Migration einer Standalone Spring-Anwendung nach Architektur von Microservices über Kubernetes

Fakultät für Mathematik und Informatik

Bachelorarbeit

vorgelegt von

Fabian Halbig

(Matrikelnummer: 39832818)

am 20.01.2022

Prüfer: Prof. Dr. Thomas Zimmer

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Inhaltsverzeichnis

Abkürzungsverzeichnis

Admin Administrator

AKID Atomarität, Konsistenz, Isolation und Dauerhaftigkeit

API Application Programming Interface

AWS Amazon Web Services

BC Bounded Context

CD Continous-Delivery

CI Continous-Integration

CNCF Cloud Native Computing Foundation

DDD Domain Driven Design

DevOps Development und Operations

ER-Modell Entity-Relationship-Modell

GCP Google Cloud Plattform

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IaC Infrastructure as Code

IP Internet Protocol

JSON JavaScript Object Notation

JWT JSON Web Token

PaaS Platform as a Service

PVC Persistent Volume Claim

REST Representational State Transfer

SOA Service Oriented Architecture

SQL Structured Query Language

UML Unified Modeling Language

URL Uniform Resource Locator

YML Yet Another Markup Language

1. Einleitung

1.1. Motivation

In den letzten Jahren hat der Begriff Development und Operations (DevOps) immer mehr an Wichtigkeit gewonnen und die damit verbunden Themenbereiche, wie Kubernetes und GitOps, werden in der Zukunft weiter an Bedeutung gewinnen, was unter anderem von den Google-Search-Trends für die Suchanfragen in der Grafik 1.1 abzuleiten ist. DevOps umfasst viele verschiedene Bereiche des Softwareentwicklungszyklus, wobei ein wichtiger und interessanter Bereich die Kubernetes-Technologie in Verbindung mit Container-Anwendungen ist, da die Containerisierung zahlreiche Vorteile bietet, wie beispielsweise kontinuierliche Entwicklung, Integration und Deployment. Diese Vorteile werden langfristig zu einer Revolutionierung des Softwareentwicklungsprozesses führen, weshalb die Verwendung von Kubernetes in Produktionsumgebungen in der Vergangenheit stark zugenommen hat.

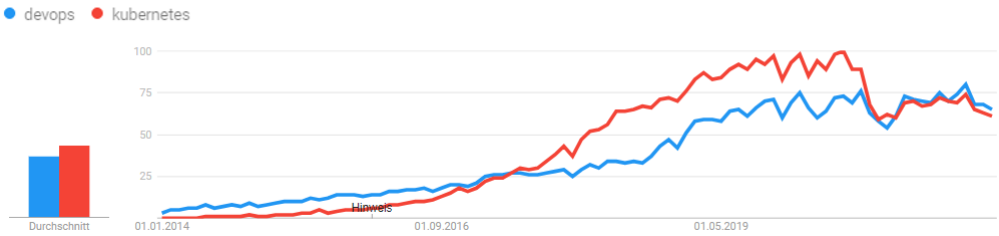

Abbildung 1.1: Google-Search-Trends für DevOps und Kubernetes [Google, 2022]

Allerdings unterscheidet sich die Architektur der bisherigen Anwendungen zu dem Architekturansatz von Microservices, der in Kubernetes verwendet wird, was eine Migration von einer traditionellen Anwendung zu Kubernetes nicht leicht realisierbar macht und oft zu komplexen Problemen während der Migration führt. Ein Problem, das häufig auftritt, ist es das Datenmodell aus der bestehenden Anwendung in Kubernetes durch eine verteilte Datenhaltung in kleineren Datenmodellen mit verschiedenen und unabhängigen Datenbanken umzusetzen, ohne die Eigenschaften Atomarität, Konsistenz, Isolation und Dauerhaftigkeit (AKID) in der Datenhaltung zu vernachlässigen. Aus diesem Grund beschäftigt sich diese Arbeit mit einer Beispielmigration einer Standalone Spring-Boot Anwendung hinzu einer Microservice-Architektur für Kubernetes, um die Möglichkeiten zur Anpassung von bereits bestehenden Datenmodellen bei einer Migration aufzuzeigen.

1.2. Zielsetzung

Das Ziel des ausgewählten Themas „Migration einer Standalone Spring-Anwendung nach Architektur von Microservices über Kubernetes“ ist es, die Möglichkeiten der Speicherung von persistierenden Daten, die dauerhaft und fortlaufend von einem System verwaltet werden, bei einer Migration von einer Standalone-Anwendung hinzu Kubernetes zu betrachten.

Dazu wird in dem qualitativen Forschungsprojekt durch eine Fallstudie beispielhaft eine Migration einer bereits bestehenden Spring-Boot Anwendung, die in dem Kapitel 3 genau analysiert und dargestellt ist, durchgeführt. In dieser Einzelfallstudie wird nach der Analyse der bestehenden Anwendung eine Microservice-Architektur mit verschiedenen Domänen, wie User-Management und Course-Task-Verwaltung, nach dem Domain Driven Design (DDD) von Evans, 2015 entworfen. Für die Aufteilung der Anwendungslogik und dem damit verbundenen Datenmodell werden die Modelle von Newman, 2019 angewendet, damit eine Speicherung der Daten nach dem AKID-Prinzip bei der verteilten Datenspeicherung des neuen Systems in verschiedenen Datenbankdiensten sichergestellt ist. Dieser Entwurf wird in der Implementierung durch Docker-Images umgesetzt, die in einer Docker Compose Datei mit Datenbankcontainern und Volumes für eine dauerhafte Datenspeicherung sorgen und anschließend mit einem Befehl als Docker-Container ausführbar sind. Danach wird ein Deployment der Images für die Applikationslogik der definierten Architektur in Kubernetes durchgeführt, um die Persistierung der Daten mit Postgres-Diensten, die durch Persistent Volume Claims (PVCs) eine dauerhafte Speicherung der Daten garantieren darzustellen. Somit zeigt diese Fallstudie den gesamten Softwareentwicklungszyklus einer Migration von einer monolithischen Anwendung mit persistierenden Daten hinzu einer Microservice-Architektur in Kubernetes.
Durch dieses Vorgehen wird die Verwendung von einer Microservice-Architektur für die Migration einer monolithischen Anwendung zu Kubernetes gezeigt und bietet die Möglichkeit, die Übertragung und Aufteilung eines bestehenden Datenmodells für die Speicherung von persistierenden Anwendungsdaten hin zu einer verteilten Datenhaltung in Kubernetes zu untersuchen.

1.3. Gliederung der Arbeit

Nachdem die Forschungsfrage und der Fall der Einzelfallstudie dargestellt ist, wird in diesem Abschnitt die Gliederung und das Vorgehensmodell aufgezeigt. In dieser Arbeit wird ein deduktives Vorgehen angewendet, sodass die bereits bestehende Theorien für die Migration zu Microservices geprüft werden [Saunders, 2007, S. 120].
Als Erstes werden in dem Kapitel 2 die benötigten Grundlagen zu den Verwendeten Systemen und Modellen ausgehend von der bestehenden Literatur dargelegt. Die Arbeit ist nach dem „action research“[Saunders, 2007, S. 140] Ansatz erarbeitet, weshalb die nächsten Kapitel nach dem Wasserfallmodell von Royce, 1987, S. 330 gegliedert sind, da eine Überschneidung der beiden Modelle in der Iteration von einzelnen Schritten diese Option bietet. Deshalb wird zunächst die zu migrierende Anwendung mit dem Namen „Course-Planner“ analysiert und die nicht-funktionalen sowie funktionalen Anforderung an das neue System mit der Bezeichnung „Scheduler-Services“ auf Basis der Analyse definiert. Anschließend wir die Microservice-Architektur in dem Kapitel 4 entworfen, sodass es möglich ist in dem folgenden Kapitel die Implementierung für die Ausführung in Docker sowie Kubernetes vorzunehmen.
In dem letzten Kapitel wird der durchgeführte Fall der Migration zusammengefasst und analysiert, um die Einbindung der verwendeten Modelle für eine verteilte Datenspeicherung zu bewerten und weitere Maßnahmen abzuleiten. Abschließend wird ein Ausblick gegeben, um darzustellen, in welchen Bereichen und Technologien weitere Untersuchungen relevant für diesen Themenbereich sind.

2. Grundlagen für eine Migration zu Kubernetes

In diesem Kapitel werden die benötigten Grundlageninformationen für die Migration einer Standalone Spring-Anwendung zu Kubernetes aufgeführt. Als Erstes wird ein Blick auf die benötigten Kenntnisse über die Funktionalitäten des Spring Frameworks für Microservices gegeben. Anschließend wird auf die Details von Microservices mit Docker und Kubernetes eingegangen sowie Modelle zu der Aufteilung von Datenmodellen aufgezeigt.

2.1. Spring Boot

Spring-Boot Anwendungen basieren auf dem Spring open-source Framework, das durch zahlreiche Funktionen und Integrationen die Entwicklung einer kompletten Webanwendung oder von einzelnen Microservices vereinfacht [Reddy, 2017]. Somit bietet Spring-Boot die Möglichkeit zu entscheiden, ob eine monolithische Anwendung, wie die Beispielanwendung in dem Kapitel 3, entwickelt werden soll oder Microservices zu implementiert sind, die über Application Programming Interfaces (APIS) miteinander kommunizieren [VMware, 2021d]. Dieses Framework bietet einige Vorteile, die bei der Entwicklung von Java basierten Anwendung eine Verbesserung sowie Förderung von Programmierpraktiken ermöglicht.
Ein Vorteil von Spring-Boot ist der Spring-Initializer (`https://start.spring.io/`), der durch die zu selektierenden Abhängigkeiten (engl. Dependencies) die Build-Konfiguration durch die Build-Management-Tools Maven oder Gradle vereinfacht [VMware, 2021c]. Weiter wird die Konfiguration der Anwendung durch Annotationen von hinzugefügten Abhängigkeiten vereinfacht, da hierdurch die Option besteht mit einem @-Zeichen Metadaten in den Quelltext des Java-Projektes einzubinden. [Oracle, 2014].
Nach dieser kurzen Einführung zu dem Spring-Boot Framework werden in den nächsten beiden Abschnitten zwei der wichtigsten Spring-Funktionalitäten für die Durchführung der Fallstudie erläutert.

2.1.1. Spring-Security

Die Dependency `spring-cloud-starter-security` bietet die Möglichkeit bei Spring-Anwendungen die Implementierung von anpassbarer Authentifikation und Zugriffskontrolle vereinfacht zu realisieren.
Spring-Security ist in allen Spring Anwendungen mit einer Java Version über 8 verwendbar. Durch die Einbindung der Abhängigkeit werden die Definition von zusätzlichen Konfigurationsdateien oder die Implementierung einer aufwendigen Java-Authentifikation vermieden [VMware, 2021c]. Allerdings beschränkt man sich bei der Verwendung von Spring-Security nicht auf ein Authentifikationssystems, sondern hat die Option zu entscheiden welche Art der Authentifikation bei der Entwicklung der Anwendung verwendet wird. So ist es beispielsweise möglich die Überprüfung einer Entität über einen Username und ein Password zu implementieren, wie es bei der bestehenden Anwendung verwendet ist oder auch über das OAuth2.0 Protokoll mit OpenID Connect, das sich speziell für eine API-Autorisierung eignet, was für die Implementierung der migrierten Anwendung

und deren Architektur in den folgenden Kapiteln genutzt wird [Alex et al., 2004, S. 404]. Die Authentifikationsmechanismen werden in Spring zunächst über die entsprechenden Abhängigkeiten in der Definitionsdatei für die Abhängigkeiten des Projektes hinzugefügt, die für einige Methoden zur Authentifikation durch zusätzliche Abhängigkeiten erweitert werden muss. Zum Beispiel wird für die Implementierung eines OAuth2.0 Protokolls zusätzlich die Dependency `spring-boot-starter-oauth2-client` benötigt. Falls Spring-Security ohne weitere Konfiguration zu einem Projekt hinzugefügt ist, wird bei Start der Applikation automatisch ein zufälliges Security-Passwort erzeugt. Diese Art der automatischen Konfiguration ist durch die Definition eines Users beziehungsweise eines Administrator (Admin) in den Anwendungseigenschaften (engl. application properties) vorzubeugen, da hiermit die Standardkonfiguration überschrieben wird [Baeldung, 2021a]. Um mehr Flexibilität bei der Konfiguration der Security-Eigenschaften der Anwendung zu haben, besteht die Möglichkeit eine Spring-Konfigurationsklasse hinzuzufügen, die sich für die verschiedenen Mechanismen unterscheidet, aber für eine einfache und einheitliche Definition der Konfiguration sorgt.
Beispielweise wird diese Konfiguration für eine einfache Authentifikation mit Usernamen und Passwort durch die Klasse `WebSecurityConfigurerAdapter` erweitert, damit es in den geerbten `configure-Methoden` unter anderem möglich ist, verschiedene Arten oder Gruppen von Usern zu definieren oder Pfade anzugeben auf die auch nicht eingeloggte User zugriff haben [VMware, 2021c]. Ein klassisches Anwendungsbeispiel hierfür ist, dass bei einer Anwendung Admins und normale User mit geringeren Rechten benötigt werden und alle User die Möglichkeit haben auf die Übersichtsseite beziehungsweise Landingpage einer Website zuzugreifen, ohne eine Authentifikation vorzunehmen.
Hiermit wurde in diesem Abschnitt gezeigt, wie Spring-Security zu einem Spring Projekt hinzuzufügen ist und dies für die Authentifikation von Nutzern genutzt wird. Diese Funktionalität ist wichtig für moderne Anwendung und wird auch dazu genutzt eine Authentifikation bei einer Representational State Transfer (REST) Schnittstelle zu konfigurieren, was essenziell für die Erstellung von sicheren Microservices ist, das in der Implementierung der Microservices in dem Kapitel 5 genau dargestellt wird.

2.1.2. Spring Cloud

Das Spring Framework folgt außerdem dem Trend von DevOps und Cloud Computing durch die eigene Projektsuit „Spring Cloud“[VMware, 2021b], die verschiedene Services für ein Deployment der Spring Projekte in der Cloud beinhaltet. Eine Beispielarchitektur für die Verwendung von Spring-Anwendungen in der Cloud ist in der Abbildung A.1 zu sehen, die im folgenden genauer beschrieben wird.
Ein wichtiger Bestandteil dieser Architektur ist das Spring-Cloud-Gateway, das über die Dependency `spring-cloud-starter-gateway` in ein Projekt eingebunden wird. Das Ziel von dieser Abhängigkeit ist es einen einfachen und effektiven Weg für das Routing von APIS zu ermöglichen und dabei die Option zu bieten komplexe Themen, wie beispielsweise Authentifikation, Monitoring oder das Verstecken von Diensten, zentral zu steuern. [Schimandle, 2021]
Die Funktionsweise von einem Spring-Cloud-Gateway ist, dass ein Client eine Anfrage (engl. Request) an die Applikation erstellt und anschließend das Gateway überprüft, ob

die übermittelte Route in dem Gateway-Mapping definiert ist. Insofern die Route in dem Mapping angegeben ist, wird der Request entsprechend durch den Gateway-Web-Handler weitergesendet [Schimandle, 2021]. Die Menge an Routen, die das Gateway-Mapping abdeckt, wird in der Datei für die Anwendungseigenschaften des Gateways definiert, was für alle eingebundenen Spring-Boot Microservices durchzuführen ist, damit die Spring-REST-Services über das Gateway erreichbar sind.
Da das Gateway für die Funktionsweise des API-Routings die verschiedenen Services kennen muss, wird in der Regel zusätzlich ein Spring-Eureka-Server verwendet. Dieser Server übernimmt hierbei die Service-Discovery. Damit sich die einzelnen Services in der Service-Discovery registrieren, muss jeder exakte Ort eines Services bekannt sein, sodass das Gateway die Anfragen an den richtigen Service weiterleitet. Ein Eureka-Server ist an der Annotation `@EnableEurekaServer` in der Hauptklasse des Spring Projektes zu erkennen und bietet durch die Registrierung der Services mit einem logischen Namen die Möglichkeit, dass „Services dynamisch auf unterschiedlichen Instanzen laufen, ohne dass der Nutzer die tatsächliche URL des Services wissen muss“[Janser, 2015]. Die Services, die in der Service-Registry des Eureka-Servers hinzuzufügen sind, müssen mit der Annotation `@EnableEurekaClient` implementiert werden und eine Referenz zu dem Eureka-Server in den Anwendungseigenschaften hinterlegen.
Das Spring-Cloud Architekturmodell aus der Springdokumentation in der Abbildung A.1 zeigt, wie mit dem Spring Cloud Gateway die Anfragen von Clients auf die entsprechenden Microservices verteilt werden. Dies wird durch die Service-Registry mit einem Eureka-Server realisiert und gegebenenfalls zusätzlich mit einem Zipkin-Service für ein verteiltes Tracing unterstützt [Schimandle, 2021]. Außerdem ist in der Abbildung ein Config-Server zu sehen, der eine API für eine gemeinsame Definition der Konfiguration von Eigenschaften für alle Services bereitstellt. Durch dieses Design der Architektur ermöglicht Spring-Boot die Erstellung einer verteilten Anwendung durch Cloud-Services. Somit besteht die Möglichkeit durch Containerisierung der Spring Applikationen ein verteiltes System in Kubernetes für den Betrieb bereitzustellen.

In diesem Abschnitt wurden grundlegende Eigenschaften von dem Spring Framework zusammengefasst, die für die Implementierung einer gesicherten Standalone Anwendung oder für eine verteilte Anwendung mit einzelne Microservices relevant sind.

2.2. Microservices

In diesem Abschnitt wird die Idee und Historie von Microservices aufgezeigt, das aus dem kontinuierlichen Ziel einer besseren Kommunikation von verschiedenen Plattformen und benutzerfreundlichen Systemen abzuleiten ist.
Der erste signifikante Schritt in der Historie von Microservices entstand durch die Veröffentlichung von dem System „Enterprise Java Beans“ von IBM, da aufgrund der Probleme mit dieser Technologie eine Service Oriented Architecture (SOA) entwickelt wurde [Foote, 2021]. SOA ermöglichte Services unabhängig von den Systemgrenzen und der verwendeten Programmiersprache miteinander zu kommunizieren, sodass die Kommunikation unabhängig von der Implementierung ist [Qusay, 2005]. Dies inkludiert den ersten

Grundgedanken von Microservices von lose gekoppelten Services. Allerdings war dieses Konzept von einer losen Kopplung nicht so weit verbreite durch SOA, sondern wurde erst durch die steigende Popularität von REST Web-Services in Verbindung mit der Verwendung von Containern, die eine Anwendung mit allen benötigten Dateien in einer Einheit definieren, und Volumes bekannt, wobei ein Volume einen logischen Teil eines Massenspeichers darstellt. Durch diese Verbindung des ursprünglichen Gedanken der SOA mit den neuen Möglichkeiten durch REST, Container-Orchestrierung und Volumes ist der heute bekannten Ansatz von Microservices entstanden [Rodríguez et al., 2016, Docker, 2021g]. Der heutige Ansatz von Microservices ist ein Anwendungsentwicklungsansatz in dem voneinander unabhängige Prozesse implementiert werden, die durch APIS untereinander kommunizieren, wodurch jeder Microservice in einem System ein autonomes System darstellt [García, 2020, S. 168]. Diese Art der Anwendungsentwicklung ermöglicht die Entkopplung der Services und bietet den Vorteil, dass nicht nur ein Schnitt nach einer 3-Schichtenarchitektur durch vertikale Microservices möglich ist, wie es in García, 2020, S. 36 beschrieben ist, sondern auch die Möglichkeit zu einem horizontalen Schnitt in der Anwendungsarchitektur zu berücksichtigen ist. Diese horizontalen Services zeichnet ein großes Potenzial aus, da diese eine einfache Erweiterbarkeit der Anwendung ermöglichen, wodurch für eine Erweiterung der Funktionalität lediglich ein neuer Service integriert wird, ohne bestehende Microservices zu verändern [Richardson, 2020]. Dies ist an die Voraussetzung gebunden, dass die Schnittstellen der bereits implementierten Services die benötigte Kommunikation mit dem neuen Service anbieten beziehungsweise der neu hinzugefügte Service auf bereits bestehende Endpunkte zurückgreift. Darüber hinaus ist durch diesen Ansatz der Microservices geboten verschiedene Technologien zu nutzten und verwendete Tools schnell auszutauschen, da alle Services über APIS kommunizieren und somit jede Technologie einsetzbar ist, die eine Integration von API Schnittstellen anbietet.

Im Optimalfall greift jeder Microservice nur auf eine Datenbank zu, was bei einer Migration einer monolithischen Anwendung zu Microservices zu einer Aufteilung des Datenmodells führt. Deshalb ist eine wichtige Entscheidung bei der Verwendung von Microservices, welche Pattern verwendet werden, um die Kommunikation der Services mit den verteilten Datenbanken zu realisieren. Deshalb zeigt die folgende Auflistung, welche Pattern nach Newman, 2019, S. 125ff bei einer Migration zu einer Microservice-Architektur für die Aufteilung der Datenbank zu betrachten sind.

- Shared Database
- Database View
- Database Wrapping Service
- Database-as-a-Service Interface
- Synchronize Data in Application
- Tracer Write

- Split Table
- Move Foreign-Key Relationship to Code

Die Aufzählung startet mit einigen Pattern, die nur als Brückenlösung für die Datenbank bei einer inkrementellen Migration zu Microservices genutzt werden sollten, da diese Modelle bei einer vollständigen Microservice-Architektur nur für Spezialfälle einzusetzen sind.
Dadurch ist es beispielsweise möglich, mit der Verwendung einer „Shared Database“ einzelne Services aus dem Monolith auszugliedern, ohne die Datenbank bereits aufteilen zu müssen [Newman, 2019, S. 127]. Dieses Modell sollte nur innerhalb des Migrationsprozesses genutzt werden, außer es handelt sich um statische Daten, die von mehreren Services verwendet werden, wie Länderlisten oder Auflistungen von Staaten. Das Pattern „Database View“ sollte ausschließlich für die Zeit während der Migration angewendet werden, wenn es unpraktisch ist, das Datenbankschema bereits bei Start der Migration aufzuteilen. Deshalb ist dieses Modell zu vermeiden, wenn das Endziel ist, die Informationen des Datenbankschemas über mehrere Services darzustellen [Newman, 2019, S. 134]. Ein ähnlichen Anwendungsansatz verfolgt das Modell „Database Wrapping Service“ nach Newman, 2019, S. 135, da darauf zurückgegriffen werden sollte, wenn die grundlegende Struktur schwierig aufzuteilen ist. Ein Pattern, das in einem Spezialfall über die Migration hinaus zu verwenden ist, hat die Bezeichnung „Database-as-a-Service Interface“. Dies wird verwendet, wenn mehrere Services nur lesend auf eine große Menge von Daten zugreifen müssen. Für die Umsetzung des Prinzips muss sichergestellt sein, dass alle Services nur lesend auf die geteilte Datenbank zugreifen und zudem der Prozess der Datenaktualisierung definiert wird [Newman, 2019, S. 137]. Wenn während einer Migration der Monolith und die Microservices auf gleiche Daten zugreifen müssen, ist das Modell „Synchronize Data in Application“ anzuwenden, was aber nach Newman, 2019, S. 148 einen großen Aufwand in der Implementierung birgt, da die Synchronisation von dem Monolithen und den Microservices sichergestellt werden muss. Ein ähnliches Pattern wie „Synchronize Data in Application“ ist ein „Tracer Write“, wobei es hier zwei Quellen der Wahrheit (engl. sources of truth) gibt [Newman, 2019, S. 149 ff.].

Im Gegensatz zu den bereits dargestellten Modellen sind die letzten beiden Pattern „Split Table“ und „Move Foreign-Key Relationship to Code“ dauerhaft in einer Microservice-Architektur in verschiedenen Anwendungsfällen einsetzbar. Da diese zudem eine essenzielle Rolle für diese Fallstudie haben, werden diese Modelle im Folgenden mit den verbundenen Vor- und Nachteilen dargestellt.
Das Pattern „Split Table“ wird verwendet, wenn in einem System eine Tabelle über zwei oder mehrere Servicegrenzen hinweg benötigt wird [Newman, 2019, S. 171 f.]. Graphisch ist hierfür beispielhaft eine Ausgangssituation auf der linken Seite der Abbildung 2.1 aufgezeigt. In diesem Teil der Abbildung ist zu sehen, dass die Tabelle `Projekte` von den zwei verschiedenen Unternehmensbereichen Projektmanagement und Controlling benötigt wird. Das Projektmanagement benötigt die Informationen aus der Tabelle, um Detailinformationen zu den Projekten zu erhalten und das Controlling benötigt auf der

anderen Seite die Tabelle, um den finanzielle Aspekt zu betrachten. Deshalb sollten bei einer Microservice-Architektur die beiden Services abstrahiert voneinander designend werden und somit auch eine eigene Datenbank für jeden Service definiert werden, da die beiden Bereiche jeweils unterschiedliche Spalten der Tabelle benötigen, was in der Abbildung 2.1 durch die gestrichelten Rechtecke dargestellt ist. Dementsprechend ist die Tabelle auf die beiden Services aufzuteilen, wie es auf der rechten Seite der Abbildung dargestellt ist. In der Abbildung hat zwar jede Tabelle eine `ProjektID` als Primärschlüssel, aber unterschiedliche Attribute, wodurch die ursprüngliche Tabelle auf die beiden Services aufgeteilt ist. Dieses Pattern ist am besten anzuwenden, wenn in dem Monolith eine Tabelle für verschiedene Bereiche genutzt wird, die durch verschiedene Microservices in der Architektur dargestellt werden.

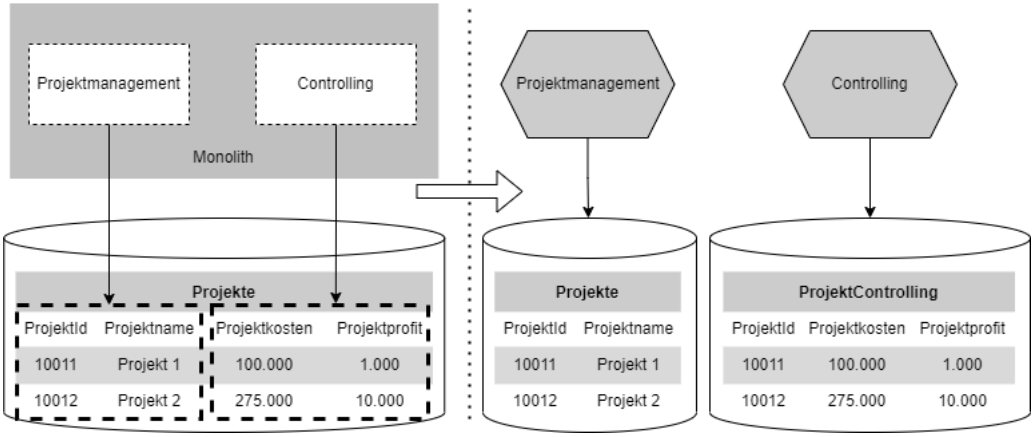

Abbildung 2.1: Split-Table Modell [Newman, 2019, S. 171]

Für eine andere Art von Anwendungsfall in einer Microservice-Architektur ist das Modell „Move Foreign-Key Relationship to Code“ nach Newman, 2019, S. 173 zu betrachten. Durch dieses Modell wird ermöglicht Tabellen, die zuvor in der monolithischen Anwendung in einer Datenbank durch einen Fremdschlüssel verknüpft waren, auf zwei Services und somit auf zwei Datenbankservices aufzuteilen. In dem oberen Teil der Abbildung 2.2 ist diese ursprüngliche Konstellation der Tabellen beispielhaft abgebildet. Aus den Tabellen ist zu entnehmen, dass die Tabelle `Produkte` benötigte Informationen zu der Produktion eines Produktes enthält. Die `Verkäufe` Tabelle zeigt in diesem Beispiel die verkauften Produkte und listet diese mit dem entsprechenden Fremdschlüssel für das Produkt mit weiteren Informationen, wie Umsatz von einem Verkauf eines Produktes und das Verkaufsdatum. Bei der Migration dieses Monolithen wird beispielhaft das Produktmanagement mit der `Produkte` Tabelle in einen eigenen Service ausgegliedert und ein Service für den Vertrieb entworfen, wodurch die beiden Tabellen nicht länger in einem Datenbankschema mit Fremdschlüssel zusammengefasst sind. Deshalb muss hier die Datenbank nach dem Prinzip des „Move Foreign-Key Relationship to Code“ aufgeteilt werden, was in dem unteren Teil der Abbildung 2.2 aufgezeigt ist.

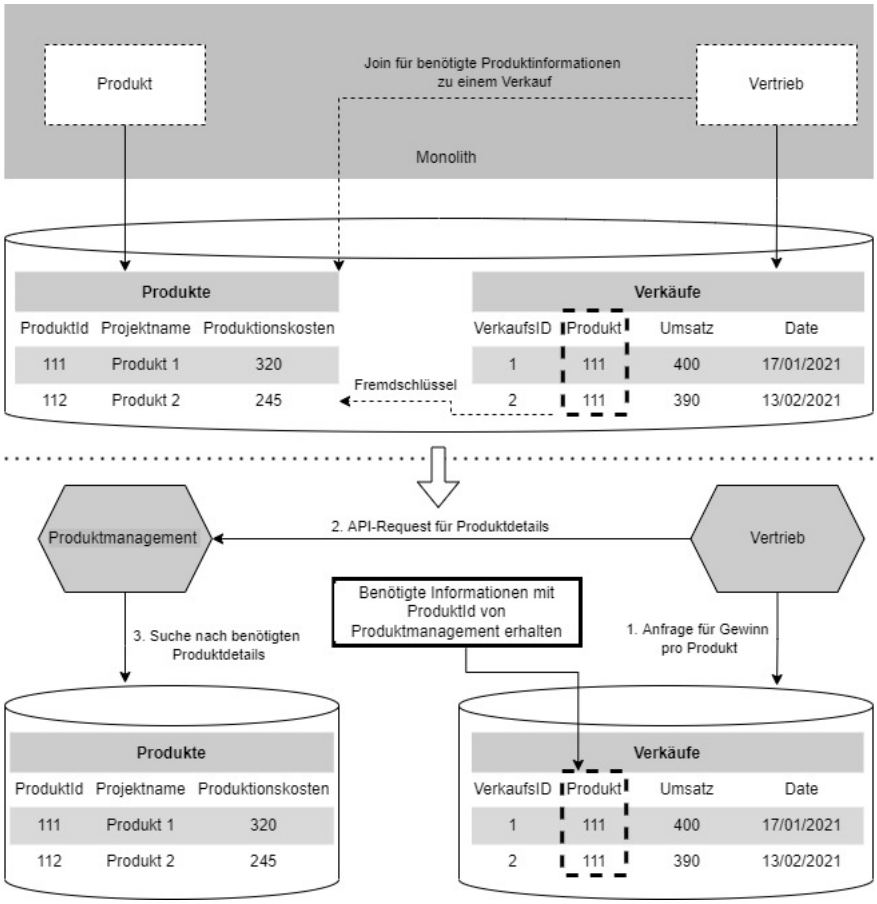

Abbildung 2.2: Abbildung einer Fremdschlüsselbeziehung [Newman, 2019, S. 174f]

Aus der Abbildung ist zu erkennen, dass die Struktur der Tabellen nicht verändert worden ist, allerdings keine Fremdschlüsselbeziehung zwischen den beiden Tabellen definiert ist, da diese in verschieden Datenbankschemata abgebildet sind. Somit ist es durch dieses Design nicht mehr möglich, die benötigten Produktinformationen für einen Verkauf durch einen `JOIN` Befehl zu ermitteln, was dazu führt, dass nun der `JOIN` in dem Code der entsprechenden Services abgebildet werden muss. Wenn bei diesem Design die Verkäufe mit Produktname, Umsatz und Produktionskosten ermittelt werden sollen, müssen zunächst in dem Vertriebs-Service alle benötigten Verkäufe selektiert werden, danach anhand der ermittelten `ProduktId` aus dem Vertriebs-Service die zusätzlichen Informationen zu den entsprechenden Produkten aus dem Produktmanagement-Service erzeugt werden, sodass alle benötigten Daten zurückgegeben werden. Festzustellen ist, dass durch dieses Modell mehr Abfragen im Vergleich zu dem Monolithen benötigt werden, um die gleichen Informationen zu erhalten, was zu einer Erhöhung der Latenz führt. Allerdings bietet diese Konzept die Möglichkeit Service-Grenzen auch über eine Fremdschlüsselbeziehung hinweg zu entwerfen, was aber nur gemacht werden sollte, wenn sichergestellt ist, dass die beiden Bestandteile nicht innerhalb eines Services darzustellen sind, da in diesem Modell die Datenkonsistenz durch die Implementierung zu gewährleisten ist [Newman, 2019, S. 178].

Zusammenfassend wurde in diesem Kapitel das Konzept von Microservices dargestellt und einzelne Modelle zur Aufteilung eines Datenmodells für eine verteilte Datenspeicherung innerhalb einer Microservice-Architektur aufgezeigt, was für die Migration in der Fallstudie von großer Wichtigkeit sein wird.

2.3. Docker Technologie

Nachdem in dem vorherigen Abschnitt die Grundlagen zu Microservices erläutert wurden, wird nun eine Technologie vorgestellt, mit der die Containerisierung von Microservices unterstützt wird.
Die Platform as a Service (PaaS) Technologie Docker bietet die Möglichkeit einen Service mit allen benötigten Abhängigkeiten in einem Container zu orchestrieren. Um eine Container-Orchestrierung von einer Anwendung durchzuführen, muss zunächst ein Dockerfile erstellt werden. Anschließend ist es möglich in der Kommandozeile durch `docker build` ein Image der Anwendung zu erzeugen. Ein Image ist ein Paket einer Software, das relativ und isoliert ist. Durch Docker ist es möglich ein Image als Container auszuführen, was eine Run-Time-Environment eines Images darstellt [Docker, 2021d]. Somit wird entweder durch den Befehl `docker run` oder per Klick in der Desktopanwendung Docker-Desktop aus einem bestehenden Image ein Container gestartet [Webb, 2020, Docker, 2021d].
Außerdem bietet Docker mit der Community-Plattform Docker-Hub eine Container-Image-Bibliothek. In dieser Bibliothek sind offizielle Images, öffentliche Images und die eigen erstellen Images zu finden, die durch den Befehl `docker pull` herunterzuladen sind. Damit die Option besteht eigen erstellte Images herunterzuladen, müssen diese erst mit dem entsprechenden Benutzernamen von Docker-Hub markiert werden und dann mit dem Kommando `docker push` in die Image-Bibliothek hochgeladen werden. Dahingegen ist ein offizielles Image ein Image, das von Docker selbst überprüft und veröffentlicht wird. Diese Images stellen Laufzeitumgebungen für Programmiersprachen, Datenspeicher und andere Dienste wie Betriebssystem zur Verfügung [Docker, 2021a]. Des Weiteren sind auch Images von anderen Usern in Docker-Hub zu finden, insofern der Ersteller des Images den Zugang auf öffentlich eingestellt hat. Diese Funktionalitäten von Docker ermöglichen Anwendung Systemunabhängig auszuführen und fördern zudem einen schnellen Bereitstellungsprozess.
Da einige Microservices eine Möglichkeit benötigten Daten persistent abzuspeichern, bietet Docker auch die Option Volumes zu erstellen, was die Popularität von dem Microservice Ansatz verstärkt. Die Datenspeicherung ist in Docker zwar auch durch die Hinzufügung von „bind mounts“ zu einem Container möglich, allerdings sind diese abhängig von der Dateistruktur und von dem Betriebssystem. Deshalb werden in den überwiegenden Fällen Volumes genutzt, da diese einfach zu managen und über mehrere Services hinweg mit einer besseren Performance als „bind mounts“ einsetzbar sind [Docker, 2021g]. Damit ein Container ein Volume zur Datenspeicherung verwendet, muss ein „mount“ des Volumes an den Container erzeugt werden, was durch den Parameter `--mount` in dem Kommando für das Starten eines Containers in Docker möglich ist.
Somit ist festzustellen, dass Docker in der Verbindung mit Docker-Hub ein wichtiges

Tool für das Deployment von Container-Anwendungen ist, die beispielsweise mit der Kubernetes Technologie bereitgestellte werden, die in dem nächsten Abschnitt genauer beschrieben wird.

2.4. Kubernetes

Containerisierung von Microservice-Anwendungen bietet einige Vorteile und hilft dabei Probleme, wie Skalierbarkeit, Zuverlässigkeit und Robustheit, zu beheben. Allerdings führen beispielsweise die Fehlerbehandlung, die persistente Speicherung von Daten in Datenbankservices oder die Kommunikation zwischen den einzelnen Services zu einigen Schwierigkeiten in dem Anwendungsdesign. Diese Nachteile können durch die Verwendung von Kubernetes, das auch abgekürzt als „k8s“ bezeichnet wird, behoben werden [Yilmaz, 2019, S. 70]. Kubernetes ist eine open-source Plattform, die von Google Inc. im Jahr 2014 der Öffentlichkeit vorgestellt wurde und im darauffolgenden Jahr an die neu gegründete Cloud Native Computing Foundation (CNCF) gespendet wurde, die das Projekt seitdem verwaltete. Mit mittlerweile mehr als 100.000 Commits auf GitHub ist Kubernetes das aktuell größte open-source Softwareprojekt.
Für Kubernetes sind verschiedenste Definitionen zu finden, da es ein schnell wachendes Ökosystem ist und so oft als Plattform bezeichnet wird mit der Plattformen gebaut werden. Die offizielle Definition ist, dass Kubernetes eine Plattform für die Verwaltung von containerisierten Services ist, die die Konfiguration und die Automatisierung erleichtert [Kubernetes, 2021d]. Dadurch bietet es die Möglichkeit komplexe Cloud-Native-Anwendungen mit einer hohen Flexibilität und Verfügbarkeit zu verwalten. Die Flexibilität wird durch die variable Anzahl an Instanzen eines Services ermöglicht, was zu einer einfachen Skalierbarkeit führt. Weiter wird durch die Ausführung der Microservices in einem Kubernetes-Cluster die Verfügbarkeit verbessert, da das Cluster sich auf den kompletten Lebenszyklus von Containern konzentriert und so bei einem Ausfall einer Instanz direkt eine neue Instanz hinzufügt, damit der Service dauerhaft verfügbar ist [Yilmaz, 2019, S. 70].
Für eine Bereitstellung von Microservices in Kubernetes ist es wichtig einen Überblick über die eigentliche Architektur und die Bestandteile der Plattform zu haben. Alle Microservices werden in Kubernetes in Clustern ausgeführt, das in der Abbildung A.2 zu erkennen ist. Ein Cluster bestimmt einen gewünschten Zustand, der durch das System dauerhaft eingehalten wird und besteht aus mindestens einer Node, in der die Services schlussendlich auf einer Maschine ausgeführt werden. Innerhalb von jedem Cluster ist die Control-Plane dafür zuständig, die Nodes und Pods in einem Cluster zu managen [Kubernetes, 2021a]. Der wichtigste Bestandteil der Control-Plane ist die Kubernetes-API, da diese eine Hypertext Transfer Protocol (HTTP) API für Enduser zur Verfügung stellt, damit die Services von außerhalb erreichbar sind und regelt außerdem die Kommunikation der Services innerhalb eines Clusters.
In einer Node sind Deployments, Services und Pods die wichtigsten Bestandteile, die über eine Yet Another Markup Language (YML) Datei definiert werden. Diese definierten Kubernetes-Objekte werden anschließend mit dem Befehl `kubectl apply -f` zu einer Node hinzugefügt. Kubernetes Services dienen unter anderem dazu hinzugefügte Microser-

vices außerhalb des Clusters verfügbar zu machen [Kubernetes, 2021c]. Dies wird benötigt, da Pods, die eine Gruppe für die Ausführung von einen oder mehreren Container mit benötigten Ressourcen und Umgebungsvariablen darstellen, zwar über eine Internet Protocol (IP) Adresse verfügen, aber diese IP-Adresse nicht außerhalb des Clusters bekannt gemacht wird [Yilmaz, 2019, S. 85]. Für das Management der Container-Anwendungen werden in Kubernetes zusätzlich Deployments definiert, die für die Versionierung der Services benötigt werden. Durch die Deployments wird ermöglicht Dienste zu einem Cluster hinzuzufügen, ein Update für einen Dienst durchzuführen, den Dienst zu einer älteren Version zurückzusetzen oder einen Dienst zu skalieren [Yilmaz, 2019, S. 90].

Nachdem die Bestandteile von Kubernetes dargestellt wurden, soll nun genauer darauf eingegangen werden, welche Tools für die Umsetzung von einem Kubernetes-Cluster zur Verfügung stehen.
Die einfachste Methode, um ein Kubernetes-Cluster zu betreiben ist Minikube zu verwenden. Durch Minikube ist es möglich nach dem Download mit dem Kommando `minikube start` Kubernetes lokal auszuführen, wobei ein Kubernetes-Cluster mit einer Node in einer virtuellen Maschine erstellt wird [Kubernetes, 2021b]. Allerdings ist Minikube ausschließlich für das ausprobieren beziehungsweise für Entwicklungstest geeignet, da das Cluster nur lokal erstellt wird und somit trotz Erstellungen von Services nicht von außerhalb erreichbar ist. Insofern Kubernetes als Produktionsumgebung genutzt werden soll, ist Minikube deshalb nicht einzusetzen. Stattdessen muss auf eine public Cloud-Lösung oder auf eine On-Premisis-Lösung für Kubernetes zurückgegriffenen werden. Für diese Cloud-Lösungen gibt es zahlreiche Anbieter, die produktionsbereite Kubernetes-Cluster anbieten. Die bekanntesten, größten und meist verwendeten Anbieter von Cloud-Lösungen sind beispielsweise Amazon Web Services (AWS), die Google Cloud Plattform (GCP) mit der Kubernetes-Engine und Microsoft Azure, die alle eine Verwaltung von Kubernetes-Cluster anbieten, aber sich in einigen Details und Konfigurationen unterscheiden, was ausführlich in dem Buch „Kubernetes A Complete DevOps Cookbook“[Karslioglu, 2020] beschrieben ist.

Mit diesen grundlegenden Informationen zu Kubernetes und Microservices wird ermöglicht in den nächsten Kapiteln die Migration einer monolithischen Spring-Boot Anwendung zu Kubernetes in dem Design einer fiktiven Fallstudie durchzuführen.

3. Anforderungsanalyse

In diesem Kapitel wird die bestehende Anwendung analysiert, damit der Entwurf für ein manuelles Deployment in Kubernetes in dem Kapitel 4 durchführbar ist. Die zu migrierende Spring-Boot Anwendung, die bereits in einem früheren Studienfach implementiert wurde, ist in einer adaptierten Version in dem GitLab-Repository (`https://gitlab.lrz.de/hm-halbig/scheduler`) gespeichert, deren Details in den folgenden Abschnitten genau dargestellt werden.

3.1. Analyse der bestehenden Anwendung

Die Web-Anwendung „Course-Planner“ hat das Ziel Anwendern die Möglichkeit zu bieten sogenannte „Courses“ zu verwalten. Um dieses Ziel zu erreichen ist die Hauptanforderung jedem registrierten User die Option zu bieten bestimmte Courses zu abonnieren und zu erstellen. In jedem Course werden verschiedene Tasks von dem Ersteller hinterlegt, damit alle User anhand eines Status pro Task einen Überblick über die abzuleistenden Courses und zu den entsprechenden Aufgaben haben. Im Folgenden werden die Anforderungen zu allen Kernfunktionalitäten der Anwendung im Detail dargestellt und das zu migrierende Datenmodell aufgezeigt.

3.1.1. Funktionalitäten der Anwendung

In diesem Abschnitt werden die Funktionalitäten der zu migrierenden Anwendung dargelegt und anhand eines Anwendungsfalldiagramms in der Abbildung 3.1 grafisch zusammengefasst. In dieser Form des Verhaltensdiagramms werden die dynamischen Aspekte des „Course-Planners“ mithilfe von einer Menge an Anwendungsfällen und anhand von Akteuren dargestellt [Booch et al., 1999, S. 128]. Zusätzlich sind die funktionalen Anwendungsfälle, die aus der bestehenden Anwendung für das neue System zu übernehmen sind, mit einer textuellen Beschreibung und einer eindeutigen ID in der Tabelle 3.1 aufgelistet.

Wie in dem Diagramm 3.1 zu sehen ist, sind in der Anwendung vier Akteure definiert. Der grundlegende Akteur ist der `Admin`, weil dieser Akteur eine Generalisierung von allen drei anderen Akteuren darstellt, da nicht alle Akteure die gleichen Rechte für die Ausführung von Anwendungsfällen haben. Weiterhin ist in der Abbildung zu erkennen, dass der Anwendungsfall mit der ID AF1, der die Erstellung von neuen Usern, die Zuweisung von User-Rollen und das Deaktivieren von Usern mit einschließt, nur von einem `Admin` ausführbar ist. Überdies ist es nur einem `Admin` möglich alle Anwendungsfälle eines `Users` durchzuführen, weshalb der Akteur `Admin` die Generalisierung eines `Users` ist. Ein `Admin` hat zudem die Option alle Courses, Status und Tasks einzusehen und zu bearbeiten, damit sichergestellt ist, dass bei auftreten von Problemen benötigte Gegenmaßnahmen durch einen `Admin` ausführbar sind.
Die wichtigste Funktion und somit auch der wichtigste Anwendungsfall der Anwendung ist der AF3, da ausschließlich Anwender auf die kompletten Funktionalitäten Zugriff haben, die zuvor von einem `Admin` durch den AF1 hinzugefügt worden sind. Insofern ein Anwender

den AF3 nicht durchgeführt hat, ist dieser in der Lage nur die allgemeinen Informationen und Dokumentationen der Anwendung zu sehen. Sobald ein `User` den Usernamen und das dazugehörige Passwort von dem `Admin` erhalten hat, ist die Durchführung von dem AF3 mit den gegebenen Referenzen für den User möglich, was in dem Anwendungsfalldiagramm durch die Generalisierung des AF3 dargestellt ist. Dies schließt auch mit ein, dass alle Nutzer und somit auch nicht angemeldete Nutzer ausschließlich auf die Home-Seite und die Login-Seite zugreifen. Insofern eine andere Seite der Anwendung aufgerufen wird, ist zunächst eine Authentifikation von einem Anwender erforderlich [Alex et al., 2004, S. 108].

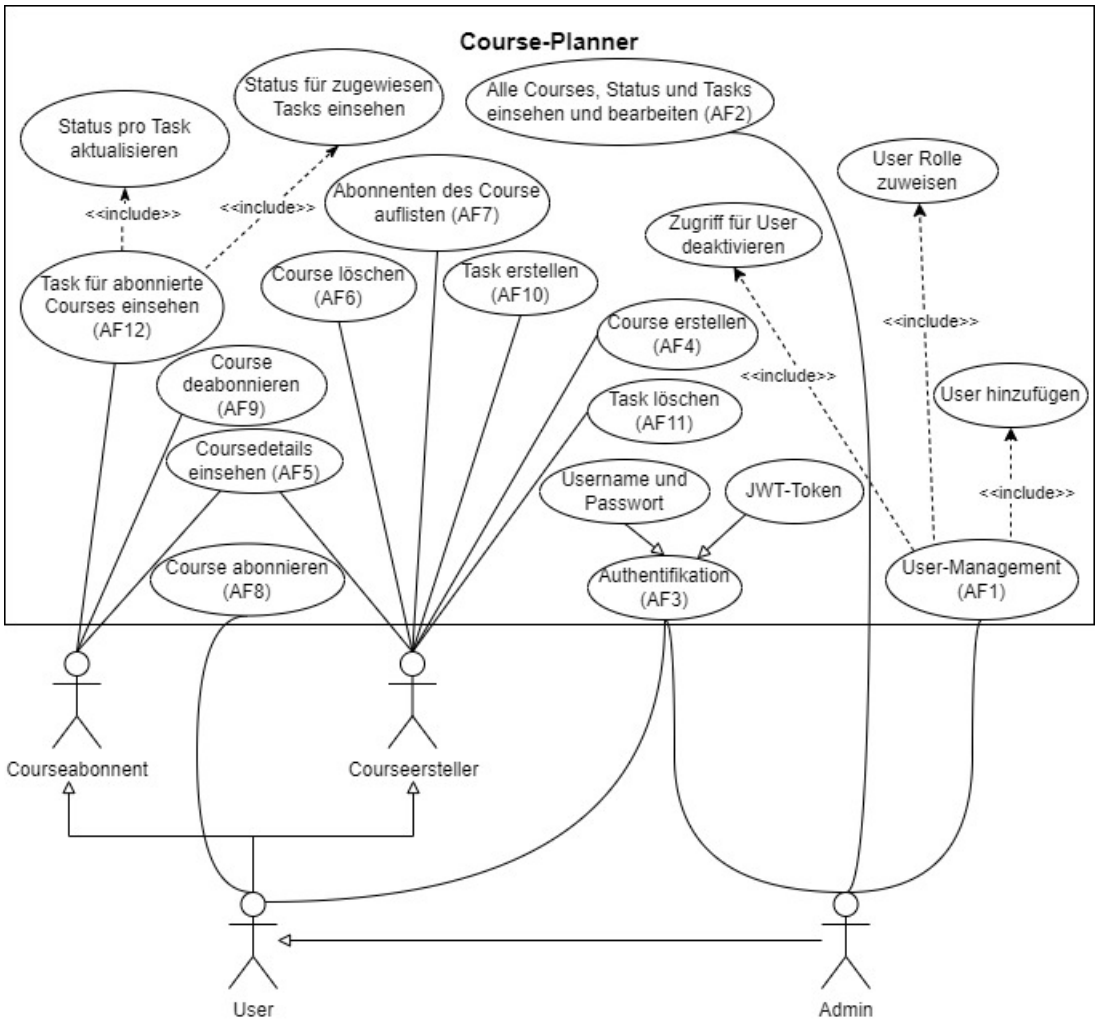

Abbildung 3.1: UML Anwendungsfalldiagramm

Nach erfolgreicher Anmeldung eines `Users` hat der Akteur die Option über die Eingabe einer Course-ID einen Course zu abonnieren oder selbst einen Course in der Courseübersicht zu erstellen. Somit wird ein `User` in `Courseabonnent` und `Courseersteller`

unterteilt, da jeder `User` jeweils ein Abonnent oder Ersteller ist, je nachdem welche Aktion ausgeführt wird. Ein `User` ist ein `Courseersteller`, sobald dieser einen eigen erstellten Course, der durch den AF4 hinzugefügt wird, in der Courseübersicht öffnet. Denn in dieser Coursedetailübersicht hat nur der `Courseersteller` die Option die Coursedetails mit der Abonnentenliste und die Anzahl an abgeschlossenen Tasks pro User einzusehen. Außerdem ist es nur einem `Courseersteller` oder einem `Admin` möglich die Anwendungsfälle AF6, AF10 und AF11 für diesen Course auszuführen. Sobald ein `User` die Details von einem zuvor abonnierten Course öffnet, ist dieser ein `Courseabonnent`, da in der Coursedetailübersicht für Abonnenten lediglich die Coursedetails mit den entsprechenden Tasks einzusehen sind. Dazu ist ein `Courseabonnent` dazu in der Lager in dieser Übersicht den Status seiner Tasks zu überprüfen und den Status pro Task nach dem AF13 entsprechend zu aktualisieren. Außerdem hat jeder `User` die Alternative mit dem AF12 eine Übersicht von allen Tasks von den abonnierten Courses einzusehen und den aktuellen Status des Tasks zu überprüfen, sowie den Status zu aktualisieren.
Durch diese Anwendungsfälle der monolithischen Anwendung wird gewährleistet, dass alle User einen Überblick über jeden abonnierten Course haben und die noch offenen Tasks einsehen und planen können, damit keine Deadline verpasst wird. Diese Anforderungen zu den Funktionalitäten müssen auch von der neuen Microservice-Anwendung erfüllt werden und bilden somit die Grundlage für den Entwurf der „Course-Service“ Anwendung in dem Kapitel 4.

3.1.2. Datenmodell der bestehenden Anwendung

In diesem Abschnitt wird das Datenmodell der Anwendung „Course-Planner“ analysiert, welches in der Abbildung 3.2 als Entity-Relationship-Modell (ER-Modell) nach Chen, 1976 aufzeigt ist.
In diesem Modell sind vier Entitäten abgebildet, die in verschiedenen Beziehungen zueinander stehen und jeweils einen festen Primärschlüssel als Attribut besitzen [Gadatsch, 2019, S. 9]. Die beiden Entitäten „Course“ und „User“ haben eine parallele Beziehung, da ein User entweder Abonnent oder Ersteller von einem Course ist. Diese Parallelität wird in dem Modell benötigt, da ein Course nur von einem User erstellt wird und somit nur ein User einem Course als Ersteller zugewiesen ist, was zu einer 1:N-Beziehung zwischen einem Course-Ersteller und einem Course führt [Gadatsch, 2019, S. 12]. Auf der anderen Seite ist es möglich, dass ein User auch Abonnent eines Courses ist. Hier ergibt sich eine N:M-Beziehung nach Gadatsch, 2019, S. 12, da ein User die Möglichkeit hat mehrere Courses gleichzeitig zu abonnieren und ein Course von einer Vielzahl an Usern abonniert wird. Aus diesem Grund wird eine parallele Beziehung zwischen den beiden Entitäten Course und User in dem Modell benötigt.
Eine Course hat eine weitere Beziehung zu der Entität „Task“, da ein Course eine variable Anzahl von Tasks hat. Somit hat ein Course eine 1:N-Beziehung zu der Task-Entität, weshalb in der Task-Tabelle in der Anwendung jeder Task einen Fremdschlüssel mit einem Primärschlüssel eines Courses hat. Die vierte Entität mit der Bezeichnung „Status“ dient zur Darstellung des aktuellen Bearbeitungsstandes eines Tasks von einem einzelnen User, weshalb ein Status genau einem Task und einem User zugeordnet ist. Somit ist einem

Status immer ein fester Primärschlüssel eines Users und eine dazugehörigen Task-ID zugewiesen. Allerdings hat ein User mehrere Status für verschiedene Tasks und zusätzlich sind einem Task auch mehrere Status zugewiesen, da für jeden User, dem der Task zur Bearbeitung durch abonnieren eines Course zugewiesen wird, ein Status erstellt wird. Dies bedeutet, dass die beiden Entitäten Task und User beide eine 1:N-Beziehung zu einem Status haben, damit jeder Status immer genau einem User und einem Task zugeordnet ist, um eine konsistente Statusspeicherung des jeweiligen Bearbeitungsstandes je User pro Task zu gewährleisten.

Dieses Datenmodell wurde durch die Definition von Entitätenklassen, die durch eine `@Entity` Annotation gekennzeichnet sind, in der Anwendung implementiert und werden somit durch Spring-Boot automatisch in einem Datenbankmodul mit den benötigten Primärschlüsseln, Fremdschlüsseln und Verknüpfungstabellen für M:N Beziehungen angelegt [García, 2020, S. 116].

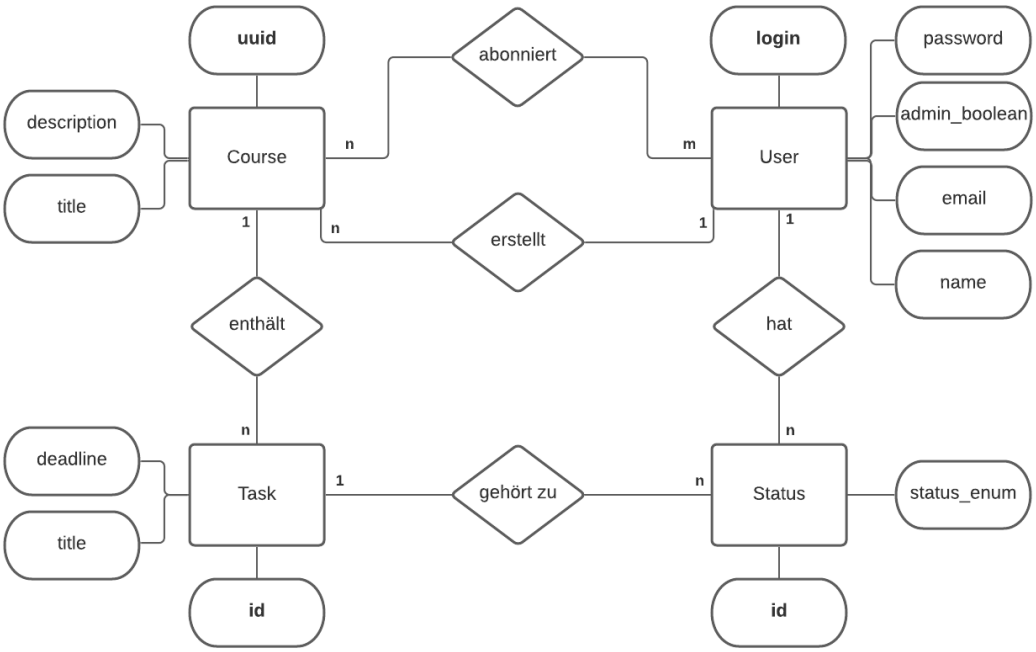

Abbildung 3.2: Datenmodell der Monolith-Anwendung

3.2. Anforderungen für eine Microservice-Architektur

Der letzte Schritt in der Analyse ist, die bestehende Architektur der Anwendung zu analysieren und darzustellen, sodass aus den bisher gesammelten Informationen zu der bestehenden Anwendung in diesem Kapitel die funktionalen und nicht-funktionalen Anforderungen für die Microservice-Architektur spezifiziert werden.

Zunächst wird das Design der bestehenden Anwendung dargestellt, um die Beschreibung der Anwendung zu komplettieren. Auffällig hierbei ist, dass trotz der monolithischen Implementierung in einem Java-Projekt innerhalb dieses Projektes das 3-Schichtenmodell nach Horn, 2007b zu erkennen ist. Dies wird deutlich durch die gewählte Projektstruktur, da die Präsentationsschicht durch die Hypertext Markup Language (HTML) Dateien in dem `ressources` Ordner abgebildet wird. Diese Präsentationsschicht greift durch die Referenzierung der Controller-Klassen auf die Applikationslogik zu. Die komplette Applikationslogik ist in dem Package mit der Bezeichnung `svc` implementiert, wobei dieses Akronym in der Bezeichnung für ein Service-Package steht. In diesem Package sind alle Funktionen für die komplette Anwendungslogik hinterlegt, die anschließend in den Controller-Klassen genutzt werden, damit über die Controller die Bereitstellung der benötigten Informationen für die Präsentationsschicht ermöglicht wird. Die Datenhaltungsschicht wird in der bestehenden Anwendung durch das relationale Datenbankmanagementsystem `H2` dargestellt, das in den Applikationseigenschaften mit den Konfigurationen, die in dem Listing 3.1 zu sehen sind, definiert ist [JavaTpoint, 2021]. Zusätzlich sind die entsprechen Datenobjekte in den Entitätenklassen definiert und JPA-Repositories zu allen definierten Entitäten angelegt, damit eine Verbindung zwischen der Applikationslogik und der Datenhaltungsschicht hergestellt wird, sodass die benötigten Daten aus dem Datenbankmodul geladen, gespeichert und gegebenenfalls gelöscht werden.

```
1 spring.h2.console.enabled = true
2 spring.datasource.url=jdbc:h2:mem:testdb
3 spring.datasource.driverClassName=org.h2.Driver
4 spring.datasource.username=sa
5 spring.datasource.password=password
6 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Listing 3.1: Definition einer H2-Datenbank

Nachdem nun der Aufbau der bestehenden Anwendung bekannt ist, werden die Anforderungen der Microservice-Anwendung definiert. Hierbei ist zunächst zwischen funktionalen und „nicht-funktionalen Anforderungen“[Horn, 2007a] zu unterscheiden, da die funktionalen Anforderungen, die in der Abbildung 3.1 dargestellt sind, für den Entwurf unverändert übernommen werden. Diese sind zusammen mit den nicht-Funktionalen Anforderungen, die für die Migration neu zu definieren sind, in der Tabelle 3.1 zusammengefasst. Im folgenden werden deshalb die abgeleiteten nicht-funktionalen Anforderungen dargestellt. Da die komplette Business- und Anwendungslogik durch eine Microservice-Architektur dargestellt wird, ist der Einsatz von einem DDD nach Evans, 2015 inkludiert, damit das System in fachliche Komponente aufgeteilt wird. Diese fachlichen Komponenten stellen nach dem Modell wiederum Services zur Kommunikation mit anderen Komponenten zur Verfügung. Durch diese Untergliederung in fachliche Komponente werden Domänen dargestellt, die eine Fachlichkeit mit Geschäftsregeln, Beziehungen und Prozessen darstellen. Außerdem ist eine Abgrenzung der fachlichen Komponenten zu definieren, was durch Bounded Context (BC) umgesetzt wird, da ein BC den geschäftlichen Kontext

und eine spezifische Abgrenzung der Organisationseinheiten darstellt [Evans, 2015, S. 2]. Des Weiteren ist ein Context-Mapping zu integrieren, damit die Grenzen der BC in einer organisatorischen Beziehung dargestellt werden, um einen BC innerhalb einer globalen Sicht der Anwendung zuzuordnen [Evans, 2015, S. 29]. Diese Anforderungen stellen sicher, dass die Microservices voneinander lose gekoppelt sind und die Kohäsion, also welcher Teil des Codes bei einer Änderung zusammen geändert werden muss, durch die BCs und Context-Maps dargestellt ist. Durch eine starke Kohäsion in dem System soll sichergestellt werden, dass bei einer Änderung so wenig Services wie möglich adaptierten werden müssen, um eine hohe Änderbarkeit zu gewährleisten [Newman, 2019, S. 17]. Zusätzlich ist durch die Implementierung der Anwendungslogik die Skalierbarkeit sicherzustellen, weshalb es wichtig ist, dass die Möglichkeit besteht eine variable Anzahl an Instanzen pro Service bereitzustellen, was auch die Zuverlässigkeit und die Erreichbarkeit des Systems positiv beeinflusst. Außerdem wird durch die Microservices die Effizienz bei der Weiterentwicklung ermöglicht, das einen schnellen Entwicklungsprozess garantiert. Unerlässlich für die Anwendung ist außerdem eine persistente und dauerhafte Datenspeicherung der User-, Task- und Coursdaten zu gewährleisten, damit die Daten konsistent an die Nutzer der migrierten Anwendung übermittelt werden.

Bezeichnung	ID	Beschreibung
User-Management	AF1	Admins ist es möglich neue User hinzufügen sowie Zugriffe von Usern zu deaktivieren.
Admin-Zugriffe	AF2	Admins haben die Option durch API-Requests alle Courses, Tasks und Status einzusehen und zu bearbeiten.
Authentifikation	AF3	Funktionalitäten der Anwendung werden nur von registrierten und angemeldeten Usern verwendet.
Course erstellen	AF4	Alle User haben die Möglichkeit beliebig viele Course mit Titel und Beschreibung zu erstellen.
Coursedetails einsehen	AF5	Der Courseersteller, Abonnenten des Courses und Admins sind in der Lage die Details einen Courses aufzurufen.
Course löschen	AF6	Der Coursersteller und Admins haben die Möglichkeit einen Course zu entfernen, insofern keine Tasks in diesem Course hinterlegt sind.
Courseabonnenten listen	AF7	Der Coursersteller und Admins haben die Option alle Abonnenten des Courses aufzulisten und die Anzahl an abgeschlossen Tasks pro User einzusehen.
Course abonnieren	AF8	Jeder User hat die Option einen Course durch Angabe der Course-ID zu abonnieren, insofern dieser nicht der Ersteller oder bereits ein Abonnent des Courses ist.
Course deabonnieren	AF9	Abonnenten haben die Möglichkeit einen Course zu deabonnieren, wobei die verbundenen Status des Users für diesen Course gelöscht werden.

Task erstellen	AF10	Dem Coursersteller und Admins ist es möglich einen Task mit Deadline für den Course zu erstellen, wobei die entsprechenden Status für die Abonnenten automatisch erzeugt werden.
Task löschen	AF11	Der Courserstellter und Admins sind in der Lage hinzugefügte Tasks zu löschen. Status, die mit dem Task verknüpft sind werden hierbei auch gelöscht.
Tasks einsehen	AF12	Jeder User sieht die Tasks mit dem Status der Abarbeitung von den abonnierten Courses.
Status aktualisieren	AF13	Jeder User hat die Option den Status seiner abzuarbeitenden Tasks zu aktualisieren.
Details des aktuellen Users ermitteln	FA14	Der Username, Vor- und Nachname, Rollen sowie E-Mail von dem aktuell angemeldeten User werden von dem System ermittelt.
Microservice-Architektur	NFA1	Die komplette Business- und Anwendungslogik wird durch durch lose gekoppelte Microservices implementiert.
Domänen definieren	NFA2	Das System wird nach dem DDD in fachliche Komponente aufgeteilt, die Service zu Kommunikation mit anderen Komponenten bereitstellen.
Abgrenzungen der Organisationseinheiten	NFA3	In dem System werden fachliche Komponenten durch BCs klar abgegrenzt.
Context-Mapping integrieren	NFA4	Ein Context-Mapping ist in dem Entwurf vorzusehen, damit die Grenzen der BCs in einer Beziehung dargestellt werden.
Skalierbarkeit des Systems	NFA5	Für die Skalierbarkeit des Systems ist es bei dem Deployment der Microservices möglich eine variable Anzahl an Instanzen pro Service bereitzustellen.
Dauerhafte Erreichbarkeit	NFA6	Bei dem Deployment des neuen Systems ist sicherzustellen, dass das System dauerhaft erreichbar ist und die Zuverlässigkeit sichergestellt ist.
Einfache Adaptierbarkeit	NFA7	Das System ermöglicht eine schnelle Anpassbarkeit und einfache Erweiterbarkeit durch neue Services.
Persistente Datenhaltung	NFA8	Die Microservice-Architektur gewährleistet eine persistente und dauerhafte Datenspeicherung der Anwendungsdaten.

Tabelle 3.1: Anforderungen an das neue System

4. Entwurf der Anwendung für ein manuelles Deployment

Basierend auf den detaillierten Überblick zu der bestehenden Anwendung und den definierten funktionalen sowie den nicht-funktionalen Anforderungen wird im Folgenden der Entwurf der Microservice-Anwendung beschrieben, um die einzelnen Schritte des Softwareentwicklungsprozess nach dem Wasserfallmodell weiterzuverfolgen [Royce, 1987, S. 329].
Als Erstes wird die Anwendungsarchitektur beschrieben und anschließend detailliert das Design der Services für die Applikationslogik mit den entsprechenden Datenspeichern dargestellt. Für einen vollständigen Entwurf werden anschließend die Graphical User Interfaces (GUIs) dargestellt und schließlich in Abschnitt 4.4 eine Validierung des Entwurfes gegen die Anforderungen, die in dem Kapitel 3 definiert sind, durchgeführt.

4.1. Modellierung der Anwendungsarchitektur

Für den Entwurf ist es zunächst wichtig, die Architektur für die Microservice-Anwendung „Scheduler-Services“ festzulegen. Die Komponenten des Entwurfs sind in der Abbildung 4.1 mit den benötigten Verbindung für die Kommunikation dargestellt. Das Architekturmodell beinhaltet zum einen vier Spring-Boot Services und zwei Angular Anwendung für die Darstellung der Applikationslogik und des Frontends. Des Weiteren wird für das Identitäts- und Zugriffsmanagement die open-source Lösung Keycloak verwendet, da die Verwaltung der Benutzeridentität ein besonders wichtiges Feature für Microservice-Anwendungen ist und durch Keycloak mit der weit verbreiteten OpenID Connect Technologie, was eine Erweiterung von OAuth2.0 ist, in dem Entwurf umgesetzt wird [Christie et al., 2017]. Für die Datenspeicherung sind die drei Datenbankservices verantwortlich, auf die jeweils nur ein Service zugreift.
Aus dieser Aufteilung sind die fachlichen Komponenten User-Management, Course-Task-Verwaltung und Userinterface nach dem DDD abzuleiten, da diese Komponenten jeweils eine zusammenhängende Fachlogik beschreiben, die in der Abbildung 4.1 mit Strichlinien abgegrenzt sind [Evans, 2015]. Die abgegrenzte Fachlogik wird durch BC beschrieben, da das User-Management nur dafür verantwortlich ist, die Userinformationen abzuspeichern, zu überprüfen und Details zu den Usern zurückzugeben. Daneben dient die Course-Task-Verwaltung zur Darstellung der Courses mit den dazugehörigen Tasks, Courseersteller und Abonnenten, was von dem User-Management abgegrenzt ist. Des Weiteren ist die Userinterface-Domäne ausschließlich dafür zuständig, die benötigten Daten für einen Anwender darzustellen, dass abstrahiert dargestellt wird, da dazu nur Informationen aus dem `Spring-Cloud-Gateway` benötigt werden. Hierdurch wird die Architektur nach Evans, 2015 in einzelne Domänen untergliedert, wobei die Kommunikation über die BC hinweg durch Context-Mapping beschrieben wird, das in der Abbildung 4.2 grafisch dargestellt ist. Durch dieses Domänenmodell wird verdeutlicht, dass die Services lose voneinander gekoppelt sind, aber die Kohäsion der Anwendungslogik durch das DDD beschrieben wird.

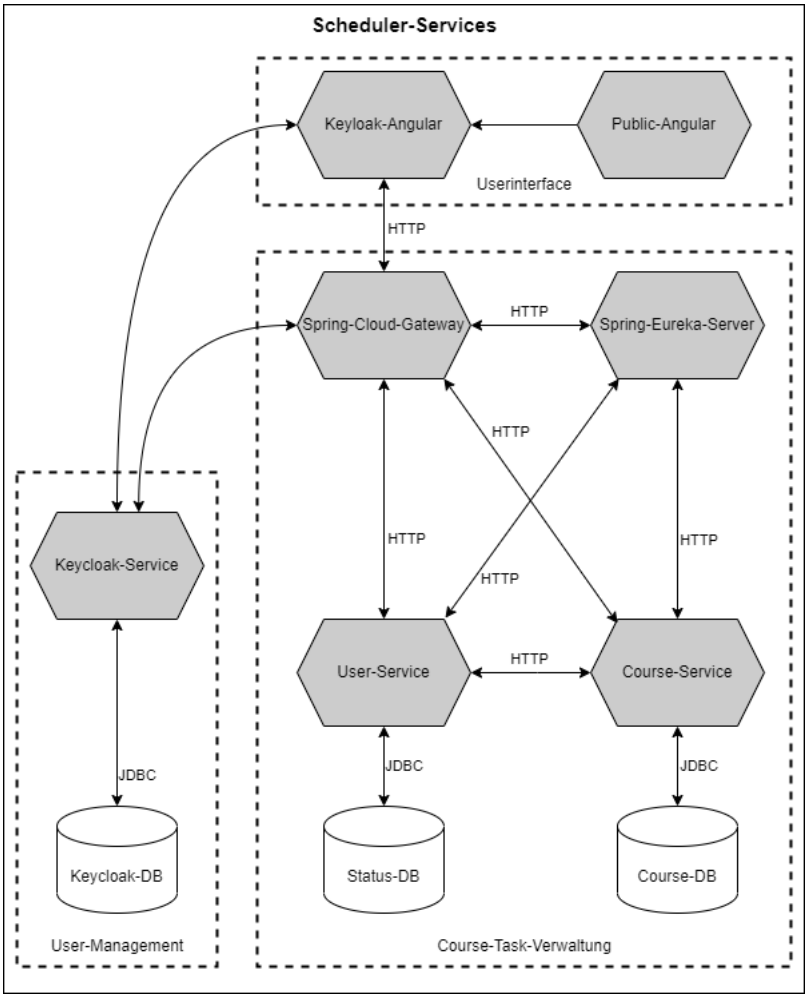

Abbildung 4.1: Microservice Architekturmodell

Nachdem ersten Überblick zu den Domänen der neuen Architektur wird aufgezeigt, wie ein großer Teil der Applikationslogik durch mehrere horizontale Spring-Services abgebildet wird. Da eine wichtige Komponente einer Microservice-Architektur eine Service-Registry ist, wird ein `Spring-Eureka-Server` als Discovery-Server in dieser Architektur eingebunden. Alle Spring-Eureka-Clients registrieren sich in diesem Service, damit alle Instanzen eines Services mit einem logischen Namen abgespeichert werden, sodass diese auch von anderen Clients gefunden werden [Janser, 2015]. Dadurch stellt der `Spring-Eureka-Server` ein Verbindungselement für die drei anderen Spring-Services in der Architektur dar.
Durch dieses Design ist es möglich ein `Spring-Cloud-Gateway` hinzuzufügen, das in der Fallstudie die Aufgabe hat, die API-Requests entweder an den `User-Service` oder an den `Course-Service` weiterzuleiten, damit die Anfragen von dem entsprechenden Services verarbeitet werden und durch das Gateway an den Initiator zurückgegeben werden. Die Verwendung eines Gateways für die Spring-Services bietet zudem die Option, die beiden

Services in der Architektur von außen nicht sichtbar zu machen, da alle Anfragen für die Services über das Gateway geleitet werden und somit der `Course-Service` und der `User-Service` nicht direkt von außerhalb des Kubernetes-Clusters erreichbar sind. Dadurch sind Information zu den Services versteckt, was dazu beiträgt, Änderungen sicherer vorzunehmen, da nur die Endpunkte und keine internen Prozesse der Anwendung bekannt sind [Newman, 2019, S. 18]. Somit ist es möglich einzelne Komponenten zu ändern oder hinzuzufügen, ohne dass dies außerhalb des Services festzustellen ist. Die beiden versteckten Services bilden zudem den Kern der Applikationslogik und führen durch die Verwendung des Eureka-Servers eine einfache Kommunikation über HTTP-REST-API-Requests untereinander durch. Diese benötigte Kommunikation der beiden Services ist zusammen mit den Schnittstellen aller Services der Applikationslogik in Form eines Unified Modeling Language (UML) Komponentendiagramms in der Abbildung 4.2 aufgezeigt, in der die Kommunikation über die BCs hinweg dargestellt ist [Booch et al., 1999, S. 126]. In diesem UML Diagramm wird verdeutlicht, dass der `User-Service` die beiden essenzielle Schnittstellen `User-Controller` und `Status-Controller` anbietet, die für die Logik der Courses und Tasks in dem `Course-Service` benötigt wird. Damit diese beiden Controller von dem `User-Service` angeboten werden, muss dieser Userdetails, wie Nachname, Vorname, Username, E-Mail-Adresse sowie die zugewiesenen Rollen pro User bereitstellen. Außerdem ist ersichtlich, dass das `Spring-Cloud-Gateway` eine Verbindung zu allen Controllern der beiden Services benötigt, damit die eingehenden Requests aus dem Frontend entsprechend dem definierten Mapping weitergeleitet werden.

In der Architektur der neuen Anwendung ist der `Keycloak-Service` innerhalb der User-Management Domäne für ein übergreifendes Identätis- und Zugriffsmanagement zuständig, um eine Implementierung der Sicherheitsmechanismen in jeden einzelnen Service zu vermeiden. Keycloak bietet die Möglichkeit ein Single-Sign-On für alle verwendeten Services, wie beispielsweise die vier Spring-Boot Services und der Angular Anwendungen, mit geringem Aufwand zu implementieren und ermöglicht außerdem die Verwendung einer vorgefertigten Login-Form sowie die Speicherung der generierten Userdaten in einer Datenbank [Keycloak, 2021c]. Des Weiteren vereinfacht Keycloak die Überprüfung von Zugriffsrechten auf spezielle Ressourcen, da verschieden Clients für mehrere Services in Keycloak angelegt werden und es möglich ist jedem User speziell definiert Rollen zuzuweisen.
Die korrekte Authentifikation wird hierbei durch das verwendete OAuth2.0 Protokoll gewährleistet, bei dem die Überprüfung der Identität und der Zugriffsberechtigungen über einen Token in dem JSON Web Token (JWT) Format funktioniert. Der Vorteil an der Verwendung von JWT-Token ist, dass diese nach erfolgreicher Registrierung eines Users alle relevanten Informationen über diesen beinhalten. So kann über das dekodieren der Token, die bei jedem HTTP-REST-API-Request in den `HttpHeaders` mitgesendet werden, Informationen zu dem User erhalten werden [Garvie, 2021]. Das umfasst Userrollen, Name des Users und Accountstatus, was die Implementierung von vielen Zugriffsmechanismen vereinfacht, das in dem Kapitel 5 mithilfe der JWT-Library genauer verdeutlicht wird [Auth0, 2021]. Daneben ist diese Lösung in den meisten gängigen Softwaretools direkt zu integrieren, was auch bei den verwendeten Tools der Fall ist, da die Keycloak-

Konfiguration in Angular durch einen `APP_INITIALIZER` hinzugefügt wird und in Spring-Boot in dem Gateway definiert werden muss, sodass nur das `Spring-Cloud-Gateway` von der Zugriffskonfiguration direkt betroffen ist, aber alle anderen Spring-Services direkt mit gesichert werden, da diese hinter dem Gateway versteckt sind. Dies ist auch in dem Architekturmodell dargestellt, da für den `Keycloak-Service` nur eine Verbindung zu dem Gateway und der `Keycloak-Angular` Anwendung modelliert ist.
Wie bereits beschrieben werden zwei Angular Anwendungen in der Frontendservice Domäne verwendet, wobei nur der `Keycloak-Angular` Service auf die Keycloak-Anwendung und auf die Spring-Services zugreifen muss, um alle Anforderungen der Anwendung zu erfüllen. Der Entwurf der beiden Angular Anwendungen wird im Detail in dem Abschnitt 4.3 beschrieben.

Durch das Architekturdesign ist es möglich, die aufgezeigten Services in einem Kuberntes-Cluster zu deployen und dabei nur insgesamt vier Services zu veröffentlichen und so die anderen Services nur innerhalb des Clusters erreichbar zu machen.

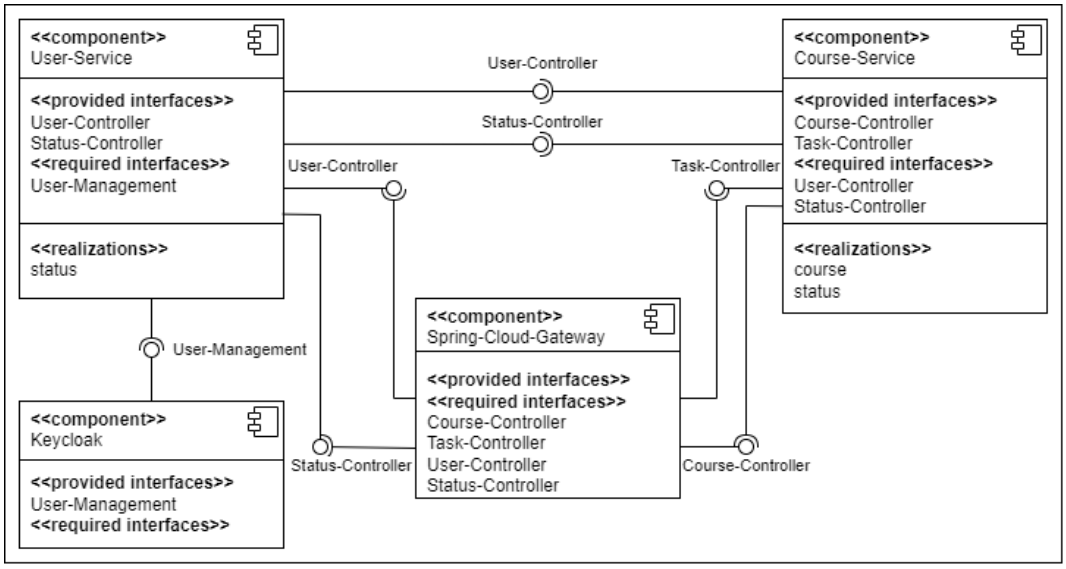

Abbildung 4.2: Domänenmodell der Applikationslogik als UML Komponentendiagramm

4.2. Anpassung des Datenmodells

In diesem Kapitel wird beschreiben, wie durch den Entwurf sichergestellt wird, dass ein Datenbankservice nur von einem Service verwendet wird [Newman, 2019, S. 5]. Dafür ist es nötig, das bereits existierende ER-Modell aus der Abbildung 3.2 aufzuteilen, was eine der größten Herausforderungen bei einer Migration darstellt. Diese Herausforderung ergibt sich, da trotz der Aufteilung des Datenmodells bei der Datenspeicherung das AKID-Prinzip sicherzustellen ist, wobei vor allem auf eine konsistente Speicherung der Daten in allen Datenbankservices zu achten ist. Dies stellt eine sehr komplexe Aufgabe dar und wird in vielen Migrationsprojekten zu einem Showstopper, da der Aufwand

für den Entwurf und die Umsetzung oft in der neuen Architektur unterschätzt wird. Deshalb wird im folgenden genau darauf eingegangen, wie das Datenmodell der bestehenden Anwendung, unter der Verwendung von den Modellen nach Newman, 2019, auf die verschiedenen Datenbankservices aus der Abbildung 4.1 aufzuteilen sind.

4.2.1. Aufteilung der Entitäten

Bevor das Datenmodell aufgeteilt wird, muss ein komplettes Verständnis das bereits bestehenden Modells sichergestellt sein. Dazu ist zunächst das bestehende ER-Modell zu analysieren, was in dem Kapitel 3.1.2 dargelegt ist, um einzuschätzen, welche Teile des Modells in einen einzelnen Datenbankservice zu gliedern sind.
Aus dieser Analyse wird abgeleitet, dass die Daten für Courses und Tasks in einem Service abzuspeichern sind, da jeder Task zu einem Course gehört und diese beiden Entitäten eng miteinander verbunden sind [Newman, 2019, S. 17]. Diese gemeinsame Darstellung ist auch in der Abbildung 4.3 ersichtlich, in der die einzelnen Datenbankschemata pro Service für das neue System abgebildet sind. Aus dem zuvor dargestellten ER-Modell wird auch die Notwendigkeit der Abspaltung der User-Entität ersichtlich, da diese für verschiedene Bereiche der Anwendung benötigt wird und so die entsprechenden Userdaten durch Keycloak in einer einzelnen Datenbank abgespeichert werden, auf die nur die Keycloak-Anwendung direkt zugreift. Außerdem wird die verbleibende Status-Entität in der `Status-DB` hinzugefügt, sodass alle aufgezeigten Entitäten aus dem ursprünglichen ER-Modell zu einem Datenbankservice zugewiesen sind. Wichtig hierbei ist festzulegen nach welchen Pattern, die in dem Kapitel 2.2 dargestellt sind, die Datenbank aufgeteilt wird. In dieser Fallstudie wird aufgrund der bisherigen Analyse, das Pattern „Move Foreign-Key Relationship to Code“ nach Newman, 2019 verwendet, da diese für eine komplette Microservice-Architektur geeignet ist. Des Weiteren sind in dem bisherigen ER-Modell die meisten Beziehungen N:M-Beziehungen, die durch dieses Pattern auf die verschiedenen Datenbankservices aufgeteilt werden.

4.2.2. Modellierung einer verteilten Datenspeicherung

In dem vorherigen Abschnitt ist die Aufteilung der Entitäten anhand des „Move Foreign-Key Relationship to Code“ Pattern definiert, dessen Anwendung auf das vorliegende Datenmodell in diesem Abschnitt dargestellt wird.
Interessant ist hierbei die Zuweisung der Status-Entität genauer zu betrachten. Neben der Hinzufügung in einer extra Datenbank wäre auch eine andere Varianten anwendbar, da es möglich wäre diese Entität beispielsweise auch in der `Course-DB` zu berücksichtigten. Dies würde allerdings die Kernidee von Microservices nicht beachten, weil dadurch ein Service für mehrere unabhängige Funktionalitäten gleichzeitig zuständig wäre [Richardson, 2020]. Aus diesem Grund ist der Status zu der `Status-DB` hinzugefügt, da ein Status immer direkt einem User zugewiesen ist und dadurch der Service, der auf diese Datenbank zugreift, die Generierung von Userdetails realisieren muss. Diese Entscheidung wird weiter durch ein Beispiel verdeutlicht, was die einfache Erweiterbarkeit der Microservice-

Architektur betrifft [Richardson, 2020]. Angenommen der Status in der Anwendung sollte durch das Feature erweitert werden, dass der entsprechende Erstellter des Tasks, also der Ersteller von dem jeweiligen Course, die Möglichkeit hat eine Bewertung für einen abgeschlossenen Status pro User zu hinterlegen. In diesem Fall müssten Änderungen an dem `Course-Service` und der dazugehörigen `Course-DB` vorgenommen werden. Dadurch ist es nicht möglich neue Status-Features unabhängig von den Tasks und Courses hinzuzufügen, wenn diese zu einer Datenbank und einem Service gehören, was darauf deutet, dass der Entwurf sich in die Richtung von einem Monolithen entwickelt [Richardson, 2020]. Wenn der Status in einer zusätzlichen Datenbank ausgegliedert wird, wie in der Abbildung 4.3 dargestellt, ist es möglich ein Status-Feature hinzuzufügen ohne den `Course-Service` und die verknüpfte Datenbank abzuändern, da die Kohäsion zwischen `Course-Service` und Status-Entität in zwei BCs aufgeteilt ist.
Da ein Status Informationen aus der `Keycloak-DB` und der `Course-DB` für eine Speicherung der Statusinformationen benötigt, enthält die `Status-DB` nach dem Modell „Move Foreign-Key Relationship to Code“ jeweils einen Fremdschlüssel des jeweiligen Primärschlüssels aus der User-Tabelle und der Task-Tabelle [Newman, 2019, S. 173]. Ein ähnliches Vorgehen wird auch bei der Course-Tabelle angewendet, da bei jedem Course ein entsprechender User als Erstellter hinzugefügt werden muss. Zusätzlich muss zu jedem Course eine Liste an Usern, die den Course abonniert haben, abgespeichert werden, damit hinterlegt ist, welche User die entsprechenden Status pro Task im Course zugewiesen bekommen. Dies ist für die Course-Ersteller in der `Course-DB` wiederum über einen Fremdschlüssel der User-ID aus der `Keycloak-DB` modelliert und durch die Hinzufügung einer Liste an User-IDs werden alle Abonnenten eines Courses abgespeichert. Durch diese Modellierung des aufgeteilten Datenbankschemata in der Abbildung 4.3 werden die Verknüpfungen der Tabellen durch Fremdschlüssel über die Grenzen der definierten BCs hinweg durch die Verwendung von dem Schema „Move Foreign-Key Relationship to Code“ von Newman, 2019, S. 170 in dieser Fallstudie dargestellt, was zu einer persistenten Datenspeicherung durch die entsprechende Implementierung des Patterns führt.

Ein Nachteil der Anwendung dieses Schemas ist die Erhöhung der Latenz in dem System, da mehrere Datenbanktransaktionen für eine Aktion benötigt werden, die davor in der Ausgangsanwendung durch einen einzigen Structured Query Language (SQL) Befehl ausgeführt wurde [Newman, 2019]. Ein Beispiel für die zusätzlichen Aktionen ist die benötigten Änderungen in den Datenbanken durchzuführen, wenn ein User einen Course abonniert. Denn hierbei wird in der neuen Architektur zunächst der User in der `Course-DB` zu der entsprechenden Abonnentenliste hinzugefügt und anschließend muss für jeden Task des neu abonnierten Courses ein Status in der `Status-DB` angelegt werden, was in der neuen Architektur zu einer Steigerung der Datenbanktransaktionen führt. Allerdings ist dieser Anstieg in der Latenz hinzunehmen, da dies nur geringfügige Transaktionen betrifft, die nur kleine Datenmengen beinhalten. Trotzdem entsteht hierbei ein etwas größerer Entwicklungsaufwand, da die Hinzufügung und das entsprechende Entfernen des Fremdschlüssel nicht mehr in der Datenbank sonder direkt im Code umgesetzt werden muss, was die Bezeichnung des angewendeten Pattern bereits beschreibt.

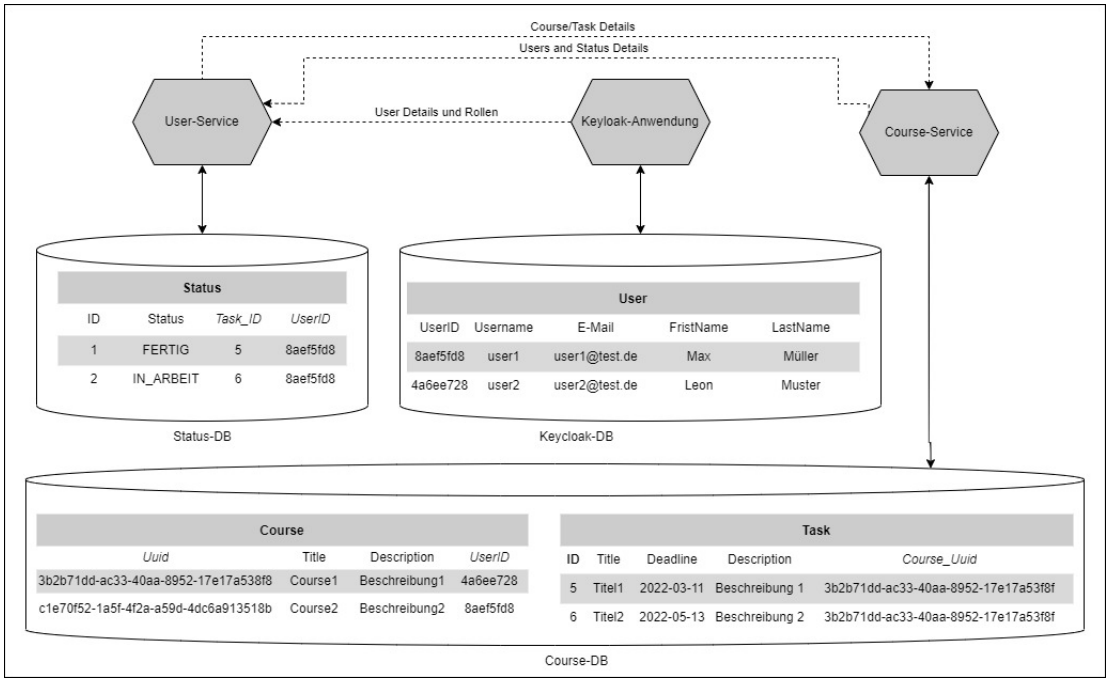

Abbildung 4.3: Datenbank-Schemata der Microservices

Viel wichtiger ist bei diesem Schema für die Aufteilung der Datenbank die Konsistenz über alle Datenbankservices durch die Aktualisierung der Fremdschlüssel in den verschieden Services bei einer Datenänderung zu gewährleisten. Nach Newman, 2019, S. 176 gibt es bei dem Modell „Move Foreign-Key Relationship to Code“ verschiedene Optionen, um die Konsistenz bei einer verteilten Speicherung sicherzustellen, wovon zwei Optionen in diesem Entwurf anzuwenden sind. Eine Option davon wird für die gespeicherten User in der `Keycloak-DB` angewendet, damit die UserID in den anderen Datenbankservices korrekt referenziert wird. Somit müssten alle Datenbanken aktualisiert werden, wenn ein User aus dem System entfernt wird. Um diesen Aufwand zu umgehen wird festgelegt, dass User nicht komplett aus dem Keycloak-User-Management entfernt werden, sondern der Zugriff für diese User deaktiviert wird. Durch diese Verfahren ist sichergestellt, dass die UserIDs korrekt abgespeichert sind, da ein User nicht gelöscht wird, aber dennoch der Zugriff auf die Anwendung von einem Anwender durch eine Deaktivierung in Keycloak zu entfernen ist [Newman, 2019, S. 177].
Etwas komplexer ist die Sicherstellung der konsistenten Speicherung von den Status mit der entsprechenden Task-ID pro User, da es möglich ist, dass der AF9 oder der AF11 durchgeführt wird, was zu einem Update in der Status-Tabelle führt. Deshalb wird bei der Ausführung von AF11 durch den Course-Ersteller zunächst überprüft, welche Status eine Referenz zu diesem Task in der `Status-DB` haben und anschließend die entsprechenden Einträge gelöscht, bevor der eigentliche Task in der `Course-DB` gelöscht wird. Durch diese Überprüfung vor dem Löschen wird sichergestellt, dass der zu löschende Task nach der Löschung in einem anderen Datenbankservice nicht mehr referenziert wird, wodurch

eine konsistente Speicherung durch das Modell „Check before deletion“[Newman, 2019, S. 176] sichergestellt ist. Das gleiche Verfahren wird für AF9 angewendet, da die erstellten Status für den User von dem Course aus der Status-Tabelle gelöscht werden müssen, da der User nach dem deabonnieren keine Tasks und somit auch keinen Status zu dem Course in der Datenbank haben darf.
Durch die Status-Entität wird verdeutlicht, dass die Aufteilung eines Datenmodells in einigen Fällen nicht trivial ist und zusätzliche Überprüfungen oder Verfahren hinzugefügt werden müssen, damit Daten trotz der Aufteilung nach dem AKID-Prinzip abgespeichert werden. Außerdem sollte vor der Implementierung evaluiert werden, welche Methoden in einem System angewendet werden, da diese verschieden Vor- und Nachteile mit sich bringen, die in dem Kapitel 2.2 aufgezeigt sind.

Aus der Anpassung des Datenmodells für diese Fallstudie wird ersichtlich, dass selbst ein relative kompaktes ER-Modell mit vier Entitäten zu einigen Schwierigkeiten und komplexen Herausforderungen in dem Entwurf für ein geteiltes Datenmodell pro Service führt. Deshalb sollte in diesem Schritt der Migration großes Augenmerk darauf gelegt werden, den Entwurf vor allem für das neue Datenmodell im Detail genau zu definieren, damit in der Implementierung größere Probleme vermieden werden. Aus diesem Grund ist eine beliebte Herangehensweise zuerst die Datenbank zu teilen, bevor die Applikation in einzelne Services unterteilt wird [Newman, 2019, S. 161]. Durch die beschriebene Anpassung des Datenmodells und der Architektur ist es nun möglich, die benötigten Services für die Applikationslogik zu implementieren und diese anschließend in Kubernetes zu deployen.

4.3. Entwurf der GUI

In diesem Abschnitt wird der Entwurf der Microservice-Anwendung durch die Beschreibung des GUI für die Darstellung der Benutzeroberfläche vervollständigt, nachdem in den vorherigen Abschnitten die Applikationslogik und die Datenspeicherung beschrieben wurde.
Wie in der Abbildung 4.1 zu sehen ist, sind in dem Entwurf zwei Angular-Anwendung für das GUI vorgesehen, die das Frontend beziehungsweise die Präsentationsschicht der Anwendung darstellen [Horn, 2007b]. Die Entscheidung zu der Modellierung des GUI durch zwei Frontend-Anwendung ist auf Basis der funktionalen Anforderungen in 3.1.1 gefallen, da eine Anforderung in diesem Kapitel beschreibt, dass öffentlich zugängliche Informationen, wie Dokumentationen und Anwendungsinformationen, benötigt werden und Seiten beziehungsweise Features, die nur nach einer Authentifikation aufrufbar sind. Aus diesem Grund wird die `Public-Angular` Anwendung definiert, um alle öffentlich zugänglichen Informationen ohne Authentifikation für die Nutzer darzustellen. Ergänzend dazu enthält die `Keyloak-Angular` Anwendung alle Features, für die eine Anmeldung eines Anwenders benötigt wird. Durch die Verlinkung der beiden Frontendservices untereinander, ist trotz der Verwendung von zwei Anwendungen ein vollständiges GUI dargestellt.

Damit der AF3 berücksichtigt wird ist eine Verbindung des `Keyloak-Angular` Services mit der Keyloak-Anwendung nötig. Somit wird im Gegensatz zu dem `Public-Angular` Service bei der `Keyloak-Angular` Anwendung durch die Verbindung mit Keycloak ein `AuthGuard` hinzugefügt, der gewährleistet, dass bei Aufruf eines Pfades der geschützten Angular-Anwendung der User zunächst auf die Login-Seite von Keycloak für eine Authentifikation mit Usernamen und Passwort weitergeleitet wird, insofern keine aktive Session für den User durch Keycloak identifiziert wird.
Für die komplette Abbildung der beiden Frontend-Anwendungen werden Daten benötigt, die in den beiden Anwendung durch verschiedene Vorgehensweisen generiert werden. Da die `Public-Angular` Anwendung ausschließlich generische Informationen darstellt, werden diese direkt in der Anwendung definiert und müssen so nicht von anderen Services dynamisch zur Verfügung gestellt werden. Deshalb benötigt diese Anwendung nur eine Verbindung zu dem `Keyloak-Angular` Service, damit der Anwender bei einem entsprechenden Aufruf auf die geschützte Angular-Anwendung weitergeleitet wird. Innerhalb des `Keyloak-Angular` Services werden die verschiedene Funktionalitäten zu den Courses, Tasks und Status dargestellt, weshalb diese Anwendung neben der Verbindung zu Keycloak eine Verbindung durch das Gateway zu den Services der Applikationslogik innerhalb der Course-Task-Verwaltungs Domäne benötigt. Diese Kommunikation wird für die Durchführung der Usereingaben benötigt, die über die BCs hinweg durch HTTP-API-Requests von dem `Keyloak-Angular` Service auf die Funktionen des `Spring-Coud-Gateway` zugreifen, was in der Abbildung 4.1 abgebildet ist. Dadurch benötigte die `Keyloak-Angular` Anwendung zur Darstellung aller geschützten Funktionalitäten nicht nur die Verbindung zu Keycloak für die Authentifikation, sondern auch Zugriff auf alle Interfaces, die bei das `Spring-Cloud-Gateway` in dem UML Komponentendiagramm in der Abbildung 4.2 von den anderen Services der Applikationslogik zusammenführt, sodass alle benötigten Funktionen in dem gesicherten Frontend-Service für den User abgebildet werden. Allerdings ist bei diesem Design zu berücksichtigen, dass die Datengenerierung durch Endpunkte der Spring-Boot Services zu kleinen Verzögerungen aufgrund der Dauer der API-Requests führen, was darin resultiert, dass beispielsweise bei Löschung eines Tasks oder Courses das Userinterface nicht direkt die aktualisierten Daten zeigt, sondern erst nach einer Aktualisierung der Seite.
Zusammenfassend ist festzustellen, dass ein Angular-Service dazu in der Lage sein muss HTTP-Requests an das Gateway zu senden und zudem eine Verbindung zu der Keycloak-Anwendung aufzubauen, da sonst eine der wichtigsten Anforderungen für die Autorisierung von Usern nach dem AF3 in der Anwendung nicht gewährleistet ist. Außerdem ist ein zweiter Angular-Service in dem Design vorgesehen, um die öffentlichen Informationen abstrahiert darzustellen.

4.4. Validierung des Entwurfs

Abschließend zu dem Entwurf innerhalb des Wasserfallmodells nach Royce, 1987, S. 329 wird eine Validierung des beschriebenen Entwurfs gegen die definierten Anforderungen in der Tabelle 3.1 durchgeführt. Durch diese Entwurfsprüfung wird die Qualität der Software sichergestellt [Benra, 2009, S. 178].

Die Validierung des Entwurfs anhand der zuvor definierten Anforderungen ist in der Tabelle 4.1 mit allen IDs der funktionalen sowie nicht-funktionalen Anforderungen dargelegt, in der für jede Anforderung spezifiziert ist, wie diese Anforderung in dem Entwurf berücksichtigt ist. Im Folgenden wird die Umsetzung besonders komplexer oder wichtiger Anwendungsfälle aus der Tabelle 4.1 genauer beschrieben.
Zuerst werden die funktionalen Anwendungsfälle des Entwurf auf die Umsetzung geprüft. Ein wichtiger Anwendungsfall ist der AF3, der durch den `Keycloak-Service` in dem Entwurf berücksichtigt ist, da Keycloak die Möglichkeit bietet eine vorgefertigte Login-Seite zu nutzten, in der sich ein User mit dem Usernamen und Passwort anmeldet und nach der Anmeldung die Entität des Users mit einem Token weiter durch Keycloak geprüft wird. Für die Unterscheidung von Coursersteller und Abonnent wird zu jedem Course die User-ID des Erstellers abgespeichert und eine Liste an User-IDs der Abonnenten hinterlegt. Durch diese Speicherung, die in der Abbildung 4.3 dargestellt ist, wird durch den Entwurf realisiert, dass es nur einem `Coursersteller` oder einem `Admin` möglich ist einen Course zu löschen und Tasks zu dem Course hinzufügen oder zu löschen, dass AF6, AF10 und AF11 betrifft. Für den AF7, ist in der Abbildung 4.2 das Interface `User-Controller` definiert, damit der `Course-Service` aus den gespeicherten User-IDs die Userdetails erhält, was auch in dem Datenbank-Schemata in der Grafik 4.3 berücksichtigt ist. Diese Schnittstelle wird von dem `Course-Controller` auch für AF8 benötigt, damit für den User pro Task des Courses ein Status erstellt wird, sodass jedem Status genau ein Task und einem User zugeordnet ist, was auch für die Anwendungsfälle AF9, AF10 und AF11 gewährleistet wird. Durch diese Beschreibung in Verbindung mit der Begründung für die Umsetzung der funktionalen Anforderungen in der Tabelle 4.1 ist deutlich, dass in dem beschriebenen Entwurf alle funktionalen Anforderungen berücksichtigt sind.

Als Nächstes wird für die Validierung die Berücksichtigung der nicht-funktionalen Anforderungen detailliert dargestellt. Zuerst ist festzuhalten, dass die lose Kopplung der Services aus der NFA1 und die leichte Abänderbarkeit des Systems nach NFA7 erfüllt sind, da durch die Microservice-Architektur in der Abbildung 4.1 die Option besteht Services auszutauschen oder neue Micorservices hinzuzufügen, ohne eine Abänderung aller Services zu implizieren. Nach dem DDD von Evans, 2015 ist die Domäne User-Management durch den `Keycloak-Service` dargestellt, die Domain Userinterface aus den beiden Angular-Anwendungen aufgebaut und die Course-Task-Verwaltung durch die Spring-Boot Services abgebildet, wodurch die NFA2 in dem Entwurf berücksichtigt ist. Die Domains sind durch BC abgegrenzt, damit die Geschäftslogik pro Fachlichkeit aufgeteilt ist, was zusätzlich in der Abbildung 4.1 grafisch gekennzeichnet ist. Bei der Umsetzung von NF3 in dem Entwurf, ist zu argumentieren, dass der `User-Service` über die BCs zwischen Course-Task-Verwaltung und User-Management hinweg agiert. Allerdings ist dieser Service in dem Entwurf klar dem BC Course-Task-Verwaltung zugeordnet, da dieser hauptsächlich für die Verarbeitung von Status-Features zuständig ist und nur zusätzlich einen User-Controller anbietet, der lediglich Userinformationen aus dem JWT-Token des aktuellen Users anbietet. Diese zusätzliche Funktionalität wurde zu diesem Service hinzugefügt, da die Userinformationen meist im Zusammenhang zu

einem Status benötigt werden, wodurch der Service als Status-Service zu interpretieren ist. Die Anforderungen mit den IDs NFA5 und NFA6 sind durch die Architektur der lose gekoppelten Microservices erfüllt, da die Architektur die Möglichkeit bietet in einem Kubernetes-Cluster deployed zu werden, was in dem Abschnitt 5.2 dargestellt wird. Essenziell ist die Umsetzung des NF8 für die Migration, der durch die definierten Datenbankservices in der Abbildung 4.1 in Verbindung mit Volumes und PVCs eine Datenspeicherung über die Laufzeit von einem Cluster oder einem Docker-Container hinaus gewährleistet und die Konsistenz dieser Daten durch die verwendeten Pattern sichergestellt ist, was in der Abbildung 4.3 festgehalten ist.

Durch diese Erläuterung der Umsetzung der einzelnen Anwendungsfälle ist zusammenfassend festzustellen, dass innerhalb des Entwurfs alle definierten Anforderung aus der Tabelle 3.1 modelliert sind und somit der nächste Schritt des Wasserfallmodells in dem folgenden Kapitel durchführbar ist.

ID	Validierung	Begründung der Berücksichtigung im Entwurf
AF1	✓	Der `Keycloak-Service` ist in der Architektur modelliert, damit durch diese open-source Lösung die Option besteht für Admins besteht User hinzuzufügen oder zu sperren.
AF2	✓	Admins ist es möglich durch API-Requests an das Gateway auf alle Courses, Tasks und Status zuzugreifen und diese zu adaptieren.
AF3	✓	Der `Keycloak-Service` ist für die Authentifikation der User durch Usernamen und Passwort oder durch generierte JWT-Token vorgesehen.
AF4	✓	Die Erstellung von Courses wird durch die Schnittstelle `Course-Controller` gewährleistet.
AF5	✓	Die Speicherung von Coursedetails wird von dem `Course-Service` gewährleistet und werden für die User über den `Course-Controller` ausgegeben.
AF6	✓	Bereits erstellte Courses sind durch den entsprechenden Endpunkt von dem `Course-Controller` zu löschen.
AF7	✓	Der `Course-Service` speichert alle User-IDs der Abonnenten und stellt diese über den `Course-Controller` zur Verfügung. Die Informationen zu den abgeschlossenen Tasks werden dabei über den `User-Service` generiert.
AF8	✓	Das Abonnieren eines Courses wird durch den `Course-Controller` in Verbindung mit dem `Status-Controller` durchgeführt.
AF9	✓	Das Deabonnieren eines Courses wird durch den `Course-Controller` in Verbindung mit dem `Status-Controller` durchgeführt.
AF10	✓	Das Erstellen von Tasks wird durch den `Task-Controller` in Verbindung mit dem `Stauts-Controller` ermöglicht.

AF11	✓	Das Löschen von Tasks wird durch den `Task-Controller` in Verbindung mit dem `Stauts-Controller` ermöglicht.
AF12	✓	Die Speicherung von Tasks wird von dem `Course-Service` gewährleistet und werden über den `Task-Controller` ausgegeben.
AF13	✓	Die Aktualisierung des Status zu einem Task wird durch den `Status-Controller` durchgeführt.
AF14	✓	Die Userdetails werden durch den JWT-Token in dem `User-Controller` generiert.
NFA1	✓	In dem Architekturmodell in der Abbildung 4.1 ist erkenntlich, dass die Business- und Anwendungslogik durch verschieden und lose gekoppelte Microservices abgebildet wird [García, 2020, S. 168].
NFA2	✓	In dem Entwurf ist erkenntlich, dass drei Domänen nach Evans, 2015 in dem Entwurf modelliert sind.
NFA3	✓	Die Organisationseinheiten sind durch BC abgegrenzt, was in dem Architekturmodell 4.1 grafisch dargestellt ist.
NFA4	✓	Das Context-Mapping ist in der Abbildung 4.2 ersichtlich, da durch dieses Modell die Beziehungen der BCs aufgezeigt sind.
NFA5	✓	Durch das Deplyoment der Architektur in Docker Compose oder Kubernetes, ist es möglich eine variable Anzahl an Instanzen pro Service zu generieren.
NFA6	✓	Eine dauerhafte Erreichbarkeit und die damit verbundene Zuverlässigkeit wird über das Deployment der Services in einem Kubernetes-Cluster ermöglicht.
NFA7	✓	Durch die lose gekoppelten Services in der Abbildung 4.1 ist eine einfache Adaptierbarkeit des System gewährleistet.
NFA8	✓	Die persistente Datenhaltung wird durch die Aufteilung des Datenmodells sichergestellt und die dauerhafte Datenhaltung wird durch Volumes und PVCs für die Datenbankservices ermöglicht.

Tabelle 4.1: Validierung der Anforderungen in dem Entwurf

5. Implementierung der Microservices

Nach dem Entwurf folgt in diesem Abschnitt die Beschreibung der Implementierung, um den nächsten Schritt des Wasserfallmodells durchzuführen.
In den beiden folgenden Abschnitten wird zunächst dargelegt, wie die Services der drei zuvor dargestellten Domänen für eine lokale Ausführung durch „Docker Compose“[Docker, 2021f] implementiert sind, um die Umsetzbarkeit des Entwurfes in einer Microserviceumgebung darzustellen. Anschließend werden in dem letzten Teil des Kapitels einige Services beispielhaft in dem Kubernetes der GCP deployed, um das Deployment ausgehenden von einer Docker Compose Datei hinzu Kubernetes aufzuzeigen und die persistente Datenspeicherung in der GCP durch Datenbankdienste zu betrachten.

5.1. Implementierung für eine Ausführung in Docker Compose

In diesem Abschnitt wird die Implementierung und die Umsetzung der Domänen User-Management, Course-Task-Verwaltung und Userinterface, die in der Abbildung 4.1 definiert sind, für eine Ausführung in Docker Compose mit einer persistenten Datenspeicherung durch Volumes für die definierten Datenbankservices dargestellt.

5.1.1. User-Management

Als erstes wird gezeigt, wie die User-Management Domäne durch das open-source Identitäts- und Zugriffsmanagementsystem Keycloak und einem Datenbankservice in Docker Compose umgesetzt ist. Die Definition der beiden Services ist in der Docker Compose YML Datei für das User-Management in dem Codebeispiel 5.1 abgebildet. Hier wird deutlich, dass für die persistente Speicherung der Daten aus dem User-Management eine MySQL-Datenbank mit dem offiziellen Docker Image `mysql:5.7` verwendet wird [Docker, 2021b]. Damit eine dauerhafte Speicherung der Daten aus der Keycloak-Anwendung sichergestellt ist, wird dem MySQL-Service das Volume `keycloak-data` hinzugefügt, sodass nach einem Stopp oder Löschung des Containers weiterhin die benötigten Daten in dem Docker-Volume vorhanden sind [Docker, 2021g]. Für die Verwendung des Volumes muss dieses auch in der YML Datei definiert werden, das durch die Hinzufügung von `volumes: keycloak-data` durchgeführt ist.
Damit die Verbindung zwischen dem `Keycloak-Service` und dem Datenbankservice aus der Abbildung 4.1 hergestellt wird, werden bei der Containerdefinition in der Compose Datei verschiedene Umgebungsvariablen durch den Schlüssel `environment` hinzugefügt [Docker, 2021f]. Bei dem MySQL-Container wird der Datenbankname, das Root-Password, der Nutzer und das Nutzerpassword festgelegt, wodurch der Keycloak-Anwendung ermöglicht wird mit diesen Referenzen direkt auf die Datenbank zuzugreifen. Bei der Containerdefinition der Keycloak-Anwendung werden anschließend die Werte dieser Variablen zur Datenbank in den Umgebungsvariablen hinzugefügt, damit der Keycloak-Service eine Verbindung zu der Datenbank aufbaut. Zusätzlich wird bei diesem Container noch ein Keycloak-Nutzer mit Usernamen und Password in den Umgebungsvariablen angegeben, sodass es nach Erstellung des Containers möglich ist, mit diesem Keycloak-Admin-Nutzer, die Keycloak-Anwendung über das Keycloak-Userinterface oder über API-Requests zu konfigurieren.

```yaml
version: '3'
services:

  keycloak-db:
      image: mysql:5.7
      container_name: keycloak-db
      environment:
        MYSQL_ROOT_PASSWORD: root
        MYSQL_DATABASE: keycloak
        MYSQL_USER: keycloak
        MYSQL_PASSWORD: password
      volumes:
        - keycloak-data:/var/lib/mysql

  keycloak:
      image: quay.io/keycloak/keycloak:latest
      container_name: keycloak
      environment:
        DB_VENDOR: MYSQL
        DB_ADDR: keycloak-db
        DB_DATABASE: keycloak
        DB_USER: keycloak
        DB_PASSWORD: password
        KEYCLOAK_USER: admin
        KEYCLOAK_PASSWORD: admin
      ports:
        - 8080:8080
      depends_on:
        - keycloak-db
```

Listing 5.1: User-Management Domäne in Docker Compose

Für die Konfiguration der Keycloak-Anwendung wird als erstes ein `realm` benötigt, das eine Gruppe von Anwendungen und Usern darstellt [Keycloak, 2021b]. In dieser Fallstudie wird ein `realm` mit der Bezeichnung `demo-realm` verwendet, in dem jeweils ein Client für die Spring-Boot Applikationen und für die geschützte Angular-Anwendung hinzugefügt ist. In dem Spring-Boot-Client mit der Bezeichnung `demo-spring-boot` sind zusätzlich zwei Client-Rollen definiert, sodass innerhalb der Spring-Boot Applikationen zwischen `ROLE_ADMIN` und `ROLE_USER` unterschieden wird, was den Akteuren aus dem Anwendungsfalldiagramm in 3.1 entspricht [Keycloak, 2021a]. Diese Client-Rollen werden den entsprechenden Usern in dem `Role-Mapping` der Keycloak-Anwendung zugewiesen. Zudem wird jedem User aus dem Standard-Client `realm-managment` die vordefinierte Rolle `view-users` zugewiesen, damit ein User die Berechtigung hat, die benötigten Userdetails über API-Requests an die Keycloak-Anwendung zu generieren. Keycloak bietet

die Möglichkeit die Konfiguration der Realms, der Clients und der Client-Rollen in eine JavaScript Object Notation (JSON) Datei zu exportieren, sodass die Konfiguration bei einem neuen Keycloak-System einfach durch einen Import der Datei durchzuführen ist. Durch die Verwendung dieser beiden Services ist der Entwurf der User-Management Domäne ohne weiteren Implementierungsaufwand in der Docker Compose Datei umgesetzt. Aus diesem Implementierungsschritt ist zu folgern, dass bei dem Entwurf der Architektur fremde Lösungen in betrachtet gezogen werden sollten, da durch die Verwendung von vorgefertigten Lösungen und Services die Möglichkeit besteht den Implementierungsaufwand zu reduzieren, wenn eine entsprechende Integration in dem Entwurf bedacht wird.

5.1.2. Course-Task-Verwaltung

Nach der Darstellung der Implementierung von der User-Management Domäne wird aufbauend auf diese beiden Services im Folgenden die Implementierung der Domäne Course-Task-Verwaltung mit den vier Spring-Services dargestellt.

Der erste Schritt der Implementierung der Domäne Course-Task-Verwaltung ist, den Eureka-Server für die Service-Registry nach Janser, 2015 zu implementieren. Für diese Implementierung wird zunächst ein neues Spring-Boot Projekt aus dem Spring-Initializer mit der Abhängigkeit `spring-cloud-starter-netflix-eureka-server` benötigt. Nach dem Download des Projektes wird, wie in dem Abschnitt 2.1.2 beschreiben, die Annotation `@EnableEurekaServer` in dem Projekt hinzugefügt und in der `application.yml` Datei der entsprechenden Server-Port und die Eureka Konfiguration hinzugefügt. Dadurch wird ermöglicht, dass alle Spring-Services in der Service-Registry mit einem Namen registriert werden, was in dem Ausschnitt in der Abbildung A.3 gezeigt ist. Danach wird das Spring-Projekt durch Maven in eine JAR-Datei konvertiert, um diese in einem Dockerfile zu referenzieren [Webb, 2020]. Dadurch besteht die Option mit dem Code aus dem Dockerfile in dem Listing 5.2 und dem Befehl `docker build .` aus dem Spring-Projekt ein Microservice in Form eines Docker-Images zu erstellen [Webb, 2020]. Dieses Vorgehen und der Code des Dockerfiles ist für alle Spring-Anwendung mit der Änderungen der Bezeichnung der JAR-Datei zu übernehmen.

```
1 FROM openjdk:11
2 ARG JAR_FILE=target/eureka-server-0.0.1-SNAPSHOT.jar
3 COPY ${JAR_FILE} service-registry.jar
4 ENTRYPOINT ["java","-jar","/service-registry.jar"]
5 EXPOSE 8761
```

Listing 5.2: Dockerfile für eine Spring-Boot-Anwendung

Unter anderem ist in dem Architekturmodell aus der Abbildung 4.1 ersichtlich, dass die Applikationslogik in zwei Spring-Services aufgeteilt ist. Aus diesem Grund ist es Notwendig ein Gateway zu implementieren, damit die Requests zu den entsprechenden Services geroutet werden. Zudem wird in dem `Spring-Cloud-Gateway` der Mechanismus für die Authentifikation von Usern übergreifend für alle Spring-Services mithilfe der Keycloak-Anwendung implementiert.

Das Routing des Gateways wird in den `application.properties` beziehungsweise in der `application.yml` Datei festgelegt. Das Routing für den `User-Service` und den `Course-Service` ist in dem Listing 5.3 aufgezeigt, das einen Ausschnitt aus dieser Datei des Gateway-Services zeigt. Aus diesem Listing ist zu erkennen, dass in dem Gateway zwei Routen unter dem Schlüssel `routes` definiert sind, was an der Anzahl der Schlüssel mit dem Wert `id` zu erkennen ist. Dadurch ist jedem der beiden Services eine Route in dem Gateway zugewiesen. Neben der `id` werden jeder Route unter den Schlüsseln `uri` und `predicates` weitere Attribute zugewiesen. Diese beiden Angaben innerhalb einer Route definieren an welchen Service ein bestimmter API-Request, der an das Gatway gesendet wird, geroutet wird. So werden anhand der Definition in dem Codebeispiel 5.3 alle Anfragen, deren Uniform Resource Locator (URL) nach der Domain mit „/api/course/“ beginnen an den Course-Service geroutet, der mit dem entsprechenden Namen aus der Service-Registry unter dem Wert der `uri` referenziert ist [Spring, 2021b]. Die gleiche Definition ist auch für den User-Service implementiert, bloß mit der Abänderung der `uri` und der Bezeichnung des Services aus dem Eureka-Service, der aus der Abbildung A.3 zu entnehmen ist.

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: course-service
          uri: lb://course-service
          predicates:
            - Path=/api/course/**
          filters:
            - RemoveRequestHeader=Cookie
        - id: user-service
          uri: lb://user-service
          predicates:
            - Path=/api/user/**
          filters:
            - RemoveRequestHeader=Cookie
```

Listing 5.3: Routingdefinition im Gateway-Service

Neben der Implementierung des Routings ist in dem `Spring-Cloud-Gateway` Service der AF3 durch die Verbindung zu Keycloak implementiert. Damit die Funktionalitäten von Keycloak in dem Gateway anwendbar sind, ist zunächst eine Security Konfigurationsklasse nötig, die durch die Annotation `@EnableWebFluxSecurity` gekennzeichnet ist. Durch diese Klasse wird innerhalb des Services definiert, dass die Authentifikation über das OAuth2.0 Protokoll mit JWT-Token durchgeführt wird [OAUTH, 2021]. Zusätzlich zu der Konfigurationsklasse wird in der `application.yml` Datei unter dem Schlüssel `keycloak-client` die Server-URL von der Keycloak-Anwendung und die Bezeichnung des verwendeten Realms hinzugefügt. Diese beiden Informationen werden auch für die Konfigu-

ration von Spring-Security unter dem Schlüsselwort `oauth2` mit dem Secret des verwendeten Clients benötigt [Alex et al., 2004, S. 327]. Diese drei Variablen werden nicht innerhalb der Spring-Anwendung festgelegt, sondern werden bei Start der Anwendung durch die Docker Compose Datei über die Umgebungsvariablen, die gegebenenfalls auf zuvor definierten Secrets basieren, hinzugefügt. Diese definierten Variablen werden in der Konfigurationsdatei bei dem `Spring-Cloud-Gateway` jeweils mit `${KEYCLOAK_SERVICE_ADDRESS}`, `${KEYCLOAK_REALM}` und `${KEYCLOAK_CLIENT_SECRET}` referenziert, damit die Variablen aus der Docker Compose Datei in den Applikationseigenschaften des Gateways dynamisch hinzugefügt werden [Baeldung, 2021b]. Durch die beschriebene Konfiguration des Gateway-Services ist die Implementierung des ersten Eureka-Clients, der durch die Annotation `@EnableEurekaClient` gekennzeichnet ist, abgeschlossen.

Als Nächstes wird die Implementierung des Eureka-Clients mit dem Namen `User-Service` für das Deployment in einer Docker Compose Datei detailliert dargestellt. Anders als die zuvor dargestellten Spring-Services bietet der `User-Service` eine REST-API für die Kommunikation mit anderen Services an und speichert zudem die benötigten Daten für die Status der User in der angebundenen Datenbank.
Damit die Daten der Anwendung in einem Datenbankservice abgespeichert werden, muss bei der Konfiguration für Docker Compose keine Änderung an den Dateien für die Anwendungseigenschaften vorgenommen werden, sondern nur die beiden Abhängigkeiten `postgresql` und `spring-boot-starter-data-jpa` zu dem Projekt hinzugefügt werden, da die Verbindung zu einem Postgres-Datenbankservice durch die Umgebungsvariablen in der Docker Compose Datei definiert wird [Silz, 2021]. Der Beispielcode 5.5 zeigt, welche Variablen benötigt werden, um einen Postgres-Datenbankservice mit der Anwendung zu verbinden. Der referenzierte und zuvor definierte Datenbankservice sorgt durch das offizielle Image von Postgres in Verbindung mit einem Volume für eine dauerhafte Speicherung der Status-Daten [Docker, 2021c]. Damit innerhalb des Posgres-Service, der in dem Beispiel durch den Containerport 5432 zu adressieren ist, in der definierten `compose-postgres` Datenbank die benötigte Tabelle für die Speicherung der Status mit den Attributen erstellt wird, ist eine Entitätenklasse in dem `User-Service` zu erstellen. Diese Entitätsklasse wird durch die Annotation `@Entity` definiert und enthält alle benötigten Attribute als Klassenvariablen [Gierkeand et al., 2021]. Dadurch wird keine aufwendige Definition des Datenbankschemas aus der Abbildung 4.3 durch SQL-Befehle benötigt, da das Schema durch die Verwendung von der JPA-Abhängigkeit automatisch erstellt wird. Zudem werden durch die Implementierung von JPA-Repositories für Entitätenklasse vereinfachte Datenabfragen ermöglicht, was die Implementierung der Logik deutlich vereinfacht [Gierkeand et al., 2021]. Resultierend aus den beschriebenen Konfigurationen ist es möglich in Docker direkt auf die Datenbank mit dem Host `localhost` und dem festgelegten Usernamen mit dem Befehl `psql -h localhost -p 5433 -U compose-postgres` zuzugreifen, wie es in der A.4 mit Beispieldaten gezeigt ist. Des Weiteren ist in der Abbildung zu erkennen, dass innerhalb des Datenbankservices die Datenbank mit der Bezeichnung `compose-potgres` automatisch erstellt wird und zudem in der definierten Datenbank die Tabelle für die Entitätstklasse `Status` durch die JPA-Abhängigkeit hinzugefügt wird. Nachdem der Verbindungsaufbau mit dem Datenbank-Container durch-

geführt ist, besteht die Option verschieden SQL-Befehle in dem Container auszuführen, um beispielsweise die Werte einer Relation zu betrachten. Dies ist auch in der Abbildung A.4 gezeigt, wobei anzumerken ist, dass das Attribut `task_id` den dazugehörigen Task aus der `Course-DB` nach dem Pattern „Move Foreign-Key Relationship to Code“ referenziert, dessen Implementierung bei der Beschreibung des `Courses-Services` dargestellt wird [Newman, 2019, S. 173].
Nachdem erläutert wurde, wie die Datenspeicherung der Status-Daten des `User-Services` implementiert ist, wird auf die Implementierung der beiden Controller-Klassen für den `User-Controller` und den `Status-Controller` aus dem Domänenmodell in der Abbildung 4.2 eingegangen. Die Controller-Klassen, die in Spring-Boot verschiedene REST-Schnittstellen definieren, sind mit der Annotation `@RestController` implementiert [VMware, 2021a]. Diese Implementierung ist Notwendig, damit jeder Controller HTTP-Requests für alle benötigten Funktionalitäten durch die Implementierung von `DELETE`, `POST` und `GET-Requests` unter Berücksichtigung der gewünschten Route pro Endpunkt anbietet [Dhalla, 2021, S. 1]. In dem `User-Controller` wurden ausschließlich REST-Endpunkt mit einem `@GetMapping` implementiert, da dieser Controller dazu dient aus den `HttpHeaders`, die bei jedem HTTP-Request mitgesendet werden, den JWT-Token zu extrahieren, damit aus diesem Token durch dekodieren die benötigten Userdetails generiert und zurück gegeben werden [Garvie, 2021]. Die Logik zu der Generierung der Details aus dem JWT-Token ist in dem `User-Service` nicht direkt in der Controller-Klasse implementiert, sondern in der Klasse `UserService` ausgegliedert. In dieser Spring-Boot Komponente sind unter anderem die benötigten Methoden für die Logik des `User-Controllers` implementiert. Hierbei ist die Methode aus dem Listing 5.4 besonders wichtig, da diese Methode den JWT -Token aus den `HttpHeaders` extrahiert und decodiert. Diese Methode wird am Anfang von allen implementierten Methoden für die Logik des `User-Controllers` verwendet, da nach der Dekodierung durch die Verwendung von `JSONObject` jeweils die benötigten Details aus dem dekodierten Token zu entnehmen sind [Oracle, 2015b]. Diese generierten Details werden dann anschließend durch eine `ResponseEntity` in der Controller-Klasse zurückgegeben, wodurch ermöglicht wird bei jeder API-Response zusätzlich zu den benötigten Daten einen HTTP-Status zurückzugeben. `ResponseEntities` werden bei allen Endpunkten der Spring-Services verwendeten, um eine einheitliche Form der Rückgaben zu gewährleisten, wodurch alle Endpunkte in den verschiedenen Services mit dem gleichen Vorgehen zu kontaktieren sind. Diese Implementierung des `User-Controllers` ist aufgrund der Verwendung des OAuth2.0 Protokolls durch die Keycloak-Anwendung und der damit verbundenen JWT-Token durchführbar. Hierdurch ist ein geringer Implementierungsaufwand für die Generierung der Userdetails entstanden, da es durch die JWT-Library möglich ist einen REST-Service für Userdetails in simpler Form zu erstellen.[Auth0, 2021].

```
1 private String getPayloadFromHeader(HttpHeaders headers) {
2     Base64.Decoder decoder = Base64.getDecoder();
3     String access_token = headers.getFirst("authorization");
4     String token = access_token.substring(7, access_token.length());
5     return new String(decoder.decode(token.split("\\.")[1]));
6 }
```

Listing 5.4: Extrahieren und decodieren eines JWT-Token

Damit der `User-Service` die eigentliche Kernaufgabe der Verwaltung und Speicherung von Status erfüllt, wurde in dem Service zusätzlich der `Status-Controller` hinzugefügt. In diesem Controller werden `@GetMappings`, `@PostMappings` sowohl als auch `@DeleteMappings` verwendet, damit es möglich ist Status zu erstellen, zu löschen, zu ändern und die gespeicherten Status auszugeben [VMware, 2021a]. Hierbei ist die Logik, wie bei dem `User-Controller`, in den Methoden der `UserService` Klasse implementiert. Wichtig ist bei dieser Controller-Klasse, dass nicht alle REST-Endpunkte von einem User angesteuert werden dürfen, da beispielsweise nur ein User mit der Client-Rolle `ROLE_ADMIN` die Status von jedem einzelnen User einsehen darf. Um dies zu gewährleisten wird in den entsprechenden Endpunkten anhand des JWT-Tokens die Rolle des Users überprüft, was in der `UserService` Klasse durch die Methode `boolean isAdmin(HttpHeaders headers)` implementiert ist. Diese Methode wird in der Implementierung des REST-Endpunktes verwendet und gewährleistet das Werfen einer `AccessDeniedException`, falls dem User nicht die benötigte Rolle in der Keycloak-Anwendung für den API-Request zugewiesen ist. In einigen Fällen ist es für die Validierung des Zugriffes auf einen Endpunkt des Controllers nötig Informationen aus dem `Course-Service` zu erhalten, da beispielsweise nur ein Course-Ersteller oder ein User mit der Client-Rolle `ROLE_ADMIN` die Status der Abonnenten eines Courses erhalten darf. Damit überprüft wird, ob der aktuelle User der Course-Creator ist, falls dieser kein Admin ist, wird über die Methode `restTemplate.exchange` ein HTTP-GET-Request an den `Course-Service` gesendet, um den Course-Ersteller durch die gegeben Course-UUID zu ermitteln, um diese Information mit dem aktuellen User abzugleichen [Oracle, 2015a]. Damit diese Methode von der RestTemplate-Klasse zu verwenden ist, muss zunächst eine entsprechende Bean in dem Spring Projekt hinzugefügt werden [Spring, 2021a]. Nach der Definition der Bean wird die „LoadBalanced“ RestTemplate-Klasse dazu verwendet REST-API-Requests an die einzelnen Services, die in der Service-Registry registriert sind, mit Body, Header und der entsprechenden HTTP-Methode zu senden. Diese Kommunikation der Services ist nötig, damit ein User nicht unbefugt auf Status von anderen Usern zugreift und alle Status eines Courses nur durch einen Admin oder durch den Course-Ersteller eingesehen werden.
Zusammenfassend zu dem `User-Service` ist festzuhalten, dass die Speicherung der Status für die User der Scheduler-Anwendung durch einen Postgres-Service in Verbindung mit JPA realisiert ist. Des Weiteren bietet der Service verschieden REST-API-Endpunkte für die Verwaltung von den Status und die Generierung der Userdetails aus dem JWT-Token an, wodurch die benötigten Controller aus der Abbildung 4.2 in dem Service implementiert sind. Zudem ist festzustellen, dass eine Kommunikation mit dem `Course-Service` zwingend notwendig ist, um die Funktionalitäten nach den funktionalen Anforderungen korrekt umzusetzen.

```
user-service:
    container_name: user-service
    image: fabianhalbig/user-service
    ports:
      - 9070:9070
    environment:
      - DISCOVERY=http://discovery:8761/eureka/
      - SPRING_DATASOURCE_URL=
        jdbc:postgresql://status-db:5432/compose-postgres
      - SPRING_DATASOURCE_USERNAME=compose-postgres
      - SPRING_DATASOURCE_PASSWORD=compose-postgres
      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
    depends_on:
      - discovery
      - keycloak
```

Listing 5.5: Spring-Service mit Verbindung zu einem Postgres-Service

Für die Vervollständigung der Beschreibung des Implementierungsschrittes für die Domäne Course-Task-Verwaltung, wird nun die Implementierung des fehlenden `Course-Service` beschrieben. Dieser ist genauso wie der zuvor beschriebenen Service als Eureka-Client implementiert, weshalb die Umgebungsvariable `${DISCOVERY}` in der Docker Compose Datei berücksichtigt wird, die auch bei der Definition des `User-Services` in der Abbildung 5.5 zu sehen ist. Diese Variable ist in den Applikationseigenschaften unter `eureka.client.serviceUrl` referenziert, damit es dem Service möglich ist eine Registrierung in dem Eureka-Server durchzuführen.
Zudem wurde bei dem `Course-Service` mit der bereits beschriebenen Vorgehensweise ein weiterer Postgres-Datenbankservice in der Docker Compose Datei mit Verbindung zu dem Spring-Service hinzugefügt, mit dem Unterschied, dass dieser Datenbankservice für die persistente Datenspeicherung das zusätzliche Volume `course-data` nutzt [Docker, 2021g]. Mit diesem Volume und der `Course-DB` werden die Task und Course Daten von dem `Course-Service` dauerhaft gespeichert. Interessant ist hierbei die Konfiguration der Datenbank durch die Spring-Data-JPA, da zwei Entitäten, die eine 1:N-Beziehung haben, zu implementieren sind, was in dem ER-Modell in der Abbildung 3.2 bereits verdeutlicht ist [Gierkeand et al., 2021]. Damit die 1:N-Beziehung durch die beiden Entiätsklassen `Course` und `Task` abgebildet ist, wird in der `Task` Klasse ein Attribut, das einen Course enthält, hinzugefügt, damit zu jedem Task ein Course abgespeichert wird. Zusätzlich wird in der `Course` Klasse eine Liste von Tasks hinterlegt, damit es möglich ist einem Course mehrere Tasks zuzuordnen. Dementsprechend wird die Beziehung der beiden Entitäten in der Datenbank des `Course-DB` Services abgebildet, wenn den beiden Attributen zusätzlich die Annotation `@ManyToOne` zugewiesen wird, damit in der Spring-Anwendung eindeutig die 1:N-Beziehung zwischen den beiden Entitäten definiert ist [Oracle, 2011]. Das Resultat dieser Konfiguration ist in der Abbildung A.5 mit einigen Demodaten zu sehen, in der erkenntlich ist, dass jedem Task in der Datenbank eine `Course-Uuid` zugewiesen ist, um die 1:N-Beziehung der Entitäten in der Datenbank zu hinterlegen.

Des Weiteren ist aus der Abbildung A.5 mit dem Datenbankauszug in A.4 erkenntlich, dass, wie bereits zuvor angemerkt, das Pattern „Move Foreign-Key Relationship to Code“ von Newman, 2019, das für die Anwendung in dem Datenbank-Schemata 4.3 entworfen ist, erfolgreich umgesetzt ist. Das ist erkenntlich, da aus den beiden Datenbankausschnitten zu sehen ist, dass die `task_id` in der `Status-DB` für die Status von den Usern korrekt referenziert wird. Dadurch ist beispielsweise aus den Abbildungen ersichtlich, dass für den Course mit dem Titel „Test Course 1“ zwei Tasks angelegt wurden und der Course von einem User abonniert worden ist, da auch genau zwei Status in der `Status-DB` abgespeichert sind. Allerdings ist für die Einhaltung des AKID-Prinzips bei dieser verteilten Datenspeicherung mit dem verwendeten Pattern in der Implementierung wichtig die Konsistenz über aller Datenbankservices hinweg sicherzustellen [Newman, 2019, S. 176]. Das betrifft in dieser Fallstudie vor allem die Aktualisierung der Status-Tabelle bei der Veränderung von der Task-Tabelle, da diese direkt über einen Fremdschlüssel in der `Status-DB` referenziert wird. Im Detail betrifft das die Anwendungsfälle mit den IDs AF8, AF9, AF10 und AF11 aus der Tabelle 3.1. Damit die Status-Tabelle bei diesen Anwendungsfällen entsprechend dem Update in der Task-Tabelle aktualisiert wird, sind im Folgenden die Anwendungsfälle mit der Methode aus der `CourseService` Klasse, in der die Konsistenz durch die Implementierung gewährleistet wird, festgehalten. Dadurch wird ein Überblick über die kritischen Implementierungen für die Sicherstellung des AKID-Prinzips geschaffen und die angewendete Logik erklärt.
In dem Fall, wenn ein User einen Course abonniert muss für alle bereits bestehenden Tasks des Courses jeweils ein Status für den neuen Abonnenten hinzugefügt werden. Dies ist in der Methode `createStatusOnSubscription` implementiert, da sonst ein User in der Lage wäre einen Course zu abonnieren, aber keine Status zu den enthaltenen Tasks einzusehen ist. In dieser Methode wird nach der Registrierung des Users in der Abonnentenliste des Courses in einer For-Schleife über die Tasks des Courses iteriert und für jeden Task über das `RestTemplate` der Endpunkt des `User-Services` zur Erstellung eines Status auf Basis des `HttpHeaders` und der Task-ID kontaktiert. Somit ist sichergestellt, dass bei dem Abonnieren eines Courses die benötigten Status hinzugefügt werden. Eine ähnliche Implementierung ist in der Methode `deleteAboAndStatusOfUser` für AF9 zu erkennen, da hier genauso über die Tasks, die in dem Course enthalten sind, iteriert wird. Der Unterschied hierbei ist, dass ein HTTP-DELETE-Request über das `RestTemplate` mit der Task-ID und der dazugehörigen User-ID an den `User-Service` gesendet wird, um den Status pro Task für den aktuell angemeldeten User zu löschen, bevor dieser von der Abonnentenliste des Courses gelöscht wird. In `createTaskAndStatus` ist die Logik für AF10 implementiert. In dieser Methode wird zunächst die Erstellung des neuen Tasks sichergestellt und anschließen für jeden Abonnenten durch eine For-Schleife ein Status für den neu hinzugefügten Task erstellt, damit jeder Abonnent zu jedem Task des Courses genau einen Status hat. Dies wird durch einen HTTP-POST-Request mit Task-ID und User-ID aus der Abonnentenliste an den `User-Service` gewährleistet. Der letzte Anwendungsfall bei dem die Konsistenz über die beiden Postgres-Services hinweg sichergestellt werden muss, ist der AF11. Die Logik von diesem Fall ist in der Methode `deleteTaskAndStatus` festgehalten, in der durch einen HTTP-Request die Status, die eine bestimmte Task-ID referenzieren, gelöscht werden. Durch diese Logik nach dem

Modell „Überprüfe vor der Löschung“, das auch bei der Implementierung von AF9 angewendet ist, wird sichergestellt, dass jeder Task, der in der Status-Tabelle referenziert wird, tatsächlich in der Task-Tabelle vorhanden ist [Newman, 2019, S. 176]. Hiermit wurde aufgezeigt bei welchen Anwendungsfällen ein Datenaustausch über HTTP-API-Requests zwischen dem `User-Service` und dem `Course-Service` stattfinden muss, damit das AKID-Prinzip trotz der verteilten Datenspeicherung gewährleistet ist. Anzumerken ist, dass der Anwendungsfall Course löschen nicht aufgezählt ist, da nach dem Modell „Erlaube keine Löschung“ von Newman, 2019, S. 177 ein Course nur zu löschen ist, wenn kein Task in diesem Course vorhanden ist, was bedeutet, dass bei der Löschung eines Courses nicht auf die Aktualisierung der Statustabelle geachtete werden muss.

Dementsprechend sind die wichtigsten Implementierungsschritte für die Erstellung des `Course-Service` dargestellt, wodurch die Services der Course-Task-Verwaltung komplettiert sind. Abschließend ist aus der Implementierung der Services für die Course-Task-Verwaltung abzuleiten, dass der Code aus der monolithischen Anwendung in diesem Bereich nur für die Entitätenklassen zu übernehmen ist, aber nicht für die Controller- und Serviceklassen, obwohl die Logik der einzelnen Teilbereiche nicht verändert wurde. Dies ist auf die neue Konzeption der Anwendungsarchitektur und der Aufteilung des Datenmodells zurückzuführen, da zunächst die User- und Zugriffsverwaltung ausgegliedert ist und zusätzlich durch die Aufteilung von Course, Task und Status Daten in zwei Datenbankservices API-Schnittstellen zur Kommunikation der Spring-Services untereinander und mit anderen Services zu implementieren sind.

5.1.3. Userinterface

In diesem Abschnitt wird die Implementierung der beiden Angular-Services für die Userinterface Domäne dargestellt, damit der Implementierungsschritt des Wasserfallmodells komplettiert ist. In den beiden Services `Public-Angular` und `Keycloak-Angular` wird genauso, wie in der monolithischen Anwendungen, die open-source Erweiterung Bootstrap hinzugefügt, um vorgefertigte HTML-Klassen, Module, Formen usw. in den Frontendservices zu nutzen [Bootstrap, 2021].
Aus dem Architekturdiagramm in der Abbildung 4.1 ist zu sehen, dass der Angular-Service mit der Bezeichnung `Public-Angular` für die Darstellung der Informationen, auf die auch nicht registrierte und angemeldete Nutzer zugreifen sollen, zuständig ist. Das ist daran zu erkennen, da dieser Service keine Verbindung zu der Keycloak-Anwendung hat. Dadurch ist die Implementierung dieses Services relativ trivial, da durch den Befehl `ng new` ein neues Angular Projekt zu erstellen ist und bei dieser Anwendung nur eine Komponente mit dem Befehl `ng generate component` hinzugefügt wird, um die Features der Anwendung zu implementieren [Angular, 2021]. Neben der Beschreibung der Anwendungsdetails, die in der HTML-Datei der erstellten Komponente in HTML-Tags hinzugefügt werden, muss die Verbindung zu dem `Keycloak-Angular` Service hergestellt werden. Dies ist in der Funktion `goToKeycloak` in der `app.component.ts` Datei implementiert, die durch die Umgebungsvariable den User auf den Pfad des `Keycloak-Angular` Services bei Betätigen des Login-Buttons weiterleitet. Damit diese Methode bei einem

Buttonclick ausgeführt wird, ist in der `app.component.html` bei dem Button die Option `(click)=goToKeycloak()` hinzugefügt. Weitere Implementierungsschrittes sind für diesen Service nicht notwendig, da die beschriebene Implementierungen ausreichend ist, um Anwendungsinformationen zu zeigen und den User, falls nötig auf die von Keycloak geschützte Angular-Anwendung weiterzuleiten.

Umfangreicher ist allerdings die Implementierung des `Keycloak-Angular` Services, da hierbei die Keycloak-Anwendung für die Authentifikation und die Spring-Services für die Ausführungen der Usereingaben eingebunden werden müssen. Des Weiteren ist ein Routing in dieser Anwendung hinzugefügt, da diese anders als der `Public-Angular` Service aus mehr als einer Komponente besteht [Angular, 2021].
Für die Implementierung der Authentifikation des Users für alle Routen des gesicherten Services ist es nötig ein `AuthGuard` in der Anwendung hinzuzufügen und in der `app.module.ts` Datei als Provider nach dem Tutorial von Krzywiec, 2021 zu definieren. Die Schlussendliche Implementierung der Authentifikationsmechanismen ist in der Klasse `auth.guard.ts` implementiert, die von der Klasse `KeycloakAuthGuard` die benötigten Methoden erbt [Krzywiec, 2021]. Durch diese Konfiguration der Angular-Anwendung ist es möglich in dem Routing die benötigten Pfade durch den Keycloakmechanismus für nicht eingeloggte User zu sperren. Bevor dies in der `app-routing.module.ts` Datei durch die Angabe von einer Komponente und `canActivate: [AuthGuard]` pro Pfad konfiguriert wird, muss zunächst das Routing-Modul durch den Befehl `ng new routing-app --routing --defaults` in der Anwendung hinzugefügt werden [Angular, 2021]. Durch diese Konfiguration der Angular-Anwendung mit der Integration von Keycloak ist es nur eingeloggten User möglich die Features des `Keycloak-Angular` Services zu nutzen. Dementsprechend wird ein User in die von Keycloak bereitgestellte Login-Seite weitergeleitet, wenn dieser in der `Public-Angular` Anwendung den Login-Button betätigt und aktuell kein Session-Token vorhanden ist. Falls ein User einen geschützten Pfad des `Keycloak-Angular` Services aufruft und kein Session-Token existiert, wird dieser zunächst genauso für eine Authentifikation auf die Login-Seite weitergeleitet und nach erfolgreicher Anmeldung auf die angefragte Seite geleitet. Die verschiedenen Sessions sind in der Keycloak-Anwendung pro User in der Userübersicht hinterlegt, wie es in der Abbildung A.7 für den „user1“ gezeigt ist. Durch diese Art der Authentifikation und dem daraus resultierenden JWT-Token ist es beispielsweise möglich über den `User-Controller` die benötigten Userdetails, die in dem Frontend angezeigt werden müssen, über API-Requests zu realisieren.
Generell wird in dem `Keycloak-Angular` Service das `HttpClientModule` verwendet, um benötigte Daten aus den Spring-Services zu erhalten und die Funktionalitäten in den Spring-Anwendungen zu verwenden, was eine Kommunikation mit dem Backendservices von der Angular-Anwendung über HTTP entspricht [Angular, 2021]. Ein Beispiel hierfür ist in der Initialisierungsmethode der `app.components.ts` Datei zu finden. In dieser Methode ist durch einen HTTP GET-Request implementiert den Usernamen des aktuell angemeldeten User in der „NavBar“[Bootstrap, 2021] auf allen Seiten anzuzeigen. In dem Listing 5.6 ist die `ngOnInit` Methode abgebildet, die bei der Initialisierung der Komponente aufgerufen wird [Angular, 2021]. Dadurch, dass diese Methode in der `AppComponent`

Klasse implementiert ist, wird diese Funktion bei jeder Komponente aufgerufen. Damit in dieser Methode das HttpClient Modul zu verwenden ist, wird in dem Konstruktor der Klasse das Modul als Variable mit der Bezeichnung `http` initialisiert. Dementsprechend wird diese Variable in der Funktion verwendet, damit über `this.http.get` ein GET-Request an den Endpunkt des `Spring-Cloud-Gateways` gesendet wird, um die benötigten Daten zur Darstellung des Usernamens zu erhalten. Da nicht alle zurückgegebenen Daten benötigt werden und von dem GET-Request ein Observable-Objekt zurück gegeben wird, muss als Nächstes die `subcribe` Methode aufgerufen werden, damit nur der Username aus den zurückgegebenen Daten durch `data["userName"]` extrahiert wird und in einer Klassenvariable gespeichert wird [Angular, 2021]. Entsprechend wird diese Variable mit `{{userName}}` innerhalb der HTML Datei referenziert, damit der generierte Wert in dem Frontend für den Nutzer sichtbar ist. Nach dieser Implementierungsweise sind in dem `Keycloak-Angular` Service unter anderem auch die Auflistung der Courses, Tasks und Status in anderen Komponenten realisiert.

```
1 ngOnInit(): void {
2   this.http.get("api/user/current/details")
3     .subscribe((data)=>this.userName=data["userName"]);
4 }
```

Listing 5.6: Ermittlung des Usernames in Angular

Eine ähnliche Implementierung ist für die benötigten POST-Request angewendet, was nun anhand von der Umsetzung von dem AF10 verdeutlicht wird. Nachdem ein User einen Course erstellt hat, gelangt dieser über die `course-detail-creator` Komponente auf den Pfad zu der Task-Erstellung. In der dafür zuständigen Komponente ist in der HTML Datei eine `fromGroup` mit der Bezeichnung `taskForm` hinzugefügt, damit der User durch insgesamt drei HTML-Input-Tags den Task-Titel, die Task-Beschreibung und die Deadline angibt [Bootstrap, 2021]. Durch die Spezifikation von `(ngSubmit)="onSubmit()"` in dem Form-Tag, ist es dem User möglich bei ausreichenden Informationen den Erstellen-Button zu betätigen, der die Methode `onSubmit` in der `task-create-component.ts` Datei aufruft, die in dem Listing 5.7 abgebildet ist [Angular, 2021]. Bei Aufruf dieser Methode wird zunächst der Body, der für den HTTP-POST-Request benötigt wird, in einem JSON Format definiert [Rodríguez et al., 2016, S. 23]. Hierbei werden zunächst die drei Eingaben des Users über den `formControlName` in die Variable `body` übernommen und anschließend unter dem Schlüssel `course` der JSON Variable die Referenz zu dem dazugehörigen Course aus dem bereits vorhandenen Daten dargestellt. Die Referenz zu dem Course ist nötig, da jeder Task genau einen Course referenziert, weshalb der Course in dem Body des POST-Requests mit anzugeben ist. Nachdem die Variable `body` vollständig erstellt ist, wird in der Methode über `this.http.post` ein HTTP-API-Request an den entsprechenden Endpunkt erstellt, in der auch die `body` Variable übergeben wird [Angular, 2021]. Durch diesen Request wird in dem `Course-Service` ein neuer Task zu dem Course hinzugefügt und wie bereits bei der Implementierung der Applikationslogik beschrieben zusätzlich ein Status pro Abonnent für den neuen Task

erstellt, um eine konsistente Datenspeicherung von Status und Tasks zu gewährleisten. Nachdem die Erstellung innerhalb der Applikationslogik in der Domäne Course-Task-Verwaltung abgeschlossen ist, werden die von dem User eingegebenen Werte in der `taskForm` durch `this.taskForm.reset({})` auf die ursprünglichen Werte zurückgesetzt, um gegebenenfalls eine weitere Taskerstellung zu ermöglichen [Angular, 2021]. Diese Implementierungsweise wird außerdem für die Erstellung von Courses in der Anwendung verwendet. Für die weiteren Anwendungsfälle, wie AF6, AF9 oder AF11 sind ebenfalls Methoden für Button-Aktionen in den Type-Script-Dateien hinzugefügt, allerdings mit der Methode `this.http.delete`. Die Aktualisierung eines Status von dem User durch einen Button-Click wird durch HTTP-POST-Requests in Angular realisiert. Allerdings wird hier kein Body für den Request erstellt. Durch diese Beispiele ist dargestellt, wie der `Keycloak-Angular` Service durch HTTP-Requests die Funktionalitäten und die Daten aus den Spring-Services für die Darstellung der Userinterface Domäne verwendet.

```
onSubmit() {
  const body = {
    title: this.taskForm.value.title,
    longDescription: this.taskForm.value.description,
    deadline: this.taskForm.value.deadline,
    course: {
        uuid: this.course.uuid,
        userId: this.course.userId,
        title: this.course.title,
        description: this.course.description,
        subscriber: this.course.subscriber
     }
  }
  this.http.post("api/course/task", body, {observe: "response"})
    .subscribe(data => this.course = data);
  this.taskForm.reset({})
}
```

Listing 5.7: Implementierung eines POST-Requests im Frontend

In den beiden Codebeispielen 5.6 und 5.7 ist auffällig, dass bei der URL des HTTP-Requests keine Domäne spezifiziert ist. Dies ist der Fall, da in der Angular-Anwendung eine Proxy-Konfiguration durch die `proxy.conf.json` Datei hinzugefügt worden ist, in der die Target-URL für die Requests, die mit „/api/“ starten, festgelegt ist [Angular, 2021]. Damit diese Konfiguration der beiden Angular-Anwendungen jeweils in einem Docker Container verwendbar ist, muss zusätzlich die `nginx.conf` Datei hinzugefügt werden und bei dem `Keycloak-Angular` Service neben den Standard-Konfiguration die `location /api` mit der Target-URL konfiguriert werden, damit der Service das Ziel bei Ausführung der Docker Compose Datei lokalisiert. Nachdem die zusätzlichen Dateien hinzugefügt sind, muss der Befehl `ng build --prod` ausgeführt werden, bevor aus den Anwendungen über die Dockerfiles ein Docker-Image pro Service erstellt wird.

Festzustellen ist bei der Implementierung der Angular-Services, dass ein Großteil der HTML-Templates aus der zu migrierenden Anwendung übernommen wurden, da sich der Aufbau dieser Templates nicht komplett verändert hat. Allerdings erzeugt die Implementierung der Userinterface Domäne einen zusätzlichen Aufwand für die Umsetzung der Verbindung zu den Spring-Boot Services in den Type-Script Dateien der Komponenten, der bei einer Migration nicht zu vernachlässigen ist.

Nachdem alle Services implementiert sind und dementsprechende Docker-Images für jeden Service erstellt sind, ist es möglich, die Docker-Images mit den benötigten Umgebungsvariablen und Volumes in der Docker Compose Datei hinzuzufügen, um die Compose Datei mit allen aufgezeigten Services aus der Abbildung 4.1 durch den Befehl `docker-compose up` auszuführen [Docker, 2021f]. Durch diesen Befehl wird für jeden definierten Services ein Container erstellt, der für jeden Service und Datenspeicher eine isolierte ausführbare Einheit darstellt, wie es in der Abbildung 5.1 gezeigt ist [Docker, 2021d]. Durch die definierten Schnittstellen der Services in der Docker Compose Datei wird so die komplette Anwendung „Scheduler-Services“ mit einem Befehl lokal in Docker ausgeführt. Dabei sorgen die drei definierten Volumes, die jeweils in dem entsprechenden Datenbankservice referenziert sind, für eine dauerhafte Datenspeicherung über die Laufzeit eines Containers hinweg [Docker, 2021g].

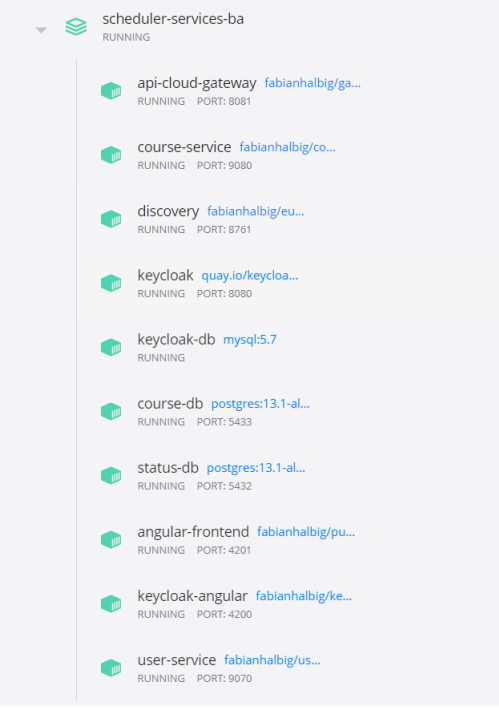

Abbildung 5.1: Erstellte Container durch Docker Compose

5.2. Deployment in Kubernetes

In dem letzten Abschnitt der Implementierung wird beispielhaft ein Deployment von Services aus zwei Domänen in Kubernetes dargestellt, um den letzten Schritt von Docker zu Kubernetes bei dieser Migration aufzuzeigen. Dadurch wird die Speicherung von persistierenden Daten der Beispielanwendung in Kubernetes aufgezeigt. Die Services des User-Managements und der Course-Task-Verwaltung werden in dem Kubernetes der GCP mithilfe von YML Dateien deployed, die in dem GitLab-Repository (`https://gitlab.lrz.de/bachelorarbeiten/bachelorarbeit-halbig/bachelorarbeit-halbig.git`) hinterlegt sind. Für das Deployment der Services ist es als erstes notwendig eine Registrierung auf der Plattform durchzuführen, damit es möglich ist mit dem Befehl `gcloud container clusters create scheduler-services --region europe-west1 --node-locations europe-west1-b,europe-west1-c` ein Cluster zu erstellen, um nach der Erstellung `Services` und `Deployments` in dem Cluster zu definieren [Google, 2021c].

5.2.1. Datenbankdienste

Nachdem ein Cluster in GCP erstellt ist, wird zunächst dargestellt, wie die drei benötigten Datenbankservices `Keycloak-DB`, `Course-DB` und `Status-DB` in dem erstellten Cluster zu realisieren sind.
Damit die Daten, die in den Datenbankservices in Dateien abgespeichert sind, persistent abgespeichert werden, wird vor der Erstellung von jedem Datenbankservice in der Plattform eine PVC mit einer entsprechenden Speicherklasse definiert [Google, 2021c]. Die Konfiguration der PVC für die `Keycloak-DB` ist in dem Codebeispiel 5.8 abgebildet, in der zu sehen ist, dass als erstes die Speicherklasse mit der Bezeichnung `keycloak-storageclass` mit den beiden `node-locations` des Clusters konfiguriert ist. Darunter ist die Definition der PVC zu sehen, die die Speicherklasse referenziert und die benötigte Speichergröße angibt. Nachdem die PVCs für eine persistente und dauerhafte Datenspeicherung auch nach der Löschung der Services oder des Clusters in der GCP konfiguriert sind, ist es nun möglich, die Postgres-Datenbanken zu erstellen. Dafür ist für jeden der drei Datenbankservices eine YML Datei mit der Definition von den beiden Typen `Deployment` und `Service` zu erstellen. In dem `Deployment` wird die Bezeichnung des Datenbankservice festgelegt und die erstellte PVC unter dem Schlüssel `volumes` referenziert. Außerdem wird hier genauso, wie in der Docker Compose Datei, ein User und ein Passwort für die Datenbank in den Umgebungsvariablen definiert und das offizielle Posgres Docker-Image referenziert [Docker, 2021c]. Wichtig ist hierbei die Verwendung von Secrets durch den „Google Secret Manager“[Google, 2021b], der ermöglicht API-Schlüssel, Zertifikate und Passwörter in der GCP verschlüsselt zu speichern, was dementsprechend für das Passwort der Datenbanken zu gewährleisten ist. Bei der Erstellung der Secrets ist im Normalfall bei der Bereitstellung einer realen Produktionsumgebung darauf zu achten, dass die dazugehörigen YML Dateien nicht in Repositories hinterlegt werden, da sonst die Sicherheit der Anwendungen nicht gewährleistet ist. Denn dadurch besteht die Gefahr, dass Schlüssel, Zertifikate und Passwörter öffentlich zugänglich sind. Somit ist eine weitere YML Datei für die Konfiguration des Secrets mit der Bezeichnung `db-secret` essenziell, damit das

Datenbankpassword für alle Services zugänglich ist, aber in verschlüsselter Form abgespeichert ist. Dieses Secret wird dann in den `Deployments` der verschiedenen Services durch die Secretbezeichnung und dem Schlüssel des benötigten Wertes referenziert, damit das Passwort nicht im Klartext angegeben ist. Des Weiteren ist ergänzend zu dem `Deployment` in der gleichen Datei ein `Service` konfiguriert, um die Pod-Endpunkte, die durch das `Deployment` erstellt werden, in „einer einzigen Ressource zu gruppieren“[Google, 2021a]. Nachdem die Konfiguration der GCP-Datenbankservices abgeschlossen ist, werden diese mit dem Befehl `kubectl apply -f` in dem Cluster hinzugefügt. Dieser Befehl ist für alle definierten YML Dateien für das Kubernetes Deployment der Anwendung anwendbar, da dieser auch für die Konfiguration der Dienste der Applikationslogik in dem folgenden Kapitel verwendet wird.

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: keycloak-storageclass
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  replication-type: regional-pd
allowedTopologies:
  - matchLabelExpressions:
      - key: failure-domain.beta.kubernetes.io/zone
        values:
          - europe-west1-b
          - europe-west1-c
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: keycloak-pv
spec:
  storageClassName: keycloak-storageclass
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 200Gi
```

Listing 5.8: PVC von einem Datenbankservice in GCP

5.2.2. Dienste für die Applikationslogik

Sobald die drei benötigten Datenbanken aus der Abbildung 4.3 in Kubernetes durch die zuvor dargestellte Konfiguration zur Verfügung stehen, ist die Bereitstellung der dazugehörigen Dienste der Applikationslogik in Kubernetes möglich.

Am einfachsten ist die Konfiguration der Keycloak-Anwendung, da hier die Elemente aus der Compose Datei in die YML Datei für das Kubernetes `Deployment` und dem `Service` übernommen werden. Allerdings ist hier anders als bei der Ausführung durch Docker Compose die Datenbank, die unter der Umgebungsvariable `DB_DATABASE` definiert ist, manuell in dem Datenbankservice hinzuzufügen. Für die manuelle Hinzufügung muss mit dem Befehl `kubectl exec -it <Pod-Name> -- psql -U <Postgres-User>` auf einen Pod des Datenbankservices zugegriffen werden und in dem nächsten Schritt durch `CREATE DATABASE <Datenbank-Name>` die benötigte Datenbank hinzugefügt werden, die dann mit der gegebenen Bezeichnung in dem `Deployment` referenziert wird [Google, 2021c]. Nach dieser Hinzufügung wird die Ressourcendatei für die Keycloak-Anwendung in der Node des Clusters als Kubernetes-Objekt hinzugefügt. Sobald ein Endpunkt für den `Keycloak-Service` in der GCP-Übersicht erstellt ist, wird über diesen Endpunkt ermöglicht auf die Anwendung zuzugreifen und ein Realm mit dem Namen „demo-realm“ zu erstellen, um die `realm-export.json` Datei aus der zuvor dargestellten Konfiguration für die Erstellung der Clients und Rollen zu importieren, wodurch die User-Management Domäne erfolgreich in dem Kubernetes der GCP deployed ist.

Nachdem die Keycloak-Anwendung in Kubernetes sowie die benötigten Datenbankdienste erreichbar sind und die Userverwaltung bereits möglich ist, werden die Spring-Boot Services durch weitere YML Dateien für Kubernetes konfiguriert. Hierzu müssen die Applikationseigenschaften der Spring-Boot Services leicht adaptiert werden, damit der Eureka-Server in dem Kubernetes-Cluster von dem `Spring-Cloud-Gateway`, dem `Course-Serivce` und dem `User-Service` gefunden wird. Deshalb wird die `defaultZone` bei den Services geändert und die Projekte in eine JAR-Datei konvertiert, um ein Docker-Image für jeden Service mit einem speziellen Tag für die Kubernetes-Konfiguration zu erstellen [Docker, 2021e]. Nach der Erstellung der Images müssen diese durch den Befehl `docker push` in Docker-Hub hochgeladen werden, damit die Images global verfügbar sind, wodurch eine Referenz auf diese Images mit dem gegebenen Tag in den Ressourcendateien ermöglicht wird.
Die Konfiguration des Eureka-Servers ist von den anderen Spring-Services zu differenzieren, da dieser Dienst mit dem Diensttypen `NodePort` erstellt wird. Dies wird benötigt, damit der Dienst über die IP-Adresse eines beliebigen Knotens mit dem Wert des `nodePort` erreichbar ist [Google, 2021a]. Des Weiteren wird ein StatefulSet für den Eureka-Server definiert, damit die Pods für diesen Service eine eindeutige und dauerhaftee Identität sowie einen stabilen Hostname haben [Google, 2021d]. Außerdem wird eine `ConfigMap` konfiguriert, damit die Service-Adresse des Eureka-Servers spezifiziert ist und anschließend in dem StatefulSet referenzierbar ist. Der Knotenpunkt für den Eureka-Server mit der Bezeichnung `eureka-lb` und der Service des StatefulSet sind mit allen anderen Services in der Abbildung A.9 dargestellt, die einen Auszug aus der Dienstübersicht des erstellten Clusters zeigt. Dadurch ist das Image des Eureka-Services in der GCP unter einem stabilen Hostnamen deployed und bietet eine Service-Registry für die anderen Spring-Services an.
Bevor das Gateway in Kubernetes deployed wird, muss ein Secret für die Keycloak-Referenzen, wie beispielsweise die Keycloak-Client-ID oder das Keycloak-Client-Secret

analog zu dem `db-secret` erstellt werden. Dies ist Notwendig, um die benötigten Informationen für den Authentifikationsmechanismus verschlüsselt in der Ressourcendatei einzufügen. Nach der Erstellung der benötigten Informationen durch ein Secret und des `Deployments` mit den Umgebungsvariablen wird das `Spring-Cloud-Gateway` durch einen Dienst mit dem Typen `LoadBalancer` in der Node hinzugefügt. Durch die Verwendung dieses Diensttypen wird automatisch ein Netzwerk-Load-Balancer und eine stabile IP-Adresse für diesen Service erstellt, „auf die von außerhalb des Projekts zugegriffen werden kann“[Google, 2021a]. Dieser Service muss genauso wie die Keycloak-Anwendung eine IP-Adresse haben, die von außerhalb des Clusters erreichbar ist, da hierdurch API-Requests ermöglicht werden, die von einem Admin oder auch von der Frontendanwendungen genutzt werden müssen, um den Anwendungszweck zu erfüllen. Durch diese Konfigurationen ist es möglich das `Spring-Cloud-Gateway` in der GCP zu deployen und auf die beiden benötigten Services aus dem Routing zuzugreifen, deren Konfiguration nun beschrieben wird.
Anders als bei den bisher beschriebenen Spring-Services wird bei den Ressourcendateien des `Course-Services` und dem `User-Service` kein Diensttyp angegeben, da bei keiner Typdefinition der Standard-Typ `ClusterIP` in dem Cluster verwendet wird. Die beiden Services stellen dadurch eine stabile IP-Adresse in Kubernetes zur Verfügung, auf die ausschließlich von Services innerhalb des Clusters zugegriffen wird. Deshalb sind die beiden Spring-Services mit dem Typ `ClusterIP` hinter dem `Spring-Coud-Gateway` versteckt, da die Services von außerhalb des Clusters nur durch das Gateway erreichbar sind, wie es in dem Abschnitt 4.1 dargestellt ist. Durch diese Umsetzung wir das Prinzip des „Information Hiding“[Newman, 2019, S. 18] umgesetzt, da die Endpunkte von dem `Course-Service` und dem `User-Service` nur durch das Gateway adressierbar sind und somit so wenig wie möglich von der Domäne Course-Task-Verwaltung öffentlich zugänglich ist. Bei diesen beiden Services muss genauso wie bei der Keycloak-Anwendung die Referenzierte Datenbank in der Umgebungsvariable `SPRING_DATASOURCE_URL` in dem dazugehörigen Datenbankservice erstellt werden. Dies Erstellung ist erforderlich für das Deployment der Services in GCP, damit ein korrekter Verbindungsaufbau gewährleistet ist, wie es in der Abbildung 5.2 beispielhaft für den `Course-DB` Service aufgezeigt ist [Google, 2021c].
Somit wurde in diesem Abschnitt der Konfigurationsaufwand für das Deployment von Microservices, die zuvor in Docker Compose ausgeführt wurden, in dem Kubernetes von GCP aufgezeigt. Hieraus ist abzuleiten, dass nur geringfügige Änderungen an den Docker-Images vorgenommen werden müssen, da die spezifische Konfiguration für Kubernetes in den YML Konfigurationsdateien durchgeführt wird.
Zusammenfassend wurde in diesem Kapitel aufgezeigt, wie die Architektur der Services aus der Abbildung 4.1 komplett in Docker Compose ausgeführt wird. Danach wurde beispielhaft dargelegt, wie die Konfiguration von Services von einem Deployment in Docker Compose zu Kubernetes adaptiert wird.
Dabei ist das Deployment in Docker Compose als Möglichkeit zu betrachten, um die Implementierung der funktionalen Anforderung innerhalb der kompletten Architektur und somit auch domänenübergreifend zu testen. Allerdings sind nicht alle nicht-funktionalen Anforderungen in der Docker Compose Umgebung bereitzustellen, wie beispielsweise die Skalierbarkeit, Zuverlässigkeit und globale Erreichbarkeit, da das Docker Compose

Deployment nur lokal ausgeführt wird. Deshalb ist für die Validierung der Erfüllung der nicht-funktionalen Anforderungen ein Deployment in Kubernetes nötig, da hier die Anzahl an Instanzen pro Service festzulegen ist und die Services beispielsweise durch einen `LoadBalancer` global erreichbar sind. Zusätzlich beschreibt jede YML Datei einen Zustand, der durch das Kubernetes der GCP versucht wird dauerhaft einzuhalten, wodurch die Zuverlässigkeit des Systems gewährleistet wird. Des Weiteren wird in Kubernetes als auch in Docker durch Volumes beziehungsweise PVC eine dauerhafte Datenspeicherung der generierten Daten gewährleistet. Die Persistenz der Daten wird durch die korrekte Umsetzung der spezifizierten Pattern des Enwurfs in der Implementierung der Spring-Boot-Services nach Newman, 2019 sichergestellt.
Abschließend ist dadurch festzuhalten, dass die Implementierung der funktionalen Anforderungen durch Docker Compose getestet, validiert und die Machbarkeit der entworfenen Microservice-Architektur nachzuweisen ist, aber für die Validierung einiger nicht-funktionale Anforderungen aus der Tabelle 3.1 ein Deployment der Services aus den Domänen User-Management und Course-Task-Verwaltung in Kubernetes nötig ist. Durch die beschrieben Erfüllung der definierten Anforderungen in dem Kapitel 3 ist festzustellen, dass die monolithische Anwendung durch das dargelegte Vorgehen der Fallstudie mit den Pattern nach Newman, 2019 und einem DDD erfolgreich zu einer Microservice-Architektur in Docker Compose und Kubernetes migriert wurde.

```
C:\Users\Fabian>kubectl exec -it course-db-b6987cd-zt5g7 -- psql -U courseUser
psql (10.19 (Debian 10.19-1.pgdg90+1))
Type "help" for help.

courseUser=# CREATE DATABASE courses;
CREATE DATABASE
courseUser=# \l
                                  List of databases
    Name    |   Owner    | Encoding |  Collate   |   Ctype    |     Access privileges
------------+------------+----------+------------+------------+---------------------------
 courseUser | courseUser | UTF8     | en_US.utf8 | en_US.utf8 |
 courses    | courseUser | UTF8     | en_US.utf8 | en_US.utf8 |
 postgres   | courseUser | UTF8     | en_US.utf8 | en_US.utf8 |
 status     | courseUser | UTF8     | en_US.utf8 | en_US.utf8 |
 template0  | courseUser | UTF8     | en_US.utf8 | en_US.utf8 | =c/courseUser            +
            |            |          |            |            | courseUser=CTc/courseUser
 template1  | courseUser | UTF8     | en_US.utf8 | en_US.utf8 | =c/courseUser            +
            |            |          |            |            | courseUser=CTc/courseUser
(6 rows)
```

Abbildung 5.2: Erstellung einer Datenbank in einem Postgres-Pod

6. Ergebnisse der Fallstudie

In dem letzten Kapitel werden zunächst die Ergebnisse aus der durchgeführten Migration von einer Standalone Spring-Anwendung hinzu Microservices in Kubernetes zusammengefasst und die aus dieser Fallstudie resultierenden Ergebnisse in einem Fazit dargelegt. Abschließend wird ein Ausblick auf die sich über die Fallstudie hinaus ergebenden Möglichkeiten bei einer Migration von einem monolithischen Design hinzu einer Microservice-Architektur in Kubernetes gegeben.

6.1. Zusammenfassung

In der durchgeführten Fallstudie wurde anhand der Phasen Anforderungsanalyse, Entwurf und Implementierung des Wasserfallmodells eine Migration der monolithischen Spring-Anwendung „Course-Planner“ hinzu Microservices in Kubernetes durchgeführt [Royce, 1987]. Dadurch wurde entsprechend der Zielsetzung in dem Kapitel 1.2 innerhalb eines gesamten Softwareentwicklungszyklus einer Migration zu einer Microservice-Architektur aufgezeigt, wie ein bestehendes Datenmodell mit persistierenden Daten in eine verteilte Datenhaltung während der Migration aufgeteilt wird.
Für diese Migration ist eine Analyse der bestehenden Anwendung in dem Kapitel 3 essenziell gewesen, um die funktionalen Anforderungen für die Migration abzuleiten und zudem ein ER-Modell nach Chen, 1976 zu erstellen und zu analysieren, damit ein Überblick über die Anwendung und die damit verbunden Daten geschaffen wird. Des Weiteren war es notwendig die nicht-funktionalen Anforderungen an das neue System zu definieren, damit der Entwurf der Architektur diese Anforderungen berücksichtigt. Durch die detaillierte Analyse und Anforderungsdefinition war es möglich in dem Kapitel 4 den Entwurf für die Microservice-Anwendung „Scheduler-Services“ zu kreieren. Bei dem Entwurf wurden zunächst BC nach dem DDD von Evans, 2015 definiert, die dann mit den Services pro Domain in einem Architekturmodell nach dem Microserviceansatz modelliert wurden [Richardson, 2020]. Bei dem Architekturmodell war es wichtig auch die Kommunikation über die BCs hinweg, beispielsweise durch ein UML Komponentendiagramm als Domänenmodell, darzustellen. Nachdem die Architektur mit verschiedenen Datenbankservices definiert wurde und eindeutig ist welcher Service auf eine eigene Datenbank zugreift, war es essenziell die Aufteilung und die Anpassungen des bestehenden Datenmodells festzulegen. In der Fallstudie wurden dazu die bestehenden Methoden von Newman, 2019 verwendet, damit trotz der Aufteilung eine persistente Datenspeicherung durch das Architekturdesign und der entsprechenden Implementierung der verwendeten Modelle sichergestellt wird. Dadurch sind die beiden gesetzten Ziele aus dem Kapitel 1.2 zur Anwendung von den Modellen nach Newman, 2019 und die Berücksichtigung des DDD erfüllt. Jedoch wäre es möglich gewesen das neue Datenbankmodell in einem anderen Design zu modellieren, allerdings ist der beschrieben Ansatz nach durchgeführten Architekturtests und Evaluierung von anderen Designs für die definierten Anwendungsfälle und das Ziel der Arbeit am besten für diese Einzelfallstudie geeignet. Damit gewährleistet ist, dass alle Anforderungen durch das dargestellte Design in dem Entwurf berücksichtigt sind, ist eine Validierung des Entwurfs durchgeführt worden [Benra, 2009, S. 178].

Nach Fertigstellung des Entwurfs, bei dem bereits einige Tests zur Umsetzung der Architektur durchgeführt wurden, war es möglich die einzelnen Services zu implementieren. Dabei wurden zunächst die Services zum testen lokal und einzeln ausgeführt. Nach abgeschlossener Implementierung wurden die Services des Microservice-Entwurfs gemeinsam durch Docker Compose lokal ausgeführt, um die Machbarkeit des Entwurfs zu prüfen und somit das gegebene Ziel der Ausführung durch Docker Compose in dem Kapitel 1.2 zu erfüllen [Docker, 2021f]. Anschließend wurde mit geringfügigen Abänderungen an den Applikationseigenschaften ein Deployment der Services, die für die Speicherung der Daten und die Applikationslogik verantwortlich sind, in einem Kubernetes-Cluster von der GCP durchgeführt. Anhand dieser Deployments ist die persistente Speicherung der Anwendungsdaten durch Postgres-Dienste in Verbindung mit PVCs in Kubernetes dargestellt, da durch die PVCs die gespeicherten Daten auch nach Löschung der Datenbankdienste oder des kompletten Clusters weiterhin zur Verfügung stehen, sodass in dieser Fallstudie unter Berücksichtigung der definierten Ziele eine Migration von einer monolithischen Spring-Anwendung über Microservices hinzu Kubernetes dargestellt ist.

6.2. Fazit

In diesem Abschnitt wird das Fazit zu dem Ziel dieser Fallstudie, das auf die Untersuchung der bestehenden Modellen zur Aufteilung eines Datenmodells aus einer monolithischen Anwendung zu einer verteilten Datenspeicherung in einer Microservice-Architektur festgelegt wurde, dargestellt.
Bei der Durchführung der Fallstudie ist festgestellt worden, dass ein Verständnis zu dem Aufbau, den Anforderung und dem Datenmodell der zu migrierenden Anwendung unerlässlich ist. Dieses tiefgehende Verständnis ist Notwendig, da sonst bereits bestehende Anforderungen in dem Entwurf nicht umgesetzt werden oder die Aufteilung des Datenmodells fehlerhaft modelliert wird, wenn der komplette Umfang des aufzuteilenden Modells nicht vorliegt. Deshalb ist eine detaillierte Analyse für das nötige Verständnis zu der monolithischen Anwendung ausschlaggebend für einen erfolgreichen Entwurf des neuen Systems, da der Entwurf auf Basis der definierten Anforderungen und aus den Informationen über die bestehende Anwendung modelliert wird. Außerdem sollten bei dem Entwurf auch fremde Lösungen, wie die Verwendung von open-source Projekten, in die Betrachtung mit einbezogen werden, da die Verwendung von bereits existierenden Projekten und Lösungen den Implementierungsaufwand verringert. Dies ist an der Architektur der Fallstudie zu begründen, da durch die Einbindung von Keycloak für das Identitäts- und Zugriffsmanagement in der User-Management Domäne nur die Konfiguration entsprechend definiert werden musste und dadurch kein zusätzlicher Implementierungsaufwand für die Erstellung eines Identitäts-Service entstanden ist [Keycloak, 2021c]. Allerdings muss bei der Einbindung von fremden Lösungen bereits bei dem Entwurf der Architektur durch Tests sichergestellt werden, dass die ausgewählte Lösung problemlos in die Architektur zu integrieren ist, damit nicht in der Implementierung die Unbrauchbarkeit der modellierten Lösung festgestellt wird. Deshalb bietet sich an bereits vor der Implementierung Architekturtests von bestimmten Teilen des Systems durchzuführen, um den Entwurf umsetzbar zu gestalten.

Des Weiteren muss innerhalb des Entwurfs bereits die Aufteilung des Datenmodells definiert werden, wobei in der Fallstudie auf die definierten Modelle von Newman, 2019 zurückgegriffen wurde. Durch die Einbindung der Modelle „Move Foreign-Key Relationship to Code“[Newman, 2019, S. 173], „Check before deletion“[Newman, 2019, S. 176] und „Do not allow deletion“[Newman, 2019, S. 177] in dem Entwurf, ist die Persistenz der Daten sichergestellt, wobei bereits kritische Stellten, bei denen innerhalb des Codes die Persistenz sichergestellt werden muss, innerhalb des Entwurfs zu identifizieren sind. Dementsprechend ist festzustellen, dass die verwendeten Modelle von Newman, 2019 für diese Einzelfallstudie erfolgreich für die Aufteilung des Datenmodells eingesetzt wurden. Allerdings muss genau analysiert werden, welches Modell bei den einzelnen Aufteilungen des bestehenden Datenmodells verwendet wird, wie es in dem Kapitel 2.2 beschrieben ist und zusätzlich auf Brückenlösungen verzichtet werden, da diese oft einen großen Implementierungsaufwand mit sich bringen und nicht für einen Dauerhaften Zeitraum in der Microservice-Architektur verwendet werden sollten.
Aufgrund dieser Ergebnisse der Fallstudie ist festzuhalten, dass die verwendeten Modelle von Newman, 2019 in dieser Einzelfallstudie umsetzbar sind. Für eine allgemeingültige Aussage über die Verwendung der Modelle müssten weitere Migrationsprojekte mit anderen Systemen, die ein komplexeres Datenmodell oder auch einen anderen Aufbau beziehungsweise andere Tools in der ursprünglichen Implementierung verwenden, betrachtete werden, um zu beobachten, ob diese auch bei der Verwendung von weiteren Modellen von Newman, 2019 zu dem gleichen Ergebnis, wie diese Fallstudie, gelangen. Für die Erfolgreiche Umsetzung einer Migration mit diesen Modellen ist es unerlässlich einen detaillierten Entwurf der Architektur und des geteilten Datenmodells auszuarbeiten, da bei einem fehlerhaften, nicht detaillierten oder nicht umsetzbaren Entwurf die Implementierung und das Deployment der Services aufgrund der höheren Komplexität großes Potenzial zeigt zu scheitern. Somit ist ein Entwurf und die damit verbundene Analyse des bestehenden System zusammen mit der Entscheidung zu den anzuwendenden Modellen ausschlaggebend für eine erfolgreiche Migration zu Microservices in Kubernetes.

6.3. Ausblick

In dem letzten Abschnitt wird ein Ausblick auf eine komplette Produktionsumgebung mit einer Continous-Integration (CI) und Continous-Delivery (CD) Plattform für implementierte Microservices in Kubernetes gegeben und anschließen auf die Möglichkeiten der Verwendung von GitOps mit „Argo CD“[CD, 2021] eingegangen.
Damit die Produktionsumgebung der monolithischen Anwendung vollständig durch Kubernetes abgebildet wird, muss zunächst ein Deployment der bereits implementierten Userinterface Domäne in Kubernetes erstellte werden. Dies wurde in der Fallstudie nicht durchgeführt, da für die Darstellung der Datenspeicherung nach dem AKID-Prinzip das Deployment der Services aus den anderen beiden Domänen ausreichend ist. Diese zusätzlichen Deployments müssten für die fehlende Domain manuell in der Node des Clusters hinzugefügt werden, weshalb eine Verwendung einer CI/CD Pipeline über die Migration hinaus sinnvoll ist, die in der Abbildung A.10 allgemeingültig für das DevOps Umfeld dargestellt ist. Eine CI/CD Pipline sorgt dafür, dass nach der Implementierung von Änderungen ein automatisierter Test und anschließender Build der Anwendung durchgeführt wird und dementsprechend die Änderungen und neu definierte Dienste durch den CI Server ausgerollt werden [GitLab, 2021]. Ein Vorteile der Einbindung von einer

CI/CD Pipline nach einer erfolgreichen Migration ist bei der Versionierung der Services zu sehen, da es möglich ist eine neue Version eines Services parallel zu der vorherigen Version bereitzustellen [Microsoft, 2021]. Zusätzlich können verschiedene Teams Services „unabhängig voneinander entwickeln und bereitstellen“[Microsoft, 2021]. Diese beiden Vorteile vereinfachen und beschleunigen den Bereitstellungsprozess von Software, was eine Microservice-Architektur in Verbindung mit einer CI/CD Plattform interessant für die meisten Softwarebereiche macht, da oft eine schnelle Anpassbarkeit und Bereitstellung gefordert ist.
Neben einer CI/CD Plattform besteht die Option einen der neusten Trends in dem DevOps Bereich zu folgen und GitOps in dem Kubernetes-Umfeld einzusetzen. Im Gegensatz zu CI/CD wird bei GitOps das Deployment direkt aus Git synchronisiert, was bedeutet, dass nicht der CI-Server das ausrollen übernimmt, sondern die Konfiguration in Git versioniert ist, wodurch als Quelle für der Deployments ein Git-Repository wird [Schnatterer and Mariewka, 2014]. Durch die Verwendung eines Git-Repository anstatt einem CI-Server wird hierbei auch von Infrastructure as Code (IaC) gesprochen [Schnatterer and Mariewka, 2014]. Für die Umsetzung von einem GitOps-System gibt es bereits einen großen Markt an Anbietern, wobei das open-source Projekt Argo CD eine der weit verbreitetsten Lösungen im GitOps-Bereich bereitstellt. Durch Argo CD wird ein definierter Zielzustand, der in Git festgelegt wird, dauerhaft in Kubernetes eingehalten, da Argo CD als Kubernetes-Controller implementiert ist, der dauerhaft die ausgeführten Anwendungen in dem Cluster überwacht und mit dem definierten Zustand vergleicht. Insofern Argo CD eine Abweichung von einem Live-Stand im Vergleich zu dem definierten Stand in dem Git-Repository feststellt, zeigt das GitOps-System diese Abweichung an und stellt Möglichkeiten zur Verfügung die Synchronisation des Live-Standes und dem definierten Standes automatisiert oder manuell wieder herzustellen. Dementsprechend werden auch Änderungen an dem Code des Git-Repositories automatisch in Kubernetes durch den Kubernetes-Controller angewendet, wodurch die betroffenen Deployments und Dienste aktualisiert werden [CD, 2021]. Somit wird der Kern der Infrastruktur durch Argo CD oder einem ähnlichen GitOps-System abgebildet. Dieses Design der Infrastruktur ermöglicht manuelle Deployments in Kubernetes zu vermeiden und alle Deployments über ein GitOps-System zu erstellen, was darin resultiert, dass durch die Verwendung von GitOps kein Expertenwissen über Kubernetes benötigt wird, um Kubernetes Dienste über eine GitOps-Infrastruktur bereitzustellen.

Zusammenfassend werden die Ergebnisse der Fallstudie folgendermaßen dargestellt. Durch die Fallstudie wurde aufgezeigt, wie eine monolithische Spring-Boot Anwendung und das damit Verbundene Datenmodell nach dem Wasserfallmodell und mit ausgewählten Modellen nach Newman, 2019 und der Anwendung des DDD zu einer Microservice-Architektur migriert wird. Des Weiteren wurde aufgezeigt, wie eine persistente Datenspeicherung in einer verteilten Datenspeicherung in Docker Compose und Kubernetes implementiert wird. Allerdings ist es möglich weitere Untersuchungen zu der Verwendung von weiteren Modellen von Newman, 2019 oder die Anwendung der Modelle auf anderen Systemen zu überprüfen. Des Weiteren sind die Vorteile einer Einführung von einer CI/CD Pipeline oder eines GitOps-System für die Darstellung einer komplettierten Produktionsumgebung der migrierten Anwendung dargestellt, um über die Migration hinaus für eine stabile und anpassbare Infrastruktur zu sorgen.

Literaturverzeichnis

[Alex et al., 2004] Alex, B., Taylor, L., Winch, R., Hillert, G., Grandja, J., and Bryant, J. (2004). Spring security reference. *https://docs.spring.io/spring-security/site/docs/5.2.1.RELEASE/reference/pdf/spring-security-reference.pdf*.

[Angular, 2021] Angular (2021). Angular docs. `https://angular.io/docs`. Zugriff 16.05.21.

[Auth0, 2021] Auth0 (2021). Libraries for token signing/verification. `https://jwt.io/libraries`. Zugriff 29.11.21.

[Baeldung, 2021a] Baeldung (2021a). Spring boot security auto-configuration. `https://www.baeldung.com/spring-boot-security-autoconfiguration`. Zugriff 06.11.21.

[Baeldung, 2021b] Baeldung (2021b). Using application.yml vs application.properties in spring boot. `https://www.baeldung.com/spring-boot-yaml-vs-properties`. Zugriff 28.12.21.

[Benra, 2009] Benra, J. T. (2009). *Software-Entwicklung für Echtzeitsysteme*. Springer-Verlag GmbH.

[Booch et al., 1999] Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *Das UML-Benutzerhandbuch*, volume 1. Addison-Wesley München.

[Bootstrap, 2021] Bootstrap (2021). Build fast, responsive sites with bootstrap. `https://getbootstrap.com/`. Zugriff 29.12.21.

[CD, 2021] CD, A. (2021). Argo cd - declerative gitops cd for kubernetes. Zugriff 05.01.22.

[Chen, 1976] Chen, P. P.-S. (1976). The entity-relationship model—toward a unified view of data. *ACM transactions on database systems (TODS)*, 1(1):9–36.

[Christie et al., 2017] Christie, M. A., Bhandar, A., Nakandala, S., Marru, S., Abeysinghe, E., Pamidighantam, S., and Pierce, M. E. (2017). Using keycloak for gateway authentication and authorization.

[Dhalla, 2021] Dhalla, H. K. (2021). A performance comparison of restful applications implemented in spring boot java and ms. net core. In *Journal of Physics: Conference Series*, volume 1933, page 012041. IOP Publishing.

[Docker, 2021f] Docker (2013-2021f). Overview of docker compose. `https://docs.docker.com/compose/`. Docker-Compose Dokumentation, Zugriff 28.12.21.

[Docker, 2021a] Docker (2021a). Docker official images. `https://docs.docker.com/docker-hub/official_images/`. Images and Container, Zugriff 06.11.21.

[Docker, 2021b] Docker (2021b). Docker official images. https://hub.docker.com/_/mysql. MySQL Official Image, Zugriff 28.12.21.

[Docker, 2021c] Docker (2021c). Docker official images. https://hub.docker.com/_/postgres. Postgres Official Image, Zugriff 28.12.21.

[Docker, 2021d] Docker (2021d). Docker overview. https://docs.docker.com/get-started/overview/. Images and Container, Zugriff 06.11.21.

[Docker, 2021e] Docker (2021e). docker tag. https://docs.docker.com/engine/reference/commandline/tag/. Zugriff 02.01.22.

[Docker, 2021g] Docker (2021g). Use volumes. https://docs.docker.com/storage/volumes/. Zugriff 09.12.21.

[Evans, 2015] Evans, E. (2015). Domain-driven design reference. *Definitions and Pattern Summaries. März*.

[Foote, 2021] Foote, K. D. (2021). A brief history of microservices. https://www.dataversity.net/a-brief-history-of-microservices/#. Zugriff 09.12.21.

[Gadatsch, 2019] Gadatsch, A. (2019). *Datenmodellierung*. Springer Fachmedien Wiesbaden.

[García, 2020] García, M. M. (2020). *Learn Microservices with Spring Boot*. Apress.

[Garvie, 2021] Garvie, L. (2021). Decode a jwt token in java. https://www.baeldung.com/java-jwt-token-decode. Zugriff 01.12.21.

[Gierkeand et al., 2021] Gierkeand, O., Darimont, T., Strobl, C., Paluch, M., and Bryant, J. (2021). Using jpa named queries. https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.named-queries.declaring-interfaces. Zugriff 28.12.21.

[GitLab, 2021] GitLab (2021). Ci/cd concepts. https://docs.gitlab.com/ee/ci/introduction/. Zugriff 05.01.22.

[Google, 2021a] Google (2021a). Dienste. https://cloud.google.com/kubernetes-engine/docs/concepts/service. Zugriff 02.01.22.

[Google, 2021b] Google (2021b). gcloud secrets. https://cloud.google.com/sdk/gcloud/reference/secrets. Zugriff 02.01.22.

[Google, 2021c] Google (2021c). Hochverfügbares postgresql mit gke bereitstellen. https://cloud.google.com/architecture/deploying-highly-available-postgresql-with-gke. Zugriff 02.01.22.

[Google, 2021d] Google (2021d). Statefulset. https://cloud.google.com/kubernetes-engine/docs/concepts/statefulset. Zugriff 02.01.22.

[Google, 2022] Google (2022). Google trends. https://www.google.com/trends. Zugriff 03.01.22.

[Horn, 2007a] Horn, T. (2007a). Architekturen für webanwendungen. https://www.torsten-horn.de/techdocs/sw-dev-process.htm. Techdocs, Zugriff 15.11.21.

[Horn, 2007b] Horn, T. (2007b). Vorgehensmodelle zum softwareentwicklungsprozess. https://torsten-horn.de/techdocs/webanwendungen.htm#Applikationsserver. Techdocs, Zugriff 12.12.21.

[Janser, 2015] Janser, S. (2015). Eureka – microservice-registry mit spring cloud. https://heise.de/-2848238. Zugriff 29.11.21.

[JavaTpoint, 2021] JavaTpoint (2021). Spring boot h2 database. https://www.javatpoint.com/spring-boot-h2-database. Zugriff 15.11.21.

[Karslioglu, 2020] Karslioglu, M. (2020). *Kubernetes A Complete DevOps Cookbook*. Packt Publishing Ltd.

[Keycloak, 2021a] Keycloak (2021a). Client roles. https://www.keycloak.org/docs/latest/server_admin/#con-client-roles_server_administration_guide. Zugriff 28.12.21.

[Keycloak, 2021b] Keycloak (2021b). Configuring realms. https://www.keycloak.org/docs/latest/server_admin/#configuring-realms. Zugriff 28.12.21.

[Keycloak, 2021c] Keycloak (2021c). Keylcoak - about. https://www.keycloak.org/about. Zugriff 29.11.21.

[Krzywiec, 2021] Krzywiec, W. (2021). Step-by-step guide how integrate keycloak with angular application. https://tinyurl.com/3a6ph6eu. Zugriff 29.12.21.

[Kubernetes, 2021a] Kubernetes (2021a). Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/. Zugriff 14.11.21.

[Kubernetes, 2021b] Kubernetes (2021b). Kubernetes lokal über minikube betreiben. https://kubernetes.io/de/docs/setup/minikube/. Zugriff 14.11.21.

[Kubernetes, 2021c] Kubernetes (2021c). Verwendung eines services zum veröffentlichen ihrer app. https://kubernetes.io/de/docs/tutorials/kubernetes-basics/expose/expose-intro/. Zugriff 14.11.21.

[Kubernetes, 2021d] Kubernetes (2021d). What is kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Zugriff 14.11.21.

[Microsoft, 2021] Microsoft (2021). Erstellen einer ci/cd-pipeline für microservices in kubernetes. https://docs.microsoft.com/de-de/azure/architecture/microservices/ci-cd-kubernetes#full-cicd-build. Zugriff 05.01.22.

[Newman, 2019] Newman, S. (2019). *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media.

[OAUTH, 2021] OAUTH (2021). Oauth 2.0. `https://jwt.io/libraries`. Zugriff 29.11.21.

[Oracle, 2011] Oracle (2011). Annotation type manytoone. `https://docs.oracle.com/javaee/6/api/javax/persistence/ManyToOne.html`. Zugriff 29.12.21.

[Oracle, 2014] Oracle (2014). Annotations. `https://docs.oracle.com/javaee/7/api/javax/json/JsonObject.html`. Zugriff 29.12.21.

[Oracle, 2015a] Oracle (2015a). Class resttemplate. `ClassRestTemplate`. Zugriff 29.12.21.

[Oracle, 2015b] Oracle (2015b). Interface jsonobject. `https://docs.oracle.com/javaee/7/api/javax/json/JsonObject.html`. Zugriff 29.12.21.

[Qusay, 2005] Qusay, M. H. (2005). Service-oriented architecture (soa) and web services: The road to enterprise application integration (eai). `https://www.oracle.com/technical-resources/articles/javase/soa.html`. Zugriff 09.12.21.

[Reddy, 2017] Reddy, K. S. P. (2017). *Beginning Spring Boot 2*. Springer-Verlag GmbH.

[Richardson, 2020] Richardson, C. (2020). Pattern: Microservice architecture. `https://microservices.io/patterns/microservices.html`. Microservices.io, Zugriff 16.05.21.

[Rodríguez et al., 2016] Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J. C., Canali, L., and Percannella, G. (2016). Rest apis: A large-scale analysis of compliance with principles and best practices. In Bozzon, A., Cudre-Maroux, P., and Pautasso, C., editors, *Web Engineering*, pages 21–39, Cham. Springer International Publishing.

[Royce, 1987] Royce, W. W. (1987). Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338.

[Saunders, 2007] Saunders, M. N. K. (2007). *Research methods for business students*. Financial Times/Prentice Hall, Harlow, England New York.

[Schimandle, 2021] Schimandle, T. (2021). Spring cloud – tracing services with zipkin. `https://www.baeldung.com/tracing-services-with-zipkin`. Zugriff 06.11.21.

[Schnatterer and Mariewka, 2014] Schnatterer, J. and Mariewka, P. (2014). Automatisierungsgehilfen: Gitops-tools im vergleich. `https://cloudogu.com/de/blog/gitops-tools`. 06.01.21.

[Silz, 2021] Silz, K. (2021). Running spring boot with postgresql in docker compose. `https://www.baeldung.com/spring-boot-postgresql-docker`. Zugriff 28.12.21.

[Spring, 2021a] Spring (2021a). The ioc container. https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-introduction. Zugriff 29.12.21.

[Spring, 2021b] Spring (2021b). Spring cloud gateway. https://cloud.spring.io/spring-cloud-gateway/reference/html/. Zugriff 06.11.21.

[VMware, 2021a] VMware (2021a). Building rest services with spring. https://spring.io/guides/tutorials/rest/. Zugriff 16.05.21.

[VMware, 2021b] VMware (2021b). Cloud. https://spring.io/cloud. Zugriff 06.11.21.

[VMware, 2021c] VMware (2021c). Securing a web application. https://spring.io/guides/gs/securing-web/. Zugriff 06.11.21.

[VMware, 2021d] VMware (2021d). Spring security - prerequisites. https://docs.spring.io/spring-security/prerequisites.html. Zugriff 06.11.21.

[Webb, 2020] Webb, P. (2020). Creating docker images with spring boot 2.3.0.m1. https://spring.io/blog/2020/01/27/creating-docker-images-with-spring-boot-2-3-0-m1. Package up Spring Boot applications into Docker images, Zugriff 16.05.21.

[Yilmaz, 2019] Yilmaz, O. (2019). *Introduction to DevOps with Kubernetes*. Packt Publishing Ltd.

A. Anhang

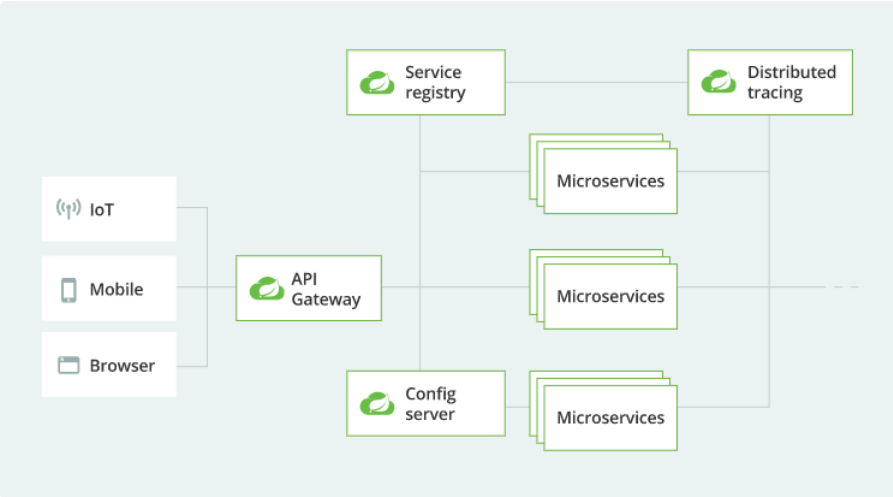

Abbildung A.1: Spring Cloud Architektur [VMware, 2021b]

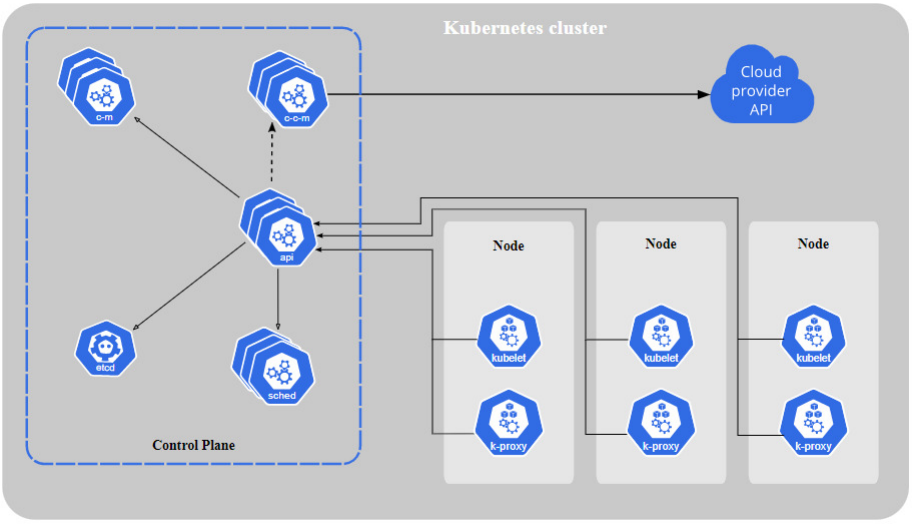

Abbildung A.2: Bestandteile eines Kubernetes-Clusters [Kubernetes, 2021a]

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - 8151b174fa18:api-gateway:8081
COURSE-SERVICE	n/a (1)	(1)	UP (1) - 5665479a49e7:course-service:9080
USER-SERVICE	n/a (1)	(1)	UP (1) - a05e8fe98057:user-service:9070

Abbildung A.3: Service-Registry des Eureka-Servers

```
/ # psql -h localhost -p 5432 -U compose-postgres
psql (13.1)
Type "help" for help.

compose-postgres=# \l
                                    List of databases
      Name       |      Owner       | Encoding |  Collate   |   Ctype    |            Access privileges
-----------------+------------------+----------+------------+------------+-----------------------------------------
 compose-postgres | compose-postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 postgres         | compose-postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 template0        | compose-postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/"compose-postgres"                  +
                  |                  |          |            |            | "compose-postgres"=CTc/"compose-postgres"
 template1        | compose-postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/"compose-postgres"                  +
                  |                  |          |            |            | "compose-postgres"=CTc/"compose-postgres"
(4 rows)

compose-postgres=# \c compose-postgres
You are now connected to database "compose-postgres" as user "compose-postgres".
compose-postgres=# \dt
            List of relations
 Schema |  Name  | Type  |      Owner
--------+--------+-------+-----------------
 public | status | table | compose-postgres
(1 row)

compose-postgres=# select * from status;
 id |  status   | task_id |              user_id
----+-----------+---------+--------------------------------------
  7 | IN_ARBEIT |      19 | e62c1ff5-d469-446c-a151-73c412210d7d
  8 | OFFEN     |      20 | e62c1ff5-d469-446c-a151-73c412210d7d
(2 rows)

compose-postgres=#
```

Abbildung A.4: Auszug aus der Status-DB

```
compose-postgres=# select uuid, title, description, user_id from course;
                 uuid                 |     title     |            description             |              user_id
--------------------------------------+---------------+------------------------------------+--------------------------------------
 2f4b8a68-9e16-4f69-b10f-755cf24a7c91 | Test Course 1 | Beschreibung für den Test Course 1 | 3b3e16d6-081f-46e0-a462-fce0ed1966e5
 15241c66-5bb9-43f8-a211-0a3058d57b9f | Test Course 3 | Beschreibung zu dem Course 3       | e62c1ff5-d469-446c-a151-73c412210d7d
 23083f96-d216-42d4-86a5-c61ddf8d0b8a | Test Course 2 | Beschreibung zu dem Test Course 2  | 3b3e16d6-081f-46e0-a462-fce0ed1966e5
(3 rows)

compose-postgres=# select * from task;
 id |  deadline  |               long_description                |          title          |             course_uuid
----+------------+-----------------------------------------------+-------------------------+--------------------------------------
 19 | 2022-01-20 | Beschreibung zu dem Task 1 aus dem Course 1   | Task 1 für Course 1     | 2f4b8a68-9e16-4f69-b10f-755cf24a7c91
 20 | 2022-01-20 | Beschreibung für den Task 2 aus dem Course 1  | Task 2 für den Course 1 | 2f4b8a68-9e16-4f69-b10f-755cf24a7c91
(2 rows)

compose-postgres=#
```

Abbildung A.5: Auszug aus der Course-DB

```
mysql> describe USER_ENTITY;
+-----------------------------+-------------+------+-----+---------+-------+
| Field                       | Type        | Null | Key | Default | Extra |
+-----------------------------+-------------+------+-----+---------+-------+
| ID                          | varchar(36) | NO   | PRI | NULL    |       |
| EMAIL                       | varchar(255)| YES  | MUL | NULL    |       |
| EMAIL_CONSTRAINT            | varchar(255)| YES  |     | NULL    |       |
| EMAIL_VERIFIED              | bit(1)      | NO   |     | b'0'    |       |
| ENABLED                     | bit(1)      | NO   |     | b'0'    |       |
| FEDERATION_LINK             | varchar(255)| YES  |     | NULL    |       |
| FIRST_NAME                  | varchar(255)| YES  |     | NULL    |       |
| LAST_NAME                   | varchar(255)| YES  |     | NULL    |       |
| REALM_ID                    | varchar(255)| YES  | MUL | NULL    |       |
| USERNAME                    | varchar(255)| YES  |     | NULL    |       |
| CREATED_TIMESTAMP           | bigint(20)  | YES  |     | NULL    |       |
| SERVICE_ACCOUNT_CLIENT_LINK | varchar(255)| YES  |     | NULL    |       |
| NOT_BEFORE                  | int(11)     | NO   |     | 0       |       |
+-----------------------------+-------------+------+-----+---------+-------+
13 rows in set (0.00 sec)

mysql> select ID, EMAIL, EMAIL_CONSTRAINT, FIRST_NAME, LAST_NAME, REALM_ID, USERNAME from USER_ENTITY WHERE REALM_ID="demo-realm";
+--------------------------------------+--------------+-----------------+------------+-----------+-----------+----------+
| ID                                   | EMAIL        | EMAIL_CONSTRAINT | FIRST_NAME | LAST_NAME | REALM_ID  | USERNAME |
+--------------------------------------+--------------+-----------------+------------+-----------+-----------+----------+
| 3b3e16d6-081f-46e0-a462-fce0ed1966e5 | user1@test.de | user1@test.de   | first1     | last1     | demo-realm | user1    |
| e62c1ff5-d469-446c-a151-73c412210d7d | user2@test.de | user2@test.de   | first2     | last2     | demo-realm | user2    |
+--------------------------------------+--------------+-----------------+------------+-----------+-----------+----------+
2 rows in set (0.00 sec)
```

Abbildung A.6: Auszug aus der Keycloak-DB

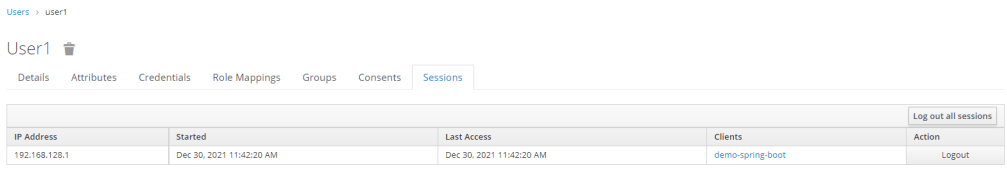

Abbildung A.7: User-Session in Keycloak

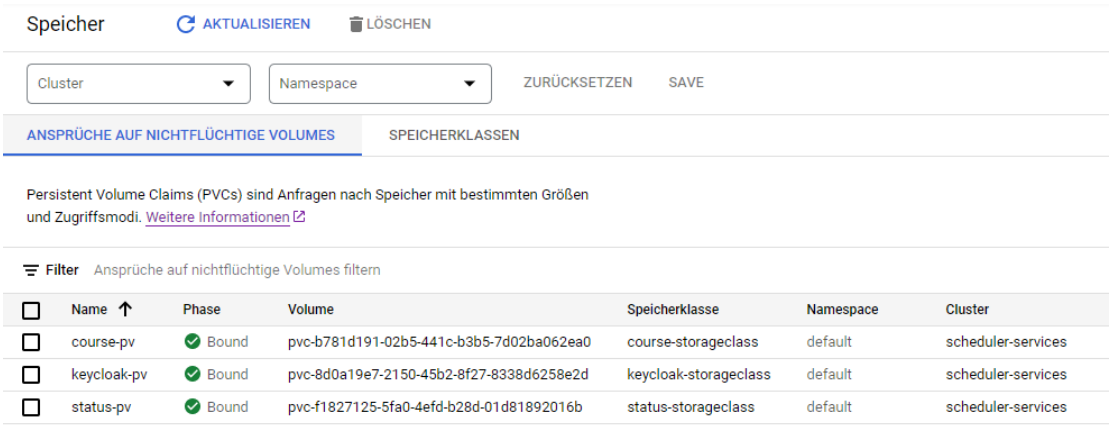

Abbildung A.8: Persistent Volume Claim in GCP

	Name	Status	Typ	Endpunkte
☐	course-db	OK	Cluster-IP-Adresse	Keine
☐	course-service	OK	Cluster-IP-Adresse	10.12.13.139
☐	eureka	OK	Cluster-IP-Adresse	Keine
☐	eureka-lb	OK	Knotenport	10.12.8.14:80 TCP
☐	gateway	OK	Externer Load-Balancer	34.77.58.84:80
☐	keycloak	OK	Externer Load-Balancer	35.195.185.201:8080
☐	keycloak-db	OK	Cluster-IP-Adresse	Keine
☐	status-db	OK	Cluster-IP-Adresse	Keine
☐	user-service	OK	Cluster-IP-Adresse	10.12.13.223

Abbildung A.9: Dienste des Beispieldeployments

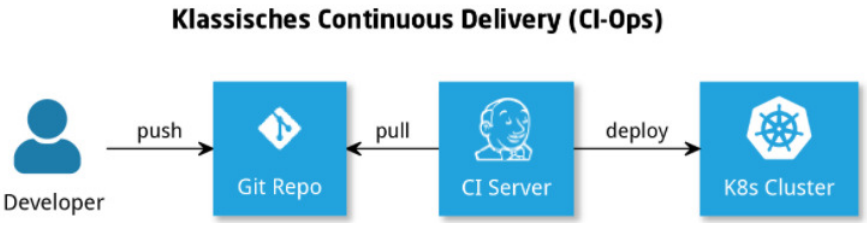

Abbildung A.10: CI/CD Pipline [Schnatterer and Mariewka, 2014]