Evaluierung von Spring-Boot-basierten Microservices: Projekterfahrungen mit Java, Hibernate und MariaDB

Marc Ziegler und Alessandro Fenrich
Technische Hochschule Mannheim
Fakultät für Informatik
Paul-Wittsack-Str. 10, 68163 Mannheim

Inhaltsverzeichnis

1. Einleitung

Die vorliegende Arbeit befasst sich mit der Planung und Implementierung einer Microservice-Architektur, welche am Beispiel einer Bibliotheksverwaltung demonstriert wird. Dies soll durch den Einsatz der Java-Frameworks Spring Boot und Hibernate sowie der Containerisierung mittels Docker-Compose einen Prototypen mit drei eigenständigen Diensten umgesetzt werden. Die Verwendung von Spring Boot ermöglicht dabei eine schnelle Erstellung von eigenständigen Anwendungen, während Hibernate als Objekt-Relational-Mapping-Lösung die Datenbankintegration erleichtert[29]. Jeder dieser Microservices übernimmt jeweils eine der klar festgelegten Aufgaben, nämlich die Verwaltung von Büchern, Benutzerdaten und der Ausleihen. Diese modulare Aufteilung soll die Vorteile einer Microservice-Architektur durch die klare Abtrennung der Aufgaben, die Verwendung von unabhängig skalierbaren Diensten und der flexiblen Bereitstellung einzelner Komponenten veranschaulichen[27].

Die Relevanz dieses Ansatzes zeigt sich durch die zunehmend komplexeren Anforderungen an moderne Softwaresysteme und beschleunigter Entwicklungszyklen. Monolithische Architekturen stoßen bei der Weiterentwicklung schnell an Grenzen, da Änderungen oft aufwendige Tests und Abstimmungen des gesamten Systems erfordern[4]. Dagegen steht eine Microservice-Architektur, welche Fachbereiche in selbstständige Dienste zerlegt, die jeweils unabhängig entwickelt, eingesetzt und skaliert werden können[4]. Dies ermöglicht unabhängige Entwicklungs- und Testzyklen einzelner Komponenten[4], wobei die Containerisierung mittels Werkzeugen wie Docker-Compose dieses Konzept mittels einheitlichen und reproduzierbaren Diensten unterstützt[63].

Zunächst werden die grundlegenden Technologien und Konzepte erklärt, insbesondere die Funktionsweisen von Spring Boot, Hibernate und Containerisierung mit Docker. Anschließend wird der entwickelte Prototyp im Detail vorgestellt. Mit der Erklärung des Aufbaus, der Funktionsweise und den Kommunikationsmechanismen soll das Zusammenspiel der drei Microservices anschaulich beschrieben werden. Abschließend erfolgt eine kritische Bewertung der verwendeten Technologien und Architekturansätze.

Ziel der Analyse ist es, fundierte Erkenntnisse über den praktischen Nutzen von Microservice-Architekturen zu gewinnen und die Wahl der richtigen Technologie zu unterstützen. Die Vor- und Nachteile, welche sich bei der gewählten Beispielanwendung ergeben, werden untersucht und die Herausforderungen von Spring Boot, Hibernate und Docker adressiert. Besonderes Augenmerk liegt dabei auf der Skalierbarkeit, Wartbarkeit und Performance der Dienste sowie auf dem Entwicklungs- und Betriebsaufwand.

2. Grundlagen und Konzepte

2.1. Microservices

Die Microservice-Architektur bezeichnet einen Architekturstill, bei welchem eine Anwendung aus vielen kleinen, eigenständigen Diensten aufgebaut wird. Jeder Microservice kümmert sich nur um eine eigen, ihm einzigartige Aufgabe inklusive eigener Logik und oft auch eigener Datenhaltung. Die Dienste kommunizieren über vordefinierte Schnittstellen, typischerweise per REST oder über Messaging-Protokolle. Diese Architektur hat zum Vorteil, dass durch die lose Koppelung der Systemkomponenten, sowie deren kleine Größe eine flexible Skalierung und Entwicklung der Anwendung erlaubt wird.[7]

Im Gegensatz zur monolithischen Architektur, bei der fast alle Funktionen innerhalb einer einzigen, zusammenhängenden Anwendungseinheit implementiert sind, fördern Microservices Modularität und Unabhängigkeit[7]. Durch die Unabhängigkeit der Services ist es bei arbeiten an einem Dienst nicht Erforderlich ein Neu-Deployment des Gesamtsystems und erlauben Continuous Integration und Continuous Deployment (CI/CD)[27]. Zudem kann jeder Dienst unabhängig skaliert und mit einem eigenen Technologie-Stack betrieben werden.[7]

Die Vorteile dieses Ansatzes liegen in der erhöhten Wartbarkeit, der verbesserten Fehlertoleranz und der Möglichkeit an der paralleler Entwicklung[7].

Gleichzeitig ergeben sich auch Herausforderungen: Die verteilte Architektur erfordert zusätzlichen Aufwand beim Monitoring, Logging und bei der Verwaltung von Abhängigkeiten[27]. Komplexitäten entstehen durch Netzwerkkommunikation, Service-Discovery, Datenkonsistenz und Sicherstellung von Transaktionen über mehrere Dienste hinweg[4]. Insbesondere in kleinen Projekten oder bei begrenzter Infrastruktur können diese Anforderungen den Nutzen übersteigen[27].

2.2. Containerisierung und Orchestrierung

Containerisierung ist ein zentraler Baustein zur Umsetzung verteilter Architekturen wie Microservices[15]. Durch Sie wird es erst möglich, Anwendungen mitsamt aller Abhängigkeiten in isolierten Umgebungen zu betreiben, welche unabhängig vom darunterliegenden Betriebssystem funktionieren[15]. Container verwenden den Kernel des Hosts gemeinsam und ermöglichen dennoch Prozess- und Ressourcentrennung über Technologien wie Cgroups und Namespaces[10], [15].

Ein weit verbreitetes Werkzeug, welches auch wir uns zu Nutzen gemacht haben, zur Verwaltung von Containerumgebungen ist Docker[15]. Für komplexere Systeme kommt häufig *Docker Compose* zum Einsatz. Dieses erlaubt die Definition mehrerer Containerdienste in einer gemeinsamen YAML-Datei, inklusive Netzwerk- und Volumenkonfiguration, für eine übersichtliche Verwaltung der Container. Durch die automatische Erstellung virtueller Netzwerke wird die Kommunikation zwischen Containern stark vereinfacht, da sich diese so auch untereinander per Servicenamen ansprechen lassen.[14]

Im Vergleich zu lokal installierter Software oder virtuellen Maschinen (VMs) haben Container einige Vorteile: Sie sind ressourcenschonender, starten schneller und lassen sich einfacher portieren[15]. Während VMs ein vollständiges Gastbetriebssystem benötigen, virtualisieren Container lediglich auf Anwendungsebene. Dies reduziert den Overhead erheblich und erlaubt eine höhere Dichte pro Host[63].

Ein Nachteil, welche die Containerisierung mit sich bringt, sind schlechtere Sicherheitsisolation gegenüber den VMs, da sich Container denselben Kernel teilen[34]. Persistente Datenhaltung sowie Netzwerk- und Speicherintegration können komplex sein, besonders im produktiven Betrieb[34], [63]. Auch der Verwaltungsaufwand wächst mit der Anzahl der Container, weshalb bei größeren Systemen zusätzliche Werkzeuge wie Kubernetes zur Orchestrierung notwendig werden. Diese erhöhen jedoch wiederum die Einstiegshürde und erfordern spezialisiertes Wissen im Bereich DevOps und Systembetrieb.[34], [63]

3. Technologieüberblick

3.1. Java und Spring Boot

Spring Boot ist ein modernes Framework zur Entwicklung von Java-basierten Anwendungen und erweitert das etablierte Spring Framework um zusätzliche Funktionen zur Vereinfachung und Beschleunigung der Anwendungsentwicklung[21]. Entwickelt wurde das Spring Boot Framework Anfang 2013 basierend auf Java mit dem Ziel, produktionsreife, eigenständig lauffähige Anwendungen mit minimalem Konfigurationsaufwand zu ermöglichen[21], [67]. Als Teil des Java-Ökosystems nutzt Spring Boot die objektorientierte Programmierung und die Java Virtual Machine (JVM), wobei es dabei auf eine enge Integration mit bestehenden Java-Werkzeugen wie Maven oder Gradle setzt[30].

3.1.1. Kernkonzepte von Spring Boot. *„Konvention vor Konfiguration“* und *automatische Konfiguration* (engl. *autoconfiguration*) stellt eines der Kernkonzepte dar. Dabei analysiert das Framework beim Start automatisch den Klassenpfad sowie vorhandene Abhängigkeiten, welche es als Basis für sinnvolle Standardkonfigurationen verwendet. Dadurch entfällt in vielen Fällen die Notwendigkeit manueller Konfiguration, und es wird der sogenannte Boilerplate-Code und Aufwand deutlich reduziert.[23], [24], [29]

Ein wesentliches Konzept stellen die *Starter-Abhängigkeiten* (engl. *Spring Starters*) dar. Diese bündeln für häufige Anwendungsszenarien (z.B. Webentwicklung, Datenbankzugriffe, Sicherheit) sämtliche benötigten Bibliotheken in einer vorkonfigurierten Abhängigkeit. Dadurch lassen sich neue Projekte mit nur wenigen Abhängigkeitseinträgen effizient aufsetzen.[29]

Ein zentrales und wichtiges Element von Spring Boot ist zudem das Prinzip der *Dependency Injection (DI)*. Über Annotationsmechanismen (z.B. `@Component`, `@Service`, `@Autowired`) können Komponenten und ihre Abhängigkeiten deklariert werden. Diese werden von dem Spring-Container bei Bedarf instanziiert und zur Verfügung gestellt. Diese entkoppelte Architektur erhöht die Modularität, Testbarkeit und Wartbarkeit der Anwendung erheblich.[3], [29]

Ein weiteres zentrales Konzept ist die Nutzung *eingebetteter Server*. Spring Boot-Anwendungen haben bereits Webserver wie Tomcat, Jetty oder Undertow integriert. Ein separates Deployment auf einem externen Applikationsserver ist dadurch nicht erforderlich. Dies vereinfacht sowohl Entwicklungs- als auch Betriebsprozesse.[13], [67]

Insgesamt verfolgt Spring Boot den sogenannten *„opinionated“* Ansatz, welcher auf vorkonfigurierten Konventionen basiert. Dieser ermöglicht eine schnelle Projekterstellung, ohne jedoch die Flexibilität für individuelle Konfigurationen einzuschränken.[13], [22]

3.1.2. Vorteile und Nutzen. Die wesentlichen Vorteile von Spring Boot liegen in einer erheblichen Steigerung der Entwicklungsgeschwindigkeit und Produktivität[13], [67]. Durch die Kernelemente wie Autokonfiguration, Starter-Pakete und eingebettete Server können Entwickler sehr schnell lauffähige Anwendungen erstellen[67]. Der Verzicht auf umfangreiche XML-Konfigurationen und die Integration moderner Entwicklungswerkzeuge (z.B. Spring Initializr, Spring Boot DevTools) tragen zusätzlich zu einem schlanken und effizienten Entwicklungsprozess bei[23].

Darüber hinaus bringt Spring Boot viele produktionsrelevante Features wie Health-Checks, Aktuator-Endpunkte, Metriken und externe Konfigurationsmöglichkeiten bereits im Standardumfang mit. Diese Aspekte erleichtern insbesondere die Integration in Continuous-Deployment- und Cloud-Umgebungen.[67]

Spring Boot ist zudem besonders gut für Microservice-Architekturen geeignet. Dessen Fähigkeit, unabhängige, modularisierte Dienste mit minimalem Konfigurationsaufwand zu erstellen und bereitzustellen, macht es zu einem guten Werkzeug für verteilte Systeme und serviceorientierte Architekturen.[24]

3.1.3. Limitierungen und Herausforderungen. Trotz der zahlreichen Vorteile sind auch bestimmte Limitierungen und Herausforderungen zu berücksichtigen. So kann die umfassende Autokonfiguration zu einem gewissen Grad an Intransparenz führen: Entwickler wissen nicht immer exakt, welche Konfigurationen oder Komponenten aktiv sind, was insbesondere beim Debugging oder bei der Fehlersuche zu Schwierigkeiten führen kann[23], [67].

Des Weiteren ist Spring Boot im Vergleich zu leichtgewichtigeren Frameworks ressourcenintensiver hinsichtlich Speicherverbrauch und Startzeit. Dies kann zu Problemen bei sehr restriktiven Betriebsumgebungen führen. Die Integration von zu vielen Funktionen und Abhängigkeiten kann außerdem zu Abhängigkeitsballung führen. Dadurch können Anwendungen komplexer und potenziell fehleranfälliger werden.[67]

Letzten Endes erfordert der Umgang mit der Abstraktionsebene von Spring Boot ein grundlegendes Verständnis der Spring-Mechanismen, insbesondere im Bereich der Abhängigkeitsinjektion und des Komponenten-Scannings. Ohne dieses Verständnis können selbst kleine Änderungen an der Konfiguration unbeabsichtigte Nebenwirkungen haben.[67]

3.2. Hibernate und ORM

Eines der größeren Herausforderungen in der Entwicklung datenbankgestützter Anwendungen ist die effiziente Verwaltung persistenter Daten. Die objektorientierte Programmierung (OOP) steht dabei in einem strukturellen Widerspruch zu den relationalen Datenbanksystemen, dabei hauptsächlich bei ihren Datenmodellen. Dieses sogenannte object-relational impedance mismatch wird dabei von dem Object-Relational Mapping (ORM)adressiert, welches eine automatisierte Abbildung zwischen objektorientierten Klassen und relationalen Tabellen ermöglicht. ORM-Frameworks abstrahieren die Interaktion mit der Datenbank und bieten Entwicklern ein deklaratives, objektzentriertes Zugriffsmuster.[6], [28]

3.2.1. Grundlagen von ORM. Object-Relational Mapping beschreibt den Mechanismus, bei welchem Daten aus einer relationalen Datenbank (z.B. Tabellen, Zeilen, Spalten) in Objekte einer Programmiersprache (z.B. Java-Klassen) überführt werden – und umgekehrt. Das Ziel ist es, den Datenzugriff für Entwickler in Objekte um zu gestalten, die den Prinzipien der Objektorientierung entsprechen. Und das ohne SQL-Befehle selbst schreiben zu müssen. ORM-Tools übernehmen dabei das Mapping zwischen der Datenbankstruktur und dem Objektmodell und verwalten die Persistenz der Objekte. Sie unterstützen Transaktionen, Lazy Loading, Caching und mehr. Durch diese wird nicht nur die Entwicklung, sondern auch die Wartung und Testbarkeit von Anwendungen erleichtert.[5], [6], [28]

3.2.2. Hibernate. Hibernate ist eines der bekanntesten und am weitesten verbreiteten ORM-Frameworks in Java. Es bietet umfangreiche Funktionen zur objekt-relationalen Abbildung, Datenbankabstraktion und zur Verwaltung von Datenpersistenz [9], [64]. Zu den zentralen Merkmalen von Hibernate zählen:

- **Automatisches Mapping:** Hibernate erlaubt die Abbildung von Java-Klassen auf Datenbanktabellen mittels Annotationen oder XML-Konfiguration. Beziehungen wie `@OneToMany` oder `@ManyToOne` können deklariert werden.[9], [11]
- **Datenbankunabhängigkeit:** Durch Verwendung von Dialekten kann dieselbe Anwendung mit verschiedenen Datenbanksystemen betrieben werden, ohne dass SQL angepasst werden muss.[9], [11], [17]
- **HQL (Hibernate Query Language):** Hibernate bietet eine objektorientierte Abfragesprache, die über Java-Objekte formuliert wird, anstatt direkt SQL zu verwenden.[11]
- **Caching:** Mehrstufige Caching-Strategien (First-Level und Second-Level Cache) verbessern die Performance durch Reduktion von Datenbankzugriffen.[64]
- **Transaktionsmanagement:** Hibernate integriert sich nahtlos in JTA (Java Transaction API) und unterstützt deklaratives Transaktionsmanagement.[17]

3.3. Relationale Datenbanken

Relationale Datenbanken folgen dem klassischen Tabellenmodell, bei welchem Daten in Tabellen mit definierten Spalten (Datenattribute) zeilenweise (Dateninstanzen) gespeichert werden. Jede Tabelle erhält einen Primärschlüssel anhand dessen sie eindeutig identifiziert werden kann, wobei Beziehungen zwischen Tabellen über Fremdschlüssel abgebildet werden. Dieser Standardansatz mit festem Schema ermöglicht eine hohe Datenintegrität.[16], [65]

Vorteile sieht man bei relationalen Datenbanken in der breiten Unterstützung sowie einfachen Programmierung und Verwaltung, da es sich bei ihnen um eine ziemlich einfach zu verstehende Art handelt[32]. Nahezu alle Tools, Frameworks und Cloud-Plattformen haben Treiber für gängige RDBMS, und durch ihre lange und häufige Verwendung gibt es eine große Community von Experten und Best Practices. Erfahrene Entwickler schätzen außerdem, dass sich durch SQL-Abfragen komplexe Operationen über mehrere verknüpfte Tabellen hinweg durchführen lassen.[8], [65]

Grenzen gibt es bei der horizontalen Skalierbarkeit von relationalen Datenbanken. Um die Kapazität zu erhöhen, muss man die Hardware des Servers verstärken (mehr CPU, RAM, schnellere SSD) oder auf einen besseren Server wechseln[32]. Dieses vertikale Skalierungs-Konzept kann schnell teuer werden und erfordert aufwändige Wartung. Zudem ist das eher statische Schema bei sich häufig ändernden oder unstrukturierten Daten klar im Nachteil. Die Strukturänderungen sind sehr aufwendig und das Modell stößt bei extrem großen Datenmengen oder sehr dynamischen Daten (z.B. in Echtzeit-Webanwendungen) an Grenzen[32]. In solchen Fällen sind relationale Systeme performancemäßig ineffizient, da sie bei hohen Lasten komplexe Joins und Transaktionen abwickeln müssen.[8]

3.4. Nicht-relationale Datenbanken

Nicht-relationale Datenbanken, oder NoSQL, umfassen eine Vielfalt an Datenmodellen. NoSQL-Systeme verzichten auf ein festes Schema und erlauben es, die für die Anwendung besten Datenstrukturen frei zu wählen. Typische Anwendungsfälle dieser Art sind sehr große, schnelle und verteilte Datenmengen.[8], [16] Dabei punkten NoSQL-Datenbanken besonders durch ihre horizontale Skalierbarkeit[33]. Sie können kostengünstig auf viele Server oder Cloud-Instanzen verteilt werden[8]. Open Source NoSQL-Systeme (z.B. MongoDB, Cassandra, Redis) bieten erschwingliche Optionen, da sie sich besonders für Cloud-Umgebungen oder schnell wachsende Datenbestände eignen, da die Kapazität einfach und schnell durch das Hinzufügen weiterer Knoten erweitert werden kann[8].

Einer der Nachteile von NOSQL ist die Abwesenheit von einer einheitlichen Abfragesprache für alle NoSQL-Datenbanken, jeder Typ verwendet eine eigene Syntax. Die Community ist insgesamt noch jünger und kleiner als bei SQL, sodass es zu speziellen Problemen nicht immer möglich ist, auf ausgereifte Standards zurückzugreifen. Häufig fehlt eine ACID-Transaktionsunterstützung. Viele NoSQL-DBs verfolgen stattdessen ein *Eventually Consistency-Modell*, bei dem die Daten

nach kurzer Zeit konsistent werden. Abfragen über verteilte Datenstrukturen können ineffizienter sein. Komplexe Joins und Abfragen müssen programmatisch realisiert werden, was erfahrene Entwickler erfordert. Aufgrund dieser Dezentralisierung kann es passieren, dass Leseanfragen kurzzeitig veraltete Daten zurückliefern (etwas, das in manchen Anwendungen tolerierbar ist, z.B. bei Social-Media-Feeds). Insgesamt gilt daher: NoSQL-Systeme bieten hohe Flexibilität und Skalierbarkeit auf Kosten von Konsistenz und Standardisierung, während relationale Systeme umgekehrt hohe Integrität und Struktur gewährleisten.[8]

3.5. Java Persistence API (JPA)

Die Java Persistence API (JPA) ist ein Standard-Framework der Java-Umgebung, welches die Abbildung zwischen Java-Objekten und relationalen Datenbanken vereinheitlicht. Anstatt manuell JDBC-Code zu schreiben, definieren Entwickler Entities als Java-Klassen und greifen über das EntityManager-Interface auf Daten zu(Siehe Kapitel 2.3.1).[21], [25]

3.6. Java Database Connectivity (JDBC)

JDBC ist eine Java-Schnittstelle für den direkten Datenbankzugriff[26]. Dafür stellt es Klassen und Interfaces für die Kommunikation über SQL-Abfragen mit relationalen Datenbanken bereit[26]. In der klassischen Zwei-Schichten-Architektur verbindet sich die Java-Anwendung über einen JDBC-Treiber direkt mit einem Datenbankserver[26]. Dadurch erzeugt die Anwendung SQL-Statements, welche der JDBC-Treiber an die Datenbank sendet, und die Ergebnisse (Result-Sets) zurückgeliefert[26]. Es existieren auch Drei-Schichten-Szenarien, in denen ein Anwendungsserver zwischen Anwendung und Datenbank vermittelt, aber auch hier arbeitet der Kern über JDBC[26]. JDBC arbeitet in einer standardisierten Zwischenschicht[26]. Sie definiert, wie eine Anwendung Verbindungsdaten übergibt und Ergebnisse ausliest, ohne dass die Java-Logik sich um die Details der jeweiligen Datenbank-API kümmern muss[26].

3.7. H2-Datenbank

Die H2-Datenbank ist ein in Java geschriebenes RDBMS mit Open-Source-Lizenz. Sie kann embedded im Java-Programm oder als eigener Server betrieben werden und bietet dabei als besonderes Feature den reinen In-Memory-Betrieb. Dabei werden alle Daten nur im Arbeitsspeicher gehalten, was für schnelle Zugriffszeiten sorgt. Durch diese Eigenschaft macht sich H2 besonders nützlich für Entwicklungs- und Testzwecke. Entwickler können mit ihr sehr schnell eine vollwertige Datenbankinstanz starten, Daten erzeugen oder migrieren und anschließend ohne großen Aufwand wieder verwerfen. H2 zeichnet sich durch eine sehr einfache Konfiguration aus und lässt sich gut in Java-Frameworks wie Spring Boot oder Hibernate (z.B. über den H2-JDBC-Treiber) integrieren. Damit wird das schnelle Aufsetzen einer Testdatenbank ermöglicht, ohne eine komplexe Infrastruktur bereitzustellen.[69]

3.8. MariaDB

MariaDB ist ein relationales Datenbankmanagementsystem, erstellt 2009 als Fork von MySQL durch die ursprünglichen MySQL-Entwickler[31]. Ziel war es, eine vollständig MySQL-kompatible Alternative zu schaffen, die Open Source bleibt[31]. Bis einschließlich MySQL Version 5.1 bzw. 7 wurde größtenteils Binärkompatibilität zwischen MySQL und MariaDB angestrebt[31]. Alle MySQL-Konnektoren (z.B. JDBC-Treiber) lassen sich ohne Anpassungen auch direkt mit MariaDB nutzen, sodass Anwendungen in der Regel problemlos zwischen beiden Datenbanken wechseln können[20]. MariaDB ist vollständig unter der GPL lizenziert und stellt die meisten Funktionen der Enterprise-DB auch in der Community-Version bereit[66]. Im Vergleich zu MySQL hat MariaDB neben dem Kerndatenbankserver verschiedene Storage-Engines eingebaut[31]. Neben dem Standard-InnoDB stehen zum Beispiel Aria (ein Crash-freundliches Äquivalent zu MyISAM) und MyRocks (für SSD-optimierte Speicherung) zur Verfügung[31]. Dies bietet Entwicklern Flexibilität, die Datenhaltung an spezielle Anwendungsfälle anzupassen[31].

3.9. Auswahlkriterien für Datenbanksysteme

Um die richtige Datenbank bereits am Anfang des Projektes aussuchen zu können, muss man sich mit ein paar Punkten auseinandersetzen. Zunächst muss der Use Case klar sein. Handelt es sich um stark strukturierte Daten mit komplexen Beziehungen, bietet sich ein relationales Modell an. Dominieren flexible/unklar strukturierte Datenmengen (z.B. Dokumente, Sensorlogs)? Bei großen, verteilten Workloads mit extrem niedrigen Latenzanforderungen können NoSQL-Systeme besser geeignet sein, da sie horizontal skalieren und schemalose Modelle unterstützen.[33]

Weitere wichtige Kriterien sind Leistung und Skalierbarkeit. Muss die Datenbank mit steigendem Datenvolumen und vielen gleichzeitigen Nutzern mithalten können? Hierbei sollte man prüfen, ob das System vertikal skalierbar ist oder zusätzlich horizontale Mechanismen (Clustering, Sharding) bietet. Sicherheit und Zuverlässigkeit sind essenziell, also muss das DBMS geeignete Authentifizierung, Verschlüsselung und Hochverfügbarkeitsfunktionen unterstützen.[16]

Ein weiteres Kriterium ist Kompatibilität und Ökosystem. Kann man die Datenbank nahtlos in bestehende Anwendungstechnologien integrieren mit zum Beispiel bereits existierenden Anschlüssen und Bibliotheken? Steht ein aktives Community- oder Unternehmenssupport-Netzwerk zur Verfügung? Eine moderne Datenbank sollte zudem zukünftige Anforderungen antizipieren, integrierte Features wie Volltextsuche, Analytics oder native Unterstützung für „Dokumente“ (z.B. als JSON/NoSQL-Modus) können künftige Entwicklungen erleichtern. Da heute viele Anwendungen in der Cloud laufen, ist Cloud-Fähigkeit (Cloud-native Architektur, automatisches Scaling, Containerisierung) ebenfalls noch zu einem wichtigen Auswahlkriterium geworden.[32]

Letztlich darf auch der finanzielle Part nicht vernachlässigt werden: Lizenzmodell (Open Source vs. kommerziell), benötigte Hardware und Wartungskosten müssen ins Budget passen.[32]

Zusammengefasst sollte die Entscheidung für ein Datenbanksystem auf einer Abwägung dieser Kriterien basieren. Leistung, Skalierbarkeit, Sicherheit, Flexibilität und die passende Datenmodellierung sind entscheidend. Nur so lässt sich die Datenbank finden, die den Anforderungen des Projekts langfristig gerecht wird.[32]

4. Konzeption des Prototyps

In diesem Kapitel wird die Konzeption des verwendeten Prototyps erläutert. Hierfür relevant sind zum einen die Kernanforderungen, die im Rahmen des Moduls Anwendungscontainer und Docker (ACD) festgelegt wurden, und zum anderen das Kreieren eines realitätsnahen Szenarios, das so auch auf dem freien Markt Verwendung finden könnte. Zunächst werden die Anforderungen und unsere Zielsetzung beschrieben, gefolgt von einer Architekturübersicht, die die einzelnen Microservices genauer unter die Lupe nimmt. Zuletzt widmen wir uns dem Datenmodell und seiner Persistenz.

4.1. Zielsetzung und Anforderungen

Den Rahmen für unseren Prototypen und somit unsere wissenschaftliche Ausarbeitung bildet das ausgewählte Thema. Die technische Anforderung hierfür ist das Erstellen und

Verknüpfen zweier Microservices mit zusätzlicher Persistenz in Form einer Datenbankanbindung an MariaDB[1]. Die zwei Microservices müssen mit der Programmiersprache Java[2] und dem Open-Source-Framework Spring-Boot[3] erstellt werden. Die Übersetzung der Persistierung von Java-Objekten in eine für relationale Datenbanken geeignete Sprache muss Hibernate[4] übernehmen. Es ist erforderlich, dass die Microservices sowie die Datenbanken als Images portierbar und als Container in Docker[5] ausführbar sind. Die Kommunikation zwischen den Services erfolgt von Container zu Container in einem eigenen Subnetz. Die Datenbanken sollen nur von ihren zugehörigen Microservices erreichbar sein, um sie vor Manipulation von außen zu schützen. Die gesamte Anwendung mit allen Services und Datenbanken muss in einer Docker-Compose-Datei definiert und mit einem Befehl ausführbar sein.

Die Wahl der fachlichen Domäne der Anwendung wurde nicht festgelegt oder beschränkt. Wir haben uns für die Simulation einer Bibliotheksanwendung im kleinstmöglichen Umfang entschieden. Sie stellt im Kern den Gedanken einer möglichen Produktivanwendung auf dem freien Markt dar. Unsere Anwendung löst die Aufgabe, das Ausleihen von Büchern in einer Bibliothek mit Nutzern und dem Bestand von Büchern zu vernetzen, damit es zu jeder Zeit eine konsistente Datenlage zur Leihe von Büchern, dem Bestand und eine Zuordnung zum ausleihenden Nutzer gibt. Um die Anwendung in Microservices zu realisieren, gibt es drei entkoppelte Microservices, die jeweils einen in sich geschlossenen Teilbereich der Fachdomäne repräsentieren und für ihre eigene Datenhaltung zuständig sind. Die drei Microservices kommunizieren mittels Hypertext Transfer Protocol (HTTP) miteinander und nutzen für die Daten das JSON-Format[6].

Ziel dieser Anwendung ist es *nicht*, Sicherheitskonzepte, Skalierbarkeit innerhalb der Anwendung durch Nebenläufigkeit oder technisch einwandfreies Transaktionsmanagement zur Datenintegrität umzusetzen. Ebenfalls ist die Effizienz der Anwendung in Bezug auf die Laufzeitnutzung zweitrangig. Sie dient lediglich als Testobjekt, um die Durchführbarkeit einer Spring-Boot-Anwendung mit Datenbankanbindung im Container als Microservices zu realisieren. Während genau dieser Implementierung stoßen wir auf Herausforderungen, die uns die Möglichkeit bieten, zu evaluieren, ob und für welche Art von Anwendungen dieser Technologie-Stack geeignet ist.

4.2. Architekturübersicht

4.2.1. Architekturansatz. Unser Prototyp verfolgt eine Microservice-basierte Architektur. Die fachlichen Domänen Nutzerverwaltung, Bücherverwaltung und Ausleihverwaltung werden durch eigenständige, entkoppelte Microservices klar getrennt und kommunizieren über konsistente, fest definierte Representational State Transfer (REST)-ful Application Programming Interfaces (API)s. Wir folgen bei der Gestaltung der APIs den Empfehlungen von Microsoft[68], einem führenden Hersteller von Softwareprodukten und Betreiber der Azure-Cloud.

1. Die Homepage der MariaDB Foundation für weitere Informationen https://mariadb.org/

2. OpenJDK für eine unabhängige Nutzung von Java https://openjdk.org/

3. Spring-Boot als Open-Source Java Framework welches auf Spring-Framework basiert https://spring.io/projects/spring-boot

4. Open-Source Java Bibliothek für das Object Relational Mapping (ORM) unter Verwendung von Java Persistence API (JPA) https://hibernate.org/

5. Isolierung von Anwendungen und Containervirtualisierung mit Docker https://www.docker.com/

6. JavaScript Object Notation (JSON) als kompaktes, gut für Menschen und Maschinen lesbares Datenformat https://www.json.org/json-en.html

Alle drei Microservices besitzen jeweils den gleichen Technologie-Stack, der aus einer Containerumgebung, einer Java-Laufzeitumgebung, dem Spring-Boot-Framework und zusätzlichen Spring-Modulen besteht. Die Module werden durch Abhängigkeiten beim Bau der Anwendung mit dem Maven-Plugin eingebunden und haben wiederum ihrerseits Abhängigkeiten, wie z.B.Hibernate, eine Java-Bibliothek zur technischen Umsetzung von JPA, oder Jackson, eine Java-Bibliothek, die der Serialisierung und Deserialisierung von Java-Objekten in JSON übernimmt. Jeder Microservice erhält durch das Einbinden von Spring-Boot einen eingebetteten Webserver. Dieser ist in der Standardkonfiguration ein Tomcat-Server und wird beim Anwendungsstart ebenfalls gestartet.

Die drei Microservices nutzen zur Persistierung jeweils eine eigene Datenbank. Für die Speicherung wurde eine relationale Open-Source-Datenbank vorgegeben – MariaDB. Somit sind alle drei Microservices voneinander unabhängig und teilen sich, falls lokal ausgeführt, nur den Host-Linux-Kernel.

4.2.2. Die Microservices. Das Usermanagement (UM) dient der Verwaltung von Nutzern in unserer Anwendung. Es bedient keine Industriestandards wie OpenID Connect (OIDC)[7] zur Authentifizierung oder die Verwaltung von Nutzerrollen mit Open Authorization (OAuth)[8]. Für den Produktivbetrieb sollte auf eine Open-Source-Lösung, z.B.Keycloak[9], gesetzt werden, die diese Aufgaben übernimmt. Unser UM ist nur für die Erzeugung und Haltung von Nutzern zuständig und ist somit exemplarisch für einen runden Versuchsaufbau anzusehen.

Der Book-Service (BS) dient dazu, neue Bücher in seinem Bestand aufzunehmen, den Bücherbestand gefiltert oder ungefiltert auszugeben, Metadaten eines bestimmten Buches zu modifizieren und den Zustand der Verfügbarkeit abzufragen oder zu ändern. Wichtig ist hier, dass der BS für seine Bücher selbst den Zustand, also die Verfügbarkeit, verwaltet. Hierdurch kann eine zu enge Bindung oder auch Abhängigkeit zwischen dem BS und dem Verleih-Service vermieden werden. Fällt z.B.der Verleih-Service aus, oder wird durch einen anderen Microservice mit der gleichen API ersetzt, ist noch immer bekannt, ob das Buch verliehen ist oder nicht.

Der Book-Rental-Service (BRS) repräsentiert das Leihverzeichnis einer Bücherei. Seine Aufgaben sind wie folgt definiert:

- Mit der Buch-ID, der User-ID und einer Leihdauer kann ein Buch ausgeliehen werden. Eine Leihe weiß immer, welcher Nutzer welches Buch und für welchen Zeitraum ausgeliehen hat.
- Wird ein Buch ausgeliehen, informiert der BRS den BS, dass das Buch mit der jeweiligen Buch-ID verliehen wurde. Somit hat der BS keine Informationen zu der Leihe – er ist entkoppelt.
- Er liefert die Informationen einer Leihe gefiltert durch die Leihe-ID, die Buch-ID oder die User-ID.
- Er bietet die Möglichkeit, ein geliehenes Buch zurückzugeben. Hierdurch wird das Buch im BS wieder auf den Status verfügbar gesetzt.

Alle drei Microservices gemeinsam bilden für unsere Anwendung ein Minimum Viable Product (MVP). Sie dienen als Testprototypen, um Herausforderungen während der Entwicklung zu identifizieren und mögliche Vorteile bzw.Nachteile abschätzen zu können.

4.2.3. Kommunikationswege und API. Microservices kommunizieren in der Regel mit leichtgewichtigen Protokollen wie z.B.HTTP und verwenden APIs die auf REST basieren[68].

7. Authentifizierung definiert durch das OIDC-Protokoll https://openid.net/developers/how-connect-works/

8. Autorisierung mittels Zugriffstokens https://auth0.com/de/intro-to-iam/what-is-oauth-2

9. Identitäts- und Zugriffsverwaltung https://www.keycloak.org/

Für die visuelle Unterstützung bei der Entwicklung einer geeigneten API, haben wir uns für OpenAPI[10] in Verbindung mit SwaggerUI[11] entschieden. Damit ist ohne das Wissen der genauen technischen Umsetzung eine unabhängige Entwicklung der Services gewährleistet. Jeder Microservice besitzt seine eigene API, die seine angebotenen Dienste darstellt. Mit der passenden Uniform Resource Identifier (URI) lassen sich die verschiedenen Endpunkte abfragen. Um den Prototypen möglichst schlank zu halten, werden an dieser Stelle keine Autorisierungsprüfungen durchgeführt. Im Produktivbetrieb sollten Endpunkte immer nur berechtigten Nutzern zur Verfügung stehen.

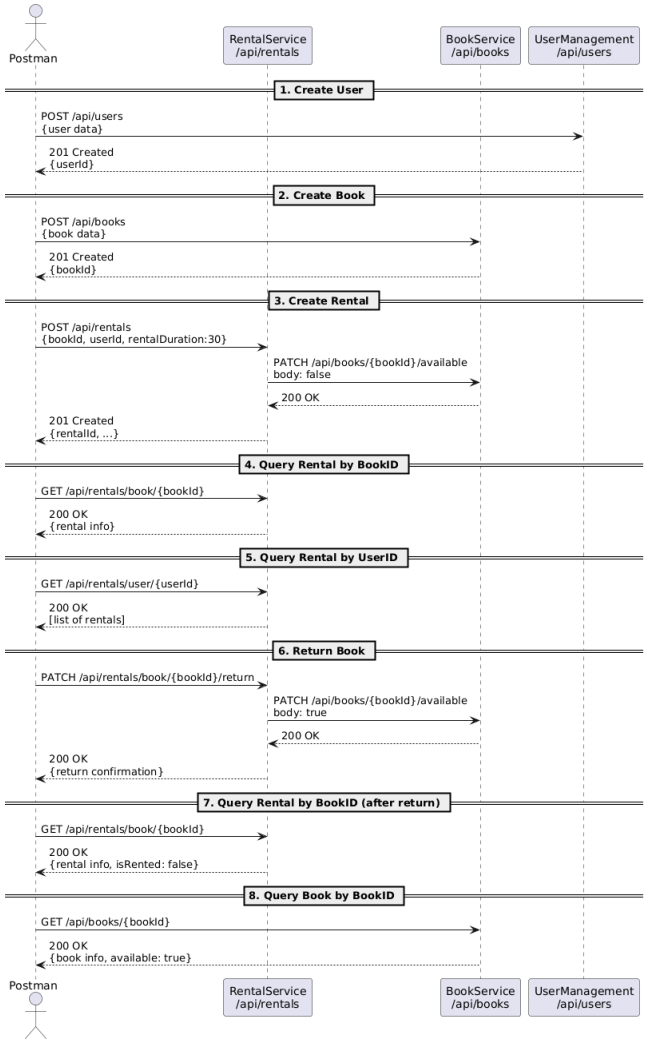

Abbildung 1. Kommunikation der Microservices

Die Abbildung (vgl. Abbildung 1 auf Seite 6) zeigt die Kommunikation zwischen dem Nutzer[12] und den einzelnen Services. Ab diesem Zeitpunkt sprechen wir weiterhin vom Nutzer, mit dem Wissen, dass Postman zur Erstellung der Anfragen und zum Empfangen von Antworten verwendet wird. Des Weiteren gehen wir davon aus, dass ausschließlich JSON-Nutzdaten zwischen Nutzer und Microservices versendet werden. Neben der Interaktion zwischen dem Nutzer und dem System findet zusätzlich eine interne Kommunikation zwischen zwei Microservices statt.

Create User – Vorab erstellt der Nutzer ein Nutzerkonto, in dem er seine Nutzerdaten mittels HTTP-Verb POST an den Endpunkt `/api/users` des UM sendet. Die Nutzerdaten für unser Beispiel umfassen den Namen, die E-Mail-Adresse und sein Passwort. Als Antwort erhält er eine eindeutige Identifikationsnummer. Mit dieser ID kann er oder ein anderer Microservice, seine Nutzerdaten mit dem HTTP-Verb GET und der URL `/api/users/userId` abfragen.

Create Book – Danach muss die Bücherei mit Büchern gefüllt werden. Durch mehrfaches Aufrufen des Endpunktes `/api/books` und dem HTTP-Verb POST lassen sich beliebig viele gleiche oder unterschiedliche Bücher erstellen. Erwartet werden Metadaten wie der Titel, der Autor und die eindeutige International Standard Book Number (ISBN). Jedem neu erzeugten Buch wird eine eindeutige Identifikationsnummer ausgestellt und als Antwort an den Nutzer gesendet. Neben den typischen, filterbaren Anfragen einzelner oder mehrerer Bücher bietet der BS die Möglichkeit, Metadaten existierender Bücher zu verändern. Zusätzlich ist die Verfügbarkeit eines Buches mit einer PATCH-Anfrage an `/api/books/bookId/available` steuerbar. Das available-Attribut einer Buch-Entität ist ein boolscher Wert, der auf *true* oder *false* gesetzt werden kann.

Create Rental – Nachdem die Datenbanken des UM und des BS mit Informationen gefüllt sind, kann der BRS sinnvoll angesprochen werden. Eine Leihe wird durch folgende Attribute definiert:

- *id* – Eine eindeutige ID für die jeweilige Leihe
- *bookId* – Repräsentiert das geliehene Buch
- *userId* – Repräsentiert den Nutzer, der das Buch ausgeliehen hat
- *rentalDate* – Das Datum, an dem das Buch ausgeliehen wurde
- *rentalDuration* – Die Dauer der Leihe in Tagen, hieraus ergibt sich, wann die Rückgabe fällig ist
- *isRented* – Der Status der Leihe

Die Informationen bookId, userId und rentalDuration müssen bei der Erstellung einer Leihe angegeben werden. Die restlichen Attribute werden bei der Erzeugung automatisch gesetzt. Der Endpunkt `/api/rentals` ist für eine POST-Anfrage zum Erstellen eines Datensatzes festgelegt und sendet bei erfolgreicher Erstellung eine Antwort mit dem gesamten Objekt. Während des Erstellens einer neuen Leihe kümmert sich der BRS um das Setzen des available-Attributes der Buch-Entität. Dabei wird der Endpunkt `/api/books/bookId/available` mit einer PATCH-Anfrage auf den Wert *false* gesetzt.

Query Rental by BookID – Der BRS bietet den Endpunkt `/api/rentals/book/bookId`, mit diesem lässt sich eine Leihe passend zur Identifikationsnummer des Buchs abfragen. Wenn das Buch in der Datenbank gelistet ist, wird eine 200-HTTP-Antwort mit dem vollständigen Rental-Objekt gesendet. Ist das abgefragte Buch nicht vorhanden, folgt eine 404-HTTP-Antwort mit einer Fehlermeldung. Das HTTP-Statuscode-Management wurde für den Prototyp absolut minimalistisch implementiert und soll hier nicht näher erläutert werden.

Query Rental by UserID – Der Nutzer hat die Möglichkeit, den BRS nach all seinen Leihen zu fragen. Hierfür dient die `/api/rentals/user/userId` API, die auf Anfrage dem Nutzer eine Liste seiner Leihen liefert.

Return Book Die Rückgabe ausgeliehener Bücher muss für den Nutzer möglich sein, daher liefert der BRS einen Endpunkt, der dies umsetzt. Analog zum Erstellungsprozess wird hier ebenfalls der BS benachrichtigt, um den available-Status des Buchs auf *true* zu setzen. Der Nutzer erhält eine Nachricht, dass die Rückgabe erfolgt ist.

Die letzten beiden Abfragen, siebtens und achtens, dienen der Überprüfung, ob sich die Datensätze der Leihe und des

10. Spezifikation zur Beschreibung von REST-ful APIs https://www.openapis.org/

11. Grafische Darstellung von OpenAPI-konformen Dokumenten https://swagger.io/tools/swagger-ui/

12. Hier wird der Nutzer bzw.die Interaktion eines Nutzers mit einer grafischen Oberfläche, durch die Software Postman simuliert https://www.postman.com/

Buchs entsprechend der Erwartung geändert haben. Die erfolgreiche Änderung wird hier vorausgesetzt und daher nicht näher erläutert. Befinden sich alle Microservices in einem gesunden Zustand, d.h.sie sind aktiv und laufen ohne Abstürze, ist mit diesem Szenario der Ablauf eines funktionstüchtigen MVP skizziert.

4.2.4. Koordination durch Docker-Compose. Bieten alle drei Microservices den gleichen Port z.B.8080 an, der neben dem Port 80 für HTTP-Anwendungen der Standard-Port ist, wird es bei der Ausführung auf dem Hostrechner zu Problemen kommen. Eine Anwendung auf einem Zielsystem ist durch die Internet Protokoll (IP)-Adresse und den Port erreichbar. Eine Firewall kann die Anfragen auf bestimmte Ports filtern und somit die Kommunikationswege festlegen. Die Thematik Firewall wird für unseren Prototypen außer Acht gelassen, da die Services nur lokal ausgeführt werden.

Teilen sich nun mehrere Anwendungen den gleichen Port und laufen auf dem gleichen System, sind sie für das Betriebssystem nicht mehr adressierbar. Bevor dies geschieht, wird die Java Virtual Machine (JVM) bei der Anfrage nach dem Port 8080 vom Betriebssystem eine Fehlermeldung erhalten und ein Starten der Anwendung verhindern. Eine Möglichkeit, dieses Problem zu beheben, ist die Veränderung der Ports, auf denen die einzelnen Microservices lauschen. Versuchen wir, die Anwendung horizontal skalieren zu lassen, d.h.jeder Microservice wird beliebig oft gestartet, stoßen wir wieder auf das exakt gleiche Problem.

Hier hilft uns Docker mit dem Ansatz der Containervirtualisierung. Der Docker-Daemon dient als Router und kann mittels Network Address Translation (NAT) ein oder mehrere virtuelle Subnetze aufspannen. In diesen Subnetzen erhält jeder Container seine eigene IP-Adresse. In jedem Container wird genau eine Instanz eines Microservices betrieben, die durch Port-Mapping von außen erreichbar ist.

Docker-Compose[13] hilft uns, die Vernetzung der einzelnen Container und ihrer zugehörigen Datenbanken in einer Yet Another Markup Language (YAML)[14]-Datei zu definieren. Mit einem Befehl lassen sich so Verbände von Docker-Containern gleichzeitig hochfahren. Für unsere Anwendung werden die drei Microservices BS, BRS und das UM zusammen mit ihren zugehörigen MariaDB-Datenbanken in einer zentralen Docker-Compose-Datei definiert. Es werden drei Bridge-Netzwerke erstellt, die für jeden Microservice mit zugehöriger Datenbank ein eigenes Subnetz bilden. Damit über den Hostrechner auf die Microservices zugegriffen werden kann, werden ihre Ports – 8080 – an die Hostrechner-Ports in aufsteigender Nummerierung d.h.8080, 8081, 8082 abgebildet. Mit dieser Herangehensweise kann auf die Datenbanken von außerhalb des Subnetzes *nicht* zugegriffen werden. Zusätzlich erfolgt das Abbilden auf die IP-Adresse 127.0.0.1 um zu verhindern, dass standardmäßig die 0.0.0.0 gewählt wird. Wird diese Wahl nicht getroffen, werden beim Abbilden alle Netzwerkkarten des Host-Rechners angesprochen, was zu einer möglichen Freigabe in das Netzwerk des Hostrechners führt. Mit diesen zwei sehr einfach umsetzbaren Konfigurationen, wird unser System und unsere Datenbanken besser vor Angriffen von außen geschützt.

Jede Datenbank wird mit einem Volume verbunden, das hilft die Daten des Container zu persistieren. Der Speicherort der Datenbank innerhalb des Containers, wird auf einen Speicherort auf dem Hostrechner abgebildet. Ohne diese Konfiguration, entstehen bei jedem Absturz oder Neustart der Datenbankcontainer, irreparable Datenverluste. Für das Abbilden muss nicht zwangsläufig ein Pfad auf dem Hostrechner gewählt werden. Das Einhängen eines Network Attached Storage (NAS) der sich im gleichen Netzwerk wie der Hostrechner befindet, ist ebenfalls möglich. Startet ein Datenbankcontainer neu, greift er auf die Daten des Hostrechners zu.

Es werden noch weitere Definitionen in der YAML-Datei benötigt, wie z.B.die Deklaration des Bauprozesses mit den passenden Dockerfiles, oder das Setzen von Umgebungsvariablen zum dynamischen Starten von Anwendungscontainern. Weitere Entscheidungen für das Setzen bestimmter Konfigurationen werden im Abschnitt 5 genauer erläutert.

13. Docker-Compose zum gleichzeitigen Start mehrerer Container https://docs.docker.com/compose/

14. YAML ist eine plattformunabhängig Sprache zur Datenserialisierung https://yaml.org/

4.3. Datenmodell und Persistenz

Das Datenmodell des Prototyps (vgl. Abbildung 2 auf Seite 7) ist zweckmäßig reduziert und bildet eine einfache Bibliotheksanwendung ab. Entitäten des UM sind unabhängig von anderen Datensätzen und benötigen keine zusätzlichen Fremdschlüssel. Gleiches gilt für die Book-Entitäten des BS. Ein Buch verwaltet seine Verfügbarkeit über das Attribut *available* selbst. Dadurch entsteht eine Entkopplung der Microservices BS und BRS, da ein Buch keine Informationen über Ausleihen oder ausleihende Nutzer besitzen muss. Die Rental-Entitäten werden im BRS verwaltet, besitzen spezifische Attribute zur Beschreibung des Leihprozesses und sind über die Fremdschlüssel *bookId* und *userId* mit den Tabellen des BS bzw. UM verknüpft.

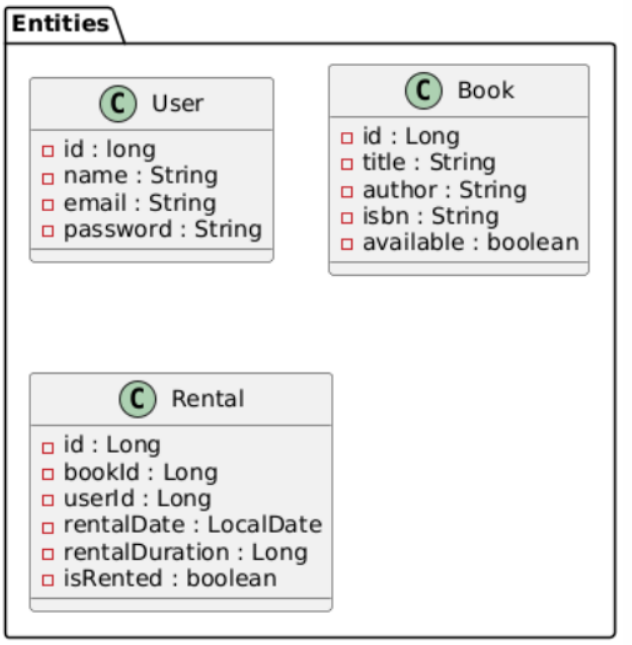

Abbildung 2. Die Entitäten der drei Microservices

Die von den Microservices verarbeiteten Daten müssen dauerhaft persistiert werden. Um diese Daten, die vom Nutzer in die Microservices gelangen, speichern zu können, braucht es verschiedene Serialisierungs- und Deserialisierungs-Schnittstellen. Für den hier beschriebenen Anwendungsfall werden Daten im JSON-Format mittels HTTP-Protokoll an die Microservices gesendet. Die Microservices kommunizieren untereinander ebenfalls auf diese Weise. Die Microservices werden in der Programmiersprache Java geschrieben, die in ihrer Standardbibliothek keine native JSON-Verarbeitung vorsieht. Somit ist das Implementieren einer eigenen Übersetzungslogik von JSON in Java-Entitäten erforderlich.

Spring-Boot nimmt uns durch das Einbinden des Spring-Boot-Starter-Web Moduls, diese Arbeit ab. Bei der Kompi-

lierung unserer Anwendung mit Maven[15], wird die Jackson-Bibliothek im Klassenpfad eingebunden [60]. Die Jackson-Bibliothek übernimmt die Aufgabe, JSON-Objekte, die als assoziative Arrays mit Key-Value-Paaren angesehen werden können, in Java lesbare Objekte zu übersetzen [45]. Die Geschäftslogik der Anwendung arbeitet ab diesem Zeitpunkt mit Java-Objekten und muss die Übersetzung nicht übernehmen.

Die Objekte werden zu diesem Zeitpunkt im Random-Access Memory (RAM) des Prozesses gehalten. Das Halten von Daten im RAM ist insofern problematisch, dass bei einem Anwendungsabsturz alle Daten verloren gehen. Um das zu verhindern, braucht es eine dauerhafte Speicherung auf einer Festplatte. Die Persistenz von Dateien ohne die Nutzung eines Database Management System (DBMS) birgt die Gefahr fehlerhafter Datensätze aufgrund des fehlenden Transaktionsmanagements. Ein Transaktionsmanagement regelt den Zugriff auf Daten, insbesondere wenn beispielsweise mehrere Prozesse gleichzeitig schreibend auf eine Ressource zugreifen möchten. MariaDB regelt für uns die Speicherung und den Zugriff auf Daten. Bei MariaDB handelt es sich um eine relationale Datenbank, sie legt also ihre Daten tabellarisch an. Zur Erstellung und Abfrage von Datensätzen nutzen relationale Datenbanken Structured Query Language (SQL).

Die Java-Standardbibliothek bietet hierfür keine native Übersetzung von Java-Code in SQL-Befehle. Wir verwenden für unseren Prototyp das ORM in Verbindung mit dem JPA. Hierbei werden Java-Objekte, auch Entitäten genannt, durch das ORM in SQL-taugliche Tabellen transformiert. Die JPA definiert hierfür einheitliche Zugriffsmethoden für Java-Anwendungen. Die technische Umsetzung von ORM und JPA übernimmt die Hibernate-Java-Bibliothek. Hibernate und die JPA-Schnittstelle werden durch das Modul Spring-Boot-Starter-Data-JPA in der Maven-Konfigurationsdatei dem Klassenpfad zur Verfügung gestellt[39]. Die Java-Anwendungen benötigen abschließend noch eine Möglichkeit zur Kommunikation mit der Datenbank. Zwar arbeitet MariaDB mit SQL, jedoch ist die konkrete Umsetzung der Abfragesprache für jede Datenbank spezifisch, sodass ein Vermittler erforderlich ist. Hierfür verwenden wir das MariaDB-Java-Client-Modul, das als Schnittstelle zur Datenbank dient. Es übersetzt die von Hibernate generierten SQL-Befehle in die spezifische Datenbanksyntax, den sogenannten Dialekt. Auch der Verbindungsaufbau und das Senden der Befehle werden mithilfe dieses Moduls realisiert. Die Integration der einzelnen Module und die Implementierung in unsere Anwendung mittels Spring-Boot wird in Abschnitt 5 erklärt.

5. Implementierung

In diesem Kapitel wird auf die Implementierung der Microservices mittels Spring Boot, die Anbindung an die Datenbank, die Umsetzung von ORM und JPA, den Bau der Docker-Images sowie die Realisierung einer gemeinsamen Infrastruktur mithilfe von Docker Compose eingegangen. Ziel ist es, die technische Umsetzung so weit zu erläutern, dass sowohl die Vorteile als auch die Nachteile für die anschließende Bewertung in Abschnitt 6 ersichtlich werden.

5.1. Aufbau unserer Spring Boot Microservices

Die Abbildung 3 unten zeigt, wie verschiedene Komponenten innerhalb der Microservices zusammenspielen. Die Abbildung ist in zwei Teilbereiche unterteilt, die miteinander verbunden sind. Der erste Teilbereich bildet den Startprozess von Spring Boot ab, dessen automatisierter Ablauf beim Anwendungsstart die Module Spring-Boot-Starter-Parent und Spring-Boot-Starter-Web verwendet [61]. Der zweite Teilbereich veranschaulicht die von uns implementierten Komponenten und zeigt deren logische Nutzungsreihenfolge.

15. Maven als Werkzeug für den Bau von Projekten mit unterschiedlichen Abhängigkeiten https://maven.apache.org/

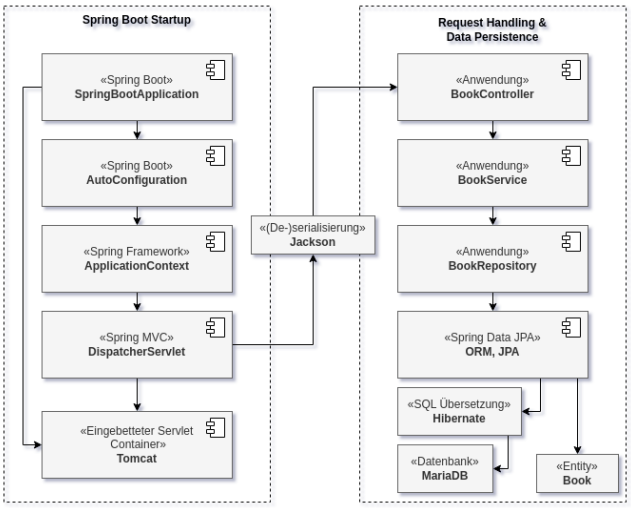

Abbildung 3. Grobgranulare Struktur der Microservices

5.2. Der Spring Boot Startprozess

Der Start einer Spring-Boot-Anwendung beginnt mit der Klasse, die mit `@SpringBootApplication` annotiert ist. Diese Annotation fasst drei Kernkomponenten zusammen [41].

- `@Configuration` markiert die Klasse als Konfigurationsquelle für Spring [61].
- `@EnableAutoConfiguration` aktiviert die automatische Konfiguration und erkennt automatisch benötigte Spring-Beans und Frameworks [2].
- `@ComponentScan` startet den Komponentenscan und registriert alle im Projekt vorhandenen Spring Beans [55].

Nach dem Start initialisiert Spring den `ApplicationContext`. Dieser verwaltet alle Beans und ihre Abhängigkeiten [57]. Das zugrundeliegende Prinzip ist Inversion of Control (IOC)[54]. Dabei übernimmt das Framework die Kontrolle über die Instanziierung und Verwaltung der als Bean markierten Objekte. Die Dependency-Injection sorgt dafür, dass Abhängigkeiten automatisch in die jeweiligen Klassen injiziert werden [52].

Im nächsten Schritt startet der eingebettete `Servlet-Container`, im Standardfall ist dies Tomcat [46]. Tomcat stellt das `DispatcherServlet` bereit. Dieses zentrale Servlet verarbeitet alle eingehenden HTTP-Anfragen. Das `DispatcherServlet` delegiert die Anfragen an passende Controller, deren Zuordnung anhand von URL-Abbildungen erfolgt. In Abbildung 3 werden Anfragen schließlich an den `BookController` weitergeleitet. Spring-Boot automatisiert in den Standardeinstellungen den Prozess von der initialen Konfiguration und dem Start der Anwendung bis zur Bereitstellung des `DispatcherServlet` und der Übergabe der Anfragen an die Controller-Schicht.

5.3. Von der Anfrage zur Datenpersistierung

Für die folgenden Unterkapitel dient der BS als Beispiel. Die Implementierung in den anderen beiden Microservices ist analog erfolgt.

5.3.1. Klassenaufteilen und Aufgabentrennung. Der BS folgt einer klaren Aufgabenteilung und wendet somit die Empfehlung der Spring-Dokumentation an [42]. Er gliedert den Code in die folgenden Schichten mit definierten Verantwortlichkeiten.

- Präsentationsschicht – Verarbeitet HTTP-Anfragen und liefert Antworten

- Geschäftslogik – Steuert die fachliche Verarbeitung der Daten
- Datenzugriff – Verwaltet das Lesen und Schreiben von Daten in der Datenbank

Der `BookController` empfängt HTTP-Anfragen und repräsentiert die Präsentationsschicht. Spring-Boot leitet die Anfragen über das `DispatcherServlet` an den Controller weiter. JSON-Daten werden dabei durch das Jackson-Modul automatisch in Java-Objekte deserialisiert. Der Controller reicht die Anfragen an die Service-Schicht weiter. Die Klasse `BookService` kapselt die Geschäftslogik. Sie steuert, wie die Daten verarbeitet und in der Datenbank gespeichert werden. Für den Datenzugriff verwendet der Service das `BookRepository`. Dieses Interface erweitert JpaRepository und stellt Methoden zum Zugriff auf die Datenbank bereit. Die Implementierung dieser Schnittstelle wird von Spring Data JPA und Hibernate zur Laufzeit automatisch erzeugt. Die Entität Book definiert die Datenstruktur. Sie ist mit der Annotation `@Entity` versehen und beschreibt die Abbildung der Felder auf die Datenbanktabelle.

Alle Klassen sind als Spring-Beans registriert[37]. Dies geschieht über die Annotationen @RestController, @Service und @Repository. Spring verwaltet die Instanzen und injiziert sie bei Bedarf automatisch in abhängige Klassen[52]. Für diese automatische Injektion wird die Annotation `@Autowired` verwendet. Sie greift bei der Erzeugung neuer Instanzen, bei denen Attribute Referenzen auf Objekte anderer Klassen sind, auf verfügbare Beans im Application-Context-Container zu [51]. Dadurch entsteht eine lose Kopplung zwischen den Schichten, bei der ein Austausch der referenzierten Objekte jederzeit möglich ist.

Der Vorgang wird anhand von diesem Beispiel exemplarisch erklärt. Eine HTTP-POST-Anfrage an `/api/books` enthält die Daten für ein neues Buch im JSON-Format. Das `DispatcherServlet` empfängt die Anfrage und übergibt sie an den `BookController`. Dort wird die Methode `addBook(Book book)` aufgerufen. `Jackson` deserialisiert die Anfrage und liefert ein Book-Objekt. Der Controller ruft daraufhin die Methode `addBook(Book book)` im `BookService` auf. Die Service-Schicht prüft die Daten und übergibt das Objekt an die Methode `save(Book book)` des `BookRepository`. Hibernate generiert den entsprechenden SQL-Befehl und speichert das neue Buch in der Datenbank.

Es wird gezeigt, dass diese Struktur die Präsentationsschicht, die Geschäftslogik und den Datenzugriff des BS klar trennt.

5.3.2. Das Objekt-Relationale-Modell. Die Microservices nutzten ORM, um Java-Objekte mit relationalen Datenbanktabellen zu verknüpfen [53]. Dieses Abbilden erleichtert die Entwicklung, da der Zugriff auf die Datenbank über Java-Klassen erfolgt. Spring-Boot stellt in Kombination mit dem Modul Spring-Data-JPA die notwendige Infrastruktur bereit [39]. Spring-Data-JPA abstrahiert die JPA-Spezifikation und integriert sie in das Spring-Framework. Hibernate übernimmt das eigentliche Mapping zwischen Java-Objekten und relationalen Tabellen sowie die Generierung und Ausführung der SQL-Befehle [12].

Entitäten werden durch mit `@Entity` annotierte Klassen definiert[49]. In der Klasse `Book` kennzeichnet die Annotation `@Entity` die Klasse als persistente Entität. Mit Annotation `@Table` wird der Name der zugehörigen Datenbanktabelle angegeben. Jedes Feld, das in der Datenbank gespeichert wird, ist mit der Annotation `@Column` versehen. So wird beispielsweise das Attribut `title` als Spalte in der Tabelle `books` abgebildet. Der Primärschlüssel wird mit `@Id` markiert und kann mit `@GeneratedValue` automatisch generiert werden. Hibernate verwaltet für uns den Lebenszyklus der Entitäten. Es speichert neue Objekte, lädt bestehende Objekte aus der Datenbank und synchronisiert Änderungen. Wir arbeiten dabei ausschließlich mit den Java-Objekten, wodurch das explizite Schreiben von SQL-Befehlen nicht erforderlich ist. Das ORM führt zu einer klaren Trennung zwischen Anwendungslogik und Datenzugriff. Es erhöht die Wartbarkeit und vereinfacht die Entwicklung von Microservices mit Datenbankanbindung.

5.3.3. Verwendung der JPA-Schnittstelle. Die Microservices nutzten das Modul Spring-Data-JPA, um den Zugriff auf die Datenbank zu vereinfachen[39]. Dieses Modul abstrahiert die JPA und integriert sie nahtlos in das Spring-Framework[56]. Der Datenzugriff erfolgt über Repository-Schnittstellen, die von JpaRepository erben[58]. Wir haben dafür eine eigene Schnittstelle mit dem Namen BookRepository erstellt und von JpaRepository geerbt. Diese Repository-Schnittstellen werden mit der Annotation @Repository als Spring-Komponenten gekennzeichnet und zur Laufzeit automatisch als Bean bereitgestellt [37].

Durch die Vererbung von JpaRepository stehen grundlegende CRUD-Operationen ohne weitere Implementierung zur Verfügung [19]. Zusätzliche Abfragen können mit der Annotation @Query als eigene JPQL[16]-Abfragen definiert werden [49]. Spring Data JPA übergibt die Abfragen und Entitäten an Hibernate. Hibernate bildet dabei die JPA-Implementierung und übernimmt das Abbilden der Java-Objekte auf die relationalen Tabellen sowie die Ausführung der generierten SQL-Befehle. Wir arbeiten ausschließlich mit Java-Objekten, während Hibernate die Datenbankinteraktionen verwaltet. Dieses Konzept reduziert den Implementierungsaufwand und ermöglicht einen klar strukturierten, einfach lesbaren Code.

5.3.4. Anbindung an MariaDB. Die Spring-Boot-Anwendung bindet eine MariaDB-Datenbank als persistente Datenspeicherschicht an. Dafür nutzt sie das Mariadb-Java-Client-Modul als JDBC-Treiber[1]. Dieser Treiber stellt die Verbindung zwischen der Spring-Boot-Anwendung und der MariaDB-Datenbank her und kann als Abhängigkeit in der Maven-Konfigurationsdatei eingebunden werden[38]. Die Verbindungskonfiguration erfolgt in der Datei `application.properties` [43]. Dort wird die Datenbank-URL unter `spring.datasource.url` angegeben. Benutzername und Passwort werden unter `spring.datasource.username` und `spring.datasource.password` konfiguriert. Zusätzlich legt die Eigenschaft `spring.jpa.hibernate.ddl-auto` fest, wie Hibernate das Datenbankschema verwaltet. Darüber hinaus definiert die Eigenschaft `spring.jpa.properties.hibernate.dialect` den für Hibernate zu verwendenden SQL-Dialekt [59]. Die Eigenschaft `spring.jpa.properties.hibernate.format_sql` steuert die Ausgabe der von Hibernate erzeugten SQL-Anweisungen im Log. Mit dem Wert `true` erfolgt diese Ausgabe formatiert und besser lesbar, was die Analyse und Fehlersuche während der Entwicklung unterstützt. Wir haben uns für den Ansatz entschieden, die Konfigurationswerte als Umgebungsvariablen zu setzen [47]. Spring Boot unterstützt das automatische Überschreiben der Konfigurationswerte durch entsprechende Umgebungsvariablen beim Start der Anwendung.

Für eine funktionierende Anbindung an eine Datenbank sind also drei Komponenten erforderlich.

- der JDBC-Treiber, hier das Mariadb-Java-Client-Modul
- die korrekte Konfiguration der Verbindungsparameter
- und ein erreichbarer MariaDB-Datenbankdienst

5.4. Containerisierung mit Docker

Die Ausführung der Microservices findet in Docker-Containern statt. Die Containervirtualisierung erlaubt es mit Docker-Images beliebig viele Container, auch Instanzen genannt, des jeweiligen Microservices zu starten. Sie gibt uns zusätzlich die Möglichkeit, einzelnen Container zu kapseln, um unbefugten

16. Java Persistence Query Language (JPQL) https://docs.oracle.com/html/E13946_04/ejb3_langref.html#ejb3_langref_stmnttypes

Zugriff zu verhindern. Damit die Container mit ihren Anwendungen reibungslos laufen, braucht es eine exakte Definition des Image-Bauvorgangs. Lassen sich die Container aus Images einwandfrei starten, braucht es einen Prozess zum gemeinsamen Hochfahren aller Container. Dies ist notwendig, da die Microservices voneinander und von Datenbanken abhängig sind. Die Struktur der Dockerfiles wird wie zuvor am Beispiel des BS demonstriert und ist analog zu den Dockerfiles der anderen Microservices. Die gesamte Anwendung besitzt im Wurzelverzeichnis eine gemeinsame Docker-Compose-Datei. Sie startet alle Microservices und ihre Datenbanken mit dem Docker-Compose-Run-Befehl.

5.4.1. Die Dockerfiles. Die Abbildung 4 zeigt exemplarisch die Image-Bauanleitung des BS. Als Basis-Image dient das `eclipse-temurin:17-jre-alpine`. Dieses Image enthält die notwendige Java-Runtime für die Ausführung von Java-Anwendungen. Es basiert auf Alpine-Linux[17] einer sehr schlanken Linux-Distribution. Ziel ist es, ein möglichst kleines Basis-Image zu verwenden, um die Container nicht unnötig aufzublähen und den Bauprozess zu beschleunigen. Im Container wird das Arbeitsverzeichnis auf `/opt/app` gesetzt. Anschließend wird das erstellte Java Archive (JAR)-Artefakt `bookService-v1.0.0.jar` in das Verzeichnis kopiert. Über `EXPOSE 8080` wird der vom Microservice genutzte Port deklariert. Der Container startet die Anwendung schließlich mit dem `ENTRYPOINT`-Befehl, indem die JAR-Datei per `java -jar` ausgeführt wird.

```
# Start with a base image containing Java runtime
FROM eclipse-temurin:17-jre-alpine

# Set the working directory inside the container
WORKDIR /opt/app

# Copy the .jar file into the container
COPY target/bookService-v1.0.0.jar /opt/app

# Expose the port that your application listens on
EXPOSE 8080

# Run the .jar file
ENTRYPOINT ["java", "-jar", "/opt/app/bookService-v1.0.0.jar"]
```

Abbildung 4. Bauanleitung für ein Image des BS

Das JAR-Artefakt entsteht im Bauprozess mithilfe des Spring-Boot Maven-Plugins. Dieses Plugin übernimmt das Packen der Anwendung und aller Abhängigkeiten in ein ausführbares JAR. Der Befehl `mvn clean package` erzeugt dabei im Target-Verzeichnis das Artefakt `bookService-v1.0.0.jar`[44]. Dieses Artefakt bildet die Grundlage für die Containerisierung und kann ohne zusätzliche Konfiguration im Container ausgeführt werden.

Die Anwendung im Container ist beim Start und zur Laufzeit über Umgebungsvariablen konfigurierbar. Spring-Boot unterstützt die automatische Zuordnung von Umgebungsvariablen zu Konfigurationseigenschaften. Dadurch können beispielsweise Datenbankverbindungen ohne Anpassung des Container-Images angepasst werden. Das Ziel ist das Erreichen einer möglichst hohe Flexibilität, ohne Änderungen an der Anwendung selbst durchzuführen. Jede Änderung der Anwendung erzwingt die Wiederholung des gesamten Bauprozesses. Ein weiterer Vorteil ist der dynamische Wechsel zwischen der Test- und der Produktionsumgebung.

5.4.2. Die Docker-Compose-File. Für die koordinierte Ausführung der Microservices BS, BRS, UM und ihrer Datenbanken wird Docker-Compose eingesetzt. Dabei handelt es sich um ein Werkzeug zur Definition und Verwaltung mehrerer Container-basierter Dienste. Die Konfiguration erfolgt in einer `docker-compose.yml`-Datei.

```
bookservicemariadb:
  image: mariadb
  container_name: bookservice-mariadb
  environment:
    MARIADB_ROOT_PASSWORD: root_password
    MARIADB_DATABASE: book_service_db
    MARIADB_USER: book_service_user
    MARIADB_PASSWORD: book_service_super_secret
  volumes:
    - ./book_Service/docker_data/mariadb:/var/lib/mysql
  networks:
    - bookservice-network
  restart: unless-stopped

▷ Run Service
bookservice:
  build:
    context: ./book_Service
    dockerfile: Dockerfile
  image: bookservice:latest
  container_name: bookservicecontainer
  ports:
    - "127.0.0.1:8080:8080"
  environment:
    SPRING_DATASOURCE_URL: jdbc:mariadb://bookservicemariadb:3306/book_service_db
    SPRING_DATASOURCE_USERNAME: book_service_user
    SPRING_DATASOURCE_PASSWORD: book_service_super_secret
    SPRING_JPA_HIBERNATE_DDL_AUTO: update
  depends_on:
    - bookservicemariadb
  networks:
    - bookservice-network
  restart: unless-stopped

networks:
  bookrentalservice-network:
    driver: bridge
  bookservice-network:
    driver: bridge
```

Abbildung 5. Docker-Compose-Datei des Prototypen

In der YAML-Datei (vgl. Abbildung 5 auf Seite 10) wird zunächst für jeden Microservice und seine Datenbank ein separates Bridge-Netzwerk erstellt. Diese Netzwerke kapseln die beteiligten Container und ermöglichen ihnen die Kommunikation untereinander, ohne Zugriffsmöglichkeit von außen. Exemplarisch wird dies anhand des BS mit seiner MariaDB-Datenbank beschrieben.

Der erste Dienst stellt eine MariaDB-Datenbank bereit. Das verwendete Image basiert auf der offiziellen MariaDB-Distribution. Wichtige Parameter wie der Name der Datenbank, der Benutzername und das Passwort werden über Umgebungsvariablen gesetzt. Zusätzlich wird ein lokales Volume eingebunden, um die Persistenz der Datenbankdaten über Container-Neustarts hinweg sicherzustellen.

Der zweite Dienst bildet den eigentlichen BS ab. Das zugehörige Container-Image wird mittels Dockerfile aus dem BS-Ordner erzeugt. Der Container veröffentlicht den Port 8080 lokal, sodass der Service über die Adresse 127.0.0.1:8080 erreichbar ist. Auch hier erfolgt die Konfiguration der Datenbankverbindung dynamisch über Umgebungsvariablen. Die Variable *SPRING-DATASOURCE-URL* legt die Verbindungs-URL zur MariaDB-Datenbank fest. Sie enthält den Hostnamen des Datenbankcontainers, den Port sowie den Namen der Zieldatenbank. Über *SPRING-DATASOURCE-USERNAME* und *SPRING-DATASOURCE-PASSWORD* werden der Benutzername und das zugehörige Passwort für die Authentifizierung mit der Datenbank spezifiziert. Zusätzlich steuert *SPRING-JPA-HIBERNATE-DDL-AUTO* das Verhalten von Hibernate beim Datenbankschema-Management. Der gesetzte Wert *update* bewirkt, dass Hibernate das bestehende Schema beim Start der Anwendung automatisch an die aktuellen Entitäten anpasst. Alle Variablen werden beim Start des Containers von Docker Compose in den Umgebungsraum des Prozesses übergeben und von Spring-Boot automatisch in die jeweilige Anwendungskonfiguration übernommen. Durch die depends-on-Direktive wird sichergestellt, dass der Datenbankcontainer vor dem Start des Microservices gestartet wird.

Beide Container kommunizieren ausschließlich über das definierte interne Netzwerk `bookservice-network`. Docker-

17. Das Basis-Image der Distribution hat eine Größe von lediglich 5MB https://hub.docker.com/_/alpine

Compose erleichtert somit das Testen und den Betrieb der Microservices im lokalen Umfeld und schafft eine konsistente Umgebung für alle beteiligten Komponenten.

6. Bewertung der eingesetzten Technologien

In diesem Kapitel beschreiben wir unsere gesammelten Erkenntnisse, die durch die notwendige Recherche und die Erstellung der Microservices entstanden sind. Dabei werden die Erkenntnisse der jeweiligen Technologien in Vorteile und Herausforderungen eingeordnet.

6.1. Microservices mit Spring Boot

6.1.1. Positive Aspekte. Wenn wir über positive Eigenschaften von Spring-Boot sprechen, muss zuerst die enorm hohe Entwicklungsgeschwindigkeit genannt werden. Einen funktionierenden Microservice in ungefähr 25 bis 30 Stunden – Recherche inbegriffen – zu entwickeln, ist für eine komplexe Programmiersprache wie Java keine Selbstverständlichkeit. Hierfür sind gerade für den Start die offiziellen Spring-Boot-Tutorials hervorzuheben, die Entwicklern Schritt für Schritt die wichtigsten Komponenten erklären. Mit dem Spring-Initializr lassen sich innerhalb weniger Klicks vollständige Maven-Projekte erzeugen[62].Mithilfe von IOC und Dependency Injection wird die Verantwortung für die Verwaltung von Objekten an Spring-Boot übergeben, was zu einem geringeren Planungs- und Implementierungsaufwand führt. Das ORM mittels JPA und Hibernate vereinfacht die Datenhaltung enorm, da das Schreiben von SQL-Anweisungen weitestgehend entfällt. Die explizite Deklaration durch JPQL gibt Entwicklern dennoch die nötige Flexibilität, um spezifische Zugriffe auf die Datenbank zu definieren. Der in Spring-Boot eingebettete Servlet-Container mit seinem Tomcat-Server nimmt für den Einstieg die Komplexität einer Server-Konfiguration. Es müssen lediglich Endpunkte beschrieben werden, während sich Spring-Boot um Anfragen und Antworten kümmert. Die zentrale Konfiguration in der `application.properties` ist einfach umzusetzen und aufgrund der Vermeidung von XML gut lesbar. Es entfallen hierdurch Konfigurationen im Quellcode, was eine klare Trennung der Aufgaben fördert und den Quellcode nicht unnötig belastet.

Wird das Maven-Plugin in der Maven-Konfigurationsdatei deklariert, lassen sich mit Maven alle direkten Abhängigkeiten und transitiven Abhängigkeiten einfach über das Einfügen von Spring-Starter-Modulen einbinden. Das Maven-Plugin kümmert sich beim Bau der Anwendung um das Einfügen der Bibliotheken in den Klassenpfad. Zusätzlich wird der kompilierte Quellcode als ausführbare JAR zur Verfügung gestellt. Die Tatsache dass diese JARs alle Abhängigkeiten in einer Datei vereinen und somit unabhängig sind, ist wichtig für die Funktionsweise von Microservices und die Anwendung von DevOps-Ansätzen. Außerdem wird durch die ZIP-Komprimierung Speicherplatz gespart.

Ein wichtiger Pluspunkt ist der Open-Source-Status des Spring Projekts, welches durch eine große Anzahl an Entwicklern – ungefähr 1.200 – weiterentwickelt wird [50]. Eine strukturierte Sicherheitspolitik[35], die auf Meldungen der Community aufbaut und regelmäßige Sicherheitsupdates gegen Common-Vulnerabilities-and-Exposures(CVE)[36], sorgen für resiliente Anwendungen, die in der Zukunft bestehen können. Open-Source-Projekte leben von einer aktiven Community, die sich einbringt und bestehenden Code kontinuierlich weiterentwickelt. Spring-Boot besitzt eine umfangreiche Community, die entweder aktiv am öffentlichen Repository arbeiten oder sich gegenseitig durch frei nutzbare Community-Starters unterstützten [40]. Eine Umfrage von Jetbrains unterstreicht die breite Verwendung von Spring Boot [18].

6.1.2. Herausforderungen. Die größte Herausforderung hat sich bei dem Studium der Dokumentation ergeben. Spring-Boot ist ein sehr umfangreiches, über viele Jahre gewachsenes Framework. Entsprechend groß ist die Unterstützung für eine Vielzahl von Java-Versionen, was sich im Umfang der Dokumentation widerspiegelt. Aufgrund der großen Anzahl bestehender Legacy-Projekte, richten sich viele Konzepte nach Strukturen dieser Zeit. Es ist somit oft nicht ersichtlich, welche beschriebenen Ansätze veraltet und durch neuere ersetzt wurden. Betreibern von Legacy-Projekten bleiben neue Spring-Boot-Features oft verwehrt [48], da diese für aktuellere Java-Versionen konzipiert werden [50]. Dies ist zwar die eigentlich logisch positive Konsequenz des technischen Fortschritts, sorgt aber für eine Inkonsistenz der Quellcodes von Legacy-Betreibern.

Ein weiterer negativer Punkt betrifft die Startzeit von Spring-Boot-Anwendungen. Aufgrund der Vielzahl von Modulen, die geladen werden und den automatischen Prozessen, die bei Anwendungsstart stattfinden, braucht eine vergleichsweise kompakte Anwendung wie der BS bis zu 10 Sekunden bis zur Betriebsfähigkeit. Die ist für Containertechnologien im Cloud-Umfeld, in der Container regelmäßig gestartet und gestoppt werden, eine beachtliche Zeit.

Ein Aspekt der negativ auffällt, ist die Image-Größe nach der Erstellung durch die Dockerfile. Eine Alpine-Basis-Image hat ungefähr einen Speicherbedarf von 5 Megabyte. Der BS nimmt als ausführbare JAR-Datei mit den notwendigsten Abhängigkeiten etwas über 60 Megabyte ein. Das Image des BS hat eine finale Größe von 238 Megabyte, wodurch in etwa 175 Megabyte auf Layer fallen, die mit der Java-Runtime-Environment zusammenhängen. Zum Vergleich, ein in der Programmiersprache Go entwickelter Microservice ähnlichen Umfangs, hat nach dem Bau eine Größe von 9 Megabyte. Dies zeigt ganz offensichtlich, das Java Anwendungen generell einen beachtlichen Overhead erzeugen.

6.2. ORM mit Hibernate

6.2.1. Positive Aspekte. Ein wesentlicher Vorteil von Hibernate ist die Reduktion von Boilerplate-Code: Entwickler können sich auf die Geschäftslogik konzentrieren, während das Framework die persistenzbezogenen Aufgaben übernimmt. Zudem verbessert Hibernate durch Caching und Verbindungs-Pooling die Laufzeiteffizienz und Skalierbarkeit von Anwendungen. Die hohe Flexibilität bei der Modellierung von komplexen Datenbeziehungen macht Hibernate besonders geeignet für größere Unternehmensanwendungen mit komplexem Datenmodell.[11], [17], [64]

6.2.2. Herausforderungen. Die Verwendung von Hibernate bringt selbstverständlich auch einige Herausforderungen mit sich. Eines der größten Probleme besteht in der *Verdeckung der zugrunde liegenden Datenbankoperationen*: Trotz erfolgreicher Abstraktion vom SQL-Zugriff führt dies zu einer eingeschränkten Kontrolle über die tatsächlich generierten SQL-Befehle. In seltenen Fällen, besonders bei komplexen Joins oder großen Datenmengen, kann dies zu *Performanceproblemen* führen, wenn etwa nicht optimierte Abfragen entstehen oder unerwartetes Lazy Loading auftritt.[9]

Ein weiteres Risiko besteht im sogenannten *N+1-Select-Problem*. Dies kann auftreten, wenn beim Zugriff auf assoziierte Objekte wiederholt einzelne Datenbankabfragen ausgelöst werden, anstatt die Daten in einer einzigen, effizienten Abfrage zu laden. Obwohl Hibernate Mechanismen wie `fetch joins` oder `batch fetching` zur Verfügung stellt, erfordert deren korrekte Anwendung ein gutes Verständnis der internen Abläufe.[9]

Zudem kann die anfängliche *R*DBMSKomplexität der Konfiguration, etwa beim Mapping komplizierter Objektstrukturen oder der Integration mit existierenden Datenbankschemas zu höherem Aufwand führen. Auch die Lernkurve ist nicht zu unterschätzen: Entwickler müssen sich intensiv mit dem Lebenszyklus von Entitäten, Transaktionsgrenzen, Fetch-Strategien und

Caching-Verhalten vertraut machen, um die Möglichkeiten von Hibernate gut zu nutzen.[9]

6.3. Datenpersistenz mit MariaDB

Zu den Vorteilen von MariaDB zählt ihre weitgehende Kompatibilität mit MySQL, sowohl auf syntaktischer Ebene (SQL-Dialekt) als auch im Hinblick auf die Nutzung bestehender Treiber, wie etwa dem verbreiteten JDBC-Treiber. Dadurch ist es möglich bestehende Anwendungen mit minimalem Aufwand von MySQL auf MariaDB umzustellen. Dies ist insbesondere für Projekte mit langer Wartungshistorie oder Abhängigkeiten zu MySQL-basierter Software relevant.[20]

Ein weiterer technischer Pluspunkt liegt in der Unterstützung erweiterter Storage-Engines, wie zum Beispiel Aria oder MyRocks. Diese ermöglichen spezifische Optimierungen wodurch MariaDB an unterschiedliche Anwendungsanforderungen angepasst werden kann. Zudem hat sich um die Datenbank eine aktive und hilfbereite Community gebildet, welche auch MariaDB durch ihr Können und ihre Leidenschaft vorantreiben. Das steht im Gegensatz zu MySQL, welches nach der Übernahme durch Oracle einer proprietären Steuerung unterliegt.[31]

Trotz der angestrebten MySQL-Kompatibilität zeigen sich ab MYSQL Version 8 zunehmende Unterschiede[20], [31]. Neue Funktionen oder Syntaxerweiterungen aus MySQL werden in MariaDB gar nicht oder in anderer Form übernommen[31].

Ein weiterer Nachteil ist die begrenzte Unterstützung für moderne Datenformate, insbesondere JSON[66]. Während MySQL native JSON-Funktionen bietet, ist die Implementierung in MariaDB noch nicht soweit vorangeschritten, was die Arbeit mit semi-strukturierten Daten erschwert und bei bestimmten Anwendungen (z.B. REST-Backends) zu Einschränkungen führen kann[66].

Schließlich sind Hochverfügbarkeitslösungen in MariaDB komplexer zu implementieren als bei konkurrierenden Datenbanksystemen. Zwar existieren Mechanismen wie Galera Cluster, jedoch ist deren Einrichtung und Betrieb technisch anspruchsvoll und erfordert detailliertes Fachwissen, gerade im Hinblick auf Netzwerk- und Replikationsmanagement.[31]

7. Fazit

Im Rahmen dieser Arbeit wurde die Entwicklung eines Prototyps auf Basis einer Microservice-Architektur im Anwendungskontext einer Bibliotheksverwaltung realisiert und analysiert. Dabei sind drei weit verbreitete Technologien zum Einsatz gekommen. Spring Boot, Hibernate und Docker Compose sind sowohl bei der Erstellung eigenständiger Dienste als auch deren Verwaltung und Ausführung im Container-Umfeld im Einsatz. Ziel der Arbeit ist es, die theoretischen Konzepte moderner verteilter Systeme mit praktischer Anwendungsentwicklung zu verknüpfen und daraus Erkenntnisse über die technische Umsetzbarkeit, Stärken und Herausforderungen des Microservice-Ansatzes zu gewinnen.

Die Umsetzung des Prototypen mit getrennten Services für Bücher-, Nutzer- und Ausleihverwaltung hat die Vorteile der Microservice-Architektur deutlich aufgezeigt: Durch die klare Trennung der Verantwortlichkeiten können die einzelnen Microservices unabhängig voneinander entwickelt, getestet und verteilt werden. Die Implementierung der Anwendung mit Spring Boot erwies sich als zügig, da das Framework mittels automatischer Konfiguration, vorgefertigter Komponenten und integrierten Servern den Entwicklungsaufwand erheblich reduziert. Hibernate ermöglicht eine effiziente objekt-relationale Abbildung und vereinfacht den Datenbankzugriff. Die Containerisierung mit Docker Compose erleichtert schließlich das Starten der Dienste und ermöglicht eine reproduzierbare Testumgebung, die sowohl lokal als auch in verteilten Szenarien einsetzbar ist.

Trotz dieser positiven Aspekte wurden auch Herausforderungen identifiziert. Die gestiegene Systemkomplexität durch verteilte Komponenten, insbesondere hinsichtlich Konfiguration, Kommunikation und Datenkonsistenz, macht deutlich, dass Microservices nicht per se eine einfache Lösung darstellen. Zudem erfordert der produktive Einsatz ein tiefgreifendes Verständnis für Themen wie Sicherheit, Resilienz und das Logging auf Systemebene, die für den vorgestellten Prototypen noch nicht von Relevanz sind. Des Weiteren kann je nach Anwendungsfall ein tiefes Verständnis des Spring Frameworks erforderlich sein, um spezielle, anwendungsspezifische Konfigurationen vorzunehmen. Hierbei wird der anfangs angenehme Einstieg, durch eine hohe Komplexität des Spring-Frameworks überschattet, die viele Stunden Einarbeitung benötigt. Wenn Speicherplatz und Rechenressourcen ein ausschlaggebendes Argument sind, sollte sich aufgrund des ziemlich hohen Grundverbrauchs von Java-Anwendungen, nach schlankeren Lösungen umgesehen werden.

Für die Umsetzung von JPA und ORM mittels Hibernate können nur bedingt schlussfolgernde Aussagen getroffen werden. Hierfür müssen weitreichende Performance- und Integrationstests durchgeführt werden, die den Rahmen dieses Projektes sprengen. Bei den getätigten Aussagen beziehen wir uns überwiegend auf der bereits verfügbaren Lektüre, oder gemachte Erfahrungen. Die Anwendung von Hibernate ist durch die Integration im Starter-JPA-Modul von Spring-Boot sehr simpel und für den Einstieg in ein Microservice-basierendes Java-Projekt gut geeignet. Hierbei ergaben sich keine größeren Komplikationen oder Hindernisse. Wird das JPA-Interface gemäß der Spezifikation richtig umgesetzt, können auch Hibernate Alternativen eingebunden und genutzt werden.

MariaDB ist zum einen durch seinen Open-Source-Status für freie Projekte, wie z.B.Forschungsarbeiten gut geeignet und zum anderen ist die Verfügbarkeit von stets aktuellen Images im Docker-Hub ein großer Pluspunkt im Microservices-Umfeld. Auch die Möglichkeit, Datensätze von MySQL in MariaDB zu migrieren, vorausgesetzt es handelt sich um eine gültige Version, ist für Unternehmen mit bestehender MySQL-Datenbank, sicherlich von Bedeutung. Auf die Performance wird sich in dieser Arbeit ebenfalls auf bereits bestehende Lektüre bezogen, da eine sinnvolle Analyse mit diesem Projekt nicht zielführend ist.

Insgesamt kann die Arbeit zeigen, dass die Microservice-Architektur mit Spring Boot und Docker Compose eine mögliche Grundlage für modulare, wartbare und skalierbare Anwendungen bietet. Für Entwickler und Softwareunternehmen, die sich auf Java-Anwendungen spezialisiert haben oder bereits Java-Anwendungen betreiben und warten, kann Spring Boot eine Möglichkeit sein ihre alten monolithischen Architekturen aufzugeben, um sich moderneren Ansätzen zu widmen. Entwickler die sich noch nicht auf eine Programmiersprache festgelegt haben, ist eine schlankere Alternative wie z.B.GO zu empfehlen. Hierbei sollte immer der Technologietrend und der gegenwärtige Arbeitsmarkt beobachtet werden. Die Nutzung einer anderen Open-Source-Datenbank wie z.B.PostgreSQL ist ohne größere Probleme möglich. Hierfür muss lediglich der PostgreSQL-Treiber in der Maven-Konfigurationsdatei eingebunden werden und in den Konfigurationsdateien für Spring-Boot und Docker-Compose die entsprechenden Argumente ausgetauscht werden. Eine Änderung am Quellcode ist nicht erforderlich. Zu beachten gilt, dass bestehende MariaDB-Datensätze nicht zu PostgreSQL kompatibel sind.

Die Ziele der Arbeit wurden erreicht: Es wurde ein funktionierender Prototyp erstellt, die eingesetzten Technologien wurden auf ihre Eignung hin untersucht, und es konnten relevante Erkenntnisse für die Bewertung von Architekturentscheidungen in verteilten Softwaresystemen gewonnen werden. Für weiterführende Arbeiten bieten sich eine vertiefte Auseinandersetzung mit Orchestrierungslösungen wie Kubernetes, eine systematische Performanceanalyse unter Last sowie die Integration von Monitoring- und Sicherheitsmechanismen an.

Abkürzungen

IP	Internet Protokoll
BS	Book-Service
UM	Usermanagement
API	Application Programming Interfaces
BRS	Book-Rental-Service
JAR	Java Archive
JPA	Java Persistence API
JVM	Java Virtual Machine
IOC	Inversion of Control
MVP	Minimum Viable Product
NAT	Network Address Translation
NAS	Network Attached Storage
ORM	Object Relational Mapping
RAM	Random-Access Memory
SQL	Structured Query Language
URI	Uniform Resource Identifier
DBMS	Database Management System
JSON	JavaScript Object Notation
JPQL	Java Persistence Query Language
HTTP	Hypertext Transfer Protocol
ISBN	International Standard Book Number
OIDC	OpenID Connect
REST	Representational State Transfer
YAML	Yet Another Markup Language
OAuth	Open Authorization

Literatur

[1] „About MariaDB Connector/J“, MariaDB KnowledgeBase. (), Adresse: https://mariadb.com/kb/en/about-mariadb-connector-j/ (besucht am 01. 06. 2025) (siehe S. 9).

[2] „Auto-configuration :: Spring Boot“. (), Adresse: https://docs.spring.io/spring-boot/reference/using/auto-configuration.html (besucht am 01. 06. 2025) (siehe S. 8).

[3] R. Bitz, „Framework für einen zusammengesetzten Datenspeicher“, Magisterarb., Institut für Parallele und Verteilte Systeme, 2018. Adresse: https://web.archive.org/web/20200213124913id_/https://elib.uni-stuttgart.de/bitstream/11682/10554/1/composite-data-storecomposite-data-store_RB.pdf (siehe S. 2).

[4] D. Cassisi, „Entwicklung robuster Microservice-Architekturen mit Spring Boot und Spring Cloud“, Hochschule Furtwangen, Furtwangen im Schwarzwald, Deutschland, Seminararbeit, 2025, Abgerufen am 02. Juni 2025. Adresse: file:///C:/Users/Marc%20Ziegler/Downloads/KMT_Seminararbeit_Cassisi.pdf (siehe S. 1, 2).

[5] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser und P. Flora, „Detecting performance anti-patterns for applications developed using object-relational mapping“, in *Proceedings of the 36th International Conference on Software Engineering*, Ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, S. 1001–1012. DOI: 10.1145/2568225.2568259. Adresse: https://doi.org/10.1145/2568225.2568259 (siehe S. 3).

[6] T.-H. P. Chen et al., „An empirical study on the practice of maintaining object-relational mapping code in Java systems“, Mai 2016, S. 165–176. DOI: 10.1145/2901739.2901758 (siehe S. 3).

[7] I. Corporation, *Was sind Microservices und Microservices-Architektur?*, Abgerufen am 02. Juni 2025, 2025. Adresse: https://www.intel.de/content/www/de/de/cloud-computing/microservices.html (besucht am 02. 06. 2025) (siehe S. 2).

[8] DataStax, *SQL im Vergleich zu NoSQL: Vorteile und Nachteile*, Updated: May 29, 2025; Abgerufen am 2. Juni 2025, 2025. Adresse: https://www.datastax.com/de/blog/sql-vs-nosql-pros-cons (besucht am 02. 06. 2025) (siehe S. 3, 4).

[9] M. GmbH. „Was ist ORM? Vorteile, Nachteile und beliebte Frameworks erklärt“. (2025), Adresse: https://www.mindtwo.de/lexicon/orm-object-relational-mapping (besucht am 01. 06. 2025) (siehe S. 3, 11, 12).

[10] M. L. Hesse, *Containerisierung einer Webanwendung, automatische Integration und Bereitstellung ihrer Softwareänderungen*, Abgerufen am 2. Juni 2025, 2022. Adresse: https://monami.hs-mittweida.de/frontdoor/deliver/index/docId/14547/file/BA_52260_Manuel_Lucas-Hesse_geschwaerzt.pdf (siehe S. 2).

[11] Hibernate. „Hibernate ORM“. (), Adresse: https://hibernate.org/orm/ (besucht am 01. 06. 2025) (siehe S. 3, 11).

[12] „Hibernate-ORM- java“, Hibernate. (), Adresse: https://hibernate.org/orm/ (besucht am 01. 06. 2025) (siehe S. 9).

[13] IBM. „Was ist Java Spring Boot?“ (), Adresse: https://www.ibm.com/de-de/topics/java-spring-boot (besucht am 01. 06. 2025) (siehe S. 2).

[14] D. Inc., *Docker Compose — Docker Docs*, Abgerufen am 2. Juni 2025, 2025. Adresse: https://docs.docker.com/compose/ (siehe S. 2).

[15] D. Inc., *What is a Container? — Docker*, Abgerufen am 2. Juni 2025, 2025. Adresse: https://www.docker.com/resources/what-container/ (siehe S. 2).

[16] InterSystems, *Relationale vs. nicht-relationale Datenbank: Hauptunterschiede für modernes Datenmanagement*, Abgerufen am 2. Juni 2025, 2025. Adresse: https://www.intersystems.com/de/resources/relationale-vs-nicht-relationale-datenbank-hauptunterschiede-fur-modernes-datenmanagement/ (siehe S. 3, 4).

[17] „Java Hibernate: Effektives ORM-Framework für effizientes Datenbankmanagement“. (), Adresse: https://remotescout24.com/de/blog/1180-java-hibernate-orm-framework (besucht am 01. 06. 2025) (siehe S. 3, 11).

[18] „Java programming - the state of developer ecosystem in 2023 infographic“, JetBrains: Developer Tools for Professionals and Teams. (), Adresse: https://www.jetbrains.com/lp/devecosystem-2023 (besucht am 26. 05. 2025) (siehe S. 11).

[19] „JpaRepository (spring data JPA parent 3.5.0 API)“. (), Adresse: https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html (besucht am 01. 06. 2025) (siehe S. 9).

[20] H. G. Limited, *Was ist MariaDB und wie unterscheidet es sich von MySQL?*, Abgerufen am 02. Juni 2025, März 2025. Adresse: https://www.hostragons.com/de/blog/was-ist-mariadb-und-was-sind-die-unterschiede-zu-mysql/#MariaDBnin_Avantajlari_ve_Dezavantajlari (besucht am 02. 06. 2025) (siehe S. 4, 12).

[21] X. Luan, „IMPLEMENTATION AND ANALYSIS OF SOFTWARE DEVELOPMENT IN SPRING BOOT“, Magisterarb., Faculty of California State Polytechnic University, Pomona, 2021. Adresse: https://scholarworks.calstate.edu/downloads/zg64ts132 (siehe S. 2, 4).

[22] K. K. M, *A Review of Spring Boot Framework Based on REST API* (siehe S. 2).

[23] M.Mythily, A. A. Raj und I. T. Joseph, „An Analysis of the Significance of Spring Boot in The Market“, in *2022 International Conference on Inventive Computation Technologies (ICICT)*, 2022. Adresse: https://www.researchgate.net/publication/362747012_An_Analysis_of_the_Significance_of_Spring_Boot_in_The_Market (siehe S. 2, 3).

[24] Microsoft. „what-is-java-spring-boot“. () (siehe S. 2, 3).
[25] F. Müller-Hofmann, M. Hiller und G. Wanner, „Die Java Persistence API“, in *Programmierung von verteilten Systemen und Webanwendungen mit Java EE*, Springer Fachmedien Wiesbaden, 2015, Kap. 9, S. 301–353. DOI: 10.1007/978-3-658-10512-9 (siehe S. 4).
[26] F. Müller-Hofmann, M. Hiller und G. Wanner, „Die Java Persistence API“, in *Programmierung von verteilten Systemen und Webanwendungen mit Java EE*, Springer Fachmedien Wiesbaden, 2015, Kap. 7, S. 301–353. DOI: 10.1007/978-3-658-10512-9 (siehe S. 4).
[27] S. Newman, *Microservices: Konzeption und design*. MITP-Verlags GmbH & Co. KG, 2015 (siehe S. 1, 2).
[28] E. E. Ogheneovo, P. O. Asagba und N. O. Ogini, „An Object Relational Mapping Technique for Java Framework“, *International Journal of Engineering Science Invention*, Jg. 2, Nr. 6, S. 01–09, 2013 (siehe S. 3).
[29] M. Raemy, *RESTful API mit Java Spring*, 2020. Adresse: https://www.unifr.ch/inf/softeng/en/assets/public/files/research/students_projects/bachelor/Bachelor_Raemy_Marc.pdf (siehe S. 1, 2).
[30] G. Rauch. „Was ist Spring Boot?“ (2021), Adresse: https://www.datacenter-insider.de/was-ist-spring-boot-a-142bdeaf5195490e0939d4136b24a692/ (besucht am 01.06.2025) (siehe S. 2).
[31] I. Redaktion, *MariaDB vs. MySQL: Ein Vergleich*, Abgerufen am 02. Juni 2025, Okt. 2022. Adresse: https://www.ionos.de/digitalguide/hosting/hosting-technik/mariadb-vs-mysql/ (besucht am 02.06.2025) (siehe S. 4, 12).
[32] E. Schicker, *Datenbanken und SQL: Eine praxisorientierte Einführung mit Anwendungen in Oracle, SQL Server und MySQL*, 5., aktualisierte und erweiterte Auflage. Wiesbaden, Deutschland: Springer Vieweg, 2017. DOI: 10.1007/978-3-658-16129-3 (siehe S. 3, 4).
[33] A. W. Services, *Relationale vs. nicht-relationale Datenbanken – Unterschied zwischen Datenbanktypen*, Abgerufen am 2. Juni 2025, 2025. Adresse: https://aws.amazon.com/de/compare/the-difference-between-relational-and-non-relational-databases/ (siehe S. 3, 4).
[34] B. für Sicherheit in der Informationstechnik (BSI), *SYS.1.6 Containerisierung*, Abgerufen am 2. Juni 2025, Feb. 2022. Adresse: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/IT-GS-Kompendium_Einzel_PDFs_2022/07_SYS_IT_Systeme/SYS_1_6_Containerisierung_Edition_2022.pdf?__blob=publicationFile&v=3 (siehe S. 2).
[35] „Spring - regelmäßige security updates CVE“, Security Advisories. (), Adresse: https://spring.io/security (besucht am 28.05.2025) (siehe S. 11).
[36] „Spring - security policy“, Security Policy. (), Adresse: https://spring.io/security-policy (besucht am 28.05.2025) (siehe S. 11).
[37] „Spring Boot - Beans und Dependency Injection“. (), Adresse: https://docs.spring.io/spring-boot/reference/using/spring-beans-and-dependency-injection.html (besucht am 28.05.2025) (siehe S. 9).
[38] „Spring Boot - Benutzung Maven Plugin“. (), Adresse: https://docs.spring.io/spring-boot/maven-plugin/using.html (besucht am 01.06.2025) (siehe S. 9).
[39] „Spring Boot - Benutzung von JDBC, JPA“. (), Adresse: https://docs.spring.io/spring-boot/reference/data/sql.html (besucht am 28.05.2025) (siehe S. 8, 9).
[40] „Spring boot - community starters“, GitHub. (), Adresse: https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-starters/README.adoc (besucht am 28.05.2025) (siehe S. 11).
[41] „Spring Boot - Einführung erste Anwendung“. (), Adresse: https://docs.spring.io/spring-boot/tutorial/first-application/index.html (besucht am 28.05.2025) (siehe S. 8).
[42] „Spring Boot - Empfohlene Code Struktur“. (), Adresse: https://docs.spring.io/spring-boot/reference/using/structuring-your-code.html (besucht am 28.05.2025) (siehe S. 8).
[43] „Spring Boot - Konfigurationsliste für Application.properties“. (), Adresse: https://docs.spring.io/spring-boot/appendix/application-properties/index.html (besucht am 28.05.2025) (siehe S. 9).
[44] „Spring Boot - Nested JARs“. (), Adresse: https://docs.spring.io/spring-boot/specification/executable-jar/nested-jars.html (besucht am 31.05.2025) (siehe S. 10).
[45] „Spring Boot - Serialisieren und Deserilisieren mit JSON“. (), Adresse: https://docs.spring.io/spring-boot/reference/features/json.html (besucht am 28.05.2025) (siehe S. 8).
[46] „Spring Boot - Servlet Web Application“. (), Adresse: https://docs.spring.io/spring-boot/reference/web/servlet.html (besucht am 01.06.2025) (siehe S. 8).
[47] „Spring boot - tips: Configuration“, Spring Tips: Configuration. (), Adresse: https://spring.io/blog/2020/04/23/spring-tips-configuration (besucht am 01.06.2025) (siehe S. 9).
[48] „Spring Boot - Voraussetzungen“. (), Adresse: https://docs.spring.io/spring-boot/system-requirements.html (besucht am 28.05.2025) (siehe S. 11).
[49] „Spring Data JPA - JPA Query Methods“. (), Adresse: https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html (besucht am 01.06.2025) (siehe S. 9).
[50] „Spring framework - (gibthub) versions“, GitHub. (), Adresse: https://github.com/spring-projects/spring-framework/wiki/Spring-Framework-Versions (besucht am 28.05.2025) (siehe S. 11).
[51] „Spring Framework - Autowiring Richtig nutzen“. (), Adresse: https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-autowire.html (besucht am 26.05.2025) (siehe S. 9).
[52] „Spring Framework - Dependency Injection“. (), Adresse: https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html (besucht am 26.05.2025) (siehe S. 8, 9).
[53] „Spring Framework - Einleitung in ORM mit Spring“. (), Adresse: https://docs.spring.io/spring-framework/reference/data-access/orm/introduction.html (besucht am 27.05.2025) (siehe S. 9).
[54] „Spring Framework - IoC Inversion of Control“. (), Adresse: https://docs.spring.io/spring-framework/reference/core/beans/introduction.html (besucht am 26.05.2025) (siehe S. 8).
[55] „Spring Framework - Klassenpfad Scan und Verwaltung von Komponenten“. (), Adresse: https://docs.spring.io/spring-framework/reference/core/beans/classpath-scanning.html (besucht am 26.05.2025) (siehe S. 8).
[56] „Spring Framework - Overview“. (), Adresse: https://docs.spring.io/spring-framework/reference/overview.html (besucht am 26.05.2025) (siehe S. 9).
[57] „Spring Framework - Spring Container als Grundlage“. (), Adresse: https://docs.spring.io/spring-framework/reference/core/beans/basics.html (besucht am 26.05.2025) (siehe S. 8).
[58] „Spring guide - accessing data with JPA“, Getting Started — Accessing Data with JPA. (2025), Adresse: https://spring.io/guides/gs/accessing-data-jpa (besucht am 28.05.2025) (siehe S. 9).
[59] „Spring guide - accessing data with MySQL“, Getting Started — Accessing data with MySQL. (), Adresse: https://spring.io/guides/gs/accessing-data-mysql (besucht am 01.06.2025) (siehe S. 9).
[60] „Spring guide - erstellen eines RestControllers mit HTTP verben“, Getting Started — Building a RESTful Web

Service. (), Adresse: https://spring.io/guides/gs/rest-service (besucht am 28. 05. 2025) (siehe S. 8).
[61] „Spring guide - spring boot anwendung bauen“, Getting Started — Building an Application with Spring Boot. (2025), Adresse: https://spring.io/guides/gs/spring-boot (besucht am 28. 05. 2025) (siehe S. 8).
[62] „Spring initializr“, Spring Initializr. (), Adresse: https://start.spring.io (besucht am 26. 05. 2025) (siehe S. 11).
[63] C. Staff, „Software erklärt: Was ist Containerisierung?“, Feb. 2025, Abgerufen am 2. Juni 2025. Adresse: https://www.coursera.org/de-DE/articles/containerization (siehe S. 1, 2).
[64] D. Storm, „Analyse und Einsatzmöglichkeiten des ORM-Frameworks Hibernate im DELECO R-Umfeld und prototypische Umsetzung eines Anwendungsszenarios“, Bachelorarbeit, Magisterarb., Hochschule Mittweida (FH), Fakultät Mathematik/Naturwissenschaften/Informatik, 2010, S. 51. Adresse: https://monami.hs-mittweida.de/frontdoor/deliver/index/docId/1175/file/Bachelorarbeit_Hibernate.pdf (siehe S. 3, 11).
[65] TecnoDigital, *Relationale Datenbanken: Eine Einführung*, Letzte Aktualisierung: 25. Juni 2023; Abgerufen am 2. Juni 2025, 2023. Adresse: https://informatecdigital.com/de/Relationale-Datenbanken-%E2%80%93-eine-Einf%C3%BChrung/ (siehe S. 3).
[66] Tianzhou, *MySQL vs. MariaDB: a Complete Comparison in 2025*, 9 min read; Updated on Mar 01, 2025; Abgerufen am 02. Juni 2025, März 2025. Adresse: https://www.bytebase.com/blog/mysql-vs-mariadb/ (besucht am 02. 06. 2025) (siehe S. 4, 12).
[67] VMSoftwarehouse. „Spring Framework vs. Spring Boot – Vor- und Nachteile“. (2023), Adresse: https://vmsoftwarehouse.de/spring-framework-vs-spring-boot-vor-und-nachteile#:~:text= (besucht am 01. 06. 2025) (siehe S. 2, 3).
[68] „Web API design best practices - azure architecture center — microsoft learn“, Web API Design Best Practices - Azure Architecture Center — Microsoft Learn. (), Adresse: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design (besucht am 30. 05. 2025) (siehe S. 5).
[69] A. A. Zubaydi, *Datenbanktests mit H2-Datenbank*, Abgerufen am 02. Juni 2025, 2024. Adresse: https://conciso.de/datenbanktests-mit-h2-datenbank/ (besucht am 02. 06. 2025) (siehe S. 4).