur ist aufgrund der Integration von Eureka, Spring Cloud Config Server und Spring Boot Admin Server umfassender und bietet erweiterte Funktionen für Service Discovery, zentrale Konfigurationsverwaltung und Monitoring. Dies macht die

Architektur komplexer, aber auch leistungsfähiger in größeren, verteilten Systemen. Die Quarkus-Architektur ist schlanker und verzichtet auf einige Komponenten, die in der Spring-Architektur vorhanden sind, Das liegt daran, dass Quarkus bestimmte Funktionen wie den Spring Cloud Config Server und den Spring Boot Admin Server nicht unterstützt.

Vergleich der Struktur von customers-service in der IDE

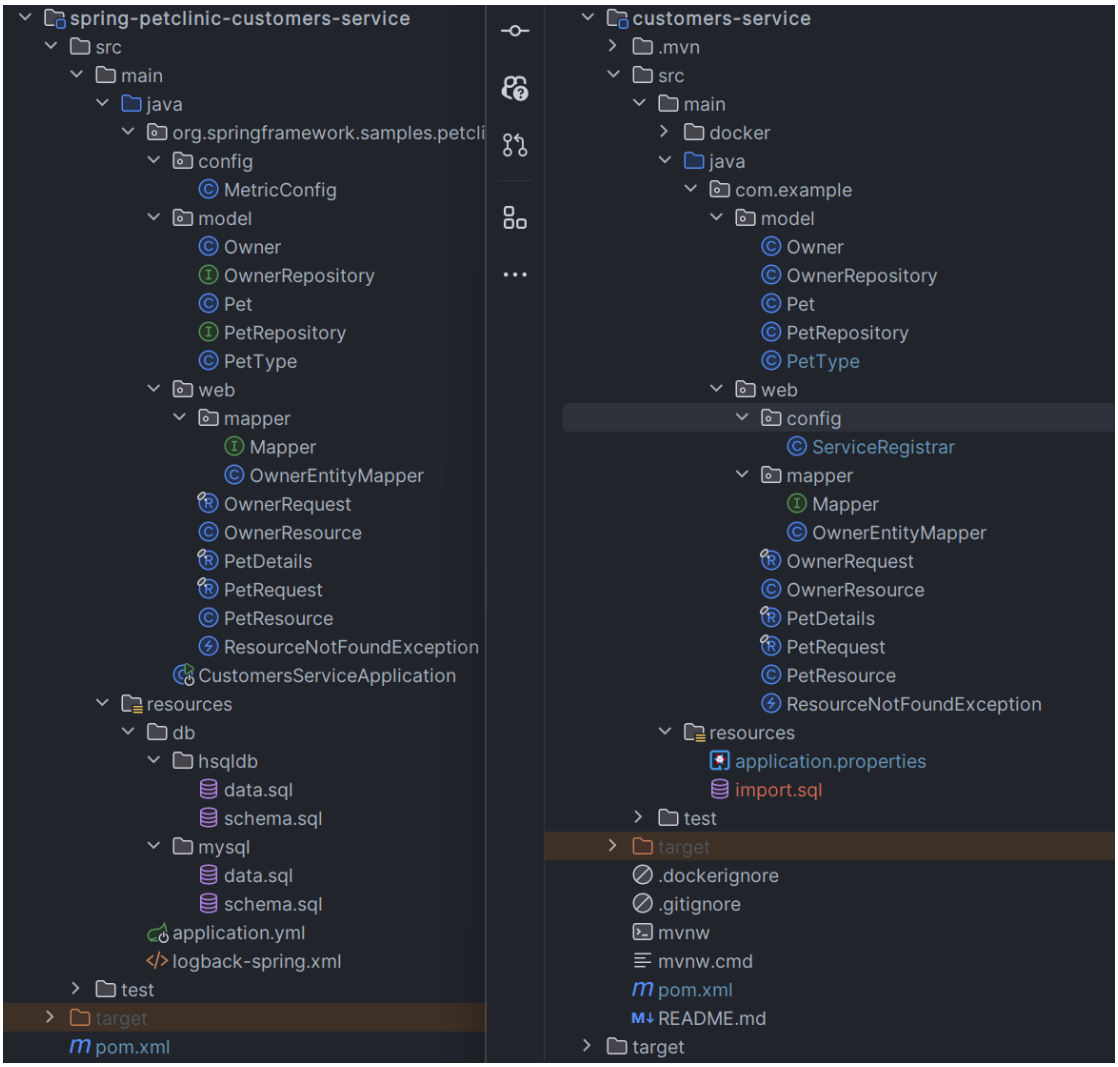


Abbildung 4.3: Struktur von customers-service in beiden Frameworks

Hinsichtlich der Projektstruktur und -größe weisen beide Implementierungen signifikante Ähnlichkeiten auf, was auf einen vergleichbaren Wartungsbedarf in der Zukunft hindeu-

tet. Dennoch gibt es einige bemerkenswerte Unterschiede. So enthält die Spring-Version beispielsweise eine MetricConfig-Klasse, die für das Sammeln und Überwachen von Anwendungsmetriken zuständig ist. Diese Klasse nutzt Micrometer, um Leistungsdaten zu erfassen, was besonders nützlich für das Monitoring der Anwendung ist. In der Quarkus-Version fehlt eine direkte Entsprechung dieser Klasse, allerdings werden alternative Methoden für das Monitoring eingesetzt.

Ein weiterer Unterschied besteht darin, dass in der Quarkus-Implementierung keine Application-Klasse wie CustomersServiceApplication erforderlich ist, die eine Hauptmethode enthält, wie es in Spring Boot der Fall ist. Quarkus generiert standardmäßig eine Main-Methode, um den Bootstrapping-Prozess zu erleichtern und wartet anschließend auf das Herunterfahren der Anwendung [22]. Darüber hinaus wird in der Quarkus-Version die ServiceRegistrar-Klasse verwendet, um den CustomersService bei Consul zu registrieren, was die Service Discovery ermöglicht. Ein ähnlicher Mechanismus wird auch für den VisitsService und VetsService angewendet. Diese spezifische Implementierung wird im 4.1.6 Vergleich der Discovery-Services Abschnitt ausführlicher behandelt.

4.1.2 Vergleich der Datenmodellierung

In diesem Abschnitt wird die Datenmodellierung des Owner-Datenmodells im customers-service in Spring und Quarkus verglichen, einschließlich der Verwendung von JPA-Annotationen sowie Lombok und Panache.

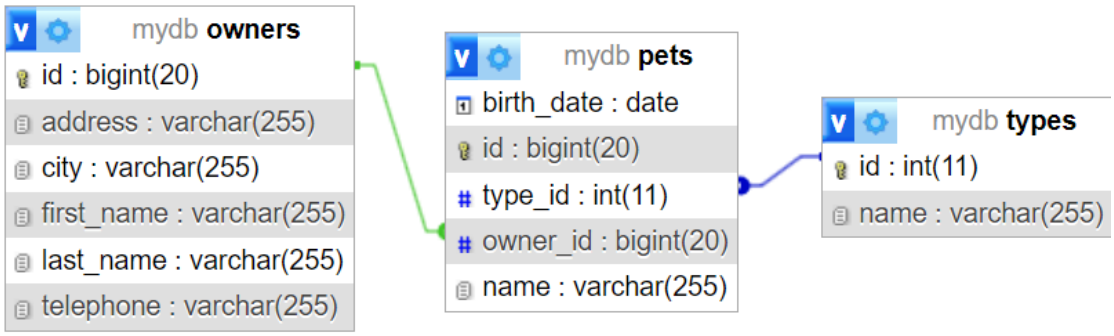


Abbildung 4.4: Datenmodell des customers-services

Wie in Abbildung 4.5 gezeigt, sind die Owner-Datenmodelle in Spring und Quarkus sehr ähnlich. Beide nutzen fast die gleichen JPA-Annotationen, um die Daten zu definieren und zu prüfen. Die Art, wie die Beziehungen zwischen Owner und Pet modelliert werden, ist ebenfalls fast gleich. Der Hauptunterschied liegt darin, dass Quarkus mit Panache weniger Code benötigt, da CRUD-Operationen direkt in der Entität durchgeführt werden können, ohne ein separates Repository zu brauchen. In der aktuellen Implementierung wird jedoch trotzdem ein Repository verwendet, um mehr Flexibilität und eine klarere Struktur zu haben. In Spring werden CRUD-Operationen immer über separate Repository-Klassen wie JpaRepository ausgeführt, was oft zu mehr Code führt.

Abbildung 4.5: Vergleich der owner-Klasse von customers-service in beiden Frameworks

4.1.3 Vergleich der Repository-Implementierungen

In diesem Abschnitt wird verglichen, wie der Datenzugriff in Spring und Quarkus umgesetzt wird. Dazu werden die OwnerRepository- und PetRepository-Klassen betrachtet, die für die Verwaltung und den Zugriff auf die Daten in der PetClinic-Anwendung zuständig sind.

Spring Data JPA Repository

```
public interface OwnerRepository extends JpaRepository<Owner, Integer> { }  5 usages  Dapeng
```

Abbildung 4.6: Spring Boot OwnerRepository

```
public interface PetRepository extends JpaRepository<Pet, Integer> {  3 usages  Dapeng +2

    /**
     * Retrieve all {@link PetType}s from the data store.
     * @return a Collection of {@link PetType}s.
     */
    @Query("SELECT ptype FROM PetType ptype ORDER BY ptype.name")  1 usage  Dapeng
    List<PetType> findPetTypes();

    @Query("FROM PetType ptype WHERE ptype.id = :typeId")  1 usage  Maciej Szarlinski
    Optional<PetType> findPetTypeById(@Param("typeId") int typeId);


}
```

Abbildung 4.7: Spring Boot PetRepository

In Spring wird für den Datenzugriff häufig Spring Data JPA verwendet. JPA steht für Java Persistence API und ist eine Schnittstelle, die es Entwicklern ermöglicht, Datenbankoperationen in Java-Anwendungen zu verwalten, ohne dass sie sich um die Details der Datenbank kümmern müssen [9]. Das OwnerRepository in Spring erbt von JpaRepository<Owner, Integer>. Dadurch werden Standard-Datenbankoperationen wie Erstellen, Lesen, Aktualisieren und Löschen (CRUD) automatisch bereitgestellt, ohne dass zusätzlicher Code geschrieben werden muss. Dies macht die Arbeit für Entwickler einfacher und schneller.

Zusätzlich können in Spring benutzerdefinierte Abfragen mit der @Query-Annotation definiert werden. Zum Beispiel enthält das PetRepository Methoden wie findPetTypes() und findPetTypeById(int typeId). Diese Abfragen werden durch @Query-Annotationen definiert, die eine spezifische Datenbankabfrage festlegen. Diese Flexibilität ist einer der großen Vorteile von Spring Data JPA.

Quarkus Panache Repository

```
@ApplicationScoped  berraqui
public class OwnerRepository implements PanacheRepository<Owner> {

}
```

Abbildung 4.8: Quarkus OwnerRepository

```
@ApplicationScoped  berraqui
public class PetRepository implements PanacheRepository<Pet> {
    public List<PetType> findPetTypes() {  1 usage  berraqui
        return getEntityManager().createQuery( s: "SELECT ptype FROM PetType ptype ORDER BY ptype.name", PetType.class)
                .getResultList();
    }

    /**
     * Find a PetType by its ID.
     * @param typeId the ID of the PetType.
     * @return an Optional containing the PetType if found, or empty if not found.
     */
    public Optional<PetType> findPetTypeById(int typeId) {  1 usage  berraqui
        return getEntityManager().createQuery( s: "FROM PetType ptype WHERE ptype.id = :typeId", PetType.class) TypedQuery<PetType>
                .setParameter( s: "typeId", typeId)
                .getResultStream() Stream<PetType>
                .findFirst();
    }
}
```

Abbildung 4.9: Quarkus PetRepository

In Quarkus wird der Datenzugriff durch Panache vereinfacht. Panache ist eine Quarkus-spezifische Bibliothek, die die Entwicklung Ihrer Hibernate-basierten Persistenzschicht vereinfacht. Ähnlich wie Spring Data JPA übernimmt Panache den größten Teil des sich wiederholenden Boilerplate-Codes für Sie [15]. Das OwnerRepository in Quarkus implementiert PanacheRepository<Owner>. Dadurch sind Standard-Datenbankoperationen, wie bei Spring, sofort verfügbar, ohne dass zusätzliche Methoden geschrieben werden müssen. Der Code ist dadurch kürzer und einfacher zu lesen.

Für komplexere Abfragen, wie sie im PetRepository benötigt werden, verwendet Quarkus den EntityManager direkt. Im Gegensatz zu Spring, wo diese Abfragen durch Annotationen definiert werden, werden in Quarkus die Abfragen direkt im Code geschrieben. Zum Beispiel werden die Methoden findPetTypes() und findPetTypeById(int typeId) direkt im Repository definiert, indem der EntityManager verwendet wird, um die Datenbankabfragen durchzuführen. Dies bietet mehr Kontrolle über den Datenzugriff, erfordert aber auch ein tieferes Verständnis von Datenbankoperationen.

4.1.4 Vergleich der Datenbankkonfigurationen

In diesem Abschnitt wird die Konfiguration von In-Memory-Datenbanken in den Frameworks Spring und Quarkus verglichen. Es wird aufgezeigt, wie beide Frameworks diese Datenbanken einrichten und welche Unterschiede es dabei gibt.

Quarkus H2-Datenbankkonfiguration

H2 ist eine leichtgewichtige, in Java geschriebene relationale Datenbank. Sie läuft vollständig im Speicher und eignet sich hervorragend für Tests oder Entwicklungsumgebungen, da sie schnell ist und keine externe Installation benötigt [5]. In Quarkus wird die H2-Datenbank direkt in der Datei application.properties konfiguriert. Die H2-Datenbank läuft im Speicher, was bedeutet, dass die Daten nur während der Laufzeit verfügbar sind und bei jedem Neustart der Anwendung gelöscht werden. Diese Art der Datenbankkonfiguration eignet sich besonders gut für Tests, da die Datenbank bei jedem Start frisch ist.

```
# Datasource configuration
quarkus.datasource.db-kind=h2
quarkus.datasource.jdbc.url=jdbc:h2:mem:default;DB_CLOSE_DELAY=-1
# Hibernate ORM configuration
quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.hibernate-orm.sql-load-script=import.sql
```

Abbildung 4.10: Quarkus H2-Datenbankkonfiguration

In dieser Konfiguration wird festgelegt, dass die H2-Datenbank im Speicher (mem) läuft und dass die Datenbank bei jedem Neustart gelöscht und neu erstellt wird (drop-and-create). Das SQL-Skript (import.sql) wird verwendet, um die Datenbank mit Testdaten

zu füllen.

Spring HSQLDB-Datenbankkonfiguration

HSQLDB ist ebenfalls eine relationale In-Memory-Datenbank, die vollständig in Java implementiert wurde. HSQLDB ist sehr schnell und flexibel und wird oft in Spring-Projekten verwendet, um schnelle Datenbanklösungen für Tests zu bieten [11]. In Spring wird die Konfiguration der HSQLDB-Datenbank über den Spring Cloud Config Server bereitgestellt, der die Konfigurationsdateien zentral verwaltet. Anstatt die Datenbankeinstellungen direkt in der lokalen application.yml-Datei zu speichern, greift Spring auf den Config Server zu, um die notwendigen Informationen zu erhalten. Dies bietet den Vorteil, dass die Konfiguration zentral geändert werden kann und sich diese Änderungen automatisch auf alle Microservices auswirken.

```yaml
spring:
  application:
    name: customers-service
  config:
    import: optional:configserver:${CONFIG_SERVER_URL:http://localhost:8888/}


---
spring:
  config:
    activate:
      on-profile: docker
    import: configserver:http://config-server:8888
```

Abbildung 4.11: Spring Application.yml von customers-service

Ein Beispiel für eine solche Konfiguration (siehe die Abbildung 4.12) wird unter der URL http://localhost:8888/customers-service-docker.yml abgerufen. Dabei wird die In-Memory-Datenbank HSQLDB so eingerichtet, dass beim Start der Anwendung die Datenbank automatisch mit einem vordefinierten Schema aus der Datei schema.sql und Testdaten aus der Datei data.sql initialisiert wird. Beide Dateien liegen im Ressourcenverzeichnis (classpath*:db/hsqldb/). Diese Konfiguration sorgt dafür, dass die Datenbank bei jedem Start frisch initialisiert wird, was besonders für Tests nützlich ist, ähnlich

wie bei Quarkus, wo ebenfalls eine frische Datenbankinstanz für jeden Testlauf erzeugt wird.

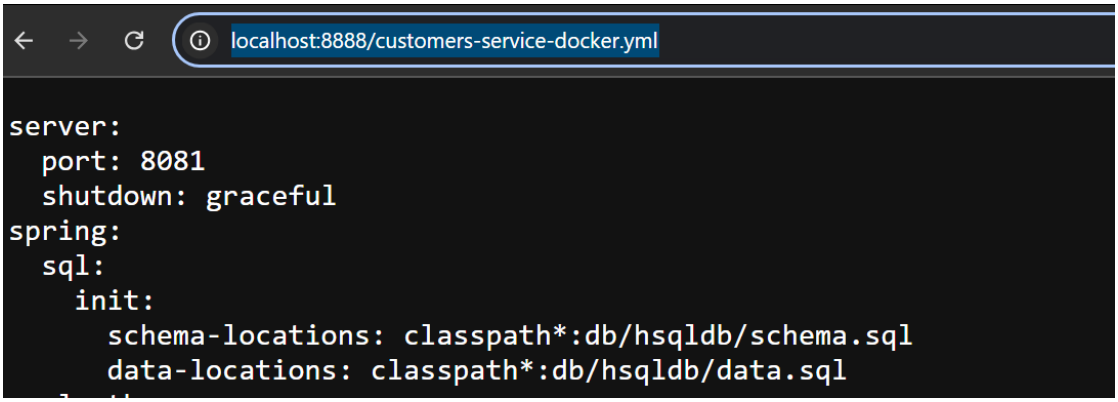


Abbildung 4.12: Spring Boot Konfiguration von customers-service

Vergleich der beiden Konfigurationen

- **Datenbanktypen:** In Quarkus wird die H2-Datenbank verwendet, während in Spring die HSQLDB-Datenbank zum Einsatz kommt. Beide Datenbanken sind In-Memory-Datenbanken, die keine externe Installation benötigen und gut für Tests geeignet sind.
- **Datenbank-Initialisierung:** In Quarkus wird die H2-Datenbank direkt in der Anwendung konfiguriert und bei jedem Neustart der Anwendung gelöscht und neu erstellt. In Spring hingegen wird die HSQLDB-Datenbank über den Config Server konfiguriert. Das bedeutet, dass die Konfiguration zentral verwaltet und von dort abgerufen wird.
- **Zentrale Konfiguration in Spring:** Ein Vorteil von Spring ist die zentrale Verwaltung der Konfigurationen über den Spring Cloud Config Server. Dadurch können Änderungen an der Konfiguration zentral vorgenommen werden und müssen nicht in jeder einzelnen Anwendung vorgenommen werden. Dies ist besonders in großen Microservices-Architekturen von Vorteil. Quarkus hingegen speichert die Konfiguration lokal in jeder Anwendung, was zwar einfacher ist, aber weniger Flexibilität bietet, wenn Änderungen für mehrere Anwendungen vorgenommen werden müssen.

4.1.5 Vergleich der REST-APIs

In diesem Abschnitt wird die Implementierung der REST-APIs zwischen Quarkus und Spring verglichen. Beide Frameworks ermöglichen die Erstellung von RESTful Webservices, dennoch unterscheiden sich die Ansätze in einigen Details.

Quarkus REST-APIs

In Quarkus werden die REST-APIs mit den Jakarta RESTful Web Services umgesetzt [23]. Hierbei wird die Annotation @Path verwendet, um festzulegen, unter welcher URL die API erreichbar ist. Um verschiedene HTTP-Methoden wie GET, POST und PUT zu unterstützen, nutzt Quarkus die entsprechenden Annotationen @GET, @POST und @PUT. Die JSON-Daten, die von der API gesendet und empfangen werden, werden mit den Annotationen @Consumes(MediaType.APPLICATION_JSON) und @Produces(MediaType.APPLICATION_JSON) angegeben. Dadurch wird sichergestellt, dass die API sowohl Anfragen im JSON-Format annimmt als auch Antworten im JSON-Format zurückgibt.

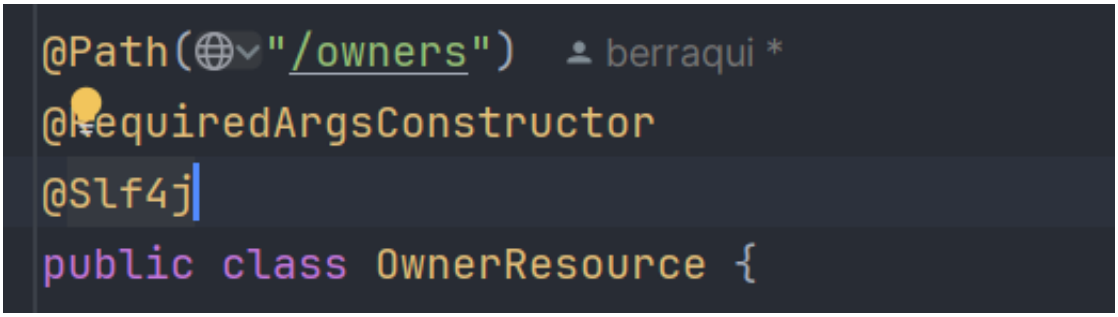


Abbildung 4.13: Verwendung Pfad-Annotation in Ressource in Quarkus

Ein Beispiel für eine **GET-Anfrage** in Quarkus sieht folgendermaßen aus:

```
@GET  berraqui
@Path("/{ownerId}")
@Produces(MediaType.APPLICATION_JSON)
@Timed(name = "owner_find_time", description = "Zeit für das Abrufen eines Besitzers")
public Response findOwner(@PathParam("ownerId") @Min(1) long ownerId) {
```

Abbildung 4.14: GET-Anfrage in Quarkus

Das Pendant für eine **POST-Anfrage** sieht ähnlich wie @GET aus, verwendet aber @POST und @Transactional, um eine neue Ressource zu erstellen und sicherzustellen, dass die Operation in einer Transaktion erfolgt:

```
@POST  berraqui
@Transactional
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Timed(name = "owner_creation_time", description = "Zeit für die Erstellung eines Besitzers")
public Response createOwner(@Valid OwnerRequest ownerRequest) {
```

Abbildung 4.15: Post-Anfrage in Quarkus

Die **PUT-Anfrage**, die zur Aktualisierung bestehender Daten dient, sieht folgendermaßen aus:

```
@PUT  berraqui *
@Path("/{ownerId}")
@Transactional
@Consumes(MediaType.APPLICATION_JSON)
@Timed(name = "owner_update_time", description = "Zeit für das Aktualisieren eines Besitzers")
public Response updateOwner(@PathParam("ownerId") @Min(1) long ownerId, @Valid OwnerRequest ownerRequest) {
```

Abbildung 4.16: Put-Anfrage in Quarkus

Spring Boot REST-APIs
in Spring Boot verwendet die Annotation @RestController, die eine spezialisierte Version des Controllers darstellt. Sie kombiniert die Funktionalitäten der Annotationen @Controller und @ResponseBody, was bedeutet, dass die Rückgabe von Daten automatisch in HTTP-Antworten serialisiert wird, ohne dass zusätzliche Konfiguration notwendig ist [2]. Zudem nutzt Spring spezialisierte Annotationen wie @GetMapping, @PostMapping und @PutMapping, um die entsprechenden HTTP-Methoden klar und deutlich abzubilden.

Ein Beispiel für eine GET-Methode in Spring sieht folgendermaßen aus:

```
@RequestMapping("/owners")  michaelisvy +6
@RestController
@Timed("petclinic.owner")
@RequiredArgsConstructor
@Slf4j
class OwnerResource {
```

Abbildung 4.17: Keine Verwendung der Path-Annotation im Controller in Spring Boot

Ein Beispiel für eine GET-Methode in Spring sieht folgendermaßen aus:

```
@GetMapping(value = "/{ownerId}")  Antoine Rey +1
public Optional<Owner> findOwner(@PathVariable("ownerId") @Min(1) int ownerId) {
```

Abbildung 4.18: GET-Anfrage in Spring Boot

Für die POST-Methode wird @PostMapping verwendet, um eine neue Ressource zu erstellen:

```
@PostMapping  Shobha Kamath +1
@ResponseStatus(HttpStatus.CREATED)
public Owner createOwner(@Valid @RequestBody OwnerRequest ownerRequest) {
```

Abbildung 4.19: POST-Anfrage in Spring Boot

Die PUT-Methode in Spring zur Aktualisierung von Daten sieht ähnlich aus:

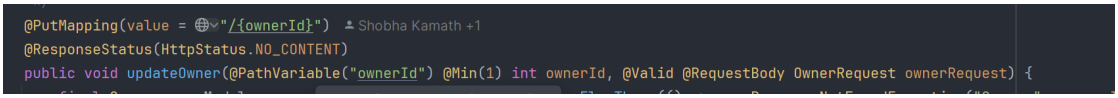


Abbildung 4.20: PUT-Anfrage in Spring Boot

Vergleich

- **Annotationen:** In Quarkus basiert die Implementierung der REST-APIs auf der Jakarta REST-Spezifikation, die Annotationen wie @GET, @POST, @PUT und @DELETE verwendet, um die verschiedenen HTTP-Anfragetypen zu definieren. Mit Spring 4.3 wurden fünf neue, spezifischere Annotationen eingeführt, die gezielt auf die einzelnen HTTP-Anfragetypen zugeschnitten sind: @GetMapping, @PostMapping, @PutMapping, @DeleteMapping und @PatchMapping [10]. Quarkus kann einfach die @Timed-Annotation von SmallRye Metrics vor die jeweilige Methode gesetzt werden. Das reicht aus, um die Metriken ohne zusätzliche Konfiguration zu aktivieren. Bei Spring Boot wird Micrometer für Metriken verwendet. Im Gegensatz zu Quarkus muss man die @Timed-Annotation auf Klassenebene setzen, und zusätzlich ist eine Konfigurationsklasse nötig, um die Metriken zu aktivieren.
- **Transaktionen:** In Quarkus wird die Annotation @Transactional explizit verwendet, um sicherzustellen, dass Datenbankoperationen in einer Transaktion erfolgen. In Spring wird das Transaktionsmanagement oft automatisch durch die Repository-Methoden wie save() gehandhabt, kann aber auch explizit mit @Transactional gesteuert werden.
- **Verarbeitung von Daten:** In Quarkus wird durch die Annotationen @Consumes und @Produces festgelegt, dass die API JSON-Daten verarbeitet. Das gibt dem Entwickler die Flexibilität, genau zu definieren, welche Art von Daten die API erwartet und zurückgibt. In Spring Boot wird die JSON-Verarbeitung durch die Kombination von @RestController und den HTTP-Mapping-Annotationen (@GetMapping, @PostMapping usw.) automatisch abgewickelt. Der Entwickler muss nicht explizit festlegen, wie die Antwort serialisiert wird, da dies von @ResponseBody übernommen wird.

4.1.6 Vergleich der Discovery-Services

in diesem Abschnitt wird der Unterschied zwischen der Implementierung der Discovery-Services in Quarkus und Spring beschrieben. Beide Frameworks bieten Mechanismen, um Microservices in einer verteilten Umgebung zu registrieren und auffindbar zu machen, jedoch gehen sie dabei unterschiedliche Wege.

Quarkus Discovery-Services

In Quarkus wird Consul als Service-Discovery-Tool verwendet. Um einen Dienst in Consul zu registrieren, wird in Quarkus eine spezielle Registrierungsklasse benötigt, die den Service beim Start automatisch registriert. Hier ist ein Beispiel für den customers-service:

```
@ApplicationScoped  berraqui *
public class ServiceRegistrar {

    @ConfigProperty(name = "consul.host")  1 usage
    String consulHost;

    @ConfigProperty(name = "consul.port")  1 usage
    int consulPort;

    void onStart(@Observes StartupEvent ev, Vertx vertx) {  new *
        ConsulClient client = ConsulClient.create(vertx, new ConsulClientOptions().setHost(consulHost).setPort(consulPort));
        client.registerServiceAndAwait(new ServiceOptions().setName("customers-service").setPort(8088).setAddress("localhost"));
    }


}
```

Abbildung 4.21: Quarkus Consul Registrierungsklasse im Customers-Services

Die Klasse ServiceRegistrar wird verwendet, um den customers-service bei Consul zu registrieren, sobald die Anwendung gestartet wird. Sie verwendet die Quarkus-abhängige Vertx-Bibliothek und die ConsulClient-API, um die Verbindung zu Consul herzustellen. Dabei werden die Host- und Port-Informationen für Consul aus den Konfigurationen entnommen. In diesem Fall wird der customers-service auf localhost:8088 registriert.

Zusätzlich wird in der application.properties die Verbindung zu Consul folgendermaßen konfiguriert:

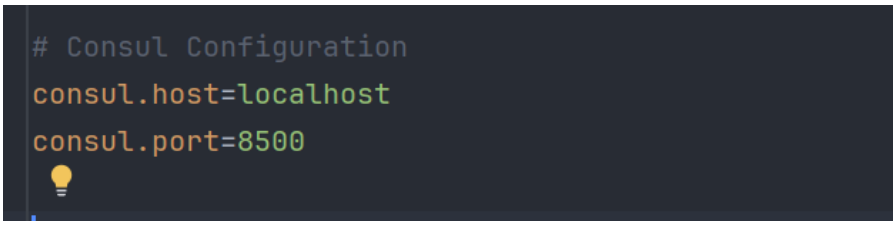


Abbildung 4.22: Quarkus Consul Konfiguration im Customers-Service

Das API-Gateway in Quarkus verwendet ebenfalls Consul zur Registrierung und Entdeckung der Microservices:

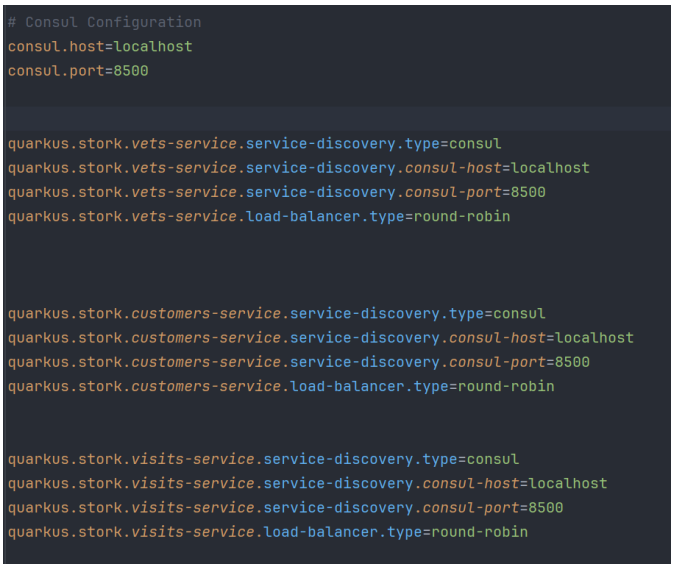


Abbildung 4.23: Quarkus Consul Konfiguration im API-Gateway

Diese Konfiguration ermöglicht es, mehrere Services wie vets-service, customers-service und visits-service dynamisch zu finden und zu verwalten. Zusätzlich zur Service-Discovery übernimmt Consul in Verbindung mit Quarkus Stork auch das Load-Balancing, sodass Anfragen gleichmäßig auf die Instanzen der Microservices verteilt werden. Die Round-Robin-Strategie sorgt dafür, dass die Last fair auf alle verfügbaren Instanzen verteilt wird. Consul fungiert dabei als zentrale Plattform für die dynamische Erkennung der Dienste, während Quarkus Stork die Lastverteilung steuert [24].

Spring Discovery-Services

Im Gegensatz zu Quarkus erfolgt die Service-Registrierung in Spring zentral über Eureka. Hier wird ein Eureka-Server verwendet, bei dem alle Services automatisch registriert werden, indem die Annotation @EnableDiscoveryClient in den jeweiligen Services verwendet wird.

Der Code für die Registrierung eines Spring-Services sieht wie folgt aus:

```
@EnableDiscoveryClient  Maciej Szarlinski +1
@SpringBootApplication
public class CustomersServiceApplication {

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

Abbildung 4.24: Spring Customers-Service Registrierung mit @EnableDiscoveryClient

Hier sorgt die Annotation @EnableDiscoveryClient dafür, dass der Service customers-service automatisch bei einem Eureka-Server registriert wird, ohne dass der Entwickler manuell eingreifen muss. Der Service startet durch den Aufruf der main-Methode.

Der Eureka-Server selbst wird durch den folgenden Code gestartet:

```
@SpringBootApplication  Maciej Szarlinski
@EnableEurekaServer
public class DiscoveryServerApplication {

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

Abbildung 4.25: Spring Konfiguration für den Eureka Discovery-Server mit @EnableEurekaServer

Durch die Verwendung von @EnableEurekaServer wird der Eureka-Server als Discovery-Service aktiviert, bei dem sich alle Microservices registrieren können.

Vergleich

In Quarkus erfolgt die Service-Registrierung über Consul, wobei jeder Dienst explizit

über eine ServiceRegistrar-Klasse registriert wird. Dies bietet eine flexible, aber etwas komplexere Methode. Die Konfiguration erfolgt über die application.properties-Datei. Spring hingegen verwendet den zentralen Eureka-Server, und die Dienste werden über die Annotation @EnableDiscoveryClient registriert, was die Implementierung einfacher und leichter wartbar macht.

4.1.7 Vergleich der API-Gateways

In diesem Abschnitt werden die API-Gateways von Quarkus und Spring verglichen. Beide haben eine wichtige Funktion bei der Kommunikation zwischen Clients und den verschiedenen Microservices. Obwohl beide leistungsstarke Lösungen anbieten, unterscheiden sie sich in ihrer Art und Weise, wie sie umgesetzt sind.

Quarkus API-Gateway
In Quarkus basiert das API-Gateway auf der Kombination von Jakarta RESTful Web Services und dem MicroProfile REST Client. Während die REST-APIs über die Standard-HTTP-Methoden definiert werden (siehe Abschnitt 4.1.5 Vergleich der REST-APIs), wird der MicroProfile REST Client verwendet, um die Kommunikation zwischen den Microservices zu vereinfachen.

Ein Beispiel für den CustomersServiceClient in Quarkus:

```
@Path("/owners")  new *
@RegisterRestClient(baseUri = "stork://customers-service")
public interface CustomersServiceClient {

    @GET  new *
    @Path("/{ownerId}")
    Uni<OwnerDetails> getOwner(@PathParam("ownerId") long ownerId);
}
```

Abbildung 4.26: Quarkus CustomerServiceClient im API-Gateway

wie in der Abbildung 4.26 registriert die Annotation @RegisterRestClient aus MicroProfile die Schnittstelle als REST-Client, der über die angegebene Basis-URL (hier

stork://customers-service) aufgerufen wird. Das API-Gateway wird also den customers-service über Stork aufrufen, wobei die aktuell verfügbare Instanz des Dienstes automatisch ermittelt wird. Stork ist in Quarkus für Service-Discovery und Load-Balancing zuständig [30]. Die Mutiny-API von Quarkus verwendet Uni, um reaktive und nicht-blockierende Anfragen zu ermöglichen [21]. Uni ist vergleichbar mit Mono in Spring und repräsentiert ein einzelnes Ergebnis, das asynchron zurückgegeben wird.

Das API-Gateway in Quarkus wird durch den ApiGatewayController implementiert:

```
@RequiredArgsConstructor  berraqui *
@Path("/api/gateway")
public class ApiGatewayController {

    @RestClient  1 usage
    CustomersServiceClient customersServiceClient;

    @RestClient  1 usage
    VisitsServiceClient visitsServiceClient;

    @RestClient  1 usage
    VetsServiceClient vetsServiceClient;




    @GET  berraqui *
    @Path("/owners/{ownerId}")
    @Produces(MediaType.APPLICATION_JSON)
    @CircuitBreaker(requestVolumeThreshold = 4)
    @Timeout(5000)
    public Uni<OwnerDetails> getOwnerDetails(@PathParam("ownerId") long ownerId) {
```

Abbildung 4.27: Quarkus ApiGatewayController im API-Gateway

In dieser Abbildung 4.27 wird gezeigt, wie bestimmte Annotationen und Bibliotheken verwendet werden, um die Kommunikation und Ausfallsicherheit zwischen Microservices zu verbessern. @RestClient injiziert einen MicroProfile REST Client, der es ermöglicht, RESTful Webservices wie den customers-service, vets-service und visits-service anzusprechen. Dies erlaubt eine deklarative und einfache Kommunikation zwischen Microservices. @CircuitBreaker aus MicroProfile Fault Tolerance schützt den Service vor Überlastung. Der Circuit Breaker tritt in Aktion, wenn eine bestimmte Anzahl von Fehlern oder zu langen Antwortzeiten auftritt. In diesem Fall wird ein alternativer Rückgabewert (Fall-

back) bereitgestellt, um die Stabilität des Systems zu gewährleisten. @Timeout definiert ein Zeitlimit für die Ausführung eines Service-Aufrufs. In diesem Beispiel ist das Timeout auf 5000 Millisekunden gesetzt. Wenn der Service diese Zeit überschreitet, wird der Aufruf abgebrochen.

Spring API-Gateway

In Spring basiert das API-Gateway auf Spring WebFlux und verwendet den WebClient zur Kommunikation mit den Microservices. Der CustomersServiceClient in Spring sieht wie folgt aus:

```
@Component  4 usages  Antoine Rey +1
@RequiredArgsConstructor
public class CustomersServiceClient {

    private final WebClient.Builder webClientBuilder;

    public Mono<OwnerDetails> getOwner(final int ownerId) {  3 usages  Antoine Rey
        return webClientBuilder.build().get()  RequestHeadersUriSpec<capture of ?>
            .uri( uri: "http://customers-service/owners/{ownerId}", ownerId)  capture of ?
            .retrieve()  ResponseSpec
            .bodyToMono(OwnerDetails.class);
    }
}
```

Abbildung 4.28: Spring CustomerServiceClient im API-Gateway

Wie in Abbildung 4.28 dargestellt, werden die Klassen WebClient und Mono folgendermaßen verwendet. Der WebClient ist ein Spring-Werkzeug für das Senden von HTTP-Anfragen. Er arbeitet reaktiv und nicht-blockierend, sodass er Anfragen abschickt und gleichzeitig andere Aufgaben erledigen kann. Häufig wird er verwendet, um REST-Services zu erreichen, bei denen Daten gesendet oder empfangen werden [34]. Mono ist eine Klasse aus Project Reactor, die ein einzelnes reaktives Ergebnis oder ein leeres Ergebnis liefert, nützlich für die asynchrone Verarbeitung von Daten, z. B. von einem entfernten Server [25].

Das API-Gateway in Spring wird als REST-Controller implementiert, ähnlich wie in Quarkus:

```java
@RestController  Antoine Rey +1
@RequiredArgsConstructor
@RequestMapping("/api/gateway")
public class ApiGatewayController {

    private final CustomersServiceClient customersServiceClient;

    private final VisitsServiceClient visitsServiceClient;

    private final ReactiveCircuitBreakerFactory cbFactory;

    @GetMapping(value = "owners/{ownerId}")  Antoine Rey
    public Mono<OwnerDetails> getOwnerDetails(final @PathVariable int ownerId) {
```

Abbildung 4.29: Spring ApiGatewayController im API-Gateway

Wie in der Abbildung gezeigt, wird hier Spring Cloud Circuit Breaker verwendet, der eine Abstraktionsschicht über verschiedene Implementierungen von Circuit Breakern bietet [33].

Vergleich

- **REST-Client und API-Implementierung:** In Quarkus wird der MicroProfile REST Client verwendet, der auf einem Interface basiert. Dadurch können REST-Clients auf deklarative Weise definiert werden, was zu einer sauberen und einfachen API-Implementierung führt. In Spring hingegen wird der WebClient verwendet, der in einer Klasse definiert ist.
- **Reaktive Programmierung:** Beide Frameworks unterstützen reaktive Programmierung. Quarkus nutzt die Mutiny-Bibliothek und Uni, während Spring auf Project Reactor und Mono setzt. Beide ermöglichen eine effiziente Verarbeitung asynchroner Anfragen, unterscheiden sich jedoch in der API-Syntax.
- **Circuit Breaker und Fallbacks:** Sowohl Quarkus als auch Spring bieten robuste Lösungen für die Fehlerbehandlung. Quarkus verwendet die MicroProfile Fault Tolerance-Spezifikation, während Spring den ReactiveCircuitBreaker aus der Spring Cloud verwendet. Beide Mechanismen schützen das API-Gateway vor überlasteten oder fehlerhaften Services und bieten Fallback-Optionen.

4.2 Vergleich der Performance

In diesem Abschnitt wird die Performance von Spring Boot und Quarkus anhand mehrerer Kriterien verglichen. Der Vergleich fokussiert sich auf die Startzeit der drei Microservices (customers-service, visits-service und vets-service), den Ressourcenverbrauch während des Startvorgangs sowie die Skalierbarkeit der Microservices in horizontaler und vertikaler Hinsicht. Dies ist entscheidend, um zu verstehen, wie gut beide Frameworks auf die Anforderungen moderner Microservice-Architekturen und hochskalierbarer Systeme reagieren.

Alle im Rahmen dieses Vergleichs durchgeführten Tests wurden auf der in Tabelle 4.1 beschriebenen Hardware und dem in Tabelle 4.2 dargestellten Betriebssystem ausgeführt. Die Leistungsmerkmale des verwendeten Systems, einschließlich des 13th Gen Intel® Core™ i7-13700k Prozessors und der 32 GB Arbeitsspeicher, gewährleisten eine optimale Testumgebung für die Bewertung der Performance von Spring Boot und Quarkus.

Memory:	32.0 GiB
Processor:	13th Gen Intel® Core™ i7-13700k × 24
Graphics:	Nvidia RTX 4070 TI SUPER 16 GiB
Disk Capacity:	2.0 TB

Tabelle 4.1: Hardware Information

Firmware Version:	23H2
OS Name:	Windows 11 Pro
OS Type:	64-bit

Tabelle 4.2: Software information

4.2.1 Vergleich der Startzeit

In diesem Abschnitt wird die Startzeit von drei Microservices customers-service, visits-service und vets-service in den Frameworks Spring Boot und Quarkus verglichen. Zur Messung wurden zwei Shell-Skripte verwendet: start-services.sh für den Start der Services ohne Tests und start-services-with-tests.sh für den Start der Services mit Tests.

Um den Test fair zu gestalten, wurden nur die drei genannten Microservices verglichen, da die Spring Boot-Version zusätzliche Services wie den Spring Cloud Config Server und

den Spring Boot Admin Server enthält, die in der Quarkus-Version nicht unterstützt werden.

Startzeiten ohne Tests

Die Startzeiten der drei Microservices ohne Tests zeigten, dass Quarkus eine deutlich schnellere Startzeit aufweist als Spring Boot. Zum Beispiel startete der customers-service in Quarkus in 2,497 Sekunden, während er in Spring Boot 6,33 Sekunden benötigte. Ähnlich war es beim visits-service und beim vets-service, wo Quarkus ebenfalls schneller war. Insgesamt zeigt sich, dass Quarkus bei einem reinen Startvorgang ohne Tests wesentlich effizienter ist.

	Spring Boot	**Quarkus**
Customers-Service	6,33 s	2,497 s
Visits-Service	5,483 s	2,746 s
Vets-Service	6,692 s	2,792 s

Tabelle 4.3: Vergleich zwischen Spring Boot und Quarkus für verschiedene Services ohne Tests

Startzeiten mit Tests

Wenn die Services zusammen mit Tests gestartet wurden, konnte Spring Boot hingegen schnellere Ergebnisse liefern. Der customers-service startete in Spring Boot in 10,455 Sekunden, während Quarkus dafür 23,681 Sekunden benötigte. Ein ähnliches Ergebnis zeigt sich auch beim visits-service und vets-service, wo Spring Boot insgesamt effizienter war. Dies deutet darauf hin, dass Quarkus mehr Zeit benötigt, wenn Tests in den Startvorgang integriert werden.

	Spring Boot	**Quarkus**
Customers-Service	10,455 s	23,681 s
Visits-Service	11,966 s	21,929 s
Vets-Service	12,986 s	24,122 s

Tabelle 4.4: Vergleich zwischen Spring Boot und Quarkus für verschiedene Services mit Tests

Zusammengefasst lässt sich feststellen, dass Quarkus ohne Tests eine bessere Performance bietet, während Spring Boot bei der Ausführung von Tests während des Startvorgangs schneller ist.

4.2.2 Vergleich der horizontalen Skalierbarkeit und des Ressourcenverbrauchs

In diesem Abschnitt wurde die Performance beider Frameworks Quarkus und Spring Boot untersucht. Hierbei wurden zwei Instanzen des API-Gateways für beide Frameworks in der Entwicklungsumgebung IntelliJ gestartet: eine Instanz auf Port 8080 und eine weitere auf Port 8081. Um die Lastverteilung zu gewährleisten, wurde ein Load Balancer mit Hilfe von Nginx konfiguriert. Die beiden getesteten Methoden waren getOwner und getAllOwners, welche mit dem Lasttest-Tool Locust ausgeführt wurden. Während des Tests wurde die CPU- und Speicherlast mit VisualVM analysiert, um die Auswirkungen der horizontalen Skalierung auf den Ressourcenverbrauch zu überwachen.

API-Gateway-Instanzen in Consul und Eureka

Wie in Abbildung 4.30 zu sehen ist, zeigt das Consul-Dashboard für Quarkus, dass die beiden API-Gateway-Instanzen auf den Ports 8080 und 8081 erfolgreich registriert wurden. Dies bestätigt, dass beide Instanzen korrekt gestartet wurden und die Lastverteilung mit Nginx funktioniert. Dadurch kann Quarkus seine horizontale Skalierbarkeit sicherstellen.

Ebenso zeigt Eureka für Spring Boot, dass auch hier die beiden API-Gateway-Instanzen auf den Ports 8080 und 8081 erfolgreich registriert sind. Dies bestätigt, dass die Lastverteilung und die Skalierung ebenfalls korrekt funktionieren.

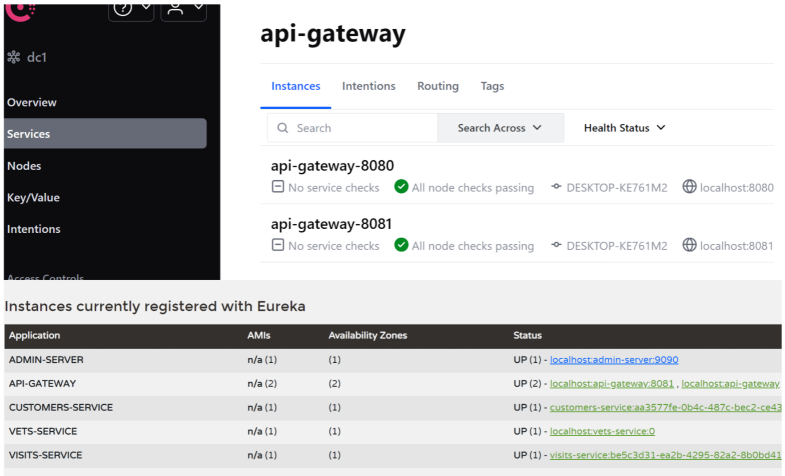


Abbildung 4.30: API-Gateway-Instanzen in Consul (Quarkus) und Eureka (Spring Boot)

Einstellung der Lasttestkonfiguration

Die Lasttestkonfiguration wurde im Locust-Tool so eingestellt, dass 2400 gleichzeitige Benutzer simuliert wurden. Wie in Abbildung 4.31 dargestellt, begann der Test mit einer Ramp-Up-Rate von 50 Benutzern pro Sekunde. Das Ziel war, über eine Laufzeit von 15 Minuten die Systeme unter intensiver Last zu beobachten und die Stabilität sowie den Ressourcenverbrauch zu bewerten.

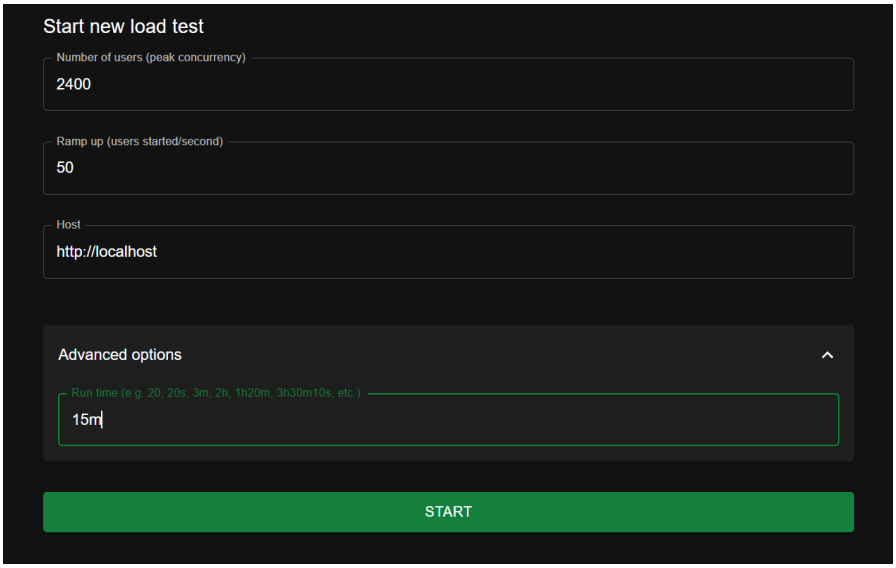
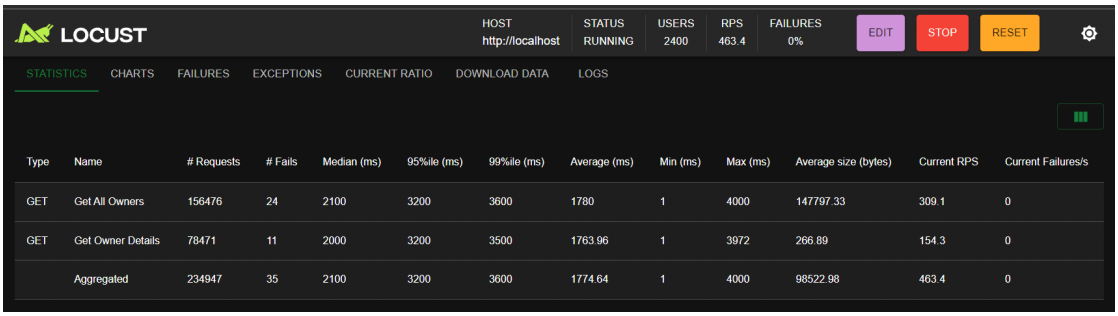


Abbildung 4.31: Locust Lasttest-Einstellungen für beide Frameworks

Quarkus Lasttest Ergebnisse

Die Locust-Testergebnisse (siehe Abbildung 4.32) zeigen, dass Quarkus bei einer gleichzeitigen Benutzersimulation von 2400 performt, was zu einer stabilen Anzahl an Requests pro Sekunde (RPS) von 463,4 führte. Die Antwortzeiten lagen im 50. Perzentil bei etwa 2100 ms, was die mittlere Zeit darstellt, die die meisten Benutzer für eine Antwort benötigten. Im 95. Perzentil, also für die 5 % der langsamsten Anfragen, stieg die Antwortzeit auf 3200 ms, und im 99. Perzentil erreichte sie 3600 ms. Der durchschnittliche Wert lag bei 1775 ms, während die minimale Antwortzeit 1 ms und die maximale 4000 ms betrug. Die durchschnittliche Größe der Antwortdaten lag bei 98.523 Bytes. Trotz der hohen Last verzeichnete Quarkus während des gesamten Tests keine Fehlanfragen, was eine beeindruckende Stabilität und Effizienz zeigt.



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	Get All Owners	156476	24	2100	3200	3600	1780	1	4000	147797.33	309.1	0
GET	Get Owner Details	78471	11	2000	3200	3500	1763.96	1	3972	266.89	154.3	0
	Aggregated	234947	35	2100	3200	3600	1774.64	1	4000	98522.98	463.4	0

Abbildung 4.32: Quarkus Lasttest

Quarkus Charts

Die Abbildung 4.33 zeigt, wie Quarkus unter Last reagiert. Mit 2400 aktiven Benutzern konnte das System konstante 500 Requests pro Sekunde (RPS) verarbeiten, wobei keine signifikanten Leistungseinbußen festzustellen waren. Die Antwortzeiten bleiben ebenfalls relativ stabil, was für die Effizienz des Frameworks spricht.

Abbildung 4.33: Quarkus Charts

Quarkus CPU-Verbrauch

CPU-Verbrauch während des Lasttests lag der durchschnittliche CPU-Verbrauch der Java Virtual Machine (JVM), gemessen mit VisualVM (siehe Abbildung 4.34), bei etwa 5 %, wobei der Spitzenwert 11,2 % betrug. Dies zeigt, wie viel Rechenleistung die Quarkus-Instanzen innerhalb der JVM beanspruchen.

Im Windows Task-Manager wurde eine Gesamt-CPU-Auslastung mit einem Spitzenwert von 72 % gemessen, während der durchschnittliche Verbrauch bei etwa 45 % lag. Dies spiegelt die Belastung des gesamten Systems während des Lasttests wider, einschließlich aller aktiven Prozesse. In Abbildung 4.35 zeigt der Task-Manager deutlich, dass die Sys-

temressourcen insgesamt stärker beansprucht wurden, verglichen mit der CPU-Nutzung der JVM, die in VisualVM beobachtet wurde.

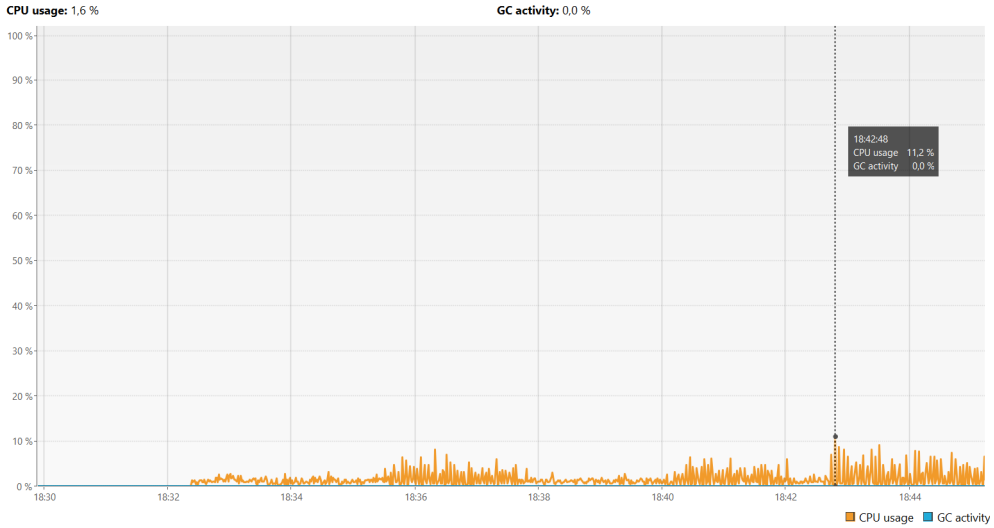


Abbildung 4.34: Quarkus CPU-Verbrauch in der JVM (VisualVM)

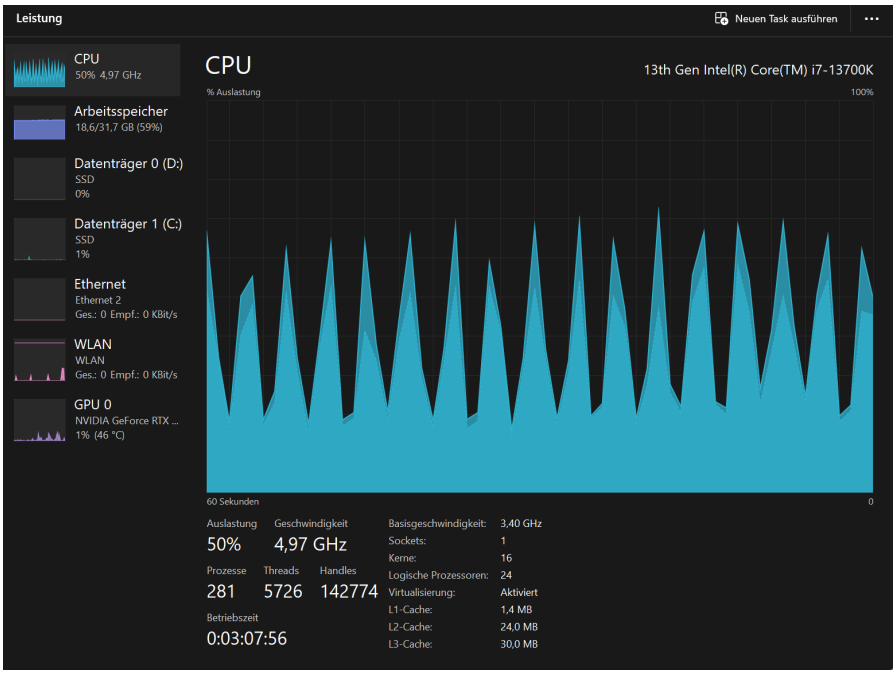


Abbildung 4.35: Quarkus Systemlast (Windows Task-Manager)

Quarkus Speicherverbrauch

Der Speicherverbrauch von Quarkus blieb ebenfalls stabil. Die Heap-Nutzung schwankte zwischen 200 MB und 510 MB (siehe Abbildung 4.36), wobei der maximale Heap-Speicher auf 580 MB festgelegt war. Dies zeigt, dass Quarkus die verfügbaren Speicherressourcen gut verwaltet und keine Speicherausfälle oder Engpässe auftreten.

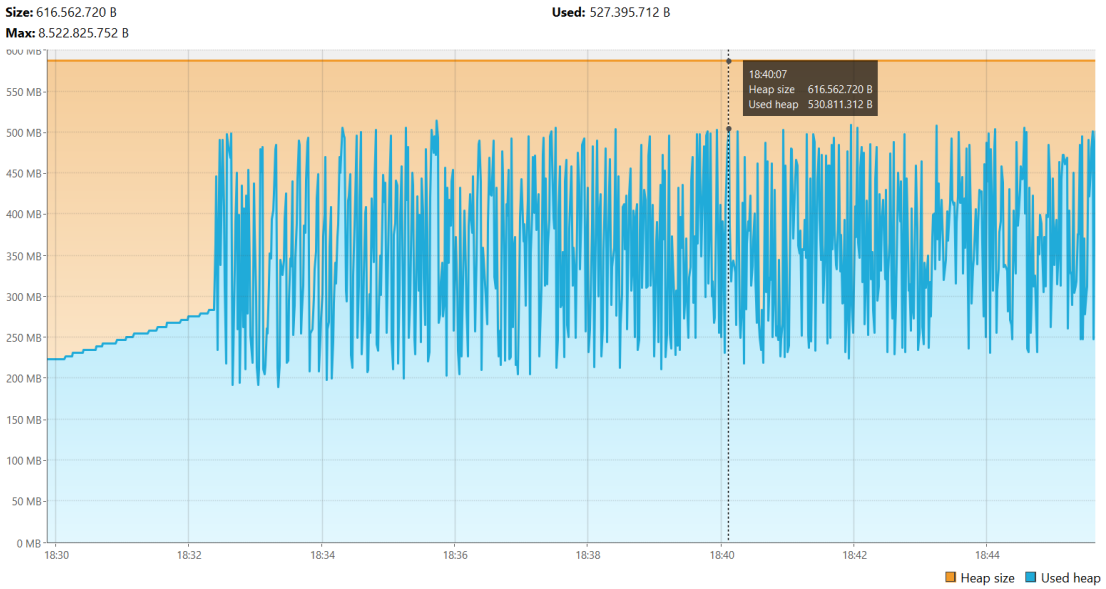


Abbildung 4.36: Quarkus Speicher-Verbrauch

Spring Boot Lasttest Ergebnisse

Die Locust-Testergebnisse (siehe Abbildung 4.37) zeigen, dass Spring Boot unter einer gleichzeitigen Benutzersimulation von 2400 ebenfalls gut performte. Die Antwortzeiten lagen im 50. Perzentil bei 2200 ms und im 95. Perzentil bei 5700 ms, während die Antwortzeiten im 99. Perzentil 6000 ms erreichten. Der durchschnittliche Wert lag bei 3395,38 ms, mit einer minimalen Antwortzeit von 5 ms und einer maximalen Antwortzeit von 8437 ms. Die durchschnittliche Größe der Antwortdaten betrug 35.969 Bytes. Die Requests pro Sekunde (RPS) beliefen sich auf 373,6, jedoch wurden hier eine Fehlerquote von 61 % festgestellt, was auf eine instabile Performance bei hoher Last hindeutet.

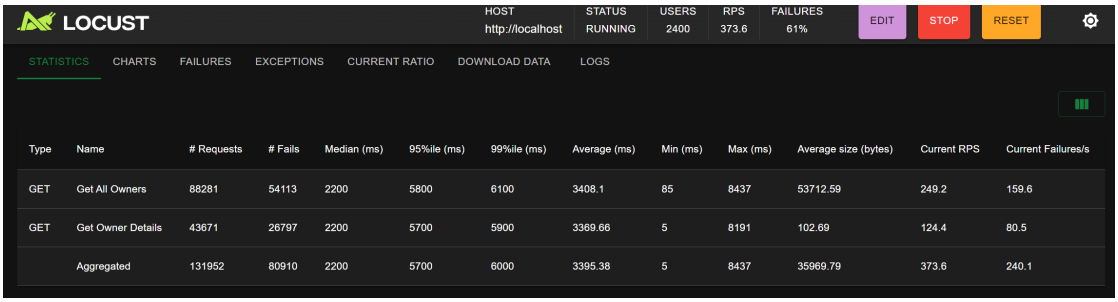


Abbildung 4.37: Spring Boot Lasttest

Spring Boot-Charts

In Abbildung 4.38 ist zu erkennen, dass die Fehlerquote ab etwa 150 Requests pro Sekunde (RPS) stetig ansteigt. Bei einer Spitzenrate von 249,4 Fehlern pro Sekunde (FPS) deutet dies darauf hin, dass das System bei dieser Anzahl von Anfragen nicht mehr in der Lage ist, alle korrekt zu verarbeiten.

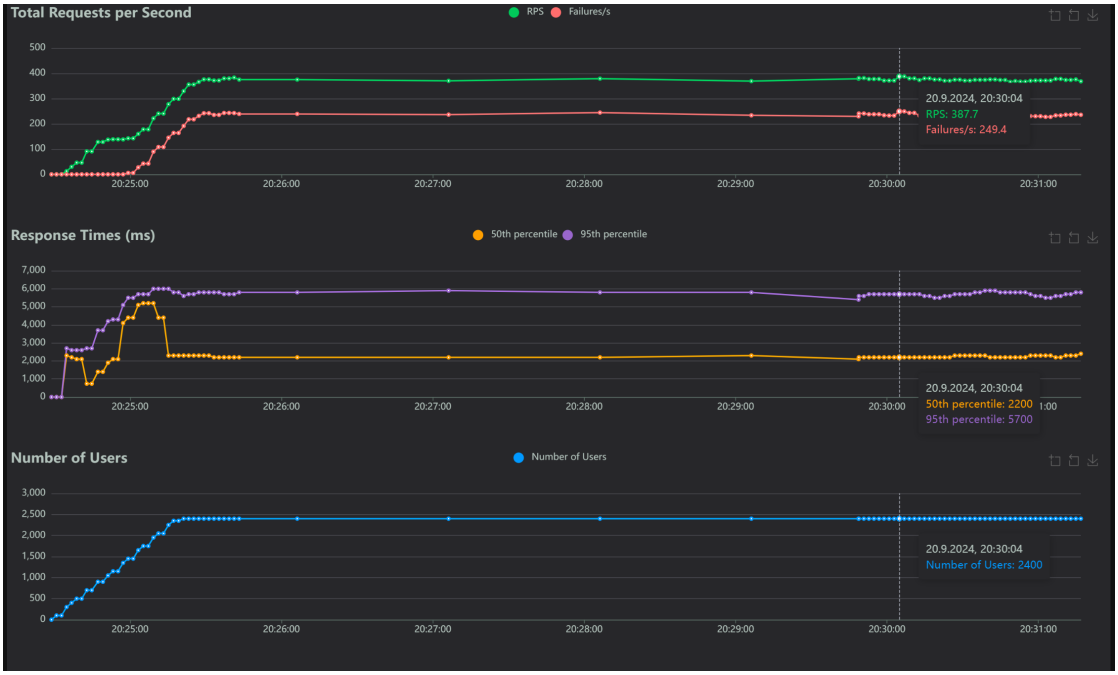


Abbildung 4.38: Spring Boot Charts

Spring Boot CPU-Verbrauch

Während des Lasttests in der Java Virtual Machine (JVM) wurde die CPU-Auslastung mithilfe von VisualVM überwacht. Die durchschnittliche CPU-Belastung lag bei 7,1 %, mit einem Spitzenwert von 9,1 % (siehe Abbildung 4.39). Dies zeigt die geringe CPU-Auslastung durch die Spring Boot-Instanzen innerhalb der JVM.

Zusätzlich dazu zeigt der Windows Task-Manager eine weitaus höhere Gesamt-CPU-Auslastung des Systems während des Tests. Hier wurde eine durchschnittliche Auslastung von 70 % beobachtet, mit einem Spitzenwert von 87 % (siehe Abbildung 4.40). Diese Messwerte verdeutlichen, dass die Systemressourcen stark beansprucht wurden, was auf die zusätzliche Last durch andere Prozesse während des Tests zurückzuführen ist.

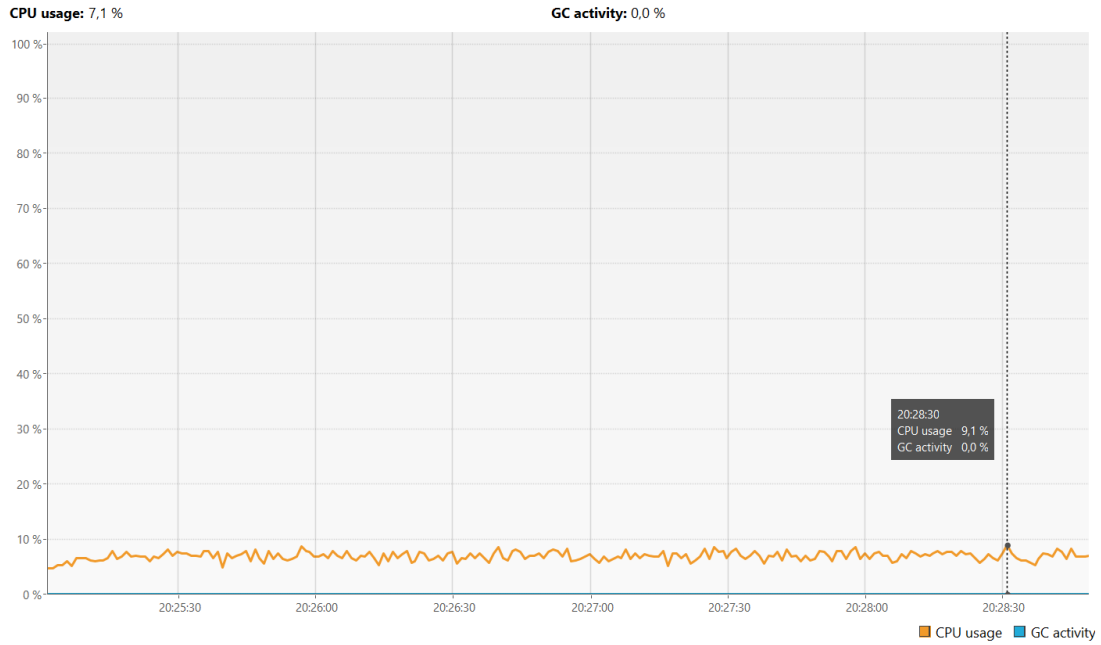


Abbildung 4.39: Spring Boot CPU-Verbrauch in der JVM (VisualVM)

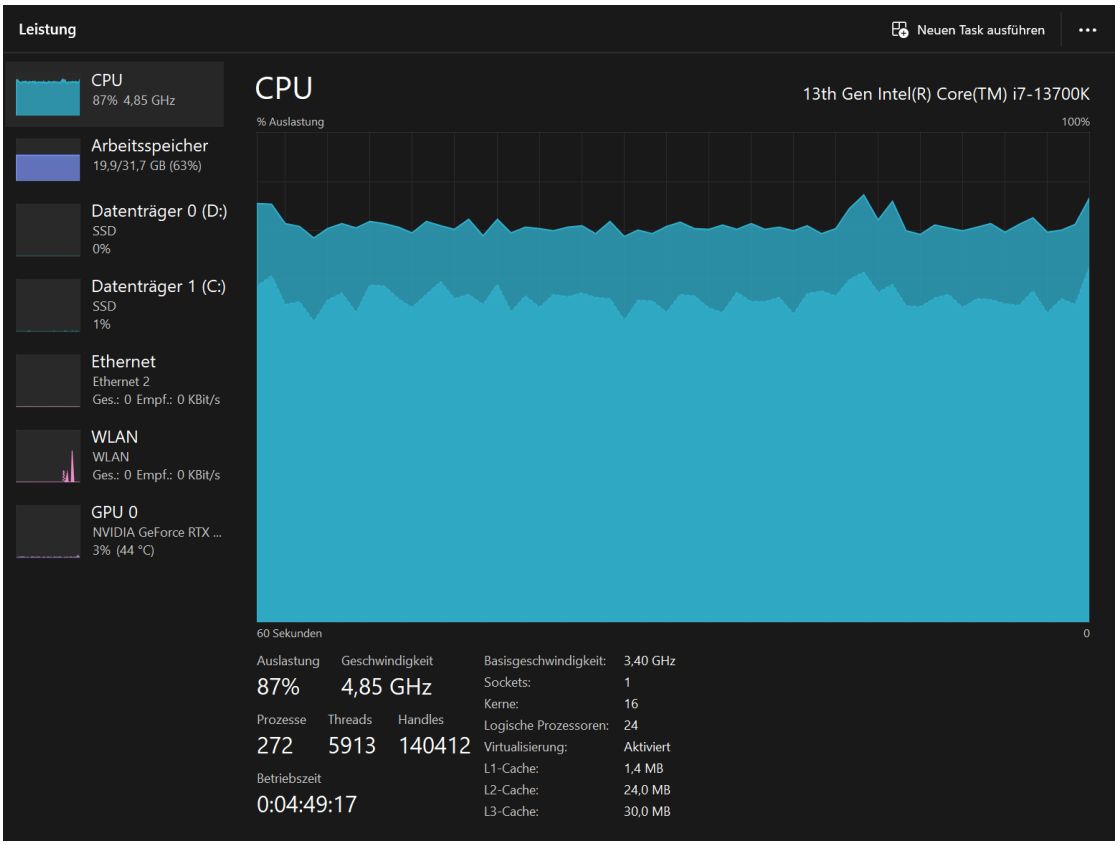


Abbildung 4.40: Spring Boot Systemlast (Windows Task-Manager)

Spring Boot Speicherverbrauch

Der Speicherverbrauch von Spring Boot zeigt, dass der Heap-Speicher stark genutzt wurde (siehe Abbildung 4.41). Der maximale Speicher war auf etwa 1,05 GB festgelegt, und der tatsächliche Verbrauch schwankte während des Tests stark. Er erreichte Spitzen von bis zu 750 MB, wobei der Speicher zwischen 100 MB und 700 MB variierte. Dies bedeutet, dass die Anwendung oft Speicher freigeben musste, was auf eine hohe Aktivität der Speicherverwaltung hinweist.

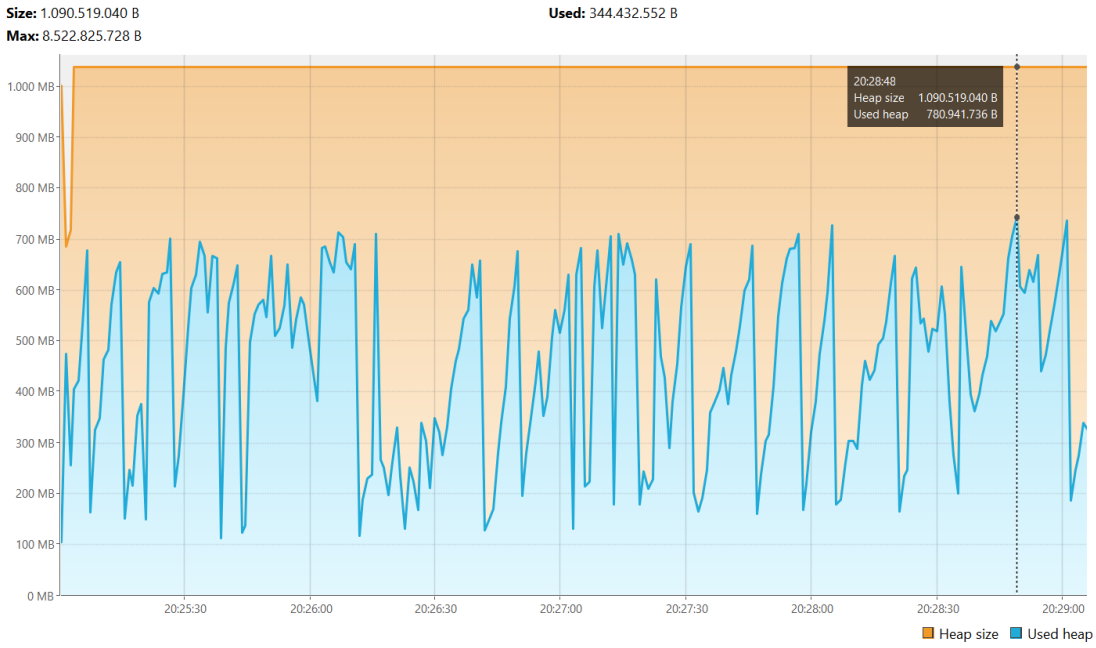


Abbildung 4.41: Spring Boot Speicherverbrauch

Vergleich

- **Lasttest-Ergebnisse:** Quarkus bewältigte 2400 gleichzeitige Benutzer mit stabilen Antwortzeiten und einer konstanten Anzahl an Requests pro Sekunde (RPS) von 463,4. Die Antwortzeiten blieben im Durchschnitt bei 2100 ms und stiegen im 95. Perzentil auf 3200 ms, ohne dass Fehlanfragen auftraten. Spring Boot dagegen hatte eine hohe Fehlerrate von 61 %, die ab etwa 150 Requests pro Sekunde (RPS) begann. Die Antwortzeiten bei Spring Boot waren ebenfalls höher, insbesondere im 95. Perzentil, wo sie auf 5800 ms stiegen, und erreichten maximal 6100 ms.
- **CPU-Verbrauch:** Beim CPU-Verbrauch war Quarkus effizienter. Die Java Virtual Machine (JVM) bei Quarkus zeigte eine durchschnittliche CPU-Auslastung

von nur 5 %, während der Spitzenwert 11,2 % betrug. Im Vergleich dazu lag die CPU-Last von Spring Boot innerhalb der JVM bei 7 %, mit einem Höchstwert von 9,1 %. Allerdings wurde im Task-Manager für beide Frameworks eine deutlich höhere CPU-Auslastung gemessen, insbesondere bei Spring Boot, wo die Systemlast Spitzenwerte von 87 % erreichte, während Quarkus bei 72 % lag.

- **Systemlast** Insgesamt zeigte Quarkus eine niedrigere Systemlast und blieb unter hoher Last stabiler als Spring Boot. Spring Boot hingegen geriet bei steigenden Anfragen schneller an seine Grenzen, was sich in einer höheren Fehlerrate, schlechteren Antwortzeiten und einer höheren Systemauslastung widerspiegelte.
- **Speicherverbrauch:** Quarkus zeigte auch beim Speicherverbrauch eine stabilere Leistung. Die Heap-Nutzung schwankte zwischen 200 MB und 510 MB, mit einer maximalen Heap-Größe von 580 MB. Spring Boot hingegen beanspruchte mehr Speicher, mit einem Maximum von 1,05 GB, und zeigte stärkere Schwankungen im Speicherverbrauch, der zwischen 100 MB und 700 MB variierte. Dies deutet auf häufigeres Garbage Collecting hin, was die Leistung beeinträchtigen könnte.

4.2.3 Vergleich der vertikalen Skalierbarkeit und des Ressourcenverbrauchs

In diesem Abschnitt wurde eine Instanz der API-Gateways von Quarkus und Spring Boot getestet, um die Auswirkungen der vertikalen Skalierung auf die Performance und den Ressourcenverbrauch zu analysieren. Hierbei wurden zwei Testdurchläufe für beide Frameworks durchgeführt: einmal mit einer Begrenzung auf 2 CPU-Kerne und einmal mit 4 CPU-Kernen.

Die beiden getesteten Methoden waren erneut getOwner und getAllOwners, ausgeführt mit dem Lasttest-Tool Locust, wie auch bereits im Abschnitt 4.2.2. Die Konfiguration der Lasttest-Szenarien blieb unverändert, um eine konsistente Vergleichbarkeit zu gewährleisten. Ziel der Tests war es, zu untersuchen, wie sich die Performance, CPU-Nutzung bei Erhöhung der CPU-Ressourcen von 2 auf 4 Kerne verändert, während der Arbeitsspeicher gleich bleibt.

Zur Messung der CPU-Auslastung in der JVM wurde wie zuvor VisualVM eingesetzt. Dabei wurden insbesondere die Antwortzeiten und die Anzahl der Anfragen pro Sekunde (RPS) verglichen.

Quarkus Lasttest-Ergebnisse

Die Locust-Testergebnisse (siehe Abbildungen 4.42 und 4.43) zeigen die Performance von Quarkus bei einer gleichzeitigen Benutzersimulation von 2400. Zwei Testdurchläufe wurden durchgeführt, einer mit 2 CPU-Kernen und einer mit 4 CPU-Kernen, um die Auswirkungen der vertikalen Skalierung zu bewerten.

- **Mit 2 CPU-Kernen:** Es wurden insgesamt 62.928 Anfragen erfolgreich bearbeitet. Die Antwortzeiten lagen im Median bei 4100 ms, während die Anfragen im 95. Perzentil 5300 ms und im 99. Perzentil 11.000 ms erreichten. Es traten keine Fehler auf, und die durchschnittliche Anfragedatenmenge betrug 98.767 Bytes. Die RPS (Requests pro Sekunde) betrugen 294,5.
- **Mit 4 CPU-Kernen:** Hier wurden insgesamt 305.000 Anfragen bearbeitet. Der Median der Antwortzeiten lag bei 1600 ms, das 95. Perzentil bei 3700 ms, und das 99. Perzentil bei 4600 ms. Auch hier gab es keine Fehlanfragen. Die durchschnittliche Anfragedatenmenge lag bei 98.722 Bytes, und die RPS stiegen auf 428,8.

Durch die Erhöhung der CPU-Kerne konnte eine erhebliche Verbesserung der Performance festgestellt werden, was durch die niedrigeren Antwortzeiten und die höhere Anzahl bearbeiteter Anfragen deutlich wird.

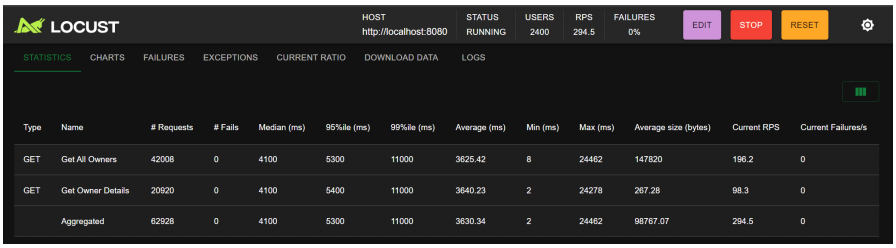


Abbildung 4.42: Quarkus Lasttest mit 2 Kerne

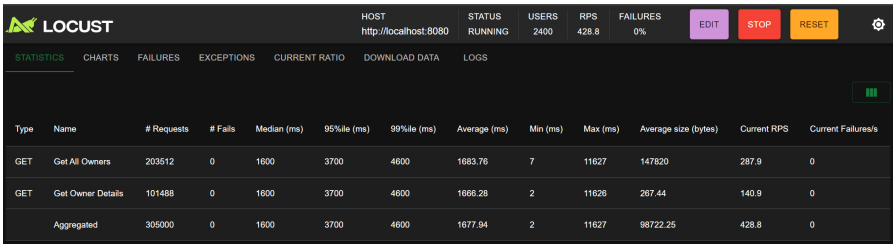


Abbildung 4.43: Quarkus Lasttest mit 4 Kerne

Quarkus CPU-Verbrauch Die VisualVM-Testergebnisse (siehe Abbildung 4.44 und 4.45) zeigen die CPU-Auslastung von Quarkus während der Lasttests bei einer gleichzeitigen Benutzersimulation von 2400.

- **Mit 2 CPU-Kernen:** Während dieses Tests lag die durchschnittliche CPU-Auslastung bei etwa 52 %, wobei Spitzenwerte von 83,5 % erreicht wurden. Diese Schwankungen in der CPU-Auslastung zeigen, dass Quarkus mit begrenzten Ressourcen stark ausgelastet war, insbesondere bei hoher Last. Die Garbage-Collection (GC)-Aktivität blieb minimal bei 0,1 %.
- **Mit 4 CPU-Kernen:** In diesem Test zeigte Quarkus eine stabilere Performance mit einer durchschnittlichen CPU-Auslastung von etwa 55 %. Der Spitzenwert erreichte hier 69,1 %, was deutlich unter dem Maximum des 2-Core-Tests liegt. Dies zeigt, dass durch die Erhöhung der CPU-Kerne eine gleichmäßigere Auslastung erreicht wurde, was zu einer effizienteren Nutzung der verfügbaren Ressourcen führte. Die GC-Aktivität blieb ebenfalls bei 0,1 %.

Der Vergleich beider Testläufe verdeutlicht, dass durch die Bereitstellung von mehr CPU-Kernen die Auslastung der einzelnen Kerne gesenkt wurde, was zu einer stabileren und gleichmäßigeren Performance führte. Insbesondere bei 4 CPU-Kernen konnten Spitzenwerte in der CPU-Auslastung vermieden werden, die bei der Nutzung von nur 2 Kernen häufiger auftraten.

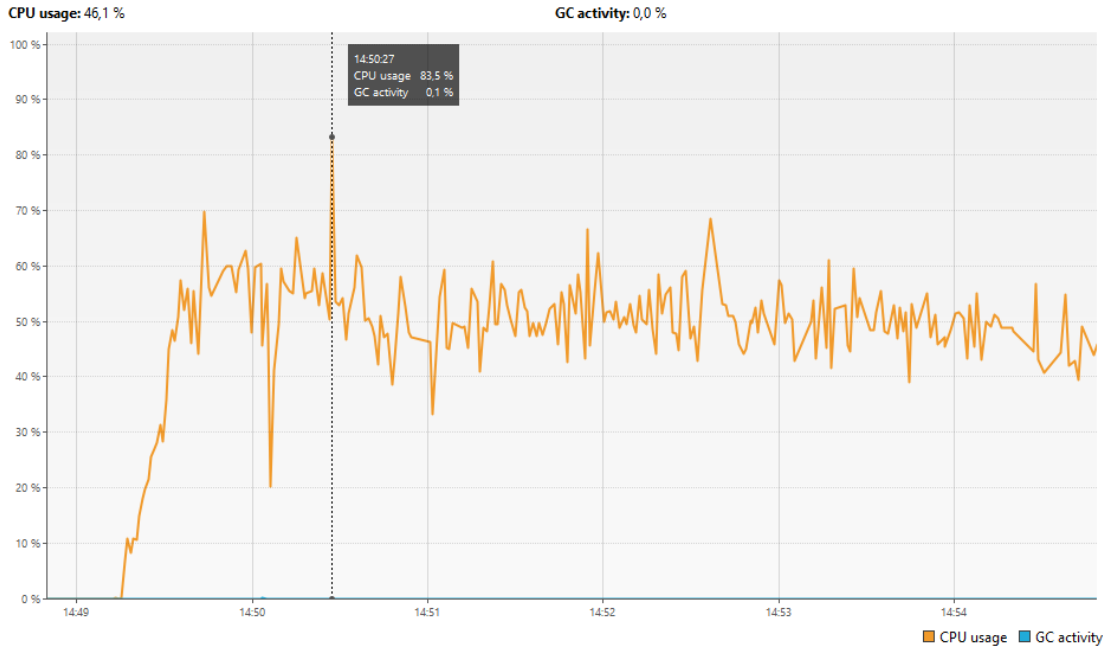


Abbildung 4.44: Quarkus CPU-Verbrauch mit 2 Kerne

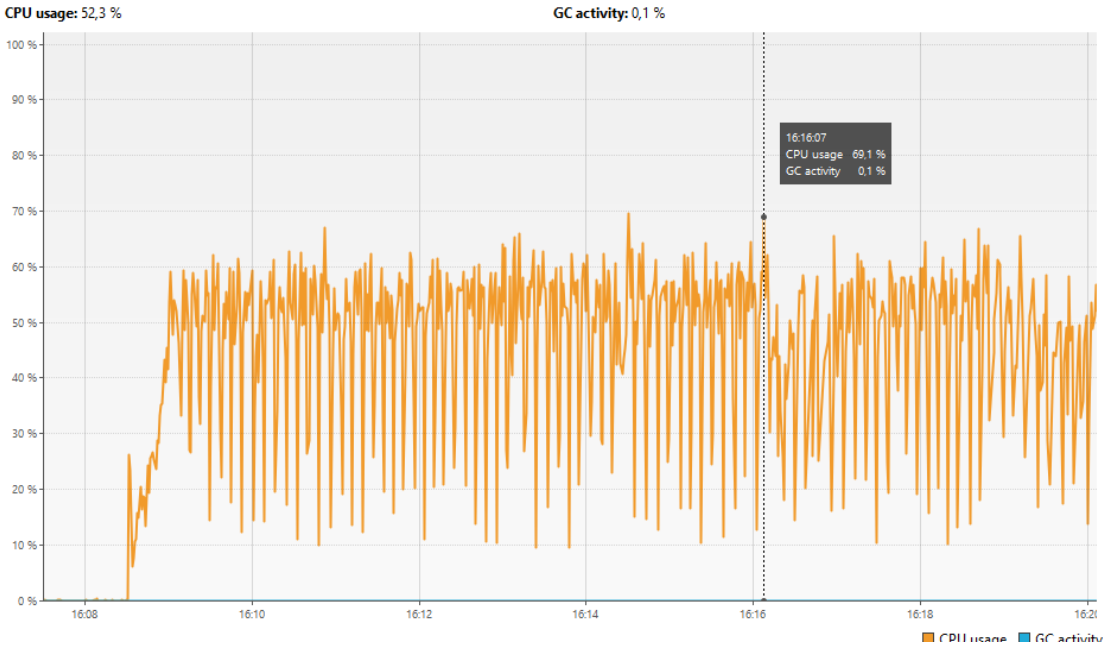


Abbildung 4.45: Quarkus CPU-Verbrauch mit 4 Kerne

Quarkus Speicherverbrauch

Der Speicherverbrauch von Quarkus blieb während der beiden Tests relativ konstant, insbesondere bei der maximalen Heap-Nutzung. Während der tatsächliche Speicherverbrauch leicht schwankte, war die Gesamtnutzung stabil. Im Vergleich zu den Tests der horizontalen Skalierung (siehe Abschnitt 4.2.2) wurde ein Rückgang des Speicherverbrauchs von etwa 10 % festgestellt. Dies deutet darauf hin, dass das System durch die vertikale Skalierung effizienter arbeitete und weniger Speicherressourcen benötigte.

Spring Boot Lasttest-Ergebnisse

Die Locust-Testergebnisse (siehe Abbildung 4.46 und 4.47) zeigen die Performance von Spring Boot bei einer gleichzeitigen Benutzersimulation von 2400.

- **Mit 2 CPU-Kernen:** Insgesamt wurden 29.320 Anfragen erfolgreich bearbeitet. Die Antwortzeiten lagen im Median bei 6800 ms, während die Anfragen im 95. Perzentil 33.000 ms und im 99. Perzentil 48.000 ms erreichten. Die Fehlerrate lag bei 81 %, und die durchschnittliche Größe der Anfragedaten betrug 41.749 Bytes. Die Anzahl der Requests pro Sekunde (RPS) betrug 242,2.
- **Mit 4 CPU-Kernen:** In diesem Durchlauf wurden 37.211 Anfragen bearbeitet. Der Median der Antwortzeiten lag bei 5600 ms, das 95. Perzentil bei 16.000 ms und das 99. Perzentil bei 21.000 ms. Die Fehlerrate sank auf 71 %. Die durchschnittliche Anfragedatenmenge betrug 48.604 Bytes, und die RPS stiegen auf 256,2.

Es zeigt sich, dass durch die Erhöhung der CPU-Kerne die Performance von Spring Boot leicht verbessert werden konnte, insbesondere in Bezug auf die Antwortzeiten und die Fehlerrate.

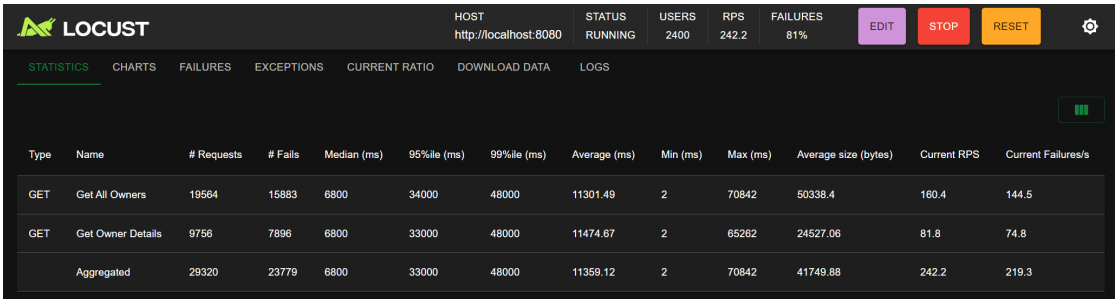


Abbildung 4.46: Spring Boot Lasttest mit 2 Cores

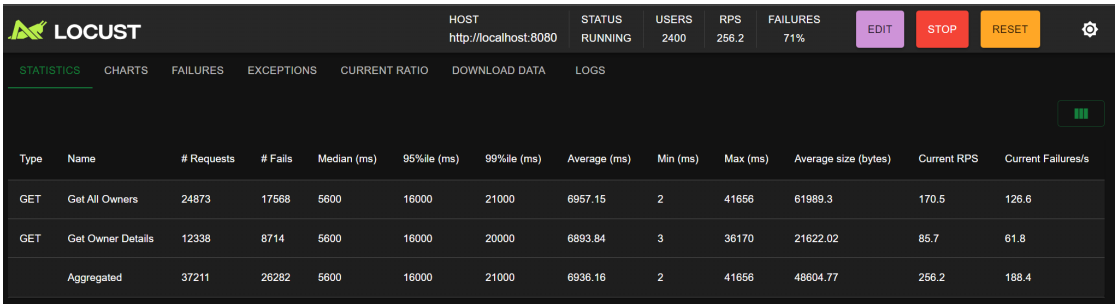


Abbildung 4.47: Spring Boot Lasttest mit 4 Cores

Spring Boot CPU-Verbrauch

Die Analyse der CPU-Auslastung während der Lasttests mit Spring Boot zeigt deutliche Unterschiede zwischen den Testläufen mit 2 und 4 CPU-Kernen.

- **Mit 2 CPU-Kernen:** Die CPU-Auslastung erreichte bei Spitzenlast 100 %, was zeigt, dass die verfügbare Rechenleistung vollständig ausgeschöpft wurde. Im Durchschnitt lag die CPU-Auslastung bei etwa 65 % (siehe Abbildung 4.48), was auf eine konstante Auslastung während des Tests hinweist. Die Garbage-Collection (GC)-Aktivität zeigte im 2-Core-Test zu Beginn des Tests eine kurzfristige Spitze von 2,9 %, stabilisierte sich jedoch schnell wieder auf 0,1 %
- **Mit 4 CPU-Kernen:** Hier war die maximale CPU-Auslastung ebenfalls bei Spitzenlast 100 %, und der Durchschnitt lag bei etwa 70 % (siehe Abbildung 4.49). Durch die Erhöhung der verfügbaren CPU-Kerne die Last etwas wenig besser verteilt und verarbeitet werden konnte. Die Garbage-Collection (GC)-Aktivität blieb minimal bei 0,5 %.

Durch den Vergleich der beiden Testläufe wird deutlich, dass die vertikale Skalierung in Bezug auf die CPU-Auslastung eine Entlastung bewirken kann. Bei 2 Kernen zeigte das System eine hohe Auslastung und reagierte insgesamt langsamer im Vergleich zum 4-Kerne-Test, bei dem die Last gleichmäßiger verteilt wurde und die Verarbeitung effizienter war. Dies führte bei 4 Kernen zu einer stabileren und schnelleren Systemleistung unter Last. Eine Erhöhung der CPU-Kapazitäten trug somit maßgeblich zur Optimierung der Performance bei.

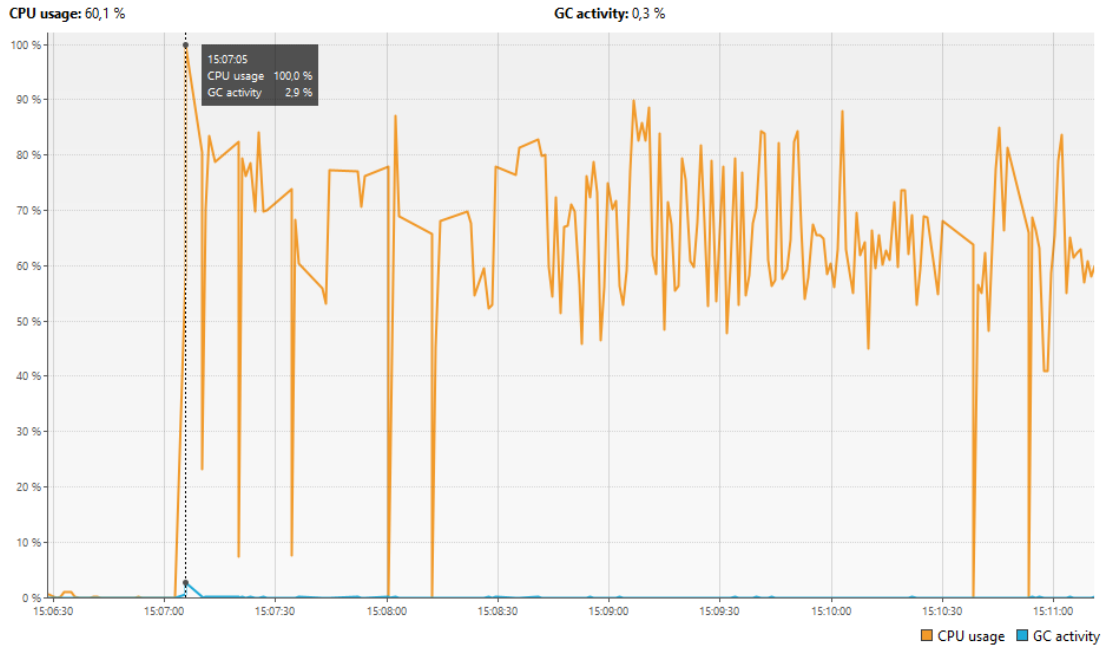


Abbildung 4.48: Spring Boot CPU-Verbrauch mit 2 Cores

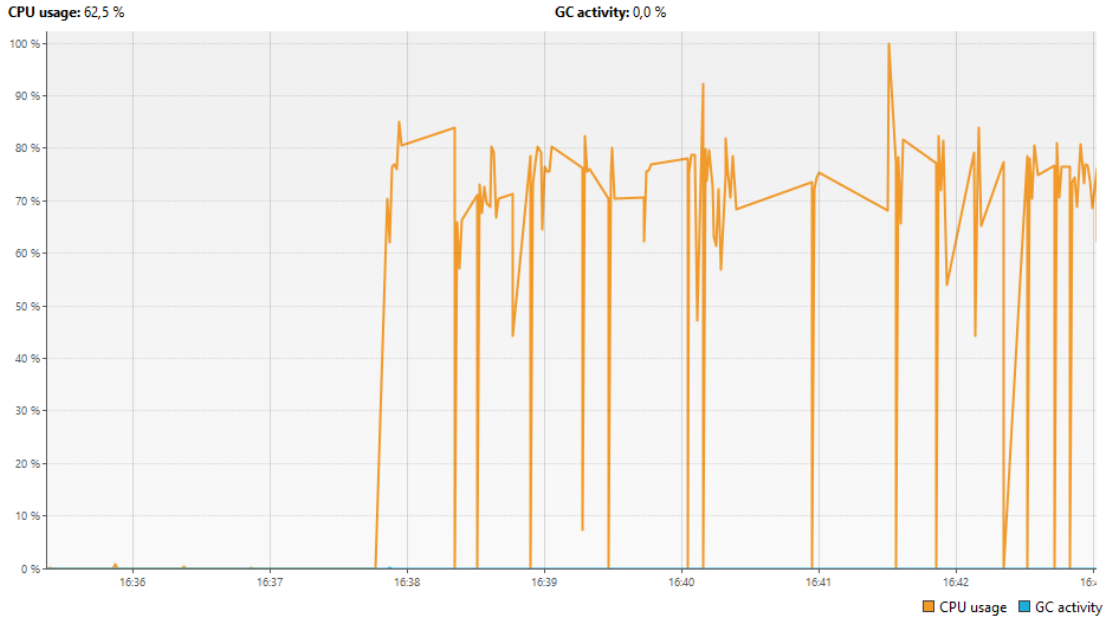


Abbildung 4.49: Spring Boot CPU-Verbrauch mit 4 Cores

Spring Boot Speicherverbrauch

Der Speicherverbrauch von Spring Boot blieb in den Tests der vertikalen Skalierbarkeit relativ konstant, ähnlich den Ergebnissen aus Abschnitt 4.2.2 zur horizontalen Skalierbarkeit. Die maximale Heap-Nutzung war unverändert und blieb auf einem stabilen Niveau. Der tatsächliche Verbrauch variierte in beiden Testdurchläufen leicht, zeigte jedoch im Vergleich zur horizontalen Skalierbarkeit einen Rückgang von bis zu 5 %. Dies deutet darauf hin, dass die vertikale Skalierung (Erhöhung der CPU-Kerne) zwar keine signifikanten Auswirkungen auf den Speicherverbrauch hatte, jedoch eine leichte Verbesserung in der Effizienz des Speicherverbrauchs erzielen konnte.

Vergleich

- **Lasttest-Ergebnisse:** Quarkus konnte in beiden Tests (mit 2 und 4 CPU-Kernen) die Last von 2400 gleichzeitigen Benutzern stabil bewältigen, ohne Fehlanfragen. Mit 4 Kernen verarbeitete Quarkus wesentlich mehr Anfragen, und die Antwortzeiten sanken im Median auf 1600 ms. Spring Boot hingegen verzeichnete bei beiden Tests eine signifikant höhere Fehlerrate. Bei 2 Kernen lag die Fehlerrate bei etwa 81 %, und selbst bei 4 Kernen blieb sie hoch bei 71 %. Die Antwortzeiten bei Spring Boot waren ebenfalls deutlich höher, vor allem im 95. Perzentil, wo sie bis zu 16.000 ms erreichten.
- **CPU-Verbrauch:** Im CPU-Verbrauch zeigte Quarkus im Vergleich zu Spring Boot eine stabilere und effizientere Auslastung. Mit 4 Kernen erreichte Quarkus eine durchschnittliche Auslastung von 55 %, während bei 2 Kernen die Auslastung moderat bei 52 % lag. Bei Spring Boot hingegen erreichte die CPU-Auslastung in beiden Tests Spitzenwerte von 100 %, was auf eine Überlastung des Systems hinweist. Im Durchschnitt lag die Auslastung bei 65 % mit 2 Kernen und bei 70 % mit 4 Kernen, was zeigt, dass die CPU bei Spring Boot in beiden Szenarien stark beansprucht wurde, ohne dass eine gleichmäßige Verteilung der Last erreicht wurde.
- **Speicherverbrauch:** Im Speicherverbrauch waren beide Frameworks relativ konstant. Bei Quarkus konnte eine Reduzierung des Speicherverbrauchs um etwa 10 % festgestellt werden, während Spring Boot eine Reduktion um etwa 5 % verzeichnete. In beiden Fällen blieb die maximale Heap-Nutzung jedoch unverändert.

5 Fazit und Ausblick

5.1 Fazit

Im Verlauf dieser Arbeit wurden die beiden Java-Frameworks Quarkus und Spring Boot detailliert hinsichtlich ihrer Eignung für die lokale Entwicklung und den Betrieb von Microservices anhand der PetClinic-Microservices-Anwendung verglichen. Beide Frameworks bieten spezifische Stärken, die sie je nach Anforderung an eine bestimmte Architektur und Umgebung besonders geeignet machen.

Quarkus zeigte sich in der Anwendungsentwicklung als ressourcenschonendes Framework, das durch seine optimierte Nutzung von Panache den Codeumfang verringert. Die geringere Menge an Boilerplate-Code macht Quarkus in Bezug auf Benutzerfreundlichkeit für Entwickler attraktiver, die eine schlankere Architektur bevorzugen. Spring Boot hingegen bietet eine tiefere Integration in das Spring-Ökosystem und ermöglicht durch eine klare und modulare Datenmodellierung eine flexible und umfangreiche Anwendungsentwicklung. Insbesondere in komplexeren Architekturen, in denen viele Dienste miteinander kommunizieren, bietet Spring mehr Konfigurationsoptionen.

In den durchgeführten Performance-Tests konnte Quarkus insbesondere durch seine schnellere Startzeit und den geringeren Ressourcenverbrauch überzeugen. Dies macht es zu einer guten Wahl in Szenarien, bei denen Anwendungen oft neu gestartet werden oder die Ressourcen limitiert sind. Spring Boot schnitt in Bezug auf Stabilität gut ab, bietet jedoch bei der Startzeit und Antwortzeit in lokalen Umgebungen weniger Effizienz im Vergleich zu Quarkus.

In Bezug auf die horizontale und vertikale Skalierbarkeit zeigte sich, dass Quarkus aufgrund seiner schnellen Startzeit und niedrigen Ressourcennutzung besser für dynamische Skalierungsumgebungen geeignet ist, in denen Microservices schnell hoch- oder heruntergefahren werden müssen. Spring Boot hingegen bietet durch seine umfassende Infrastruktur und Flexibilität eine bessere langfristige Stabilität und Anpassbarkeit in größeren,

beständigen Umgebungen, wo die effiziente Ressourcennutzung durch fortgeschrittene Konfigurationen wie den Spring Cloud Config Server optimiert werden kann.

Abschließend lässt sich festhalten, dass die Wahl zwischen Quarkus und Spring von den spezifischen Anforderungen der Anwendung abhängt. Beide Frameworks haben ihre Daseinsberechtigung und bieten je nach Szenario unterschiedliche Vorteile.

5.2 Ausblick

Für zukünftige Arbeiten bietet es sich an, beide Frameworks in realen Produktionsumgebungen zu testen, um ihre Performance und Skalierbarkeit in verschiedenen Cloud-Infrastrukturen, wie Kubernetes oder OpenShift, noch genauer zu analysieren. Darüber hinaus könnten weiterführende Untersuchungen sich auf spezifische Features konzentrieren, wie die Integration von reaktiven Programmiermodellen oder die native Kompilierung mit GraalVM. Auch die langfristige Wartbarkeit und Erweiterbarkeit von Anwendungen in beiden Frameworks wäre ein spannendes Thema für zukünftige Studien. Schließlich könnten weitergehende Vergleiche zwischen Quarkus und anderen aufkommenden Frameworks helfen, die bestmöglichen Lösungen für unterschiedliche Anwendungsfälle zu identifizieren.

Literaturverzeichnis

[1] Antoine Rey arey: *Spring PetClinic Microservices Sample Application*. 2024. – URL https://github.com/spring-petclinic/spring-petclinic-microservices. – Accessed: 2024-08-23

[2] Baeldung: *Spring @Controller vs @RestController*. – URL https://www.baeldung.com/spring-controller-vs-restcontroller. – Accessed: 2024-09-06

[3] Baeldung: *Spring Boot vs. Quarkus*. 2024. – URL https://www.baeldung.com/spring-boot-vs-quarkus. – Accessed: 2024-08-18

[4] Baker, Jordan: *History of Spring Framework and Spring Boot Framework*. 2019. – URL https://dzone.com/articles/history-of-spring-framework-spring-boot-framework. – Accessed: 2024-08-18

[5] Database, H2: *H2 Database Engine*. – URL https://www.h2database.com/html/main.html. – Accessed: 2024-09-05

[6] Deandrea, Eric: *Quarkus for Spring Developers*. Red Hat Developer, 2021. – URL https://developers.redhat.com/e-books/quarkus-spring-developers. – Accessed: 2024-08-11

[7] Dhaduk, Hiren: *What Are Microservices?* March 10, 2022. – URL https://www.simform.com/blog/what-are-microservices/. – Accessed: 2024-08-16

[8] Goncalves, Antonio: *Understanding Quarkus*. Red Hat, 2020. – URL https://developers.redhat.com/e-books/understanding-quarkus. – Foreword by Emmanuel Bernard, Accessed: 2024-08-11

[9] Gregory, G. ; Bauer, C.: *Java Persistence with Hibernate*. Manning, 2015. – URL https://books.google.de/books?id=8TgzEAAAQBAJ. – Accessed: 2024-09-01. – ISBN 9781638355229

[10] Gupta, Lokesh: *Spring MVC @GetMapping, @PostMapping Annotations*. 2024. – URL https://howtodoinjava.com/spring-mvc/controller-getmapping-postmapping/. – Accessed: 2024-09-06

[11] HSQLDB: *HyperSQL Database (HSQLDB)*. – URL https://hsqldb.org/. – Accessed: 2024-09-05

[12] IBM: *Java Spring Boot*. 2024. – URL https://www.ibm.com/topics/java-spring-boot. – Accessed: 2024-08-14

[13] Infosystems, Crest: *Top 10 Java Frameworks*. 2024. – URL https://medium.com/@crestinfosystems/top-10-java-frameworks-eb32406aa37f. – Accessed: 2024-08-11

[14] James Lewis, Martin F.: *Microservices*. 2014. – URL https://martinfowler.com/articles/microservices.html. – Accessed: 2024-08-15

[15] Janssen, Thorben: *Introduction to Panache*. 2024. – URL https://thorben-janssen.com/introduction-panache/. – Accessed: 2024-09-05

[16] Kapresoft: *Spring vs. Spring Boot*. 2024. – URL https://www.kapresoft.com/java/2024/03/06/spring-vs-spring-boot.html. – Accessed: 2024-08-18

[17] Krivov, Alexey: *Spring Boot vs Quarkus*. 2023. – URL https://maddevs.io/blog/spring-boot-vs-quarkus/. – Accessed: 2024-08-19

[18] Newman, S.: *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015. – URL https://books.google.de/books?id=jjl4BgAAQBAJ. – Accessed: 2024-08-17. – ISBN 9781491950333

[19] Newman, Sam: *Building Microservices*. O'Reilly Media, Inc., 2021. – Accessed: 2024-08-15. – ISBN 978-1492034025

[20] Quarkus: *About Quarkus*. – URL https://quarkus.io/about/. – Accessed: 2024-08-19

[21] Quarkus: *Mutiny Primer Guide*. – URL https://quarkus.io/guides/mutiny-primer. – Accessed: 2024-09-10

[22] Quarkus: *Quarkus Lifecycle Guide*. – URL https://quarkus.io/guides/lifecycle. – Accessed: 2024-08-31

[23] Quarkus: *REST JSON Guide*. – URL https://quarkus.io/guides/rest-json. – Accessed: 2024-09-06

[24] Quarkus: *Stork Guide*. – URL https://quarkus.io/guides/stork. – Accessed: 2024-08-27

[25] Reactor, Project: *Reactor Mono API Documentation*. – URL https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html. – Accessed: 2024-09-09

[26] Red Hat: *What is a Java Framework?* 2024. – URL https://www.redhat.com/de/topics/cloud-native-apps/what-is-a-Java-framework. – Accessed: 2024-08-15

[27] Richardson, C.: *Microservices Patterns: With examples in Java*. Manning, 2018. – URL https://books.google.de/books?id=QTgzEAAAQBAJ. – Accessed: 2024-08-16. – ISBN 9781638356325

[28] Satyabrata, Mr.: *Why Spring Boot: The King of Java Frameworks*. 2023. – URL https://www.tutorialspoint.com/why-spring-boot-the-king-of-java-frameworks. – Accessed: 2024-08-19

[29] Singh, Hiten P.: *Quarkus Framework: An Introduction*. 2023. – URL https://medium.com/hprog99/quarkus-framework-an-introduction-2cec6f17621e. – Accessed: 2024-08-14

[30] SmallRye: *SmallRye Stork for Quarkus*. – URL https://smallrye.io/smallrye-stork/1.0.0.Beta1/quarkus/. – Accessed: 2024-09-10

[31] Soto, Alex: Microservicilities with Quarkus. In: *InfoQ* (2023). – URL https://www.infoq.com/articles/microservicilities-quarkus/. – Accessed: 2024-08-15

[32] Sping: *Spring PetClinic Sample Application*. 2024. – URL https://spring-petclinic.github.io/. – Accessed: 2024-08-22

[33] Spring: *Spring Cloud CircuitBreaker*. – URL https://spring.io/projects/spring-cloud-circuitbreaker. – Accessed: 2024-09-09

[34] Spring: *Spring Framework WebFlux WebClient*. – URL https://docs.spring.io/spring-framework/reference/web/webflux-webclient.html. – Accessed: 2024-09-09

[35] Tozzi, Chris: *When not to use microservices: Challenges to consider*. 2024. – URL https://www.techtarget.com/searchapparchitecture/tip/When-not-to-use-microservices-Challenges-to-consider. – Accessed: 2024-08-17

[36] Walls, C.: *Spring Boot in Action*. Manning, 2015. – URL https://books.google.de/books?id=IzkzEAAAQBAJ. – ISBN 9781638353584

A Anhang

A.1 Verwendete Hilfsmittel

In der Tabelle A.1 sind die im Rahmen der Bearbeitung des Themas der Bachelorarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Tool	Verwendung
LaTeX	Textsatz- und Layout-Werkzeug verwendet zur Erstellung dieses Dokuments

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original