HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Konzeptionierung und prototypische Umsetzung eines Cloud-Marketplaces für Software-as-a-Service-Produkte

Marco Schiborr

Masterarbeit im Studiengang „Angewandte Informatik“

16. August 2022

Autor: Marco Schiborr
Matrikelnummer: 1633405
marco.schiborr@gmail.com

Erstprüfer: Prof. Dr. Arne Koschel
Abteilung Informatik, Fakultät IV
Hochschule Hannover
arne.koschel@hs-hannover.de

Zweitprüfer: Dr. Marcel Ziems
Landesamt für Geoinformation und Landesvermessung
Landesbetrieb Landesvermessung und Geobasisdaten
marcel.ziems@lgln.niedersachsen.de

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Masterarbeit selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Hannover, den 16. August 2022 Unterschrift

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen meiner Masterarbeit beigetragen haben.

Zuerst gebührt mein Dank Richard Bischof vom Landesamt für Geoinformation und Landesvermessung, der meine Masterarbeit betreut und begutachtet hat. Für die zuverlässigen und hilfreichen Anregungen sowie für die konstruktive Kritik während des gesamten Betreuungszeitraums dieser Arbeit möchte ich mich herzlich bedanken.

Ich bedanke mich nachdrücklich bei Herrn Prof. Dr. Arne Koschel und Herrn Dr. Marcel Ziems für die Betreuung dieser Masterarbeit. Des Weiteren möchte ich mich bei meinen Professoren und Kommilitonen meines Masterstudiums für das lehrreiche und angenehme Studium bedanken. Besonders möchte ich mich dafür bei Kevin Jeske bedanken.

Ebenfalls möchte ich mich bei Sven Schulz und Christin Schulze bedanken, die mir mit viel Geduld und Hilfsbereitschaft zur Seite standen. Außerdem möchte ich mich auch für das Korrekturlesen meiner Masterarbeit bedanken.

Letztlich richte ich auch ein Dankeschön an Marie Krüger sowie an meine Eltern, ohne deren Unterstützung wäre mir das Schreiben dieser Abschlussarbeit in dieser Form nicht möglich gewesen.

Marco Schiborr

Hannover, den 16. August 2022

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Quellcodes

Abkürzungsverzeichnis

1 Einführung

1.1 Motivation

Bereits kurz nach der Entstehung des Internets kamen erste digitale Plattformen auf, die den Handel von Waren und Dienstleistungen über digitale Kanäle ermöglichten. Neben der technischen Neuerung des digitalen Zugriffs, kann ihr Erfolg vor allem auf ein grundlegend verändertes Rollenverständnis der Akteure zurückgeführt werden. Anstatt dass die Plattformen selbst als Handelspartner tätig sind, bringen sie Anbieter und Käufer zusammen und bieten nützliche Funktionen, sowie einen Mechanismus zur Absicherung der Transaktionen, an.

Weitere Vorteile einer Plattform als zentraler Cloud-Marketplace für einen Anbieter ist der mögliche Nutzen aus dem vorhandenen Kundenstamm, die Senkung des Entwicklungsaufwandes für einen eigenen Vertriebskanal, die Überwachung der Nutzung sowie die Verkürzung des Zeitraums zur Bereitstellung des Produktes am Markt. Für den Käufer bieten sich Vorteile wie eine Vernetzung verschiedener Produkte, Transparenz und Vertrauen gegenüber Preisgestaltung und Produktqualität der Services, sowie eine zuverlässige Nutzbarkeit der Plattform.

Dieses Prinzip der „Plattform-Ökonomie“ ist auch bei heutigen Public Cloud-Providern (z.B. Amazon Web Services, Google Cloud Platform, Microsoft Azure etc.) zu erkennen, die allesamt Möglichkeiten zur Integration von Drittanbieter Software- und Services bieten. Dabei sind die Verantwortlichkeiten der drei Akteure (Service-Consumer, Service-Provider und die Plattform) bedeutsam (siehe Abbildung 1.1). Die Software wird nach einem Onboarding-Prozess im Produkt-Portfolio der Cloud (Cloud-Marketplace) aufgeführt. Diese Software muss nach bestimmten Prinzipien implementiert sein und bestimmten Anforderungen zur Integration genügen, beispielsweise durch die Unterstützung der Open Service Broker API (OSBAPI Spezifikation). Der Software-as-a-Service (SaaS)-Anbieter stellt service-spezifische Funktionalitäten zur Verfügung, wie beispielsweise das Updaten der Service-Instanzen oder verschiedene Preismodelle. Dieser Service steht anschließend den Plattform Kunden für die Buchung zur Verfügung. Benutzer dieses Cloud-Marketplaces können sich bei dieser Cloud registrieren und

bekommen eine eindeutige Zuweisung einer Identität. Die Authentisierung, Identifizierung und Rollenzuweisung verschiedener Benutzer ist Aufgabe des Cloud-Marketplaces. Außerdem ist es die Aufgabe des Cloud-Marketplaces, das Buchen der SaaS-Produkt durchzuführen und zu überwachen. So wird von der Cloud-Plattform die Benutzung verschiedener Produkte überwacht sowie die Abrechnung aller gebuchten Produkte zusammengeführt und einheitlich durchgeführt.

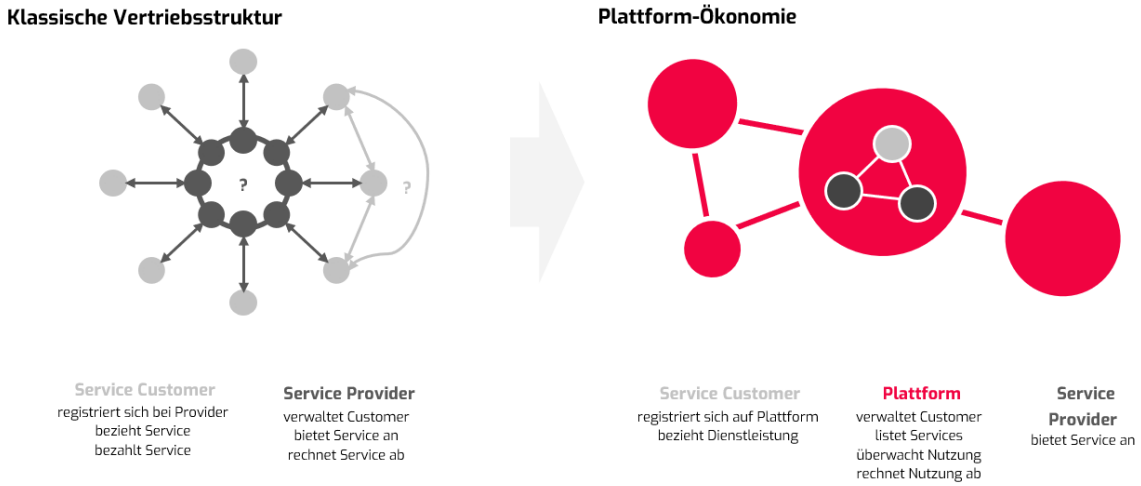


Abbildung 1.1: Klassische Vertriebsstruktur vs. Plattform-Ökonomie [LGLN 2022]

Gemäß dieses Konzeptes wird ein Cloud-Marketplace für amtliche SaaS-Produkte beim Landesamt für Geoinformation und Landesvermessung Niedersachsen (LGLN) entstehen. Das LGLN möchte damit ihre eigene Plattform „geoPlattform “ als neuen Vertriebskanal für ihre Produkte und Dienstleistungen positionieren.

1.2 Ziele

Das Ziel dieser Arbeit dient der Charakterisierung und Umsetzung einer Cloud-Plattform für den Cloud-Marketplace, die sicher und stets antwortbereit ist. Dies macht das Landesamt für Geoinformation und Landesvermessung Niedersachsen unabhängig von den Public Cloud-Anbietern und ermöglicht, die eigenen Produkte und Dienstleistungen in einem eigenen Cloud-Marketplace anzubieten.

So sind für diese Arbeit folgende Ergebnisse geplant:

- Charakterisierung der Verantwortlichkeiten und Anforderungen von Anbieter, Käufer und Plattform im Cloud-Marketplace.
- Erstellung einer Cloud-Marketplace-Architektur für SaaS-Produkte sowie eine fachliche und technische Evaluation anhand vergleichbarer Architekturen.
- Prototypische Umsetzung der vorgestellten Architektur für den Cloud-Marketplace mit den dazugehörigen Querschnittaufgaben.

1.3 Struktur

Diese Arbeit gliedert sich in folgende neun Kapitel:

In Kapitel 1 wird die Notwendigkeit des Cloud-Marketplaces motiviert, zudem werden kurz die Ziele, die Struktur und die Konventionen der Arbeit erläutert.

In Kapitel 2 erfolgt eine Beschreibung grundlegender Konzepte und Technologien dieser Arbeit. Besonders relevant sind die Grundlagen von Microservices, Cloud Computing, reaktiven Systemen, Skalierbarkeit und Event-Driven Architecture.

In Kapitel 3 werden die Akteure der Plattform, sowie die Plattform selbst, nach Anforderungen und Verantwortlichkeiten analysiert. Anhand dieser Analyse ist es möglich technische und fachliche Anforderungen der Plattform zu erheben.

Kapitel 4 definiert den Umfang der fachlichen und technischen Anforderungen. Anschließend wird ein Katalog mit funktionalen, technischen und betrieblichen Anforderungen erstellt.

In Kapitel 5 wird der Cloud-Marketplace anhand einer Event-Driven Reactive Services-Architektur entworfen. Dazu wird ein Standard für die SaaS-Produkte sowie für die Kommunikation zwischen den Akteuren definiert. Zudem werden die zentralen Workflows der Applikation im Detail erläutert. Die Dokumentation, Kommunikation und der Entwurf der Software- und Systemarchitektur wird

nach der arc42 Methodik umgesetzt.

Kapitel 6 widmet sich einer kurzen Vorstellung der prototypischen Umsetzung. Es wird eine Erläuterung zur Technologieauswahl, Implementierungs- und Konfigurationsaspekte sowie zur Bereitstellung der Software gegeben.

In Kapitel 7 findet eine konzeptionelle und eine technische Evaluation statt. Im konzeptionellen Teil wird ein Vergleich der vorgestellten Architektur mit vergleichbaren Umsetzungen evaluiert. Im technischen Teil wird die Zielsetzung anhand verschiedener Experimente auf Erfüllung überprüft.

In Kapitel 8 wird im Fazit eine Zusammenfassung der Ergebnisse und die Erfüllung der Ziele kurz erläutert. Außerdem wird ein Ausblick für Erweiterungsmöglichkeiten sowie für die nächsten Schritte gegeben.

1.4 Konventionen

Die in dieser Arbeit verwendeten Abkürzungen sind im Abkürzungsverzeichnis aufgeführt. Bei der ersten Verwendung einer Abkürzung wird die Abkürzung eines Begriffes zusammen mit der ausgeschriebenen Form in Klammern angegeben.

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers (m/w/d) verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Konzepte und Technologien dieser Arbeit vorgestellt. Besonders relevant sind die Grundlagen der Microservice-Architektur, Cloud Computing, reaktiven Systemen, Skalierbarkeit und Event-Driven Architecture.

2.1 Microservice-Architektur

Microservices sind ein Architekturmuster, das eine Anwendung als eine Sammlung kleiner, lose gekoppelter Services strukturiert, die zusammenarbeiten, um ein gemeinsames Ziel zu erreichen. Da sie unabhängig voneinander arbeiten, können sie unabhängig voneinander hinzugefügt, entfernt oder aktualisiert werden, ohne sich gegenseitig zu beeinträchtigen [Confluent 2022].

2.1.1 Microservice-Architektur im Vergleich zur monolithischen Architektur

Anders als bei einer Microservice-Architektur besteht die monolithische Architektur aus nur einem Service (siehe Abbildung 2.1). Bei der monolithischen Architektur sind alle Prozesse eng miteinander verbunden und arbeiten als ein Service. Für eine Leistungssteigerung des Systems muss das gesamte System skaliert werden. Das Erweitern, Warten oder Hinzufügen von Funktionen ist bei großen monolithischen Systemen risikobehaftet und unter Umständen weniger effizient, da ein neues Deployment der gesamten Anwendung erforderlich ist. Monolithische Architekturen können das Risiko für die Anwendungsverfügbarkeit erhöhen, da viele abhängige und eng miteinander verbundene Prozesse die Auswirkungen eines einzelnen Prozessausfalls erhöhen [Amazon-AWS 2022d].

Bei der Microservice-Architektur wird dagegen eine Anwendung in Form von

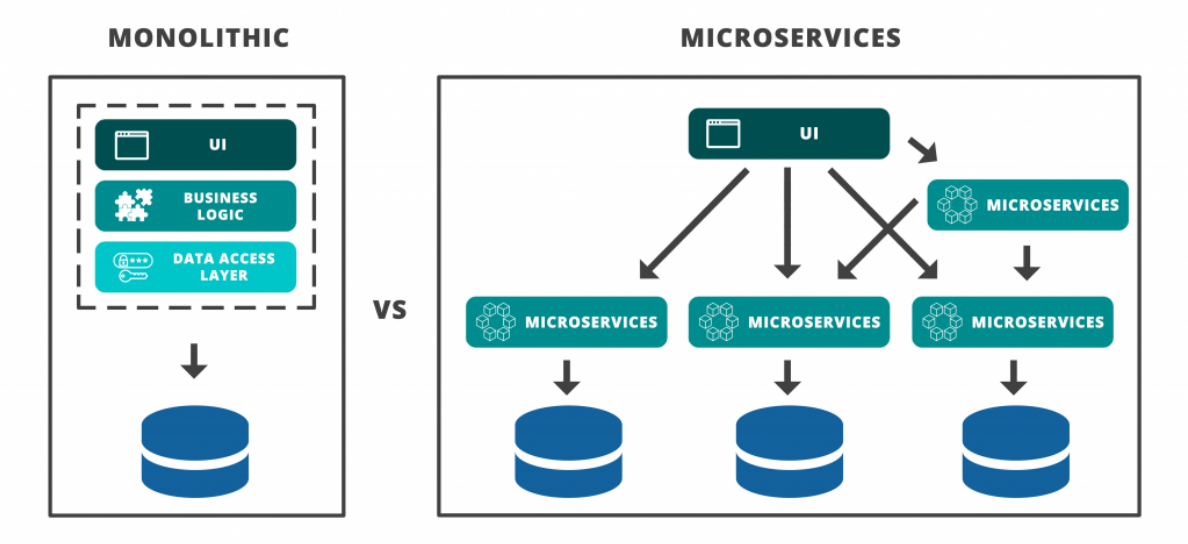


Abbildung 2.1: Monolithische Architektur im Vergleich zur Microservice-Architektur [Red-Hat 2022]

eigenständigen Komponenten erstellt, die jeden Anwendungsprozess als Service ausführen. Diese Services kommunizieren mithilfe von schlanken Application Programming Interfaces (APIs) über eine sorgfältig definierte Schnittstelle miteinander. Services werden im Hinblick auf Unternehmensfunktionen entwickelt, wobei jeder Service eine bestimmte Funktion erfüllt. Da sie unabhängig voneinander ausgeführt werden, kann jeder Service unabhängig aktualisiert, bereitgestellt und skaliert werden, um die Anforderungen an bestimmte Funktionen einer Anwendung zu erfüllen [Amazon-AWS 2022d].

2.1.2 Eigenschaften von Microservices

Amazon AWS definiert zwei wesentliche Eigenschaften von Microservices [Amazon-AWS 2022d]:

Eigenständigkeit

Jeder Komponentenservice in einer Microservices-Architektur kann entwickelt, bereitgestellt, betrieben und skaliert werden, ohne die Funktionsfähigkeit anderer

Services zu beeinträchtigen. Services müssen keinen Code und keine Implementierung mit anderen Services teilen. Die Kommunikation zwischen den einzelnen Komponenten erfolgt über klar definierte APIs [Amazon-AWS 2022d].

Spezialisierung

Jeder Service ist für eine Reihe von Funktionen ausgelegt und konzentriert sich auf die Lösung eines bestimmten Problems. Wenn Entwickler im Laufe der Zeit mehr Code zu einem Service beisteuern und der Service komplex wird, kann er in kleinere Services unterteilt werden [Amazon-AWS 2022d].

2.1.3 Vorteile von Microservices

Amazon AWS definiert fünf wesentliche Vorteile von Microservices:

Agilität

Microservices unterstützen eine Organisation von kleinen, unabhängigen Teams, die jeweils die Verantwortung für ihre eigenen Services übernehmen. Teams agieren in einem kleinen und klar definierten Kontext und haben die Möglichkeit, eigenverantwortlich und schneller zu arbeiten. Dadurch werden die Entwicklungszyklen verkürzt. Sie profitieren in erheblichem Maße von der Gesamtleistung der Organisation [Amazon-AWS 2022d].

Flexible Skalierung

Microservices ermöglichen es, jeden Service unabhängig voneinander zu skalieren, um die Nachfrage nach der von ihm unterstützten Anwendungsfunktion zu decken. Auf diese Weise können Teams die Infrastrukturanforderungen anpassen, die Kosten einer Funktion genau messen und die Verfügbarkeit aufrechterhalten, wenn ein Service eine Nachfragesteigerung verzeichnet [Amazon-AWS 2022d].

Einfache Bereitstellung

Microservices ermöglichen eine kontinuierliche Integration und Bereitstellung, wodurch es möglich wird, neue Konzepte auszuprobieren und zurückzunehmen, da

jeder Microservices unabhängig entwickelt und bereitgestellt werden kann. Die niedrigen Ausfallkosten ermöglichen Experimente, erleichtern die Aktualisierung von Code und verkürzen die Markteinführungszeit für neue Funktionen [Amazon-AWS 2022d].

Technologische Flexibilität

Microservices-Architekturen folgen nicht dem Ansatz einer Bereitstellung von Einheitslösungen. Die Teams haben die Freiheit, das geeignetste Tool zur Lösung ihrer spezifischen Probleme auszuwählen. Infolgedessen können Teams, die Microservices entwickeln, flexibel und unabhängig von anderen Teams entwickeln [Amazon-AWS 2022d].

Wiederverwendbarer Code

Die Aufteilung der Software in kleine, klar definierte Module ermöglicht es Teams, Funktionen für verschiedene Zwecke zu nutzen. Ein für eine bestimmte Funktion geschriebener Service kann auch als Baustein für einen anderen Funktionsumfang verwendet werden. Dies ermöglicht es einer Anwendung, auf sich selbst zurückzugreifen, da Entwickler neue Funktionen erstellen können, ohne Code von Grund auf neu zu schreiben [Amazon-AWS 2022d].

Resilienz

Die Serviceunabhängigkeit erhöht die Ausfallsicherheit einer Anwendung. In einer monolithischen Architektur kann der Ausfall einer einzelnen Komponente zum Ausfall der gesamten Anwendung führen. Mit Microservices überwinden Anwendungen einen kompletten Serviceausfall, indem sie die Funktionalität beeinträchtigen und nicht die gesamte Anwendung zum Absturz bringen [Amazon-AWS 2022d].

2.1.4 Herausforderungen von Microservices

Eine Microservices-Architektur bringt jedoch nicht nur Vorteile mit sich. Als Herausforderung für eine Softwarelösung mit Microservices gelten das Testen, das Versionieren, die Bereitstellung, das Logging, das Überwachen, das Debugging und die Konnektivität der Microservices [Red-Hat 2022]. Diese Nachteile sollten

nicht vernachlässigt werden, da es sonst, durch beispielsweise nicht lose gekoppelten Microserives, zu erhöhter Netzwerklast und so zu größeren Effizienzverlusten in der gesamten Applikation kommen kann. Die einzelnen Micorservices sollten daher klar getrennte Verantwortlichkeiten, eine lose Kopplung und eine schlanke API besitzen.

2.2 Container

Container machen Anwendungen unabhängiger von der Umgebung, in der sie ausgeführt werden. Sie agieren damit ähnlich einer Virtuelle Maschine (VM). Abbildung 2.2 zeigt, dass eine VM jedoch ein vollständiges Betriebssystem, Kernel sowie Applikationen enthält.

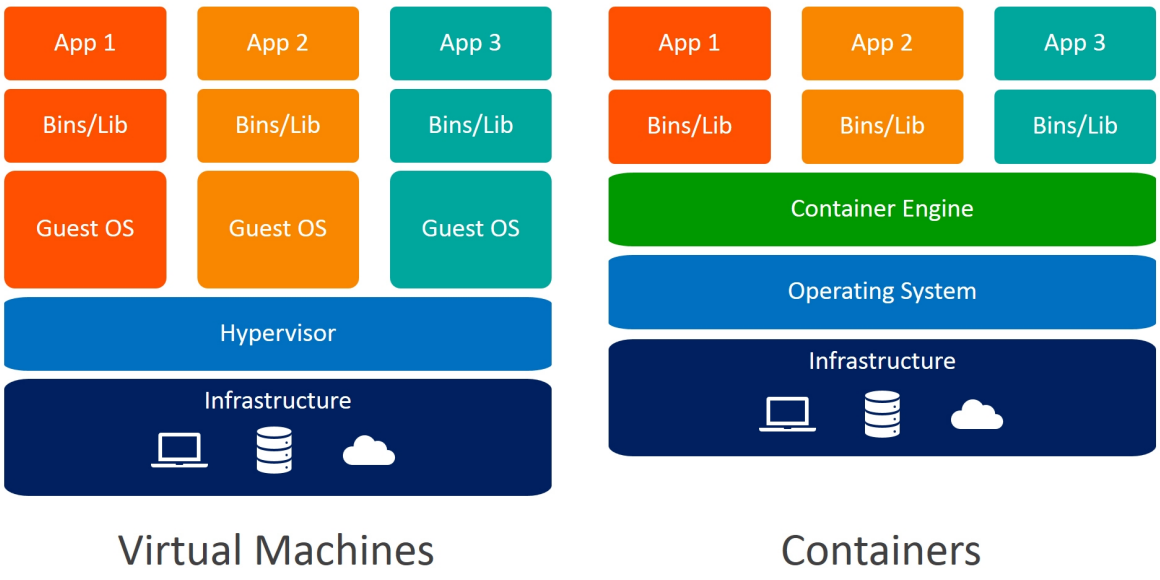


Abbildung 2.2: Virtuelle Maschinen und Container im Vergleich [Plusserver 2022]

Dies ist ineffizienter als Container, denn ein Betriebssystem und ein Kernel benötigten fest zugewiesene Mengen an RAM, CPU und Speicherkapazität. Mehrere Container teilen sich jedoch einen Betriebssystemkern. Jede Anwendung erhält

lediglich einen neuen User Space und damit eine komplett isolierte Umgebung [Plusserver 2022]. Zudem gilt Containervirtualisierung als besonders ressourcenschonend. Durch Containerisierung können Systemressourcen dem Container und Prozessen isoliert zugewiesen und verwaltet werden.

Die wohl bekannteste freie Software für Containervirtualisierung ist Docker. Docker ist eine freie Software und wird im Folgenden kurz erklärt.

2.2.1 Docker

Docker isoliert Anwendungen mit Hilfe von Containervirtualisierung. Docker vereinfacht die Bereitstellung von Anwendungen, weil sich Container, die alle nötigen Pakete enthalten, leicht als Dateien transportieren und installieren lassen. Die umfassende End-to-End-Plattform von Docker umfasst Benutzeroberflächen, Command Line Interfaces (CLIs), APIs und Sicherheitsfunktionen, die so konzipiert sind, dass sie über den gesamten Lebenszyklus der Anwendungsentwicklung hinweg zusammenarbeiten [Docker-Inc. 2022].

Dockerfiles

Um die Anwendung mit Docker zu erstellen, muss ein Dockerfile erstellt werden. Ein Dockerfile ist ein textbasiertes Skript mit Anweisungen, dass zur Erstellung eines Container-Images verwendet wird. Quelltext 2.1 zeigt ein Beispiel eines Dockerfiles. Es beschreibt die Schritte, welche zur Erzeugung eines Docker-Image führen. Das Docker-Image kann anschließend über die Docker CLI ausgeführt werden. Weitergehende Informationen zum Thema CLI, Dockerfile-Erstellung, Docker-Images, Docker-Registry und Docker-Runtime sind für diese Arbeit nicht von Relevanz.

```
1  FROM maven:3.6.1-jdk-11 AS build
2  RUN mkdir -p /workspace
3  WORKDIR /workspace
4  COPY pom.xml /workspace
5  COPY src /workspace/src
6  COPY ./src/main/resources/application.yml .
7  RUN mvn -f pom.xml clean package -e
8
9  FROM openjdk:11
10 COPY --from=build /workspace/target/*.jar service.jar
```

```
EXPOSE 8083
ENTRYPOINT ["java","-jar","service.jar"]
```

Quelltext 2.1: Beispiel Dockerfile

2.2.2 Docker Compose

Docker Compose ist ein Tool zur Definition und Ausführung von Multi-Container-Docker-Anwendungen. Eine Docker Compose-Konfiguration besteht aus einer YAML-Datei, und definiert ein oder mehrere Docker-Container mit den notwendigen Konfigurationen und Netzwerkeinstellungen. Anschließend lassen sich alle Container der Docker Compose-Datei mit einem einzigen Befehl erstellen und starten. Quelltext 2.2 zeigt ein Beispiel einer Docker Compose Datei. Dabei lassen sich Docker-Images aus einer Online Docker-Registry (oben im Quelltext 2.2) oder aus lokalen Dockerfiles (unten im Quelltex 2.2) verwenden.

```
version: '3'

services:
    postgres-database:
        container_name: postgres-database
        image: 'postgres:alpine'
        restart: unless-stopped
        volumes:
          - postgres-db:/var/lib/postgres
        environment:
          POSTGRES_PASSWORD: "changeme1"
          POSTGRES_USER: "postgres"
          POSTGRES_DB: "changeme2"
        ports:
          - "5434:5432"
        expose:
          - "5434"
        networks:
          - service-db-net

    registry-service:
        build: ../ServiceFolder
        container_name: SerivceExample
        ports:
          - "8080:8080"
        networks:
          - service-db-net
```

```
28          depends_on:
29            - postgres-database
30
31  networks:
32      service-db-net
33
34  volumes:
35      postgres-db
```

Quelltext 2.2: Beispiel Docker-Compose

2.3 Cloud Computing

Es gibt eine Reihe von Definitionen zu Cloud Computing, so definiert zum Beispiel das Bundesamt für Sicherheit und Informationstechnik (BSI) Cloud Computing folgendermaßen:

> Cloud Computing ist ein Modell, das es erlaubt bei Bedarf, jederzeit und überall bequem über ein Netz auf einen geteilten Pool von konfigurierbaren Rechnerressourcen (z. B. Netze, Server, Speichersysteme, Anwendungen und Dienste) zuzugreifen, die schnell und mit minimalem Managementaufwand oder geringer Serviceprovider-Interaktion zur Verfügung gestellt werden können [Bundesamt-für-Sicherheit-und-Informationstechnik 2022].

Die wohl bekannteste Definition stammt jedoch vom National Institute Of Standards And Technology (NIST) [Möhring, Keller und Schmidt 2017]:

> Cloud Computing ist ein Modell für den allgegenwärtigen, bequemen und bedarfsgerechten Netzzugang zu einem gemeinsam genutzten Pool konfigurierbarer Computerressourcen (z. B. Netzwerke, Server, Speicher, Anwendungen und Dienste), die schnell und mit minimalem Verwaltungsaufwand bereitgestellt und freigegeben werden können. Eine menschliche Interaktion durch den Dienstanbieter, auch Cloud Service Provider (CSP) genannt, ist im Normalfall nicht notwendig [National-Institute-of-Standards-and-Technology 2022].

Abbildung 2.3 zeigt, dass das Cloud-Modell aus fünf wesentlichen Eigenschaften, drei Service- und vier Bereitstellungsmodellen besteht, welche im Folgenden definiert werden [National-Institute-of-Standards-and-Technology 2022].

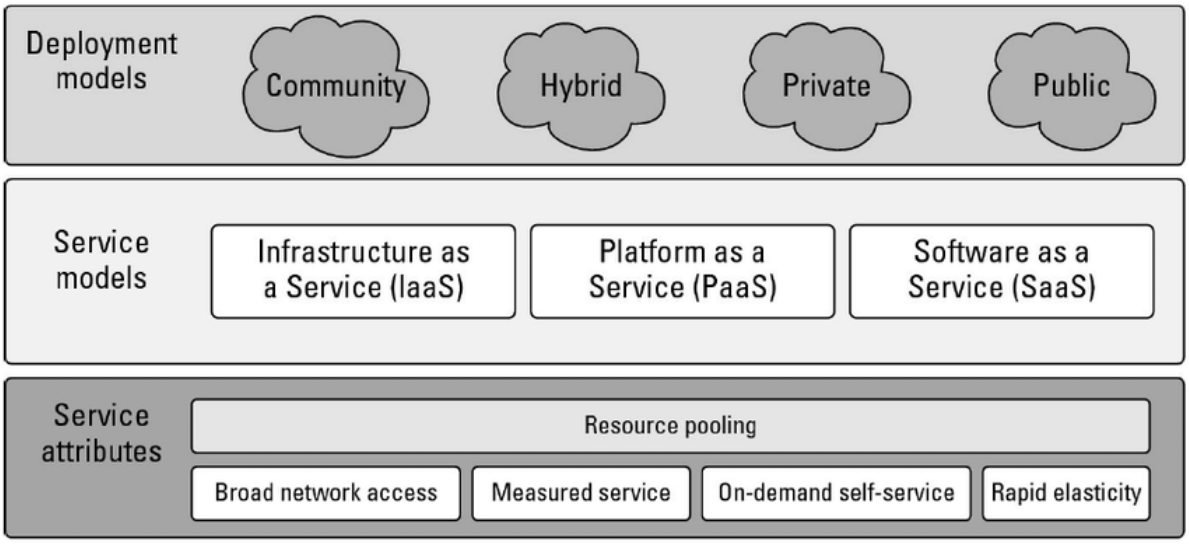


Abbildung 2.3: NIST Cloud Computing Definition [Researchgate 2022]

2.3.1 Eigenschaften

Folgende fünf Eigenschaften charakterisieren laut NIST-Definition eine Cloud-Lösung [National-Institute-of-Standards-and-Technology 2022]:

On-Demand Self Service (Diensterbringung auf Anforderung)

Ein Kunde, auch Cloud Service Customer (CSC) genannt, kann einseitig Rechenkapazitäten wie Serverzeit oder Netzwerkspeicher, je nach Bedarf automatisch bereitstellen, ohne dass ein weiterer Mensch auf der Seite der Dienstanbieter, auch Cloud Service Provider (CSP) genannt, interagieren muss.

Broad Network Access (Umfassender Netzwerkzugriff)

Die Services sind mit Standard-Mechanismen über das Netz verfügbar und nicht an einen bestimmten Client gebunden.

Resource Pooling (Ressourcen Pooling)

Die Rechenressourcen des CSP liegen in einem Pool vor, um mehrere Kunden zu bedienen (Multi-Tenant Modell), wobei verschiedene physische und virtuelle Ressourcen je nach Kundenbedarf dynamisch zugewiesen und neu angelegt werden. Es besteht eine gewisse Standortunabhängigkeit, da der Kunde in der Regel keine Kontrolle oder Kenntnis über den genauen Standort der bereitgestellten Ressourcen hat, aber in der Lage sein kann, den Standort auf einer höheren Abstraktionsebene (z. B. Land, Bundesland oder Rechenzentrum) anzugeben. Beispiele für Ressourcen sind Verarbeitung, Speicher, Arbeitsspeicher und Netzbandbreite.

Rapid Elasticity (Schnelle Elastizität)

Die Services können elastisch bereitgestellt und freigegeben werden, in einigen Fällen auch automatisch, um entsprechend der Nachfrage schnell nach außen und innen zu skalieren. Für den Verbraucher erscheinen die für die Bereitstellung verfügbaren Fähigkeiten oft unbegrenzt und können in beliebiger Menge und zu jeder Zeit angeeignet werden.

Measured Services (Messbare Dienstqualität)

Cloud Service Provider überwachen automatisch die Ressourcennutzung, indem sie eine Messfunktion auf einer für die Art des Dienstes geeigneten Abstraktionsebene nutzen (z. B. Speicher, Verarbeitung, Bandbreite, Zeit, und aktive Benutzerkonten). Die Ressourcennutzung kann überwacht, kontrolliert, gemeldet und abgerechnet werden.

2.3.2 Servicemodelle

Das NIST unterscheidet in drei verschiedene Kategorien von Servicemodellen: SaaS, Platform-as-a-Service (PaaS) und Infrastructure-as-a-Service (IaaS). Die

Servicemodelle beschreiben und definieren die Fertigungstiefe und Verantwortungsaufteilung der bereitgestellten Ressource.
Die Abbildung 2.4 zeigt den Unterschied zwischen der Onsite-IT und der traditionellen IT, mit den verschiedenen Servicemodellen. Die Grafik zeigt, welche Teile eines IT-Systems selbst aufgebaut und betreut werden müssen, sowie welche Verantwortlichkeiten von dem CSP übernommen werden. In der traditionellen IT wird das gesamte System, von der Applikation bis hin zu Daten, Betriebssystem, Server und Networking selbst verwaltet. Wie sich das bei den verschiedenen Servicemodellen ändert wird im Folgenden erläutert.

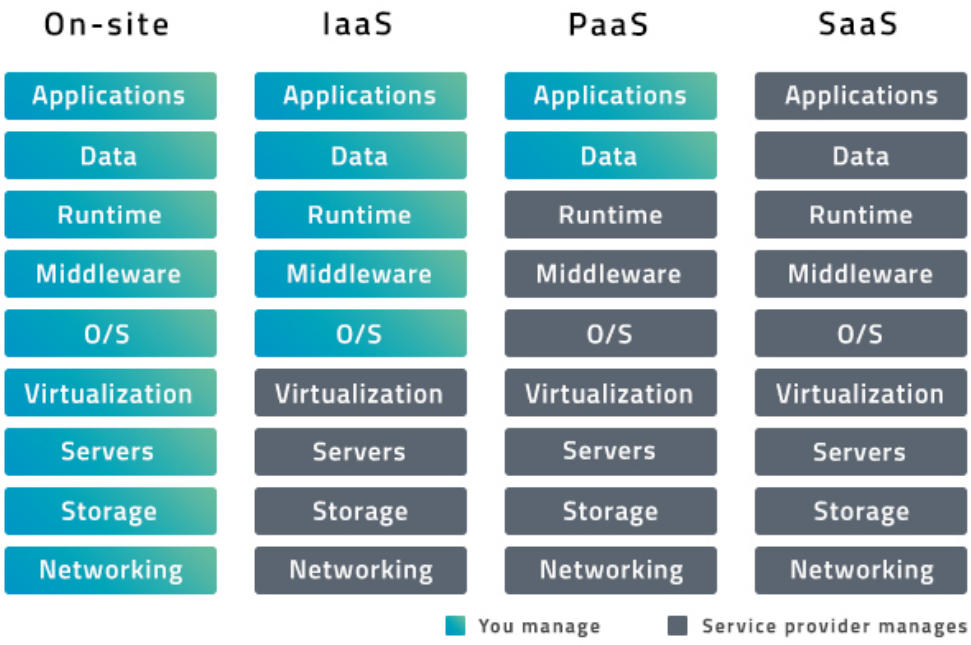


Abbildung 2.4: Vergleich der On-site IT mit den verschiedenen Servicemodellen [group24 2022]

Infrastructure-as-a-Service

Dem CSC wird die Fähigkeit zur Verfügung gestellt, verschiedene und mehrere VMs zu provisionieren. Der CSP bietet in der Regel verschiedene VMs zur Auswahl. Auf einer VM kann der CSC beliebige Software einsetzen und ausführen. Dabei kann der CSC vollständig auf die Ressourcen von der Anwendung bis zum Betriebssystem zugreifen. Die darunterliegende Cloud-Infrastruktur verwaltet der CSP [National-Institute-of-Standards-and-Technology 2022].

Platform-as-a-Service

Der CSC hat die Möglichkeit, in der Cloud-Infrastruktur selbst implementierte Software zu deployen, die von CSC erstellte oder erworbene Software wurden mit Hilfe von Programmiersprachen, Bibliotheken, Diensten und Tools erstellt, die vom CSP unterstützt werden. Der CSC hat keine Kontrolle über das Netzwerk, die Server, die Betriebssysteme oder den Speicher, aber er hat die Kontrolle über die implementierten Anwendungen und möglicherweise über die Konfigurationseinstellungen für die Hosting-Umgebung [National-Institute-of-Standards-and-Technology 2022].

Software-as-a-Service

Der CSC hat die Möglichkeit, die Anwendungen des CSP zu nutzen, die in einer Cloud-Infrastruktur laufen. Der Zugriff auf die Anwendungen erfolgt von verschiedenen Client-Geräten aus, entweder über eine Thin-Client-Schnittstelle, wie einen Webbrowser (z. B. webbasierte E-Mail), oder über eine Programmschnittstelle. Der CSC verwaltet oder kontrolliert nicht die Cloud-Infrastruktur, Netzwerk, Server, Betriebssysteme, Speicher oder sogar einzelne Anwendungsfunktionen. Der CSC stehen nur begrenzte benutzerspezifische Konfigurationseinstellungen zur Verfügung. Die restliche Verantwortung liegt bei dem CSP [Salesforce 2022].

Everything-as-a-Service (XaaS)

Neben denen durch das NIST definierten Servicemodelle haben sich bereits weitere Modelle etabliert [Fraunhofer 2022]. So gibt es eine ganze Reihe an *as a*

Service-Produkte, wie zum Beispiel:

- Function-as-a-Service (FaaS)
- Storage-as-a-Service (STaaS)
- Communications-as-a-Service (CaaS)
- Monitoring-as-a-Service (MaaS)
- Business-Process-as-a-Service (BPaaS)
- Data-as-a-Service (DaaS)
- Security-as-a-Service (SECaaS)
- Identity-as-a-Service (IDaaS)
- Backup-as-a-Service (BaaS)
- Database-as-a-Service (DBaaS)
- Data-Intensive-Computing-as-a-Service (DICaaS)

Zusammenfassen lassen sich diese unter dem Namen Everything-as-a-Service. Die Vielzahl an Modellen beschreibt, dass sich theoretisch alles als Cloud-Dienst vermarkten lässt. Eine genaue Untersuchung, inwiefern sich diese Modelle unterscheiden, ist jedoch für diese Arbeit nicht von Relevanz.

2.3.3 Bereitstellungsmodelle

Das NIST unterscheidet zwischen vier verschiedenen Modellen der Bereitstellung (siehe Abbildung 2.3 die Deployment Modelle). Hierzu zählen die Private Cloud, die Public Cloud, die Community Cloud und die Hybrid Cloud [National-Institute-of-Standards-and-Technology 2022].

Private Cloud

Die Cloud-Infrastruktur wird für die ausschließliche Nutzung durch eine einzige Organisation oder Institution bereitgestellt, die mehrere Verbraucher (z. B. Geschäftseinheiten) umfassen kann. Sie kann im Besitz der Organisation, eines Dritten oder einer Kombination aus beiden sein und wird von dieser verwaltet und betrieben. Die Private Cloud kann On- oder Off-Premise existieren [National-Institute-of-Standards-and-Technology 2022].

Community Cloud

Die Cloud-Infrastruktur wird für die ausschließliche Nutzung durch eine bestimmte Gemeinschaft von Nutzern aus Organisationen bereitgestellt, die gemeinsame Anliegen haben (z. B. Auftrag, Sicherheitsanforderungen, Richtlinien und Compliance). Sie kann im Besitz einer oder mehrerer Organisationen in der Gemeinschaft, einer dritten Partei oder einer Kombination von ihnen sein und von diesen verwaltet und betrieben werden. Die Community Cloud kann wie auch bei der Private Cloud On- oder Off-Premise existieren [National-Institute-of-Standards-and-Technology 2022].

Public Cloud

Die Cloud-Infrastruktur wird zur offenen Nutzung für die Allgemeinheit bereitgestellt. Sie kann im Besitz eines Unternehmens, einer akademischen oder staatlichen Organisation oder einer Kombination dieser Organisationen sein und von diesen verwaltet und betrieben werden. Die Public Cloud befindet sich On-Premises bei dem CSP [National-Institute-of-Standards-and-Technology 2022].

Hybrid Cloud

Die Cloud-Infrastruktur ist eine Zusammensetzung aus zwei oder mehr verschiedenen Cloud-Infrastrukturen (Private, Community oder Public). Diese bleiben zwar eigenständige Einheiten, sind aber durch standardisierte oder proprietäre Technologien miteinander verbunden, welches die von Daten- und Anwendungs-Portabilität ermöglichen (z. B. Cloud Bursting für das Load Balancing zwischen Clouds) [National-Institute-of-Standards-and-Technology 2022].

2.4 Reactive Manifesto

Das Reactive Manifesto (Reaktive Manifest) entstand 2013 aus der Zusammenarbeit der Softwareentwickler Jonas Bonér, Dave Farley, Roland Kuhn und Martin Thompson. Seither haben rund 32.000 Entwickler das Manifest unterschrieben und fast alle Publikationen zum Thema Reactive beziehen sich darauf. Es definiert Werte, nicht-funktionale Anforderungen und Prinzipien, die für eine reaktive Architektur stehen (siehe Abbildung 2.5) [Bonér u. a. 2014] [Schäfer 2022]. Das Reactive Manifesto trägt zusammen, wie moderne Softwaresysteme aufgebaut sein müssen, um als reaktiv zu gelten. So sollen Softwaresysteme „stets responsive (antwortbereit), resilient (widerstandsfähig), elastic (elastisch) und message-driven (nachrichtenorientiert) sein“ [Bonér u. a. 2014].

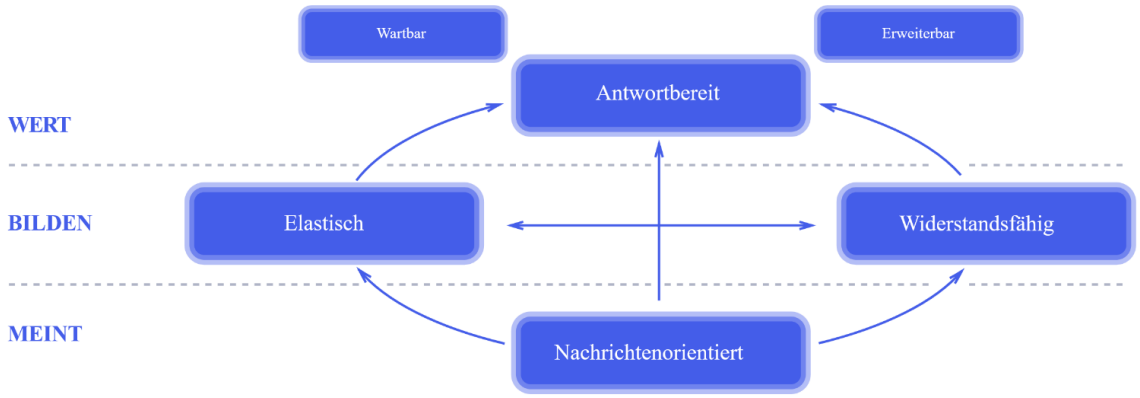


Abbildung 2.5: Reactive Manifesto [Bonér u. a. 2014]

Zudem behaupten die Autoren, dass Computersysteme, die nach diesen Anforderungen entwickelt werden, sich als anpassungsfähiger erweisen, mit weniger starr gekoppelten Komponenten und in jeder Hinsicht skalierbarer seien. Sie seien einfacher weiterzuentwickeln und zu verändern. Sie reagieren zuverlässiger und eleganter auf Fehler und vermeiden so Ausfälle. Reaktive Systeme bereiten dem Benutzer durch ihre fortwährende Antwortbereitschaft eine interaktive und höchst

befriedigende Erfahrung [Bonér u. a. 2014].

Abbildung 2.5 zeigt die Beziehung zwischen den Eigenschaften des Reactive Manifesto und beschreiben reaktive Systeme. Diese Eigenschaften werden im Folgenden erläutert:

2.4.1 Responsive

Antwortbereitschaft ist die Grundlage für Funktion und Benutzbarkeit eines Systems, deshalb ist fortwährende Antwortbereitschaft das oberste Ziel von reaktiven Systemen. Das System antwortet unter allen Umständen zeitgerecht, solange dies überhaupt möglich ist. Wichtig ist es zu beachten, dass Fehler in verteilten Systemen nur durch die Abwesenheit einer Antwort sicher festgestellt werden können. Ohne vereinbarte Antwortzeitgrenzen ist die Erkennung und Behandlung von Fehlern nicht möglich. Außerdem stiften konsistente Antwortzeiten Qualität und Vertrauen, so dass weitere Interaktion gefördert werden kann [Bonér u. a. 2014].

2.4.2 Resilient

Das System bleibt auch im Falle eines Ausfalls reaktionsfähig. Dies gilt nicht nur für hochverfügbare, unternehmenskritische Systeme - jedes System, das nicht widerstandsfähig ist, wird nach einem Ausfall nicht mehr reagieren können. Die Ausfallsicherheit wird durch Replikation, Eingrenzung, Isolierung und Delegation erreicht. Ausfälle werden in jeder Komponente getrennt eingedämmt, indem die Komponenten voneinander isoliert werden und so sichergestellt wird, dass wenn Teile des Systems ausfallen, diese unabhängig voneinander wiederhergestellt werden können, ohne das System als Ganzes zu gefährden. Die Hochverfügbarkeit wird durch Replikation sichergestellt. Der Client einer Komponente muss sich nicht um Ausfälle der Komponente befassen [Bonér u. a. 2014].

2.4.3 Elastic

Das System bleibt auch bei wechselnder Arbeitsbelastung reaktionsfähig. Reaktive Systeme können auf Änderungen der Input-Rate reagieren, indem sie Ressourcen erhöhen oder verringern. Dies setzt voraus, dass die Softwarearchitektur so umgesetzt wurde, dass das System effizient skalieren kann. Reaktive Systeme unterstützen sowohl vorausschauende als auch reaktive Skalierungsalgorithmen, um auf akute Änderungen der Input-Rate reagieren zu können. Sie erreichen Elastizität auf kostengünstige Weise auf handelsüblichen Hard- und Softwareplattformen [Bonér u. a. 2014].

2.4.4 Message Driven

Das System verwendet asynchrone Nachrichtenübermittlung zwischen seinen Komponenten. Dies stellt die Entkopplung, Isolation und Standorttransparenz sicher. Die explizite Weitergabe von Nachrichten ermöglicht Lastmanagement, Elastizität und Flusskontrolle, da Nachrichtenwarteschlangen im System gesteuert, angepasst und überwacht werden können [Bonér u. a. 2014].

Häufig wird dafür eine Message oriented Middleware (MoM) eingesetzt, welche als Empfänger, Sender und Speicher für Nachrichten agiert. Zudem ist es MoMs möglich, Nachrichten zu persistieren und für den Fall eines Service-Ausfalls die Nachrichtenübertragung zu garantieren. Nicht-blockierende, nachrichtenorientierte Systeme erlauben eine effiziente Verwendung von Ressourcen, da Komponenten beim Ausbleiben von Nachrichten vollständig inaktiv bleiben können.

2.5 Skalierbarkeit

> Allgemein wird Skalierung oder auch Skalierbarkeit als die Fähigkeit eines Systems, Netzwerks oder Prozesses verstanden, seine Größe zu verändern, bzw. zu wachsen. In der Informationstechnologie ist Skalierbarkeit die Vergrößerung von Rechenleistung durch Hinzufügen von weiteren Ressourcen [Cloudogu 2022].

Grundsätzlich wird zwischen vertikaler und horizontaler Skalierbarkeit unterschieden:

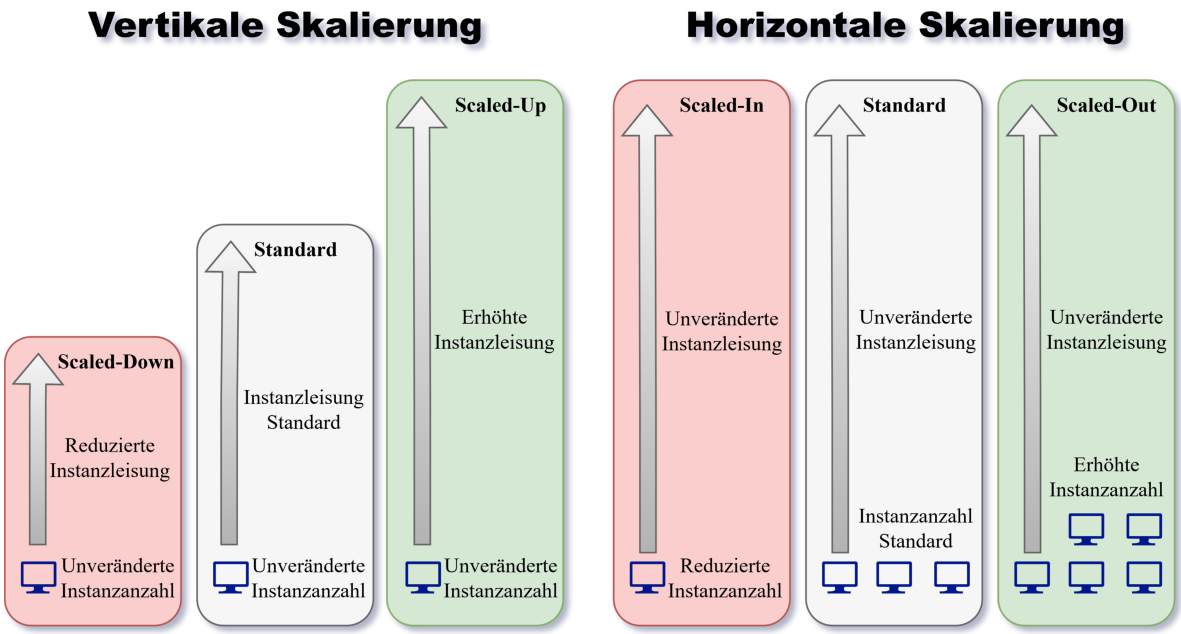


Abbildung 2.6: Vertikale (links) und horizontale (rechts) Skalierung

2.5.1 Vertikale Skalierbarkeit

Abbildung 2.6 zeigt auf der linken Seite die vertikale Skalierung. Werden innerhalb einer logischen Einheit Ressourcen hinzugefügt, so liegt eine vertikale Skalierbarkeit (Scale-Up) vor. Das bedeutet, dass die Leistungssteigerung durch das Hinzufügen von Ressourcen zu einer Instanz oder einer Maschine innerhalb des Systems entsteht. Die Leistungssenkung der vertikalen Skalierung (Scale-Down) geschieht durch das Entfernen oder das Reduzieren von Ressourcen. Die Anzahl der Maschinen oder Instanzen bleibt dabei unverändert.

Beispiele

- Das Vergrößern von Speicherplatz,
- Das Hinzufügen oder Entfernen einer CPU,
- Auf- oder Abrüstung von Arbeitsspeicher,
- Der Ein- oder Ausbau einer Grafikkarte.

2.5.2 Horizontale Skalierbarkeit

Abbildung 2.6 zeigt auf der rechten Seite die horizontale Skalierung. Bei der horizontalen Skalierung bleibt Instanz- oder Maschinenleistung unverändert. Denn eine Leistungssteigerung der horizontalen Skalierung (Scale-Out) wird durch die Erhöhung der Instanz- oder Maschinenanzahl erzielt. Dennoch hängt die Effizienz dieser Skalierung stark von der Implementierung der Applikation ab, da sich nicht jede Software gleich gut parallelisieren lässt. Das Reduzieren der Instanz- oder Maschinenanzahl nennt sich Scale-In. Anders als bei der vertikalen Skalierbarkeit ist jedoch eine Koordination durch einen Load-Balancer (Lastverteiler) notwendig.

2.6 Event-Driven Architecture

> Event-Driven Architecture (EDA) repräsentieren einen neuen Stil von Unternehmensanwendungen, bei dem Ereignisse in das Zentrum der Softwarearchitektur rücken – Ereignisorientierung als Architekturstil [Bruns und Dunkel 2010].

Das bedeutet, dass bei dem Architekturentwurf zuerst die Kommunikation und die Ereignisse analysiert werden. Anhand dieser Informationen wird der Rest des Systems entwickelt. Dies ermöglicht es, die Abhängigkeiten zwischen Komponenten gut zu erkennen, um eine Architektur mit loser Kopplung zu erstellen [Bruns und Dunkel 2010].

2.6.1 Event-Driven Services

Besteht eine EDA aus mehreren Services, spricht man von Event-Driven Services (EDS). Dieses Architekturprinzip erweitert EDA um die Microservices-Architektur. Bei EDS wird die Softwarearchitektur durch mehrere Microservices gelöst. Da EDA gut geeignet ist, Services mit loser Kopplung und einer wohldefinierten Aufgabe (starke Kohäsion) zu definieren, ergänzen sich die Ansätze.

2.7 Event-Driven Reactive Services

Wie in Abbildung 2.7 dargestellt, kombiniert Event-Driven Reactive Services (EDRS) die Ansätze des Reactive Manifesto und der EDS. EDRS sollten damit alle Anforderungen des Reactive Manifesto erfüllen, sowie gleichzeitig eine ereignisgesteuerte Architektur besitzen.

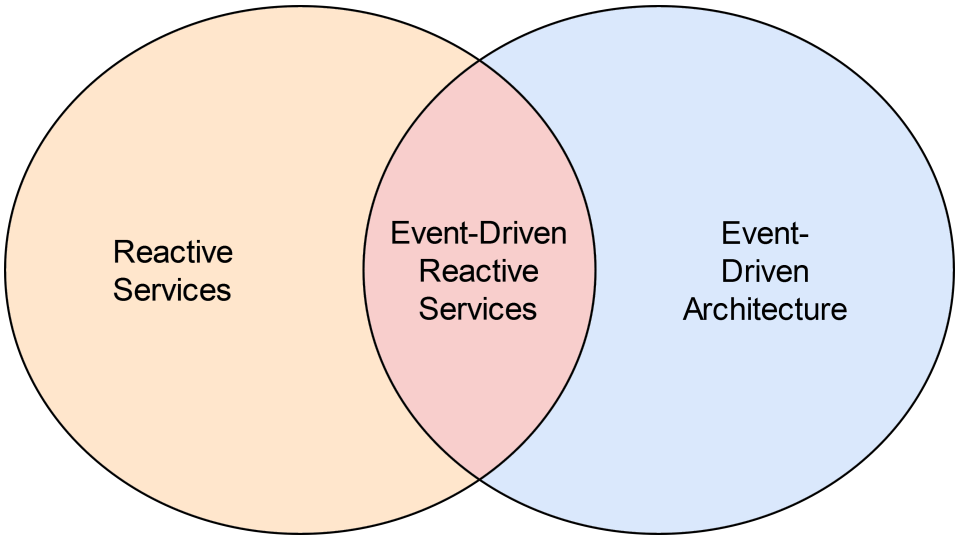


Abbildung 2.7: Event-Driven Reactive Services [Schiborr u. a. 2021]

3 Analyse

Dieses Kapitel widmet sich der Charakterisierung und Analyse der verschiedenen Verantwortlichkeiten und Anforderungen der Akteure und der Plattform. Anhand dieser Analyse können technische und fachliche Anforderungen für die Plattform definiert und erläutert werden.

3.1 Akteuranalyse

Akteure sind Personen, Gruppen, Privatpersonen oder Unternehmen, die Einfluss auf die Anforderungen an ein System haben. Für den Cloud-Marketplace lässt sich das, neben dem Plattformbetreiber, auf zwei wichtige Akteure zusammenfassen, den Service-Provider und den Service-Consumer (siehe Abbildung 1.1).

3.1.1 Service-Provider

Bei der Interaktion mit einem Cloud-Marketplace ist das erfolgreiche Vermarkten und Verkaufen der Produkte das größte Anstreben. In diesem Anwendungsfall bedeutet das konkret, das zur Verfügung stellen von SaaS-Produkten.

Use-Cases

Die Use-Cases (Anwendungsfälle) lassen sich, unter anderem, durch die Beobachtung des gesamten Lebenszyklus einer SaaS-Instanz erheben. Amazon AWS hat eine Auflistung der Aufgaben des Service-Providers für den AWS-Marketplace erstellt [Amazon-AWS 2022c]. In Abbildung 3.1 ist dargestellt, welche Aufgaben der Service-Provider im AWS-Marketplace für eine erfolgreiche Vermarktung zu erledigen hat.

Im AWS-Marketplace lassen sich SaaS-Produkte nur im Bereich des Service-Consumer oder des Service-Providers der Public Cloud provisionieren. Es muss

Abbildung 3.1: Aufgaben des Service-Providers für die AWS-Marketplace [Amazon-AWS 2022a]

ein Standard für den Lebenszyklus der SaaS-Produkte existieren. Zudem ist eine klare Definition, wie aus einer Service-Class eine Service-Instanz erstellt wird, notwendig. In welchem Cloud Userspace und unter welchen Benutzerrechten die Instanz bereitgestellt wird sollte auch Definiert sein [Valdes 2012]. [Gohad, Ponnalagu und Narendra 2012]. Die Instanz muss anschließend noch konfiguriert, upgedatet und deprovisioniert werden können. Zudem muss die Plattform ein Interface anbieten mit dem das Software-Produkt registriert werden kann.

Abbildung 3.2 zeigt eine Auflistung der Interaktionen zwischen Service-Provider und Plattform. Der Service-Provider muss auch eine Reihe an Funktionen um den Lebenszyklus einer Service-Instanz anbieten. Dazu zählen das Provisionieren, das Deprovisionieren und das Konfigurieren von verschiedenen Service-Instanzen des SaaS-Produktes. Dem Service-Consumer sollte es durch den Cloud-Marketplace möglich sein, nach der Cloud-Service-Definition *On-Demand Self-Service*, automatisch SaaS-Produkte zu erstellen, ohne dass dazu eine Interaktion mit dem Service-Provider notwendig ist. Außerdem muss der Service-Provider sein Produkt so implementieren, dass es Service-Bindings erstellen kann. Ein Service-Binding sind Zugriffsdaten, wie Benutzername, Passwort oder Uniform Resource Locator (URL) für ein SaaS-Produkt. Des Weiteren muss der Service Broker jederzeit in der Lage sein eine aktuelle Auflistung der Produktinformationen, sowie die verschiedene Preismodelle bereitzustellen.

Die Interaktion des Service-Providers mit dem Cloud-Marketplace beginnt mit der Registrierung und Anmeldung als Service-Provider auf der Plattform. Um ein Produkt zu verkaufen, muss der Service-Provider einen Service Broker für die SaaS-Produkte entwickeln und betreiben. Dieser Service Broker kann anhand der Service-Class die Service-Instanzen erstellen und bekommt die Aufträge für die Provisionierung, Deprovisionierung und für die Konfiguration von Instanzen. Der Service-Provider stellt diesen Service Broker bei der Registrierung eines SaaS-Produktes zu Verfügung und kann diese zu einem späteren Zeitpunkt noch konfigurieren. Auch das SaaS-Produkt kann der Service-Provider zu einem spä-

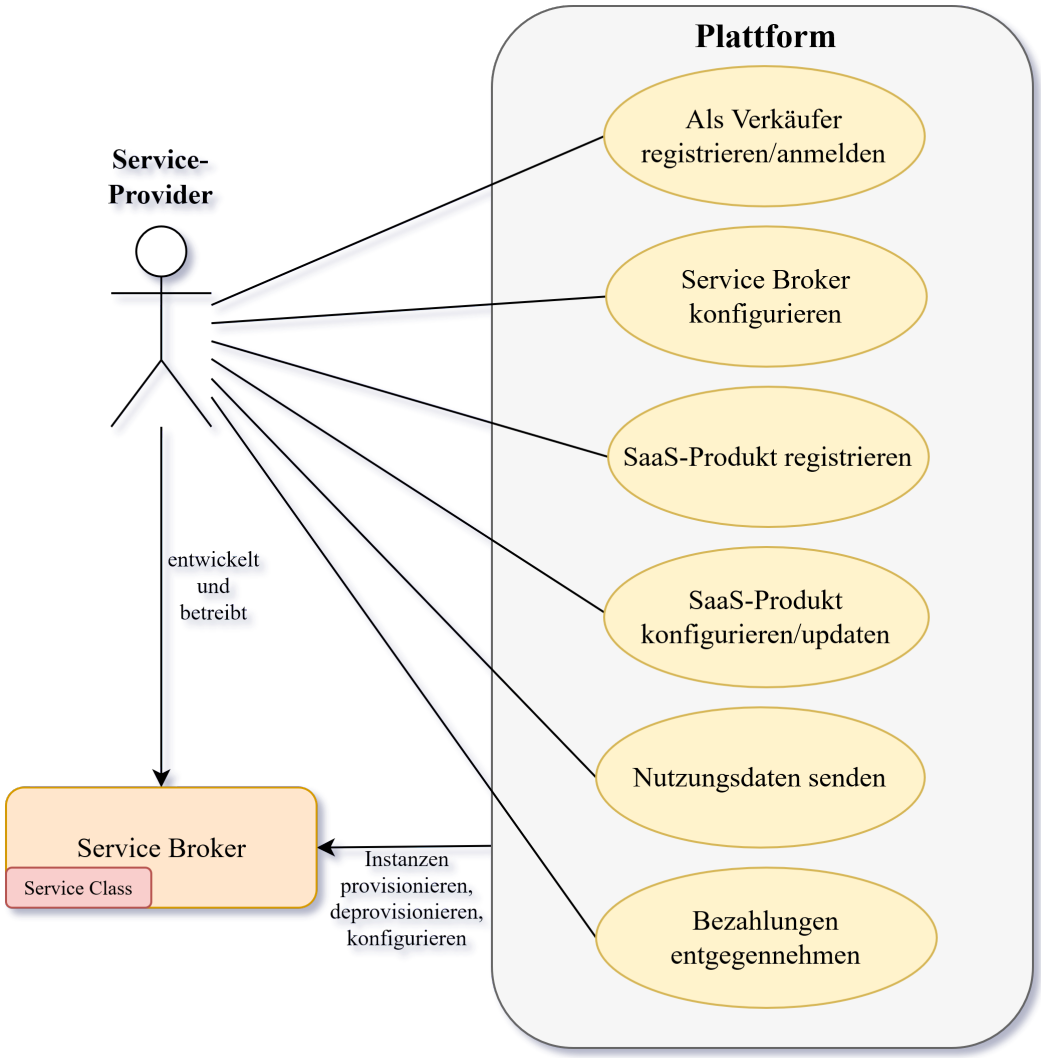


Abbildung 3.2: Use-Case Diagramm des Service-Providers

teren Zeitpunkt updaten und konfigurieren, wie zum Beispiel eine Änderung des Preises, des Produktumfanges oder der Produktinformation. Für eine korrekte Abrechnung der Instanznutzung senden die Instanzen die Nutzungsdaten an die Plattform. Zur SaaS-Produkt-Konfiguration gehört zudem auch das Aktivieren und Deaktivieren des SaaS-Produktes für den Cloud-Marketplace. So kann der Service-Provider zu einem beliebigen Zeitpunkt das SaaS-Produkt für den Cloud-Marketplace freischalten oder wieder deaktivieren.

Anforderungen an die Plattform

Die Service-Provider erwarten in erster Linie die Erfüllung der Use-Cases von Abbildung 3.2. Dazu existiert das Interesse von dem bereits vorhandenen Kundenstamm der Plattform zu profitieren. Durch die Verwendung von Funktionen der Plattform, wie zum Beispiel die Verwendung der Kundendatenverwaltung, das Monitoring oder die Verwendung des Bezahlvorgangs, kann der Service-Provider Entwicklungsaufwand einsparen. Um ein effizientes Vermarkten des SaaS-Produktes zu garantieren, braucht es ein intuitiv nutzbares und geführtes Onboarding durch das Frontend der Plattform.

Verantwortlichkeiten

Der Service-Provider hat das SaaS-Produkt so zu gestalten, wie der Cloud-Marketplace es vorgibt, da es dem Cloud-Marketplace sonst nicht möglich ist, das Produkt zu provisionieren. Der Service-Provider ist also auch zu einer standardisierten API für die Kommunikation zwischen der Plattform und dem Service Broker gebunden. Zudem ist der Service-Provider dafür verantwortlich, die Instanzen, sowie vor allem personenbezogene Daten, vor fremden Zugriff zu schützen.

3.1.2 Service-Consumer

Der Service-Consumer gilt als Kunde der Plattform und es sollte ein großes Bestreben sein, dem Service-Consumer eine gute User Experience (UX, Nutzungserfahrung) und ein gutes Vertrauen in dem Cloud-Marketplace zu bieten.

Use-Cases

Die Use-Cases des Service-Consumers lassen sich ebenso durch die Beobachtung des gesamten Lebenszyklus einer Provisionierung erheben [Amazon-AWS 2022b]. Der Umfang der Use-Cases für den Service-Consumer hängt stark von dem Umfang an zusätzlichen Funktionen ab, die von der Plattform angeboten werden.

Abbildung 3.3 zeigt eine Auflistung der Interaktionen zwischen Service-Consumer und Plattform. Der Service-Consumer benötigt zu Beginn eine Möglichkeit sich ein Benutzerprofil zu erstellen und sich als ein Benutzer anzumelden. Jedoch sollte es auch möglich sein, ohne eine Registrierung, die verfügbaren SaaS-Produkte

des Cloud-Marketplaces zu sehen. Um ein SaaS-Produkt zu buchen, muss der Service-Consumer jedoch auf der Plattform angemeldet sein. Auch für die restlichen Use-Cases muss der Service-Consumer angemeldet sein. Dann ist es dem Service-Consumer erlaubt, gebuchte SaaS-Produkte anzuzeigen, zu bezahlen, zu nutzen oder zu löschen.

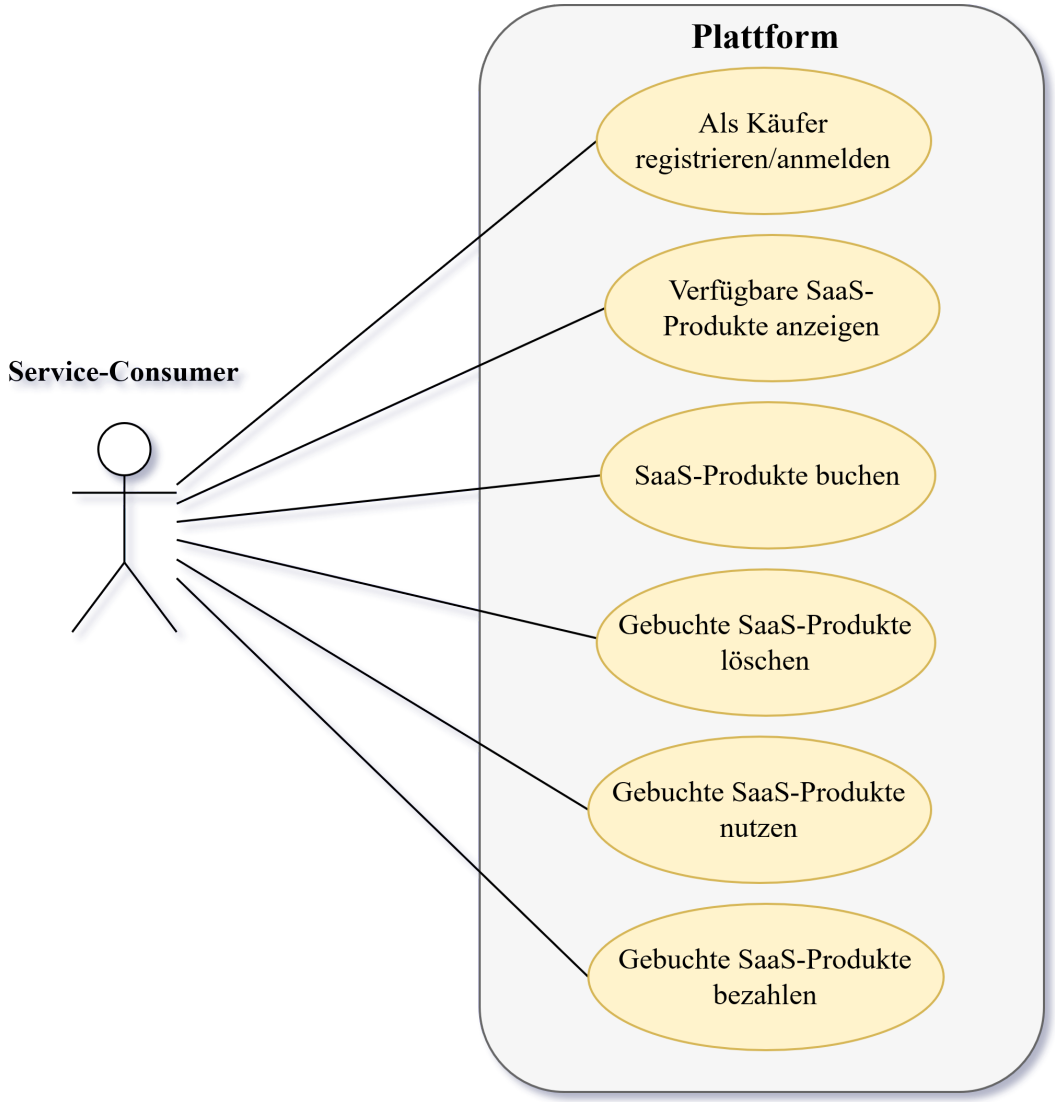


Abbildung 3.3: Use-Case Diagramm des Service-Consumers

Zusätzliche Use-Cases können durch zusätzliche optionale Funktionen der Platt-

form entstehen. Bietet die Plattform zum Beispiel eine umfangreiche grafische Überwachung der Instanzen an, dann wäre das Einsehen oder das Erstellen von Nutzungsstatistiken ein weiterer Use-Case.

Anforderungen an die Plattform

Die Service-Consumer erwarten ebenso, durch eine gut gestaltete Website, eine benutzerfreundliche Umsetzung der Use-Cases. Der Erfolg der Plattform hängt mit einem positiven Nutzererlebnis zusammen. Weitere Anforderungen an die Plattform ist eine transparente und sichere Kostenabwicklung sowie eine durchgehende Nutzbarkeit und Zugänglichkeit der Plattform.

Verantwortlichkeiten

Der Service-Consumer agiert als Kunde und muss sich als solcher ein Benutzerkonto anlegen und für alle Aktionen eingeloggt sein. Selbstverständlich muss der Service-Consumer für kostenpflichtige SaaS-Produkte bezahlen.

3.2 Plattformanalyse

In dieser Arbeit treten verschiedene Begrifflichkeiten im Zusammenhang mit der Plattform auf. Darunter zählen der Cloud-Marketplace, der Plattformbetreiber und die Plattform selbst. Für ein eindeutiges Verständnis der Begrifflichkeiten werden diese im Folgenden definiert und differenziert.

3.2.1 Begrifflichkeiten

Eine Plattform ist im Allgemeinen eine digitale Landingpage (eine speziell eingerichtete Webseite), welche für das Online-Marketing eingesetzt wird. Demnach ist eine Plattform eine Internetseite, die einen online verfügbaren Marktplatz abbildet. Auf diesem Marktplatz treffen, ähnlich wie bei einem realen Markt, Angebot und Nachfrage aufeinander. Es können unterschiedlichste Dienstleistungen oder Produkte angeboten werden.

Der in dieser Arbeit zu konzipierende Cloud-Marketplace ist eine spezielle Form

einer Plattform. Der Cloud-Marketplace dient speziell für den Verkauf von digitalen Produkten in Form von Servicemodellen, wie in diesem Fall SaaS-Produkte, welche in einer Cloud bereitgestellt werden können. Zusätzlich ist der Cloud-Marketplace, in den meisten Fällen der bestehenden Lösungen (wie beispielsweise bei Microsoft, Google oder Amazon), Teil eines Cloud-Providers. Ein Cloud-Marketplace kann dem Service-Provider und dem Service-Consumer zusätzlich verschiedene Funktionalitäten oder Servicemodelle anbieten. Der Cloud-Marketplace von bestehenden Public Cloud-Providern bietet außerdem eine Reihe an nützlichen Funktionen und Produkte, die die Transformation und die Bereitstellung von Cloud-Lösungen für Unternehmens- und Privatanwendungen beschleunigt oder vereinfacht.

Da in dieser Arbeit ein Cloud-Marketplace konzeptioniert wird, findet eine Analyse der allgemeinen Plattform und zusätzlich des spezielleren Cloud-Marketplaces statt. Zum Zeitpunkt der Anforderungserhebung des Cloud-Marketplaces wird der Fokus auf den zu erstellenden Cloud-Marketplace gerichtet. Um die Begrifflichkeiten eindeutig zu verstehen wird im Folgenden auf weitere Synonyme zur Plattform oder zum Cloud-Marketplace, wie beispielsweise das System, die Applikation, der Marketplace oder die Software, verzichtet.

3.2.2 Anforderungen und Verantwortlichkeiten

Die Anforderungen und Verantwortlichkeiten lassen sich direkt aus der Akteuranalyse des Service-Providers und Service-Consumers erheben. Die Plattform muss alle Use-Cases der beiden Akteure erfüllen. Außerdem ist es aus der Perspektive des Plattformbetreibers wichtig, dass die Plattform eine gute User Experience (UX) für alle Nutzer bietet, um gegenüber vergleichbaren Plattformen konkurrenzfähig zu sein. Zur Konkurrenzfähigkeit gehört ebenso, dass die Plattform Responsive, Resilient und Elastic nach den Grundsätzen des Reaktive Manifesto entwickelt ist. So ist die Plattform stets erreichbar und kann eine gute UX bieten.

3.2.3 Querschnittaufgaben

Um die Plattform attraktiv für den Service-Consumer und den Service Provider zu gestalten, sollte die Plattform neben einem guten User Interface (UI) und UX einige weitere Funktionen anbieten. Diese Funktionen heißen Querschnittsaufgaben. Die wichtigste Aufgabe neben dem direkten Handel der SaaS-Produkte ist eine sichere Lösung eines Identity and Access Management (IAM), welches das

Authentisieren von Nutzern ermöglicht. Das IAM sollte auch eine API für den Service-Provider anbieten, damit besitzen Service-Provider die Funktion, Nutzer für ihre SaaS-Produkte zu authentifizieren. Ebenso ist die Erhebung von Nutzungsdaten und die anschließende Kostenabrechnung eine wichtige Funktion. Weitere optionale Features für den Cloud-Marketplace sind das Überwachen der Servicenutzung, Benachrichtigungsdienste, Buchungshistorien und eine Kostenansicht.

4 Anforderungen

Aus der ausgeführten Analyse lassen sich nun Anforderungen unter der Berücksichtigung der Interessen der Akteure und der Use-Cases definieren. Diese sind nach funktionalen Anforderungen, nicht-funktionalen Anforderungen und betriebliche Anforderungen durch das LGLN gegliedert. Im Folgenden wird ein Anforderungskatalog für den Prototyp des Cloud-Marketplaces erstellt und zur Verfolgbarkeit durchnummeriert.

Funktionale Anforderungen beschreiben die zu erfüllende Eigenschaft oder zu erbringende Leistung des Softwareproduktes. Da es sich in dieser Arbeit um eine Konzeptionierung und prototypische Umsetzung handelt, bestehen die funktionalen Anforderungen überwiegend aus den grundlegenden Use-Cases aus Kapitel 3. Nicht-funktionale Anforderungen sind Qualitäts- und Leistungsanforderungen an das System, sowie weitere Randbedingungen unter denen die Funktionalität zu erbringen ist. Da es sich in diesem Rahmen um eine Konzeptionierung und prototypische Umsetzung handelt, wurde noch keine umfassende Qualitätsanalyse durchgeführt. Im Folgenden werden Qualitätsziele angenommen, welche ein widerstandsfähiges System charakterisieren.

Da dieser Cloud-Marketplace für das LGLN konzeptioniert und entwickelt wird, spielen die betrieblichen Anforderungen durch das LGLN eine wichtige Rolle. Diese Anforderungen sind teilweise nicht abstrakt und beinhalten direkte Technologie- und Funktionswünsche. Deshalb werden diese Anforderungen getrennt zu den funktionalen und nicht-funktionalen Anforderungen gesammelt, auch wenn diese sich teilweise dort einordnen lassen könnten.

4.1 Umfang

Um den Umfang an den zeitlichen Rahmen dieser Arbeit anzupassen, wird sich im Folgenden auf die kritischen Use-Cases und Softwarekomponenten des Backends konzentriert. Dadurch werden Aufgaben wie die Konzeptionierung und

Implementierung einer Webseite, untergeordnete Funktionen (wie beispielsweise die Kostenabrechnung), sowie andere optionale Funktionen und Use-Cases, nicht weiter behandelt. Zudem genügt für das LGLN die Bereitstellung der Software als Sammlung von Containern in einer Docker Compose-Datei.

4.2 Funktionale Anforderungen

1 a) Registrierung und Anmeldung von Service-Consumer und Service-Provider

Service-Consumer und Service-Provider können sich über eine Webseite ein Nutzerprofil anlegen und anschließend als Nutzer anmelden.

1 b) Registrierung von Service Brokern und SaaS-Produkten

Authentisierte Service-Provider können Service Broker und SaaS-Produkte im Cloud-Marketplace registrieren.

1 c) Konfigurieren der eigenen Service Broker und SaaS-Produkte

Authentisierte Service-Provider können ihre eigenen registrierten Service Broker und SaaS-Produkte konfigurieren und updaten.

1 d) Aktivieren und Löschen von Service Brokern und SaaS-Produkte

Authentisierte Service-Provider können ihre eigenen registrierten Service Broker und SaaS-Produkte aktivieren und löschen.

1 e) Verfügbare SaaS-Produkte anzeigen

Der Cloud-Marketplace-Produktkatalog mit allen verfügbaren SaaS-Produkten sollte frei, auch für anonyme Nutzer, zur Verfügung stehen.

1 f) Verfügbare SaaS-Produkte provisionieren

Authentisierte Service-Consumer können verfügbare SaaS-Produkte provisionieren.

1 g) Status der Service-Provisionierung einsehen

Authentisierte Service-Consumer können den Provisionierungsstatus der eigenen Service-Instanzen einsehen.

1 h) Eigene Service-Instanzen anzeigen und deprovisionieren

Authentisierte Service-Consumer können ihre eigenen Service-Instanzen einsehen und bei Bedarf deprovisionieren.

1 i) Nutzungsdaten der Instanzen entgegennehmen

Der Cloud-Marketplace bietet eine Schnittstelle für eine Übertragung der Nutzungsdaten einzelner Instanzen.

1 j) Nutzungsdaten der eigenen Instanzen einsehen

Authentisierte Service-Consumer können die Nutzung ihrer eigenen Instanzen einsehen.

4.3 Nicht-funktionale Anforderungen

2 a) Skalierbarkeit

Die Softwarekomponenten unterstützt vertikales und horizontales Skalieren.

2 b) Elastizität

Das System bleibt auch bei wechselnder Arbeitsbelastung reaktionsfähig. Softwarekomponenten mit höherer Last können getrennt skaliert werden.

2 c) Zeitverhalten

Das System antwortet unter allen Umständen zeitgerecht, spätestens nach zwei Sekunden.

2 d) Resilienz

Das System bleibt auch bei einem Ausfall einzelner Service-Instanzen reaktionsfähig.

2 e) Sicherheit

Der Cloud-Marketplace ist nur über eine wohldefinierte API erreichbar. Der Cloud-Marketplace blockiert nach drei Fehlversuchen die Anmeldung für 60 Sekunden.

2 f) Wartbarkeit

Der Cloud-Marketplace ist in kleinere modulare Softwarekomponenten gegliedert, dies macht die Software wartbar und erweiterbar.

4.4 Betriebliche Anforderungen durch das LGLN

3 a) Open Source-Technologien

Das LGLN hat das Interesse den Einsatz von Open Source-Software zu bevorzugen. Dies ermöglicht ein hohes Maß an Flexibilität für individuelle Anpassungen und liefert die Transparenz und die Überprüfbarkeit der Software.

3 b) Lose Kopplung der Softwarekomponenten

Die zu entstehenden Softwarekomponenten des Cloud-Marketplaces sollten eine lose Kopplung zueinander besitzen.

3 c) Technologie-Stack des LGLN

Technologieentscheidungen für die Softwarekomponenten bevorzugen bei vergleichbaren Technologien die bisher eingesetzten Technologien des LGLN.

3 d) Spezieller Datenschutz

Die Datenschutzrichtlinien für personenbezogene Daten müssen eingehalten werden. Eine Public Cloud kann dann für die Verarbeitung von personenbezogenen Daten verwendet werden, wenn die gesetzlichen Rahmenbedingungen erfüllt werden.

3 e) Cloudfähigkeit

Die Softwarekomponenten sollen Cloud Native implementiert werden, da der Cloud-Marketplace zu einem späteren Zeitpunkt in einer Private Cloud deployt wird.

3 f) Überwachung und Persistenz der Events

Eingehende und interne Events können überwacht werden. Zusätzlich werden Events persistiert, um zu einem späteren Zeitpunkt eine Analyse ausführen zu können.

5 Entwurf der Architektur

Dieses Kapitel widmet sich der Dokumentation, Kommunikation und dem Entwurf der Software- und Systemarchitektur der von mir entwickelten Lösung für einen Cloud-Marketplace. Zudem wird dieses Kapitel nach der arc42 Methodik [Hruschka u. a. 2022] umgesetzt. Die Methodik arc42 liefert eine Vorlage für eine systematische und prozessorientierte Dokumentation, Kommunikation und Entwurf von Software- und Systemarchitekturen [Hruschka u. a. 2022]. Sie fokussiert sich auf eine verständliche und einheitliche Dokumentation der Architektur und definiert den Inhalt sowie den Umfang der einzelnen Kapitel.

Die ersten beiden Kapitel des arc42 Templates („Einführung und Ziele“ und „Randbedingungen“), werden hier nicht wiederholt, da der Inhalt bereits in Kapitel 1, 3 und 4 ausführlich behandelt wurde.

Im Folgenden werden eine Kontextabgrenzung, eine Lösungsstrategie, eine statische Bausteinsicht und deren dynamisches Pendant (die Laufzeitsicht) erläutert. Im Anschluss folgen die Kapitel „Verteilungssicht“, „Querschnittliche Konzepte“, „Architekturentscheidungen“, „Qualitätsanforderungen“ und zum Abschluss folgt eine Einschätzung der „Risiken und technischen Schulden“.

5.1 Kontextabgrenzung

Die Kontextabgrenzung grenzt das System von allen Kommunikationsbeziehungen (Nachbarsystemen und Benutzerrollen) ab. Sie legt damit die externen Schnittstellen fest [Hruschka u. a. 2022].

Das Kapitel „Technischer Kontext“, welches technische Kontextabgrenzungen zu weiteren technischen Schnittstellen, wie Kanälen, Protokollen und Hardware erläutert, wird im Folgenden nicht ausgeführt, da der Prototyp keine Schnittstellen zu weiteren technischen Systemen besitzt.

5.1.1 Fachlicher Kontext

In Abbildung 5.1 sind die fachlichen Ein- und Ausgaben des Cloud-Marketplaces dargestellt. Diese Ein- und Ausgaben lassen sich auf die Use-Cases von Abbildung 3.2 und 3.3 zurückführen. So interagiert der Service-Consumer wie im Kapitel 3.1.2 definiert. Der Service-Provider stellt, zusätzlich zu den Use-Cases, ebenfalls Funktionen über eine definierte Schnittstelle, wie im Kapitel 3.1.1 beschrieben, zur Verfügung.

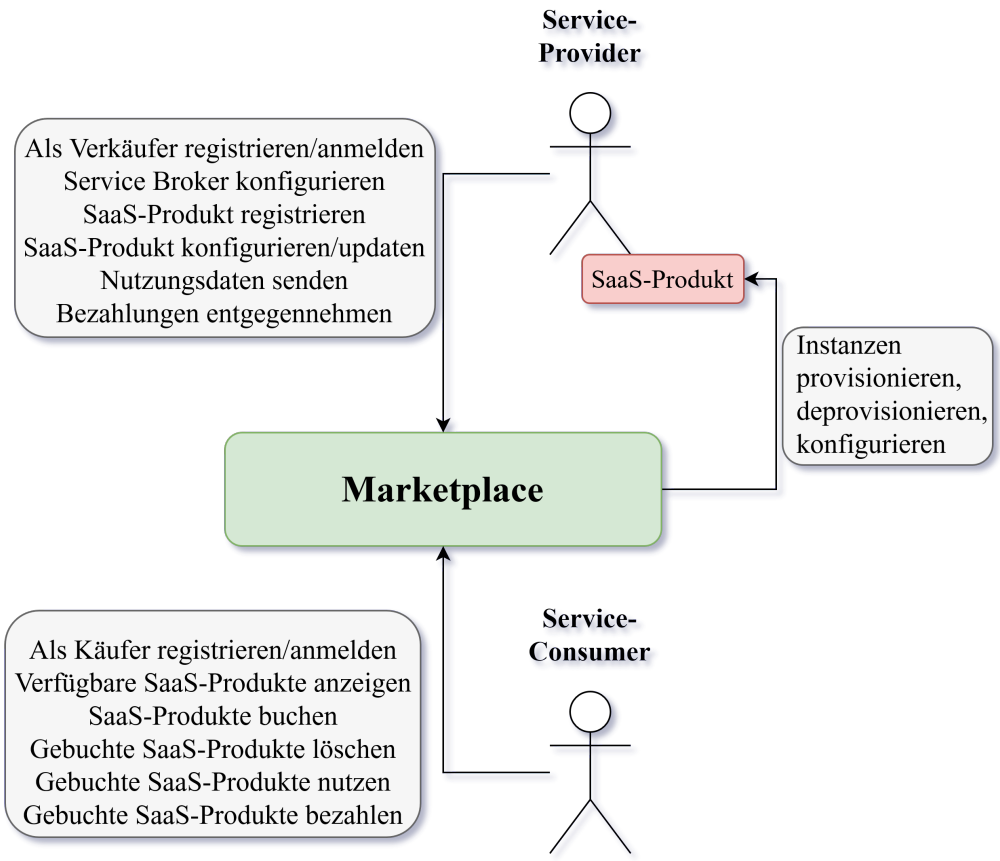


Abbildung 5.1: Fachliche Ein- und Ausgaben des Cloud-Marketplaces

5.2 Lösungsstrategie

Dieses Kapitel gibt einen kurzen Überblick über die grundlegenden Entscheidungen und Lösungsansätze, die Entwurf und Implementierung des Systems prägen.

Um das Qualitätsziel der stetigen Antwortbereitschaft des Cloud-Marketplaces zu erfüllen, wird sich bei dem Architekturentwurf an den Richtlinien des Reactive Manifesto, erklärt im Kapitel 2.4, gehalten. Zusätzlich werden eine starke Kohäsion, eine gute Wartbarkeit und Erweiterbarkeit durch die Umsetzung einer Microservice-Architektur angestrebt. Für eine lose Kopplung wird zusätzlich eine MoM für die Nachrichtenübertragung eingesetzt. Um auch bei einer großen Menge an Anfragen reaktionsfähig zu sein, muss der Cloud-Marketplace vertikales und horizontales Skalieren unterstützen. Zudem werden die Microservices als Cloud Native-Applikationen zur Verfügung gestellt. Die Verantwortung der Authentisierung und Autorisierung trägt ein zentrales IAM. Das IAM erstellt, unter Verwendung von asynchronen Verschlüsselungstechniken, Nutzer-Token für die Authentisierung der Nutzer gegenüber den Microservices. Die signierten Nutzer-Token können anschließend durch verschiedene Akteure oder Microservices mit dem Zertifikat des IAM verifiziert werden. Im Nutzer-Token sind Informationen des Nutzers, unter anderem eine Nutzer-ID und die Nutzer-Rollen, enthalten. Die Autorisierung findet im Endpunkt statt, dass bedeutet in diesem Fall, in den einzelnen Microservices.

Zusammengefasst wird der Cloud-Marketplace nach einer Cloud Native-Event-Driven Reactive Services (EDRS)-Architektur entworfen, die sich durch besondere Eignung zur Skalierbarkeit ausweist.

5.3 Bausteinsicht

Diese Sicht zeigt die statische Zerlegung des Systems in Bausteine, sowie deren Beziehungen [Hruschka u. a. 2022]. Die Bausteinsicht ist eine hierarchische Sammlung von Blackboxen und Whiteboxen, sowie deren Beschreibungen.

Die erste hierarchische Ebene ist die Kontextabgrenzung von Kapitel 5.1 und die Abbildung der Ein- und Ausgaben des Cloud-Marketplaces (Abbildung 5.1). Die nächste Ebene ist die Whitebox-Beschreibung des Gesamtsystems sowie zusätzlich die Blackbox-Beschreibungen der darin enthaltenen Bausteine. Anschließend

werden die enthaltenen Bausteine aus Ebene 1, in der Ebene 2 als Whitebox-Beschreibung, zusammen mit Blackbox-Beschreibungen der darin enthaltenen Bausteine, beschrieben. Es erfolgt demnach eine schrittweise Verfeinerung der Erklärungen durch die verschiedenen Abstraktionsebenen.

5.3.1 Gesamtsystem

Abbildung 5.2 zeigt das Gesamtsystem des Cloud-Marketplaces. Zu sehen sind die Service-Provider, die Service-Consumer und der Cloud-Marketplace selbst. Fünf Services befinden sich im Cloud-Marketplace: Service Registry, Service Provisioning Service, Service Instanzen Service, IAM und der Service Usage Service. Zusätzlich besitzt der Cloud-Marketplace eine skalierbare MoM. In Abbildung 5.2 ist die Kommunikation der Akteure und die Kommunikation mit den Service Brokern zu den internen Bausteinen des Cloud-Marketplaces abstrakt dargestellt. Eine Definition der einzelnen Schnittstellen, sowie eine Definition der API des Service-Providers folgt in den weiteren Abstraktionsebenen.

Die Service-Provider (auf der linken Seite) symbolisieren gleichzeitig ein technisches System und jeweils einen Akteur. Es kann mehrere verschiedene Service-Provider mit mehreren verschiedenen Service Brokern geben. Der Service Broker ist keine Softwarekomponente des Cloud-Marketplaces, jedoch ist eine Definition des Service Brokers notwendig, um eine einheitliche API im Cloud-Marketplace benutzen zu können. Eine Definition des Service Brokers erfolgt im Kapitel 5.3.2.

Die Service-Consumer (auf der rechten Seite) symbolisieren mehrere Akteure, die über das Frontend oder die API des Systems mit dem Cloud-Marketplace interagieren. Aus Übersichtsgründen wurde der Betreiber des Cloud-Marketplaces als Akteur aus der Abbildung herausgelassen. Der Betreiber des Cloud-Marketplaces verwaltet, installiert, konfiguriert und pflegt die einzelnen Services des Cloud-Marketplaces. Da es keine Beziehung von dem Cloud-Marketplace zum Service-Consumer gibt und da der Service-Consumer keine Komponenten des Cloud-Marketplaces besitzt, gibt es keine weitere Betrachtung des Service-Consumers als Baustein.

Verantwortlichkeiten der Cloud-Marketplace-Bausteine

Die Services des Cloud-Marketplaces haben klar getrennte Verantwortlichkeiten. Alle Microservices bieten eine Representational State Transfer (REST)-Schnittstelle für das Frontend. Im Folgenden wird eine Blackbox-Beschreibungen der

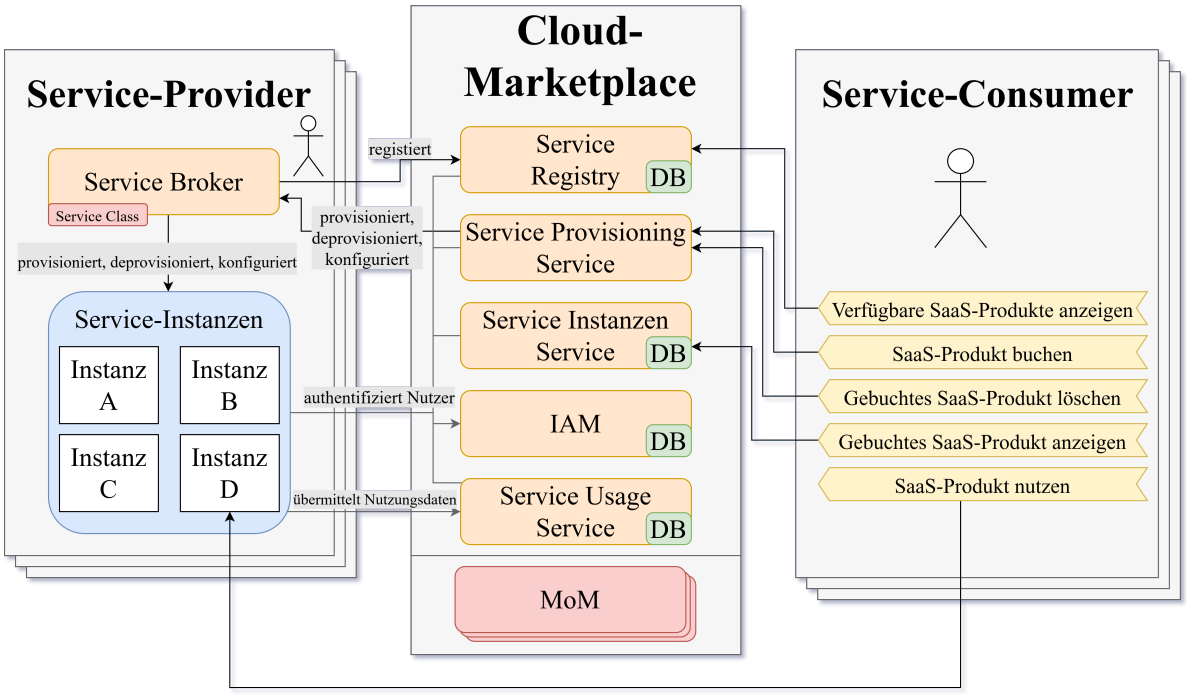


Abbildung 5.2: Gesamtsystem des Cloud-Marketplaces: Kommunikation der Bausteine nach außen

Cloud-Marketplace-Services erläutert, eine detaillierte Beschreibung der Schnittstellen erfolgt in der tiefen Abstraktionsebene der Services.

Die **Service Registry** hat die Verantwortung über die Verwaltung der Service Broker. Die Registrierung der Service Broker nimmt die Service Registry vom Frontend entgegen und führt anschließend eine Validierung und Persistierung durch. Ein detailliertes Sequenzdiagramm zur Service Broker Registrierung wird in Kapitel 5.4.1 erläutert. Auch Konfigurationen der persistierten Service Broker, wie das Aktualisieren, Erweitern, Löschen oder Aktivieren des Brokers nimmt die Service Registry entgegen. Die Registry kann Service-Providern ihre eigenen Service Brokern zurückgeben und besitzt eine Schnittstelle für die Lieferung des aktiven Cloud-Marketplace-Produktportfolios.

Der **Service Provisioning Service** hat die Verantwortung über die Provisionierung der Service-Instanz. Der Service Provisioning Service erhält Aufträge zur Instanzprovisionierung und orchestriert die Erstellung sowie den Lebenszyklus der Instanz. Ein detailliertes Sequenzdiagramm

zur Provisionierung der Instanz wird in Kapitel 5.4.2 erläutert. Auch das Erstellen von Service-Bindings, das Überprüfen des Service-Status und das Deprovisionieren von Service-Instanzen übernimmt der Service Provisioning Service.

Der **Service Usage Service** hat die Verantwortung über die Nutzungsdaten der Service-Instanzen. Der Service Usage Service nimmt die Nutzungsdaten der Instanzen entgegen, validiert und persistiert diese anschließend. Service-Instanz Nutzer können die Nutzungsdaten ihrer eigenen Instanzen anfragen.

Der **Service Instanzen Service** hat die Verantwortung über die Service-Instanzen und deren Persistierung. Der Service Instanzen Service nimmt die Erstellung der Instanz sowie die Änderungen einer Instanz entgegen, validiert und persistiert diese anschließend. Service-Instanz-Nutzer können alle aktiven, inaktiven und ehemaligen Service-Instanzen einsehen.

Das **IAM** hat die Verantwortung über die Authentisierung von allen Nutzern. Das IAM bietet eine zentrale API und Webseite für die Registrierung und Anmeldung für die Service-Consumer und -Provider. Über einem Endpunkt des IAM können Service-Consumer und -Provider einen Nutzer-Token erlangen. Zudem bietet das IAM durch die signierten Nutzer-Token und dem IAM-Zertifikat die Möglichkeit, die Nutzer zu verifizieren.

Schnittstellen

Die Schnittstellendefinition aller Interaktionen, wie REST-Schnittstellen, die Aufträge an die Service Broker oder die Nachrichtenübertragung über die MoM, erfolgt in der Abstraktionsebene des jeweiligen Service. Im Folgenden erfolgt die Definition der API des Service Brokers.

5.3.2 Service Broker

Bevor die Bausteine des Cloud-Marketplaces definiert werden, wird in diesem Kapitel ein Standard für den Service Broker definiert. Nach Vorgabe des LGLN wird der Standard der Open Service Broker API [Gunaratne u. a. 2022] für den Service Broker beschlossen. Alle Service Broker müssen demnach die Open Service Broker API erfüllen, die Service-Provider können jedoch frei entscheiden, wo der

Service Broker bereitgestellt wird. So kann der Service Broker beispielsweise in einer Public Cloud oder auf einem selbst installierten Server bereitgestellt werden. Es ist jedoch wichtig, dass die API des Service Brokers für den Cloud-Marketplace erreichbar ist.

Open Service Broker API

Die Open Service Broker API ermöglicht es unabhängigen Softwareanbietern, für beispielsweise SaaS-Produkte, Dienste für Workloads bereitzustellen, die auf nativen Cloud-Plattformen wie Cloud Foundry und Kubernetes laufen. Die Spezifikation beschreibt eine kleine Menge von API-Endpunkten, die für die Bereitstellung, den Zugriff und die Verwaltung von Service-Instanzen verwendet werden können. An dem Projekt sind Google, IBM, Pivotal, Red Hat, SAP und viele andere führende Cloud-Unternehmen beteiligt [Gunaratne u. a. 2022].

API-Endpunkte des Service Brokers

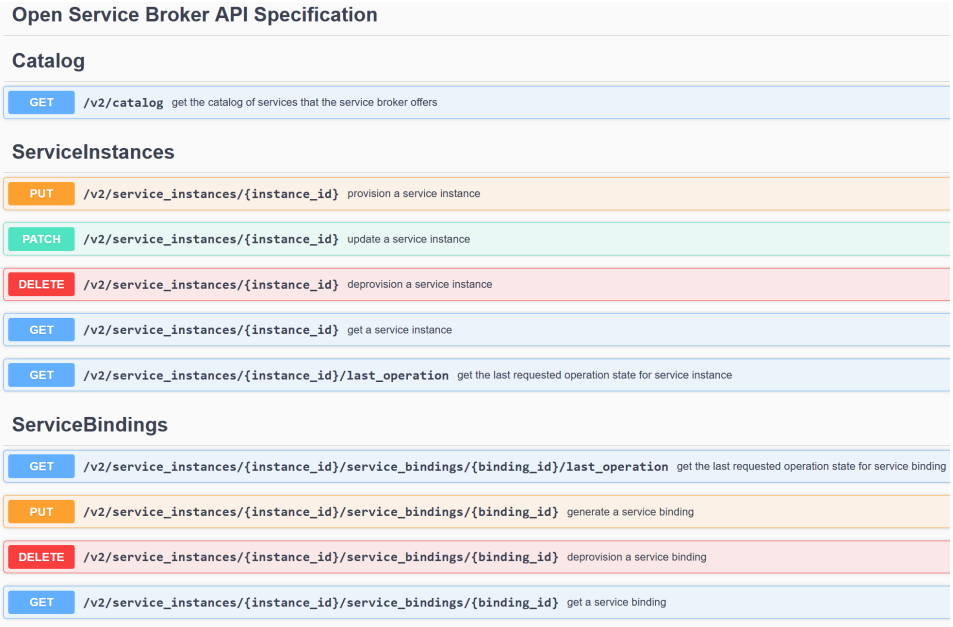


Abbildung 5.3: Open Service Broker API-Spezifikation [Open-Service-Broker-API 2022]

In Abbildung 5.3 ist eine Auflistung der API-Endpunkte der Open Service Broker API dargestellt. Es existiert eine GET-Methode für das Abrufen des Katalogs. Im Katalog ist definiert, welche SaaS-Produkte zu welchen Preismodellen angeboten werden. Demnach ist eine Liste mit mindestens einem SaaS-Produkt enthalten. In der Definition des SaaS-Produktes ist mindestens ein Preismodell sowie weitere Informationen des SaaS-Produktes enthalten. Eine Definition des Datenmodells und weitere Details zu den Schnittstellen befindet sich im Anhang auf Seite 111.

5.3.3 Message-oriented Middleware

Die Kommunikation innerhalb des Cloud-Marketplaces findet fast vollständig über die MoM statt. In Abbildung 5.4 ist die interne Kommunikation der Microservices dargestellt.

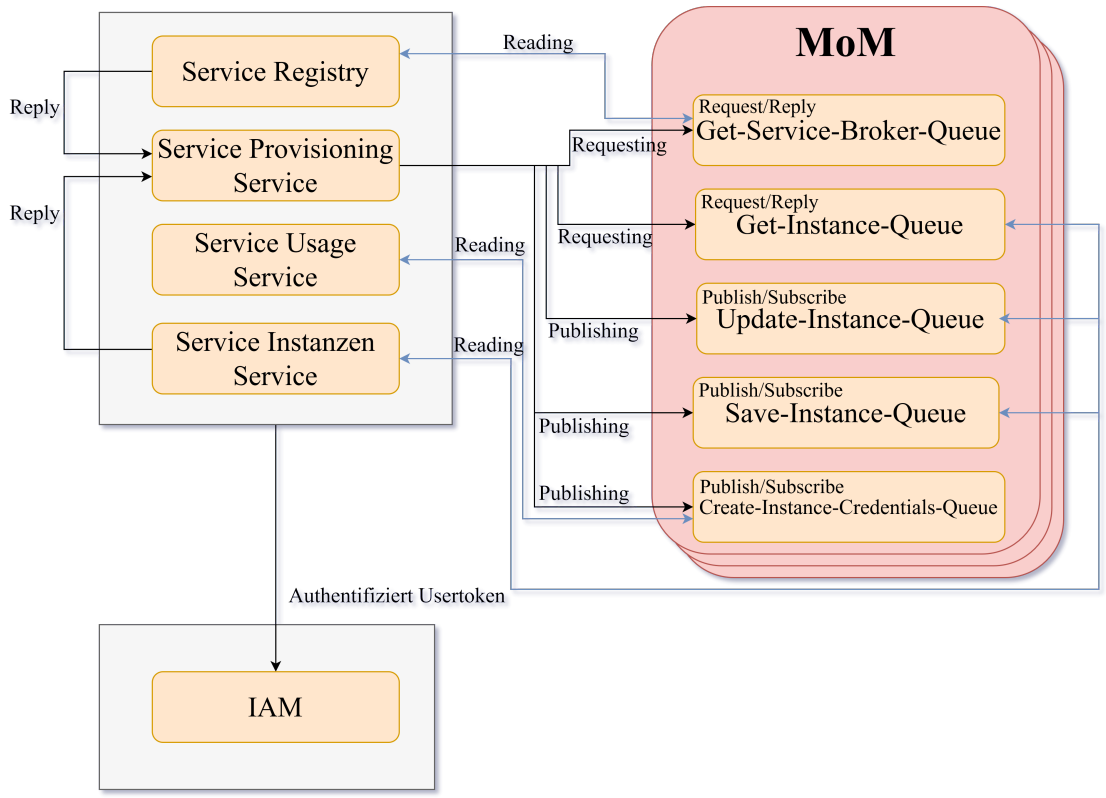


Abbildung 5.4: Gesamtsystem des Cloud-Marketplace: Interne Kommunikation der Bausteine

Da die MoM und Message-Queues in vier von fünf Services benutzt wird, werden die Message-Queues mit dem Request-Reply-Pattern und Publish-Subscribe-Pattern als Querschnittliche Konzepte in Kapitel 5.6 erklärt.

Die interne Kommunikation besteht aus zwei Request-Reply- und zwei Publish-Subscribe Message-Queues. Die vier Microservices benötigen jeweils nur die Adresse der MoM und den Namen der Message-Queue, um eine Nachricht zu senden oder um eine Nachricht zu empfangen. Alle vier Services haben ebenfalls eine Client-Verbindung zum IAM-Service, um Nutzer zu Authentifizieren sowie um Nutzer eindeutig zuordnen zu können.

Nach Vorgabe des LGLN soll bei dem Cloud-Marketplace Neural Autonomic Transport System (NATS) [NATS 2022a] als MoM eingesetzt werden. NATS ist ein Open-Source-Messaging-System mit Fokus auf Leistung, Skalierbarkeit, Benutzerfreundlichkeit, Sicherheit und Resilienz. NATS besitzt zudem Client-Bibliotheken in vielen Programmiersprachen. Es wurde im Hinblick dessen gewählt, da NATS eine aufsteigende Technologie im Cloud-Umfeld ist und von der Cloud Native Computing Foundation (CNCF) aufgenommen wurde [Cloud-Native-Computing-Foundation 2022b].

5.3.4 Service Registry

Die Service Registry hat die Verantwortung der Service Broker-Verwaltung. Neben dem Lesen aus der Message-Queue *Get-Service-Broker-Queue* erfüllt die Service Registry sieben REST-Schnittstellen. Die Message-Queue *Get-Service-Broker-Queue* dient als Getter-Funktion für Service Broker-Objekte. Der Service Provisioning Service benötigt diese Objekte, um die Kommunikation mit den Service Brokern herzustellen. Die Service Registry schickt demnach jedes Mal ein Service Broker-Objekt an die, in der Nachricht enthaltene, *reply-to* Adresse.

Service Broker-Datenbank

Die Service Registry besitzt eine Anbindung an eine relationale Datenbank. Dafür wird eine PostgreSQL-Datenbank verwendet, eine Begründung dazu ist in Kapitel 5.6.1. Die Persistierung der Service Broker-Objekte muss die Service Registry selbst übernehmen. Dazu sollte jedes Objekt eine eindeutige Identifikationsnummer (ID), einen Namen für den Cloud-Marketplace, die Besitzer-ID, die Uniform Resource Identifier (URI), einen Status und den Service Broker-Katalog besitzen.

REST-Schnittstelle

In Abbildung 5.5 sind die sieben REST-Schnittstellen der Service Registry dargestellt. Eine detaillierte Dokumentation der einzelnen REST-APIs befindet sich im Anhang auf Seite 128.

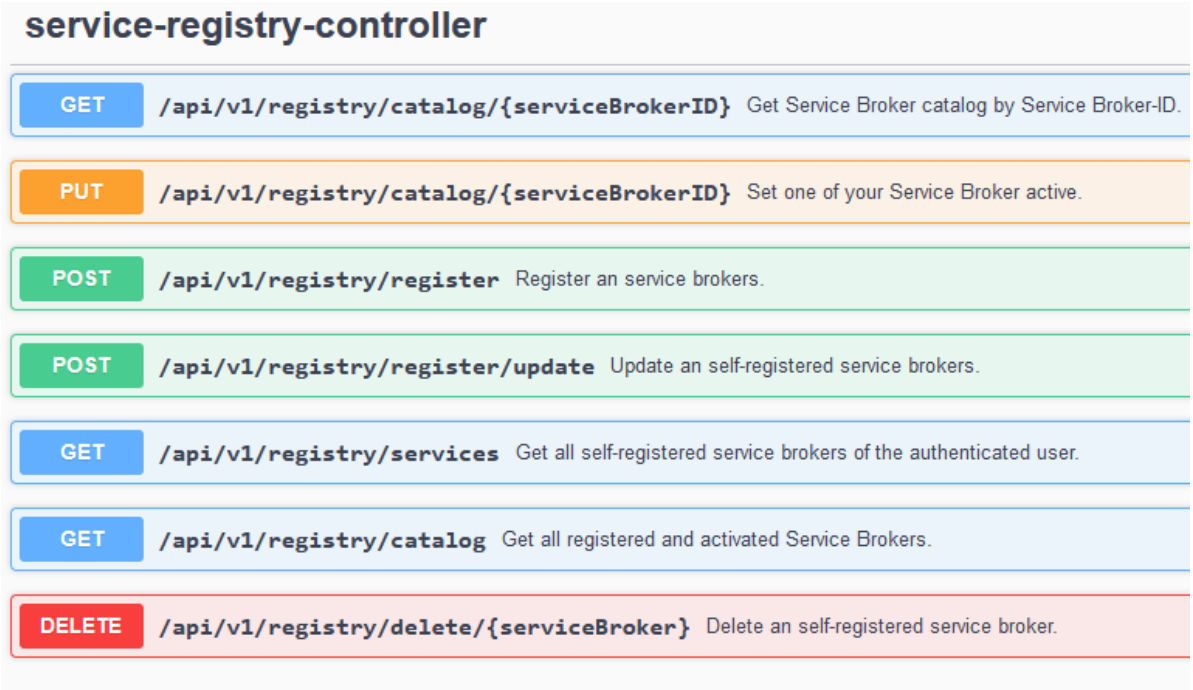


Abbildung 5.5: REST-API der Service Registry

5.3.5 Service Provisioning Service

Der Service Provisioning Service hat die Verantwortung über die Provisionierung der Service-Instanzen und orchestriert den Lebenszyklus der Service-Instanzen. Die Kommunikation mit den Service Brokern findet nur in diesem Microservice statt. Demnach ist der Service Provisioning Service der einzige Microservice, der die Open Service Broker API kennen muss. Er besitzt jedoch keine Anbindung an eine Datenbank und hat damit keine Verantwortung von Persistierungen. Dafür kommuniziert der Service Provisioning Service mit allen fünf Message-Queues

der MoM, da der Service Provisioning Service als zentraler Orchestrator für die Provisionierungen, also dem Kern des Cloud-Marketplaces, agiert. So benutzt der Service Provisioning Service die Message-Queue *Get-Service-Broker-Queue* und *Get-Instance-Queue*, um existierende Objekte von der Service Registry und dem Service Usage Service zu erlangen. Die Message-Queues *Update-Instance-Queue* und *Save-Instance-Queue* werden verwendet, um Instanz-Objekte für eine Persistierung an den Service Instanzen Service zu schicken. Die Message-Queue *Create-Instance-Credentials-Queue* wird verwendet, um die Nutzungsdatenübertragung zu erleichtern. Durch die Nachricht aus der *Create-Instance-Credentials-Queue* wird im Service Usage Service die Berechtigung und die Relationen für den Service Broker zum Service-Instanzobjekt gesetzt.

REST-Schnittstelle

In Abbildung 5.6 sind die vier REST Schnittstellen des Service Provisioning Service dargestellt. Eine detaillierte Dokumentation der einzelnen REST-APIs befindet sich im Anhang auf Seite 138.

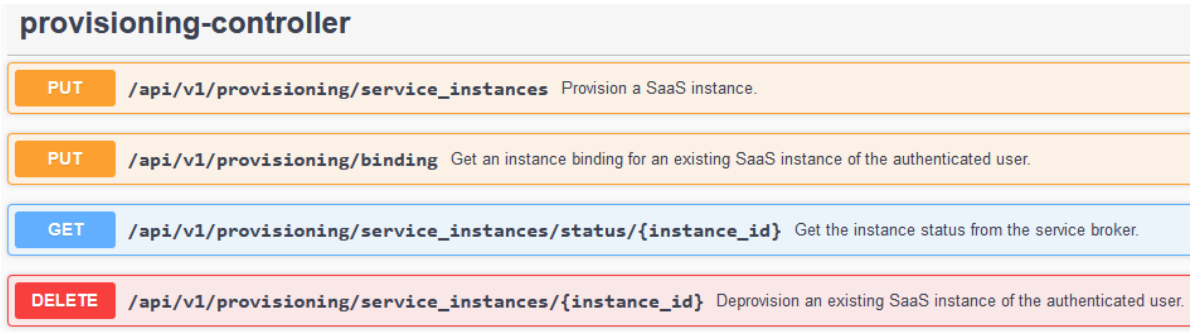


Abbildung 5.6: REST-API des Service Provisioning Services

5.3.6 Service Usage Service

Der Service Usage Service hat die Verantwortung über die Nutzungsdaten der Service-Instanzen. Der Service Usage Service besitzt demnach die Aufgabe, die

Nutzungsdaten der Serviceinstanzen entgegenzunehmen, sowie diese bei Bedarf nutzerspezifisch zur Verfügung zu stellen. Eine dynamische Erläuterung der Übertragung von Meteringdaten befindet sich im Kapitel 5.4.5.

Service Usage-Datenbank

Der Service Usage Service besitzt eine Anbindung an eine relationale Datenbank. Dafür wird eine PostgreSQL-Datenbank verwendet, eine Begründung dazu ist in Kapitel 5.6.1 zu finden. Die Persistierung der Nutzungsdaten muss der Service Usage Service selbst übernehmen. Dazu sollte jedes Objekt eine eindeutige ID, die Instanz-ID, die Instanz-Besitzer-ID, die Service Broker-ID, eine Methode für die sichere Übertragung der Nutzungsdaten, einen berechtigten Benutzer für die Übertragung und abschließend die Nutzungsdaten enthalten.

REST-Schnittstelle

In Abbildung 5.7 sind die zwei REST-Schnittstellen des Service Usage Service dargestellt. Eine detaillierte Dokumentation der einzelnen REST-APIs befindet sich im Anhang auf Seite 145.

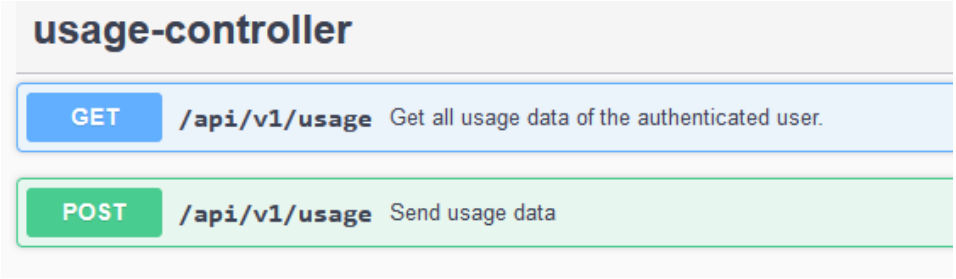


Abbildung 5.7: REST-API des Service Usage Service

5.3.7 Service Instanzen Service

Der Service Instanzen Service hat die Verantwortung über die Service-Instanzen sowie deren Persistierung und besitzt nur eine REST-Schnittstelle, um die In-

stanzen bei Bedarf nutzerspezifisch zur Verfügung zu stellen. Der Service Instanzen Service liest aus drei Message-Queues: *Get-Instance-Queue*, *Update-Instance-Queue* und *Save-Instance-Queue*. Die Message-Queue *Get-Instance-Queue* dient als Getter-Funktion für Instanz Objekte. Der Service Instanzen Service schickt demnach das Instanz Objekt an die, in der Nachricht enthaltene, *reply-to* Adresse. Aus den Message-Queues *Update-Instance-Queue* und *Save-Instance-Queue* wird gelesen, um die Instanzinformation zu persistieren.

Service Instanzen-Datenbank

Der Service Instanzen Service besitzt eine Anbindung an eine relationale Datenbank. Dafür wird eine PostgreSQL-Datenbank verwendet, eine Begründung dazu ist in Kapitel 5.6.1. Die Persistierung der Instanz-Objekte muss der Service Instanzen Service selbst übernehmen. Dazu sollte jedes Objekt eine eindeutige ID, die Instanz-ID, die Instanz-Besitzer-ID, die Service Broker-ID, die Service-ID, die Service-Plan-ID, ein Erstellungsdatum, ein Datum für die letzte Änderung des Status sowie den Status der Instanz besitzen.

REST-Schnittstelle

In Abbildung 5.8 ist die REST-Schnittstelle des Service Instanzen Service dargestellt. Eine detaillierte Dokumentation der REST-API befindet sich im Anhang auf Seite 143.

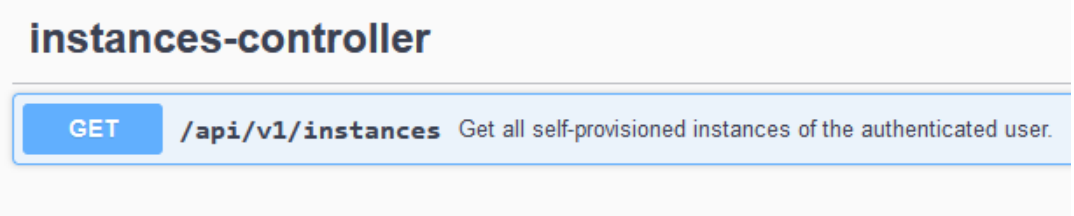


Abbildung 5.8: REST-API des Service Instanzen Service

5.3.8 Identity Access Management - Keycloak

Die IAM hat die Verantwortung über die Authentisierung, Autorisierung und Identifizierung von allen Nutzern. Nach Vorgabe des LGLN wird als IAM Keycloak verwendet. Keycloak ist eine Java-basierte Open-Source-Identity- und Access-Management-Lösung. Zusätzlich begünstigt die Anforderung 3 c) die Auswahl, denn Keycloak ist bereits im Technologie-Stack des LGLN.

IAM-Datenbank

Das IAMs, damit in diesem Fall Keycloak, benötigt eine relationale Datenbank. Bei dem Cloud-Marketplace wird dafür eine PostgreSQL-Datenbank (Begründung in Kapitel 5.6.1) verwendet. Das Anlegen von Datentabellen, sowie das Einfügen von Daten übernimmt Keycloak automatisch.

Schnittstelle

Keycloak besitzt eine sehr Umfangreiche API welche sich in der Dokumentation von Keycloak einsehen lässt [Keycloak 2022a].

5.3.9 Payment Service

Da der Payment Service noch kein Bestandteil des Prototyps sein wird, ist dieser nicht in der Übersicht von Abbildung 5.2 enthalten. Der Payment Service ist dennoch wichtig, um die Notwendigkeit des Service Usage Service nachvollziehen zu können. Denn der Service Usage Service ist nicht nur dafür da, um Nutzungsdaten entgegenzunehmen und diese für eine Statistik wieder bereit zu stellen. Im Folgenden werden die Verantwortlichkeiten des Payment Services kurz erläutert.

Einige SaaS-Produkte haben kostenpflichtige Pläne in Form von verschiedenen Preismodellen. Für diese Abwicklung der Bezahlung ist der Payment Service zuständig. Dabei werden zwischen einmaligen und dynamischen Kosten unterschieden. Das SaaS-Produkt kann einen einmaligen Betrag für die Provisionierung oder für eine einmalige Nutzung erheben. Die zweite Variante ist das Abrechnen für eine aktive Nutzung oder über eine aktive Bereitstellung eines SaaS-Produktes über die Zeit. Es sind demnach verschiedene Metriken für eine Kostenabrechnung möglich. In den, an den Service Usage Service zu übertragenen Nutzungsdaten,

sowie im gebuchten Plan ist die Einheit der Nutzungsdaten und der Preis für jede Einheit enthalten. Mögliche Einheiten sind beispielsweise ein Zeitraum oder ein bestimmter Durchsatz an Funktionen. Eine Kombination von verschiedenen dynamischen Kosten mit weiteren dynamischen oder einmaligen Kosten ist möglich.

Der Payment Service ist berechtigt diese Nutzungsdaten aus dem Service Usage Service auszuwerten, um anschließend Kostenabrechnung für die Nutzer zu erstellen. Der Service-Consumer ist verpflichtet für die kostenpflichtige Nutzung zu bezahlen. Die Umsetzung des Payment Service wird nachgelagert betrachtet.

5.4 Laufzeitsicht

Diese Sicht erklärt konkrete Abläufe und Beziehungen zwischen Bausteinen in Form von Szenarien [Hruschka u. a. 2022].

5.4.1 Service Broker-Registrierung

In Abbildung 5.9 ist der dynamische Ablauf der Registrierung von SaaS-Produkten als Sequenzdiagramm dargestellt. Für die Registrierung eines SaaS-Produktes füllt ein angemeldeter Nutzer auf der Webseite alle notwendigen Informationen aus und stellt ebenfalls seinen Service Broker für die SaaS-Instanzen in einer Umgebung seiner Wahl zur Verfügung. Diese Umgebung muss von der Service Registry aus erreichbar sein.

Nach dieser Vorbereitung kann eine Registrierung über die REST-Schnittstelle erfolgen. Die Service Registry überprüft zuerst, ob der angemeldete Nutzer für eine Registrierung von Service Brokern berechtigt ist. Im Anschluss werden die übertragenden Daten validiert. Zusätzlich wird die Erreichbarkeit des Service Brokers überprüft, indem die Service Registry nach der Open Service Broker API den Katalog des Service Brokers abruft. Hat das fehlerfrei funktioniert, wird der Service Broker in der Service Broker-Datenbank persistiert. Der Service Broker wird zu Beginn jedoch noch nicht direkt im Cloud-Marketplace-Produktkatalog angeboten, sondern befindet sich in einem *Onboarding-Status*. In diesem Zustand ist es dem Service-Provider noch möglich, Anpassungen oder Tests zu tätigen. Damit das SaaS-Produkt in dem Cloud-Marketplace-Produktkatalog aufgenommen wird, muss der Service-Provider sein Produkt dafür aktivieren.

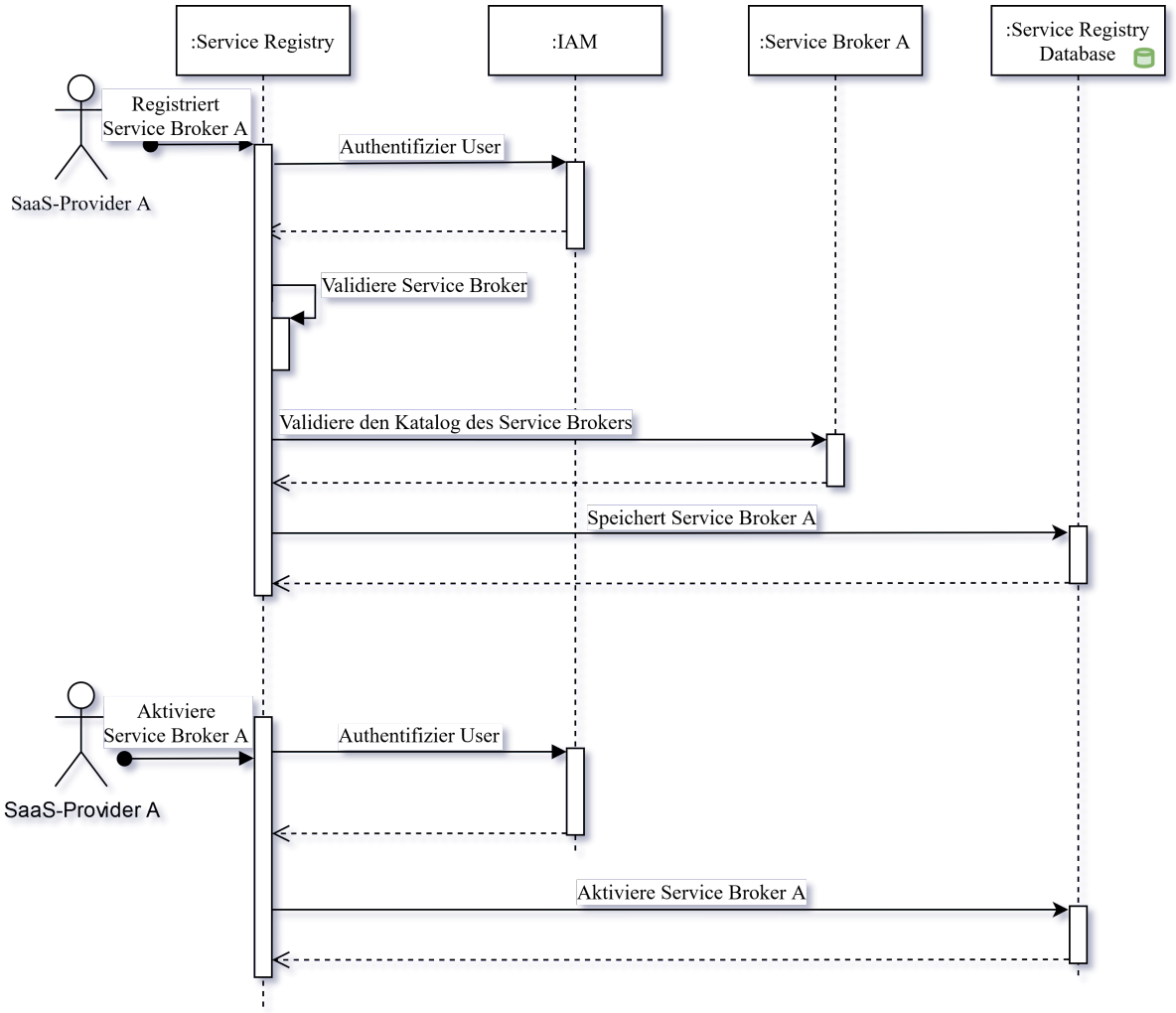


Abbildung 5.9: Sequenzdiagramm: Registrierung eines Service Brokers

5.4.2 Service-Provisionierung

In Abbildung 5.10 ist der dynamische Ablauf der Provisionierung von SaaS-Instanzen als Sequenzdiagramm dargestellt. Hier ist als Vorbereitung eine Anmeldung als Cloud-Marketplace-Nutzer, sowie das Auswählen eines SaaS-Produktes mit einem dazugehörigen Serviceplan notwendig.

Nach dieser Vorbereitung authentifiziert der Service Provisioning Service zuerst den Nutzer, um die eindeutige Nutzer-ID zu erhalten. Zusätzlich erstellt der Service Provisiong Service über die REST-Schnittstelle von Keycloak einen Instanz-Account, unter dem es später möglich ist, Nutzungsdaten zu senden. Der Ge-

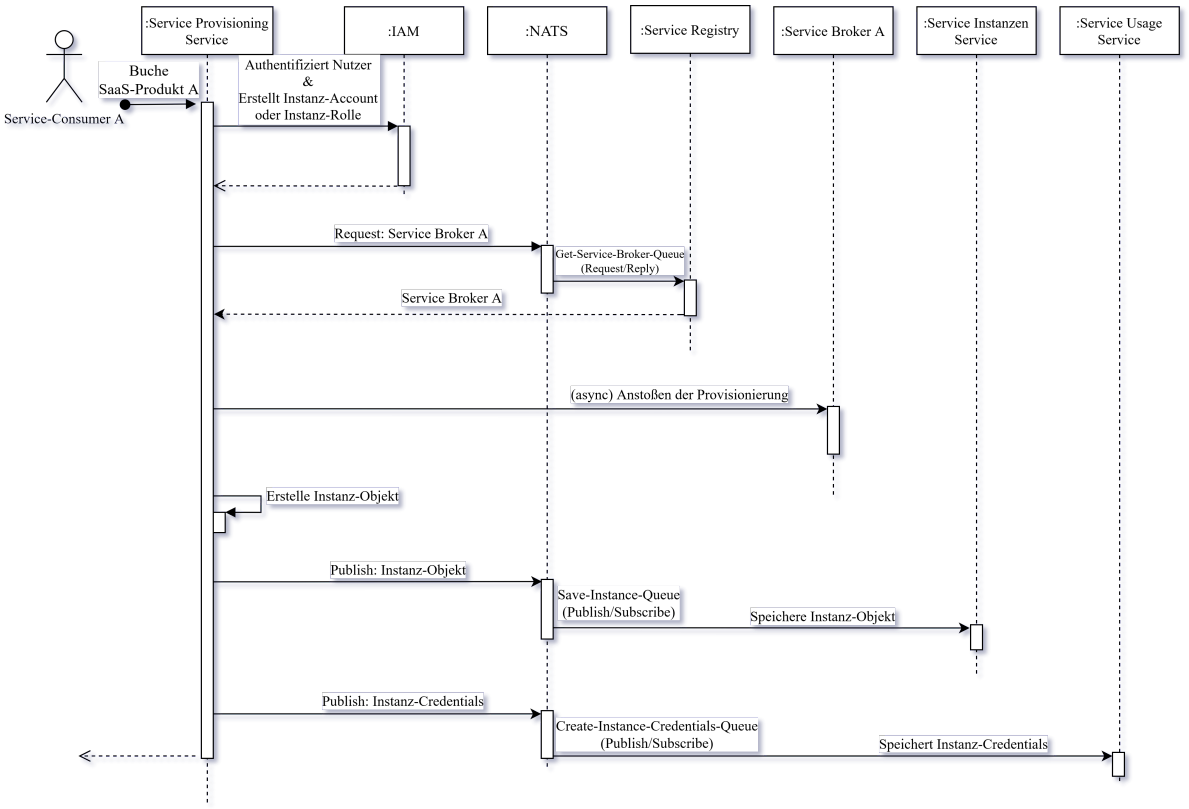


Abbildung 5.10: Sequenzdiagramm: Service-Provisionierung

samtablauf der Provisionierung erfolgt asynchron. Der Service Provisioning Service schickt einen Request an den NATS-Service, um das Service Broker-Objekt des SaaS-Produktes zu erhalten. Mit der, in dem Objekt enthaltenen, URI schickt der Service Provisioning Service nach der Definition der Open Service Broker API einen asynchronen REST-Aufruf zum Anstoßen der Provisionierung einer Service-Instanz. Währenddessen wird zu diesem Prozess ein Service-Instanzobjekt erstellt und über die NATS Message-Queue *Save-Instance-Queue* für die Persistierung weitergeleitet. Abschließend wird ebenfalls ein Instanz-Credentials-Objekt über die Message-Queue *Create-Instance-Credentials-Queue* für den Service Usage Service weitergeleitet.

5.4.3 Update Strategie für Service Broker und Servicepläne

Für den Anwendungsfall, dass ein bestehender Service Broker oder Service-Produkt überarbeitet oder konfiguriert werden soll, bedarf es einer Strategie. In Abbildung

5.11 ist ein Sequenzdiagramm für das Updaten von Service Brokern dargestellt. Die REST-Schnittstelle sollte über das Frontend des Cloud-Marketplaces aufgerufen werden. Dort werden alle notwendigen Informationen für das Konfigurieren abgefragt. Es ist möglich einzelne Service-Pläne zu konfigurieren, wie zum Beispiel ein Erhöhen des Preises, das Deaktivieren ganzer Service-Pläne oder das Hinzufügen von neuen Plänen. Ebenso sollte es möglich sein, den Namen des Produktes, die Informationsangaben des Produktes oder den Standort des Service Brokers zu ändern.

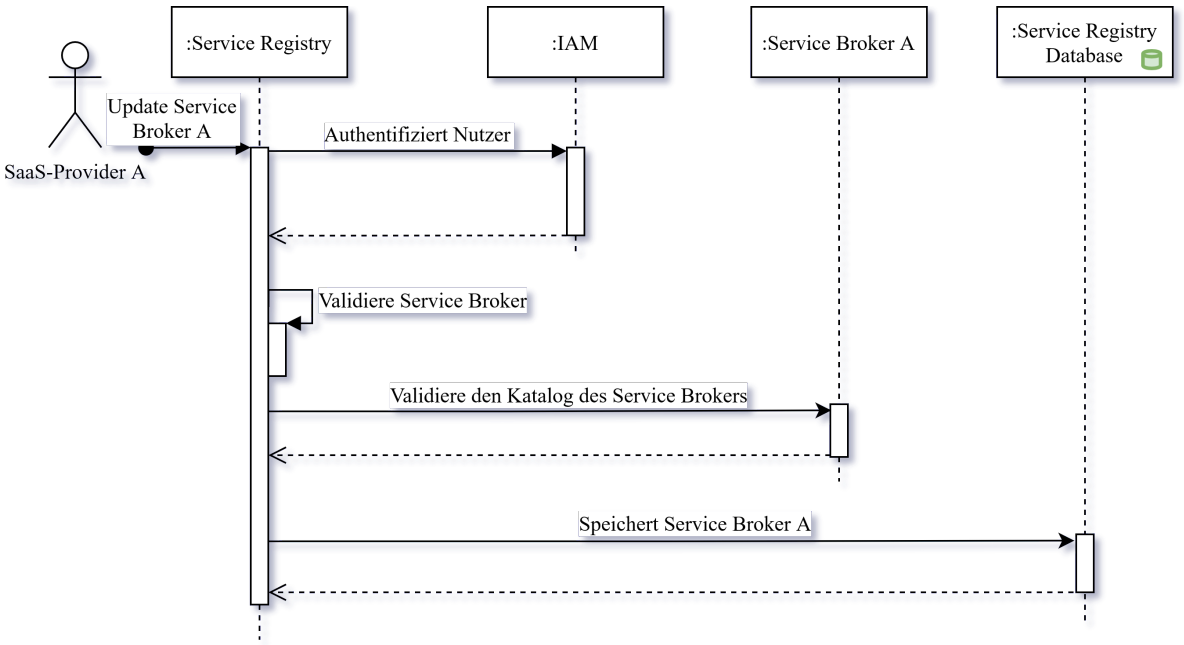


Abbildung 5.11: Sequenzdiagramm: Update-Strategie für Service Broker

Um alles über eine Schnittstelle zu ermöglichen, bedarf es nach der Authentifizierung des Service-Providers einer Validierung des *neuen* Service Brokers. Jedoch sollte der Service Broker bei einem Update nicht neu hinterlegt werden, denn der Service-Provider hat das Interesse, beispielsweise die vorhandenen Bewertungen zu behalten. Nach der Validierung und Anpassung des Service Broker-Objektes wird anschließend die Erreichbarkeit des Service Brokers überprüft, indem die Service Registry nach der Open Service Broker API einen Abruf des Service-Katalogs tätigt. Hat das fehlerfrei funktioniert, wird der erneuerte Service Broker

in der Service Broker-Datenbank persistiert.

5.4.4 Speicherung der Events

Eingehende und interne Events sollen überwacht werden können. Dies geschieht über die MoM, da über diese alle Events weitergeleitet werden. Deshalb ist es von Vorteil, wenn die MoM eine Funktion für die Überwachung und Persistierung von Events anbietet. In Abbildung 5.4 sind die fünf möglichen Events dargestellt. Prinzipiell lassen sich aus allen Message-Queues die Events persistieren. Falls das zu aufwendig oder teuer wird, würde es sich zuerst anbieten, die Events aus den beiden Request-Reply Message-Queues nicht zu persistieren, da es sich bei den beiden Message-Queues um reine Information-Getter handelt. Die anderen drei Message-Queues beinhalten dagegen wichtige Informationen zu dem Kerngeschäft des Cloud-Marketplaces und sind deshalb essentiell zu überwachen. Die Persistierung der Events erlaubt es zudem, zu einem späteren Zeitpunkt eine Analyse der Events ausführen zu können, wie es mit NATS möglich ist. Die Persistierung von Events wird in Kapitel 6.2.2 erläutert.

5.4.5 Übertragung von Meteringdaten

Die Übertragung von Meteringdaten ist eine kritische Anwendungsfunktion, da dort potenziell besonders viel Arbeitsbelastung, durch eingehende Sendungen von Nutzungsdaten, auftreten kann. Deshalb wurde bei dem Entwurf dieses Szenarios darauf geachtet, dass dieser Prozess so wenig wie möglich auf die anderen Microservices zugreifen muss, um den Service Usage Service getrennt skalieren zu können.

So ist es möglich den Service Usage Service unabhängig von der Service Registry, dem Service Provisioning Service und dem Service Instanzen Service horizontal zu skalieren, um die steigende Arbeitsbelastung erfüllen zu können. Im Provisioningsprozess wurde bereits dem Service Usage Service die Informationen über Instanz, Nutzer und Service-Provider übergeben. Dabei enthalten ist ebenfalls die Methode sowie Daten für eine sichere Übertragung der Meteringdaten. Prinzipiell gibt es dafür zwei Methoden zur Auswahl.

Bei der ersten Methode erhält die Service-Instanz einen eigenen Account im IAM.

Dieser Instanz-Account muss anschließend verwendet werden, um die Nutzungsdaten zu senden. Die Methode ist dafür geeignet, wenn die Kosten des SaaS-Produktes von seiner aktiven Nutzung abhängig ist. Jedoch muss der Instanz-Account, während der Provisionierung, dem Service Broker verschickt werden.

Bei der zweiten Methode bekommt der bestehende Service-Provider-Account für die Instanz eine Instanzrolle zugewiesen. Dafür müssen keine Accountinformationen versendet werden, jedoch muss der Service Broker selbst die Nutzungsdaten versenden und nicht mehr die Instanzen, welches zu hoher Last für den Service Broker führen könnte. Diese Methode ist dafür geeignet, wenn die Kosten des SaaS-Produktes einmalig, beispielsweise bei der ersten Nutzung, anfallen.

Sendet der Service Broker oder die Instanz die Nutzungsdaten dem Service Usage Service muss nur noch, je nach Methode, der angemeldete Nutzer oder die Rollen des Nutzers authentifiziert werden. Der Service Usage Service prüft je nach Methode, ob der Nutzer oder die Rolle des Nutzers mit der hinterlegten Information im Instanz Objekt passt und lässt sich dieses durch das IAM bestätigen.

5.5 Verteilungssicht

Die Verteilungssicht beschreibt die technische Infrastruktur, auf der das System ausgeführt wird, wie zum Beispiel Standorte, Umgebungen, Maschinen oder Prozessoren. Da der Cloud-Marketplace zu einem späteren Zeitpunkt in eine Private Cloud überführt werden soll, werden die einzelnen Microservices schon für den Prototypen containerisiert. Für die Open Source-Software Keycloak, NATS und PostgreSQL existieren schon Docker-Container. Für die restlichen Microservices werden Dockerfiles geschrieben. Die Bereitstellung mit Docker Compose wird in Kapitel 6.4 kurz erläutert.

Prinzipiell ist es möglich, dank der losen Kopplung, die Microservices, Datenbanken und die MoM alle auf verschiedene Maschinen und in verschiedenen Netzwerken bereitzustellen. Dies ist jedoch aus Performancegründen, aufgrund von steigenden Latenzen, nicht empfehlenswert. Zusätzlich sind die Übertragungen zwischen den Microservices und der MoM sowie die Übertragungen zwischen den Microservices und dem IAM vertraulich und bedürfen Schutz vor dem Auslesen Dritter. Wenn es möglich wäre, dass ein Dritter den Netzwerkverkehr zwischen den Services auslesen kann, müssten die Services für die Kommunikation über Hypertext Transfer Protocol Secure (HTTPS) oder einer anderen Transport Layer

Security (TLS)-Verbindung stattfinden, um die Daten vertraulich zu halten sowie um die Integrität der Daten zu schützen. Es bietet Vorteile, wenn die Services in einer isolierten Umgebung bereitgestellt werden, wie beispielsweise einem eigenen Netzwerk einer Private Cloud. Dann gäbe es verschiedene Möglichkeiten, die interne Kommunikation von außen zu schützen, wie beispielsweise durch eine Firewall oder einer Web Application Firewall (WAF). Wenn die interne Kommunikation sichergestellt ist, wird eine Verwendung von internen Verschlüsselungen, wie TLS oder HTTPS theoretisch nicht mehr notwendig sein. Eine Kommunikation über Hypertext Transfer Protocol (HTTP) wäre dann wesentlich performanter, jedoch müssten die verschiedenen internen Microservices dem äußeren Schutzmechanismus vertrauen. Dies widerspricht dem Zero-Trust-Prinzip, bei dem niemandem automatisch vertraut wird. Aufgrund der Einhaltung des Zero-Trust-Prinzips im LGLN wird die interne Kommunikation zwischen der MoM und den Microservices über einen verschlüsselten und integritätsgesicherten Kanal stattfinden. Der Netzwerkverkehr mit externen Akteuren muss ebenfalls jederzeit über einen verschlüsselten und integritätsgesicherten Kanal stattfinden.

5.5.1 Service Broker-Infrastruktur

Dem Service Broker ist es erlaubt in einer Umgebung zu arbeiten, die am geeignetsten für den Besitzer des Produktes ist. Der Service Broker muss nicht notwendigerweise im gleichen Netzwerk arbeiten wie der Cloud-Marketplace. Der Service-Provider hat demnach die freie Auswahl für eine passende und günstige Umgebung für den Service Broker. Eine Bereitstellung bei einem der Public Cloud-Anbieter oder auf einem gemieteten Server ist beispielsweise möglich.

5.6 Querschnittliche Konzepte

Dieser Abschnitt beschreibt übergreifende, prinzipielle Regelungen und Lösungsansätze, die an mehreren Stellen, also querschnittlich, relevant sind [Hruschka u. a. 2022].

5.6.1 Datenbanken

Alle Datenbankdaten, Nachrichten und Transferobjekte lassen sich als Objekt definieren. Programmiertechnisch lassen sich für diese Objekte Klassen definieren.

Verschiedene Datenbankobjekte, Nachrichten oder Transferobjekte lassen sich dann als Objekt dieser Klasse abbilden. Die Klasse definiert demnach das Datenmodell für den Transfer oder für eine Tabelle der Datenbank. Aufgrund dessen bietet es sich an relationale Structured Query Language (SQL)-Datenbanken für die Persistierung von Daten zu verwenden. Mit SQL-Datenbaken lassen sich, unter Verwendung von Frameworks, diese Objekte performant persistieren. Auch das Auslesen und Suchen aus der Datenbank funktioniert mit den Objekten und Attributen der Klasse.

Nach Vorgabe des LGLN wird in der Arbeit PostgreSQL verwendet, da sich diese Datenbank bereits im Technologie-Stack des LGLN anfindet.

5.6.2 Message-oriented Middleware und Message-Queues

Eine MoM agiert als Empfänger, Sender und Speicher für Nachrichten. Zudem ist es der MoMs möglich, Nachrichten zu persistieren und für den Fall eines Service-Ausfalls die Nachrichtenübertragung zu garantieren. In Abbildung 5.12 ist die Funktionsweise der MoM dargestellt. Die MoM besteht aus mindestens einem Message Broker und mindestens einer Message-Queue. In der Abbildung sind es beispielhaft zwei Message-Queues, in die der Message Broker die eingehenden Nachrichten navigiert. Der Sender adressiert seine Nachricht nur an eine Empfängergruppe, auch *topic* genannt. Der Message Broker navigiert diese Nachricht dann in die ensprechenden Message-Queues. Aus einer Message-Queue können mehrere Instanzen lesen und Nachrichten können auch in mehrere Message-Queues gesendet werden. Es gibt es verschiedene Arten der Nachrichtenübertragung, im Folgenden wird das Request-Reply-Pattern und das Publish-Subscribe-Pattern kurz erläutert, da diese in der Architektur verwendet werden.

Request-Reply-Pattern

Das Request-Reply-Pattern funktioniert wie Anfrage und Antwort. Der Sender verpackt in seine Nachricht, welche eine Anfrage einer Information oder einem Objekt enthält, eine *reply-to* Adresse. Der Receiver (Empfänger), der die Nachricht empfängt und verarbeitet, kann die Antwort anschließend an die *reply-to* Adresse schicken.

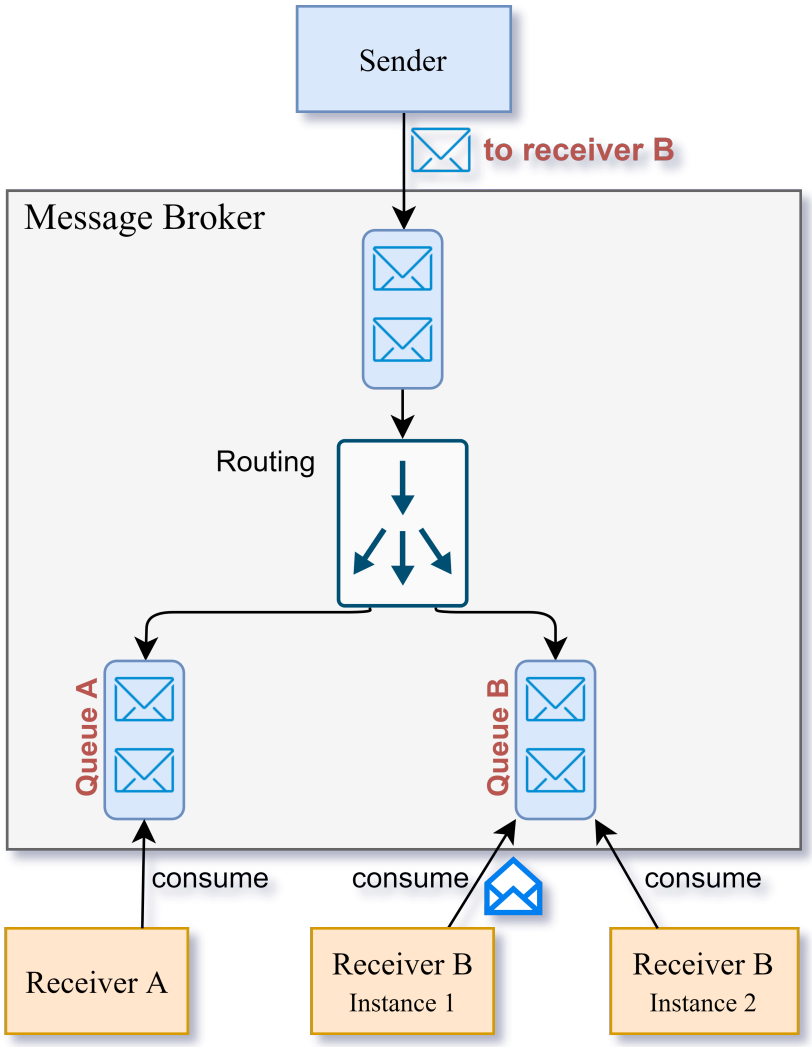


Abbildung 5.12: Funktionsweise der Message-oriented Middleware [Schiborr u. a. 2021]

Publish-Subscribe-Pattern

Das Publish-Subscribe-Pattern hingegen beinhaltet keine Antwortfunktion. Denn dieses Pattern funktioniert nach dem *fire-and-forget* (senden und vergessen) Prinzip. Der Sender schickt die Nachricht los und interessiert sich im Anschluss nicht mehr an den weiteren Prozess dahinter. Der Sender kann jedoch beispielsweise das Interesse haben, dass die Nachricht *at-least-once* (mindestens einmal) von einem Empfänger gelesen wird.

5.6.3 Authentisierung und Autorisierung von Nutzern

Bis auf das Anzeigen des Cloud-Marketplace-Portfolios sind alle Aktionen im Cloud-Marketplace schutzbedürftig. Deshalb wird bei allen schutzbedürftigen Interaktionen mit dem Cloud-Marketplace eine Anmeldung vorausgesetzt. Der Nutzer authentisiert sich über das Frontend bei der Keycloak-API und erhält einen Nutzer-Token zurück. Dieser Nutzer-Token ist nur für eine kurze Zeit gültig, kann jedoch erneuert werden. Im Nutzer-Token sind Informationen des Nutzers, unter anderem eine eindeutige Nutzer-ID sowie die Nutzer-Rollen, enthalten. Bei jeder Interaktion mit dem Cloud-Marketplace wird im Microservice überprüft, ob dieser Nutzer-Token, welcher im HTTP-Header hinterlegt ist, gültig ist, indem die Signatur des Nutzer-Tokens geprüft wird. Dadurch wird sichergestellt, das der Nutzer-Token nur von dem im Microservice definierten IAM erstellt worden ist. Diesem Nutzer-Token, mit den enthalten Informationen zum Nutzer, vertraut der Microservice anschließend und autorisiert den Nutzer anhand der in dem Nutzer-Token enthaltenen IDs und Nutzer-Rollen.

5.7 Architekturentscheidungen

In den Architekturentscheidungen werden wichtige, teure, große oder riskante Architektur- oder Entwurfsentscheidungen inklusive der jeweiligen Begründungen und Konsequenzen der Auswahl kurz erläutert [Hruschka u. a. 2022].

5.7.1 Begründung und Konsequenzen von der Message-oriented Middleware

Die eingesetzte MoM bietet Vorteile, wie das asynchrone und synchrone Kommunizieren zwischen Services. Services müssen nicht immer verfügbar sein, da die MoM Nachrichten speichern kann, bis der Empfänger-Service wieder verfügbar ist. Dies bietet eine lose Kopplung, mehr Toleranz und eine bessere Verfügbarkeit des Gesamtsystems.

Jedoch ist die Konsequenz einer MoM die zusätzliche Komplexität und der Mehraufwand für das Erstellen, Einrichten und möglicherweise Skalieren der MoM. Die MoM ist ein Single Point of Failure (SPOF), das bedeutet, dass wenn die MoM ausfällt, kann das möglicherweise Auswirkungen auf das ganze System haben.

Deshalb sind MoMs im besten Fall skaliert oder in einem Cluster aktiv. Es ratsam einen Trade-off zwischen der Anzahl an MoM-Instanzen und Kosten für die Instanzen abzuwägen. Eine kurze Anleitung für die Bereitstellung eines NATS-Clusters befindet sich im Kapitel 6.2.

5.7.2 Begründung und Konsequenzen von Microservices

In dieser Architektur werden mehrere Microservices eingesetzt. Dies hat nicht nur Vorteile gegenüber einer monolithischen Architektur wie im Kapitel 2.1 erläutert. Die verteilte Microservices-Architektur erzeugt zusätzliche Komplexität, Netzwerk-Latenzen und eine ungleichmäßige Lastverteilung. Dennoch überwiegen die Vorteile der Microservices-Architektur gegenüber die der monolithischen Architektur (siehe Kapitel 2.1.1).

5.8 Qualitätsanforderungen

Szenarien operationalisieren Qualitätsanforderungen und machen deren Erfüllung mess- oder entscheidbar [Hruschka u. a. 2022]. Im Folgenden wird eine kurze Übersicht angegeben, wie der Cloud-Marketplace reagieren muss, um die Qualitätsziele zu erfüllen. Eine ausführliche Evaluation der Qualitätsanforderungen durch Experimente befindet sich Kapitel 7.2 der Evaluation.

5.8.1 Qualitätsszenarien

- Alle Cloud-Marketplace Microservices sowie das IAM und die NATS-Services sind horizontal skalierbar.
- Der Ausfall einer Service-Instanz innerhalb von zehn Sekunden hat keinen Einfluss auf weitere Services.
- Auch bei dem Ausfall einer NATS-Instanz innerhalb von zehn Sekunden ist die Nachrichtenübertragung sichergestellt.
- Das IAM blockiert nach drei Fehlversuchen die Anmeldung für 60 Sekunden.

5.9 Risiken und technische Schulden

Dieser Abschnitt ist eine Auflistung von Risiken und technischen Schulden mit, wenn möglich, vorgeschlagenen Maßnahmen zur Risikovermeidung, Risikominimierung oder dem Abbau der technischen Schulden [Hruschka u. a. 2022].

5.9.1 Skalierbarkeit und Elastizität

Für eine dynamische Elastizität ist eine dynamische Skalierbarkeit notwendig. Dynamisches Skalieren, also das automatische Erstellen weiterer Service-Instanzen, ist in der aktuellen Lösung der Bereitstellung des Cloud-Marketplaces noch nicht umgesetzt. Mit Docker Compose ist es möglich statisch, also zu Beginn der Bereitstellung oder durch aktives Eingreifen, die Anzahl der Service-Instanzen einzustellen. Ein flexibles und dynamischeres Einstellen der Service-Instanzen ist mit Kubernetes möglich. Kubernetes bringt jedoch nicht automatisch die Funktion für ein automatisches Skalieren bei erhöhter Arbeitslast. Diese Funktion lässt für das System, flexibel und günstig, so viele Instanzen aktiv, wie benötigt werden. Es fordert ein hohes Maß an Kapazitäten ein solches System zuverlässig zu erstellen, dennoch wäre dies eine kostensenkende und qualitätssteigernde Funktion für das Produktivsystem.

5.9.2 Risiko und Alternativen zu NATS

Eine MoM birgt schon Risiken eines SPOF, wie im Kapitel 5.7.1 erläutert. Jedoch sollte auch die Technologieauswahl der MoM bedacht werden. Es gibt weitere, aktuell stärker verbreitete, Technologien für eine MoM, wie beispielsweise Kafka, RabbitMQ oder ActiveMQ. Auch nach der Entwurf- und Implementierungsphase des Cloud-Marketplaces muss stetig weiter überprüft werden, ob eine Technologie weiterhin für den Einsatz geeignet ist und ob diese Technologie kontinuierlich mit Sicherheitsupdates versorgt wird.

6 Prototypische Umsetzung

Dieses Kapitel widmet sich einer kurzen Vorstellung der wichtigen Aspekte der prototypischen Umsetzung. Es wird eine Erläuterung zur Technologieauswahl, Implementierungs- und Konfigurationsaspekte sowie zur Bereitstellung der Software gegeben.

6.1 Framework und Technologien der Microservices

Prinzipiell lassen sich Microservices unterschiedlichst mit allen gängigen Programmiersprachen umsetzen. Im Rahmen dieser Arbeit müssen die Microservices eine REST-API, eine Datenbankverbindung, eine Verbindung zum NATS-Service sowie eine Verbindung zu der Keycloak ermöglichen können.

Für eine effiziente Umsetzung der Microservices bietet es sich an Frameworks zu verwenden. Es gibt zahlreiche Frameworks, die bei der Implementierung der Microservices unterstützen können. Bekannte Frameworks für diesen Anwendugszweck sind zum Beispiel das *Spring* Framework, *Dropwizard*, *Flask*, *Falcom*, *Go-Micro* oder *Quarkus* [Vinugayathri 2022]. Diese Frameworks können den Entwicklungsaufwand reduzieren, indem sie Softwarepakete oder Funktionen, wie zum Beispiel einen REST-Controller oder ein Datenbankmodul, anbieten.

Nach Vorgabe des LGLN wird für den Prototypen und für alle Microservices das Spring Framework verwendet. Spring bietet Software-Dependencies für eine REST-API, für Datenbankverbindungen mit verschiedenen Datenbanken sowie eine Verbindung zum Keycloak, welches das Implementieren der Microservices beschleunigt.

In Abbildung 6.1 ist die Webseite *Spring Initializr* mit einer beispielhaften Erstellung eines Microservices dargestellt. Diese Webapplikation erstellt für den Microservice das initiale Spring-Softwarepaket und stellt es für die Weiterentwicklung zur Verfügung. Zudem ist es möglich, direkt oder zu einem späteren Zeitpunkt,

passende Dependencies für den Microservice aus einer langen Dependencies-Liste auszuwählen. Für eine Verbindung mit einem NATS-Service bietet NATS einen in Java implementierten Client an.

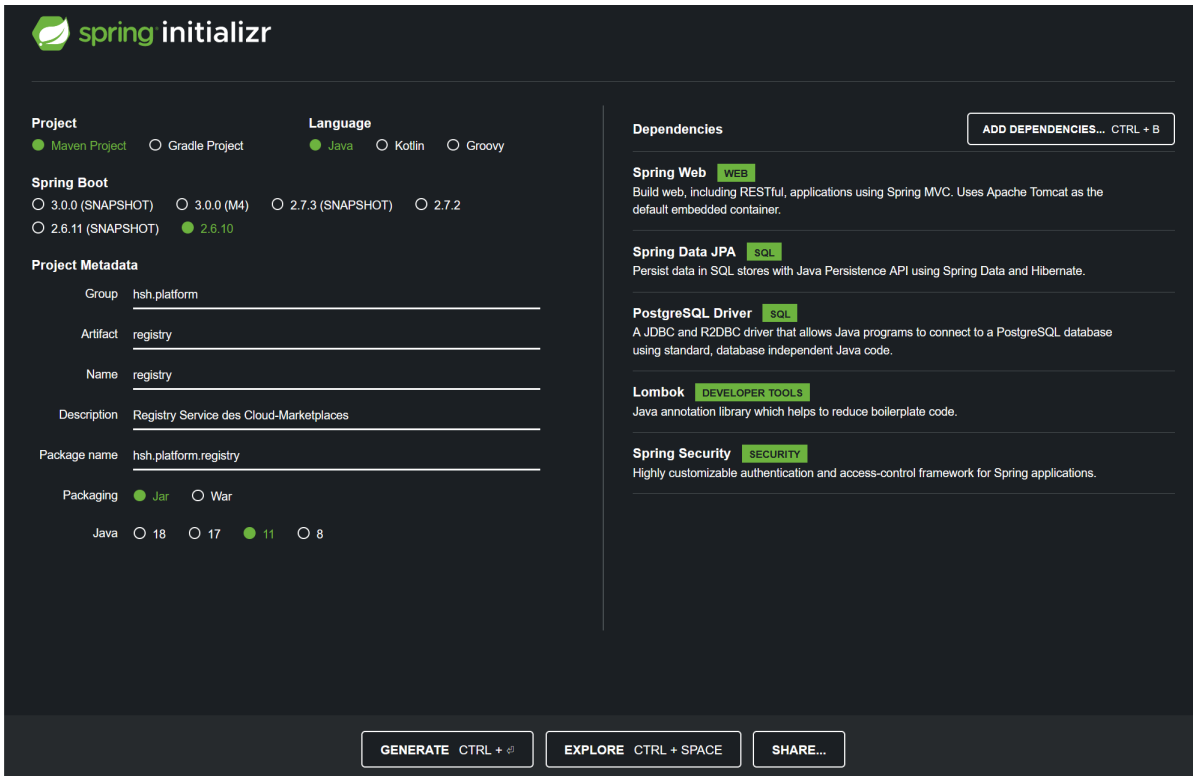


Abbildung 6.1: Spring Initializr für Microservices [VMware-Inc. 2022b]

6.1.1 Service Broker

Prinzipiell lässt sich der Service Broker auch unterschiedlichst mit allen gängigen Programmiersprachen umsetzen. Die Wahl der Programmiersprache und des Frameworks für die Implementierung des Service Brokers ist stark abhängig von der Funktionalität des Service-Produktes. Deshalb lassen sich dafür keine allgemeinen Empfehlungen aussprechen.

Aus experimentellen Gründen werden in dieser Arbeit drei kleine Service Broker

implementiert. Diese Service Broker erfüllen die grundlegende Open Service Broker API. Die Service Broker werden auch mit dem Spring Framework umgesetzt. Für die Umsetzung der Open Service Broker API wird das Spring-Modul *Spring Cloud Open Service Broker* [VMware-Inc. 2022a] verwendet. Das SaaS-Produkt simuliert in allen drei Fällen einen beispielhaften Open Source-E-Mail-Service von Baeldung [baeldung 2022]. Die Bereitstellung und die Containerisierung werden in den folgenden Kapiteln erläutert.

6.2 NATS

NATS bietet für die Bereitstellung eines NATS-Services die Installation über Docker, über einen Package-Manager oder über das Downloaden und Installieren von verschiedenen Release-Builds. Da der Prototyp des Cloud-Marketplaces über Docker Compose bereitgestellt wird, wird für NATS auch die Docker-Lösung gewählt.

Für eine hochverfügbare und ausfallsichere MoM ist es notwendig, nicht nur einen NATS-Service zu installieren. Quelltext 6.1 enthält eine Docker Compose-Konfiguration für ein NATS-Cluster. Im Cluster befinden sich drei NATS-Instanzen des Docker-Images *synadia/jsm:nightly*. Alle drei Instanzen werden mit einem Containernamen versehen und mit dem *command* Befehl miteinander in einem Cluster vernetzt. Zusätzlich erhalten alle drei Instanzen ein Volume für die Persistierung der Events.

```
1  version: '3'
2
3  services:
4      nats1:
5          container_name: nats1
6          image: synadia/jsm:nightly
7          entrypoint: /nats-server
8          command: --name NATS1 --cluster_name JSC --js --sd /data
       --cluster nats://0.0.0.0:4245 --routes nats://nats2:4245,nats
       ://nats3:4245 -p 4222
9          ports:
10           - 4222:4222
11         volumes:
12           - ./jetstream-cluster/nats1:/data
13
```

```
14     nats2:
15         container_name: nats2
16         image: synadia/jsm:nightly
17         entrypoint: /nats-server
18         command: --name NATS2 --cluster_name JSC --js --sd /data
       --cluster nats://0.0.0.0:4245 --routes nats://nats1:4245,nats
       ://nats3:4245 -p 4222
19         ports:
20           - 4223:4222
21         volumes:
22           - ./jetstream-cluster/nats2:/data
23
24     nats3:
25         container_name: nats3
26         image: synadia/jsm:nightly
27         entrypoint: /nats-server
28         command: --name NATS3 --cluster_name JSC --js --sd /data
       --cluster nats://0.0.0.0:4245 --routes nats://nats1:4245,nats
       ://nats2:4245 -p 4222
29         ports:
30           - 4224:4222
31         volumes:
32           - ./jetstream-cluster/nats3:/data
```

Quelltext 6.1: NATS JetStream-Cluster

6.2.1 NATS-Queues

Da die Microservices, welche mit dem Spring Framework erstellt sind, in Java geschrieben sind, wird für die Interaktion mit den NATS-Queues der Java-Client von *nats-io* verwendet [nats-io 2022]. In Quelltext 6.2 ist ein Beispiel für das Lesen aus einer NATS-Queue mit dem Namen *updateInstanceTopic*. Zuerst wird eine Verbindung zu dem NATS-Cluster hergestellt. Anschließend wird ein *Dispatcher*-Objekt für das Empfangen von Events erstellt. Zudem wird ein Hilfsobjekt für das Erstellen von Java-Objekten aus JSON instanziiert. In Zeile 8 des Quelltextes wird eine Funktion definiert, welche bei dem Empfangen einer Nachricht ausgeführt wird. Zuerst wird in dieser Funktion ein Java-Objekt aus dem Inhalt der Nachricht, welches als JSON versendet wird, erstellt. Im Anschluss lässt sich weitere Anwendungslogik starten.

```
1 Connection nats = Nats.connect("nats://nats1:4222,nats://nats2
     :4222,nats://nats3:4222");
2
3 Dispatcher dispatcher = nats.createDispatcher(msg -> {});
4
5 GsonBuilder builder = new GsonBuilder();
6 Gson gson = builder.create();
7
8 dispatcher.subscribe("updateInstanceTopic", msg -> {
9     String jsonResponse = new String(msg.getData(),
     StandardCharsets.UTF_8);
10
11     InstancePersistenceQueueDTO instancePersistenceQueueDTO =
     gson.fromJson(jsonResponse, InstancePersistenceQueueDTO.class)
     ;
12
13     doSomething(instancePersistenceQueueDTO);
14     });
```

Quelltext 6.2: NATS: In Message-Queues senden in Java

In Quelltext 6.3 ist ein Beispiel für das Senden in eine NATS-Queue mit dem Namen *updateInstanceTopic*. Zuerst wird eine Verbindung zu dem NATS-Cluster hergestellt. Anschließend wird ein Hilfsobjekt für das Erstellen von Java-Objekten aus JSON instanziiert. In Zeile 7 des Quelltextes wird in die Message-Queue *createInstanceCredentialsTopic* eine Nachricht gesendet. Um Java-Objekte zu verschicken, muss dieses Objekt vorher in ein JSON-Objekt konvertiert werden.

```
1 Connection nats = Nats.connect("nats://nats1:4222,nats://nats2
     :4222,nats://nats3:4222");
2
3 GsonBuilder builder = new GsonBuilder();
4 Gson gson = builder.create();
5
6 String json = gson.toJson(usageCreateCredModel);
7 nats.publish("createInstanceCredentialsTopic", json.getBytes());
```

Quelltext 6.3: NATS: Aus Message-Queues lesen in Java

6.2.2 NATS JetStream

NATS JetStream ist eine NATS-Funktion für die Persistierung von Events, die eine Message-Queue passieren. In Quelltext 6.1 ist die Erstellung eines JetStream-

fähigen Clusters dargestellt. Es gibt die Möglichkeit, die Events in eine Datenbank zu schreiben oder, wie es in der Docker Compose-Konfiguration von Quelltext 6.1 umgesetzt ist, in eine Datei. Um das Schreiben der Events in einen JetStream zu aktivieren, muss vorher ein Stream erstellt werden. In Quelltext 6.4 ist die Erstellung eines Streams für die Persistierung von Events dargestellt.

```
Connection nats = Nats.connect("nats://nats1:4222,nats://nats2
   :4222,nats://nats3:4222");
JetStreamManagement jsm = nats.jetStreamManagement();

StreamConfiguration streamConfig = StreamConfiguration.builder()
                .name("Stream1")
                .storageType(StorageType.File)
                .subjects("persistInstanceTopicStream")
                .build();

// Create the stream
StreamInfo streamInfo = jsm.addStream(streamConfig);
```

Quelltext 6.4: NATS: JetStream erstellen

6.3 Containerisierung

Nach der Implementierung der Microservices ist die Containerisierung der nächste Schritt für deren Bereitstellung. Für die Containerisierung mit Docker ist ein Dockerfile notwendig. In Quelltext 6.5 ist das Dockerfile für die Service Registry dargestellt. Das Dockerfile ist ein textbasiertes Skript mit Anweisungen, dass zur Erstellung eines Container-Images verwendet wird. Die Dockerfiles der anderen drei Microservices sind analog zu dem der Service Registry (siehe Anhang Seite 103). Das Dockerfile erstellt mit dem Befehl *mvn package* ein lauffähiges Programm und packt alle notwendigen Pakete dazu.

```
FROM maven:3.6.1-jdk-11-slim AS build
RUN mkdir -p /workspace
WORKDIR /workspace
COPY pom.xml /workspace
COPY src /workspace/src
COPY ./src/main/resources/application.yml .
```

```
RUN mvn -f pom.xml clean package -e

FROM openjdk:11
COPY --from=build /workspace/target/*.jar registry.jar
EXPOSE 8081
ENTRYPOINT ["java","-jar","registry.jar"]
```

Quelltext 6.5: Dockerfile für die Service Regitry

6.4 Bereitstellung mit Docker Compose

In einer Docker Compose-Datei befindet sich die Bereitstellung des gesamten Cloud-Marketplaces. Diese befindet sich im Anhang auf Seite 105. In dieser Docker Compose-Datei befinden sich die notwendigen Konfiguration und Netzwerkeinstellungen für mehrere Docker-Container, welche alle mit einem Befehl erstellt oder gestartet werden können.

In Quelltext 6.6 ist die Docker Compose-Konfiguration für die Servie Registry dargestellt. Der Service Registry wird Zugriff auf verschiedene Netzwerke gewährt und wartet vor dem Start auf die Container *registry-postgres*, *keycloak* und *nats1*. Der Eintrag *build: ../registry* bedeuted, dass als Docker-Image das lokale Dockerfile aus dem angegebenen Ordner gewählt wird.

```
registry-service:
    build: ../registry
    container_name: registry
    ports:
      - "8081:8081"
    networks:
      - keycloak-net
      - nats-net
      - registry-postgres-net
      - brokers-net
    depends_on:
      - registry-postgres
      - keycloak
      - nats1
    links:
      - keycloak
```

Quelltext 6.6: Registry Service-Docker Compose-Konfiguration

In Quelltext 6.7 ist die Docker Compose-Konfiguration für die PostgreSQL für die Service Registry dargestellt. Als Docker Image wird hier kein lokales Dockerfile, sondern ein öffentliches Docker Image aus dem Internet, verwendet. Der Datenbank wird zusätzlich ein Volume für die Persistierung sowie einige Environment Variablen zugewiesen. Die Konfiguration der restlichen Microservices und Datenbanken sind analog zu denen der Service Registry.

```
registry-postgres:
    image: 'postgres:alpine'
    restart: unless-stopped
    container_name: registry-postgres
    volumes:
      - postgres-registry-db:/var/lib/postgres
    environment:
      POSTGRES_PASSWORD: "registry-postgres-passwort"
      POSTGRES_USER: "postgres"
      POSTGRES_DB: "serviceBrokerDB"
    ports:
     - "5432:5432"
    expose:
      - "5432"
    networks:
      - registry-postgres-net
```

Quelltext 6.7: PostgreSQL-Konfiguration für die Service Registry

In Quelltext 6.8 ist die Docker Compose-Konfiguration für den Keycloak Service dargestellt. Die Keycloak benötigt für die Persistierung der Daten eine Datenbank. Dem Container werden über die Environment-Variablen die notwendigen Information für die Verbindung zur Datenbank übergeben. Für die prototypische Umsetzung wurden die Benutzer und Passwörter für die Keycloak und die Datenbanken als Environment-Variablen übergeben. Für ein Produktivsystem muss dieses durch eine sichere Methode, wie beispielsweise *Docker secrets*, ausgetauscht werden.

```
keycloak:
    image: quay.io/keycloak/keycloak:legacy
    container_name: keycloak
    environment:
      HTTP_ENABLED: "true"
```

```
6       DB_VENDOR: postgres
7       DB_ADDR: keycloak_db_postgres
8       DB_DATABASE: keycloak
9       DB_USER: keycloak
10      DB_PASSWORD: keycloak-password-for-postgres
11      KEYCLOAK_USER: admin
12      KEYCLOAK_PASSWORD: lgln
13      PROXY_ADDRESS_FORWARDING: "true"
14    depends_on:
15      - keycloak_db_postgres
16    ports:
17      - "8080:8080"
18      - "8443:8443"
19    networks:
20      - keycloak-net
```

Quelltext 6.8: Docker Compose-Konfiguration für den Keycloak Service

In der Docker Compose-Konfigurationen lassen sich, wie in Quelltext 6.9 dargestellt, Netzwerke erstellen. Docker-Container lassen sich anschließend diesen Netzwerken hinzufügen.

```
1  networks:
2    keycloak-net:
3      driver: bridge
4    registry-postgres-net:
5      driver: bridge
6    nats-net:
7      driver: bridge
8    brokers-net:
9      driver: bridge
10   instances-postgres-net:
11     driver: bridge
12   usage-postgres-net:
13     driver: bridge
```

Quelltext 6.9: Docker Compose-Netzwerkkonfiguration

In Docker Compose-Konfigurationen lassen sich, wie in Quelltext 6.10 dargestellt, Volumes erstellen. Volumes können anschließend an verschiedenen Docker-Container angebunden werden. Eine ausführliche Anleitung für Installation des ganzen Cloud-Marketplaces befindet sich im Anhang auf Seite 100.

```
volumes:
  postgres-keycloak-db:
    driver: local
  postgres-registry-db:
    driver: local
  postgres-instances-db:
    driver: local
  usage-instances-db:
    driver: local
```

Quelltext 6.10: Docker Compose-Volume-Konfiguration

7 Vergleich und Evaluation

Die Evaluation des Cloud-Marketplaces gliedert sich in eine konzeptionelle und eine technische Evaluation. Im konzeptionellen Teil der Evaluation wird ein Vergleich der vorgestellten Architektur mit vergleichbaren Umsetzungen evaluiert. Im technischen Teil der Evaluation wird die Zielsetzung anhand verschiedener Experimente und Untersuchungen geprüft. Die Evaluation des technischen Teils bezieht sich auf die definierten Bewertungskriterien des Anforderungskataloges aus Kapitel 4.

7.1 Konzeptionelle Evaluation anhand vergleichbarer Umsetzungen

Als vergleichbare Umsetzungen verstehen sich theoretische, alternative Architekturen sowie bereits existierende Lösungen für den Cloud-Marketplace. Theoretische alternative Architekturen sind beispielsweise eine monolithische Architektur, eine Architektur ohne eine MoM oder eine Architektur mit unterschiedlichen Strategien zur Umsetzung der Kernaufgabe des Cloud-Marketplaces. Im ersten Teil der konzeptionellen Evaluation wird daher kurz die Notwendigkeit der Microservice-Architektur im Vergleich zu alternativen Architekturen zusammengefasst.

Es existieren eine Reihe von bestehenden Lösungen für einen Cloud-Marketplace. Dazu zählen unter anderem der AWS Marketplace, Google Cloud-Marketplace, Microsoft Azure Marketplace, IBM Cloud-Marketplace, der Chrome Web Store, der internet4YOU Cloud-Marketplace oder der Business-Marketplace der Deutschen Telekom. Die meisten Cloud-Marketplaces bieten eine Reihe an Funktionalitäten und Produkte für Service-Provider und Service-Consumer an, welcher weit über das ausschließliche Vermarkten von SaaS-Produkten hinaus geht. So bieten die meisten der größten Cloud-Anbieter dieselben grundlegenden Dienste an:

- Verschieden ausgestattete Virtuelle Maschinen,
- Container und SaaS-Produkte,
- Datenbanken oder andere Speichermedien,
- Serverlose Funktionen,
- Isolierte Cloud-Netzwerke.

Die bestehenden Cloud-Anbieter mit integriertem Cloud-Marketplace unterscheiden sich darin, dass alle dieselben Dienste anbieten können, jedoch alle ihre eigene Strategie dafür entwickelt haben, diese Dienste umzusetzen. Daher wird in Kapitel 7.1.2 nur ein Cloud-Anbieter, der Microsoft Azure Marketplace, für den Vergleich zu der vorgestellten Architektur des Cloud-Marketplaces ausgewählt.

7.1.1 Vergleich zu alternativen Architekturen

Der Cloud-Marketplace wurde nach einer Cloud Native-EDRS-Architektur entworfen, die sich durch besondere Eignung zur Skalierbarkeit ausweist. Diese Skalierbarkeit ist zwingend notwendig, um die Qualitätsziele der Applikation zu erfüllen. Fehlt die Möglichkeit einer effizienten Skalierbarkeit der einzelnen Services oder Datenbanken, lassen sich die Qualitätsziele, wie eine gute Elastizität oder Resilienz, nicht mehr erfüllen. Zudem könnten, bei steigender Arbeitsbelastung des Cloud-Marketplaces, Prozesse verlangsamt werden, wodurch die UX leiden würde. Die Vorteile der Microservices, definiert im Kapitel 2.1.3, erfüllen dazu weitere Anforderung, wie beispielsweise eine gute Wartbarkeit, denn durch Microservices ist es möglich, einzelne Services zu erweitern, warten und neu bereitzustellen, ohne dass dies mit dem gesamten System geschehen muss.

Zusammenfassend ist die monolithische Architektur, eine Architektur ohne MoM oder eine nicht Cloud Native-Architektur keine bessere Alternative zur vorgestellten Architektur für den Cloud-Marketplace.

7.1.2 Vergleich zum Microsoft Azure Marketplace

Der Microsoft Azure Marketplace bietet, neben den geforderten funktionalen Anforderungen von Kapitel 4.2, eine große Menge an Funktionalitäten und Produkten für ihre Nutzer. Seit dem 1. Februar 2010 ist Microsoft Azure als Public Cloud Anbieter erstmals offiziell verfügbar und bot den Nutzern zuerst IaaS, PaaS und

SaaS an. Seitdem steigt die Anzahl der Funktionalitäten und Produkten der Public Cloud und dem Cloud-Marketplace, sowie deren Qualität stetig weiter [Stark 2022].

Der Cloud-Marketplace besitzt, wie in der Übersicht in Abbildung 7.1 dargestellt, eine ausgereifte und sehr umfangreiche Dokumentation für einen Service-Provider des Cloud-Marketplace. Die Dokumentation ist in verschiedenen Sprachen erhältlich und enthält zahlreiche Schrittanleitungen zu verschiedenen Angeboten, Produkten oder Richtlinien. Jedoch befindet sich der Microsoft Azure Marketplace weiterhin in stetiger Weiterentwicklung, wodurch es in der Vergangenheit dazu kam, dass Weblinks zu Teilaspekten der großen Dokumentation vorübergehend nicht richtig waren [Microsoft 2022b].

Der Service-Provider wird bei der Erstellung eines Produktes an eine Vielzahl von APIs und Richtlinien speziell für den Microsoft Azure Marketplace gebunden. So ist es eine Vorgabe, die Azure Active Directory und die Microsoft Accounts für die Authentifizierung von Azure-Nutzern in den Service-Instanzen anzubieten. Der Cloud-Marketplace bietet den Service-Providern jedoch auch weitere Features für eine passende Softwarelösung, wie zum Beispiel eine Anbindung an eine, von Azure bereitgestellte, MoM oder eine Cloud-Datenbank.

Für den Service-Provider gibt es demnach, nach einer möglicherweise längeren Einarbeitungsphase in die große Anzahl von Richtlinien für die Verwendung des Cloud-Marketplaces, alle Möglichkeiten, Anwendungen und Dienste für Kunden weltweit anzubieten.

Der Microsoft Azure Marketplace erfüllt zudem alle funktionalen Anforderungen des Service-Consumers. Der Service-Consumer ist verpflichtet einen Microsoft-Account zu erstellen und für die Anmeldung bei den SaaS-Produkten zu verwenden. Als authentifizierter Nutzer kann der Service-Consumer mit dem Cloud-Marketplace interagieren und zusätzlich verschiedene Servicemodelle der Microsoft Azure Cloud, überwiegend kostenpflichtig, benutzen.

Der größte Nachteil des Microsoft Azure Marketplace, im Vergleich zur eigenen Cloud-Marketplace-Lösung, ist die Gebühr für das Anbieten jedes einzelnen Produktes. Der Azure Marketplace nimmt eine Gebühr von 3% auf jede Einnahme des Produktes. Dazu kommen die Kosten für die kostenpflichtige Provisionierung verschiedener Servicemodelle in der Azure Cloud, welche nicht nur Kosten für den Service-Provider, sondern auch für den Service-Consumer sein können. Ein weiterer Nachteil ist die fehlende Freiheit für die Bereitstellung des Service Bro-

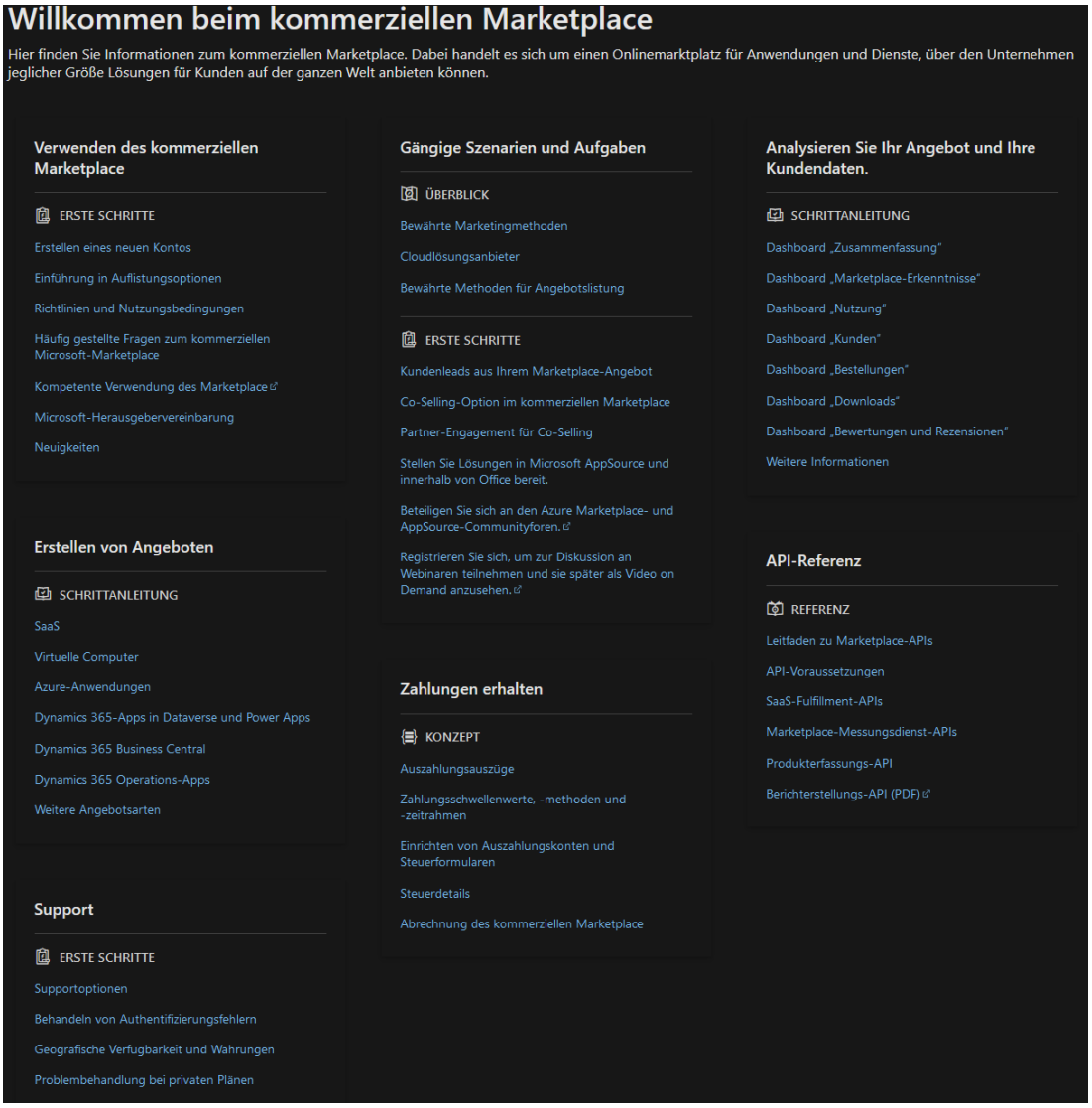


Abbildung 7.1: Microsoft Azure: Informationen zum kommerziellen Marketplace [Microsoft 2022b]

kers. Die vorgestellte Lösung für den Cloud-Marketplace erlaubt es dem Service-Provider den Service Broker an einem beliebigen Ort bereitzustellen. Neben der Erfüllung der Open Service Broker API und einer korrekten Sendung der Nutzungsdaten sind keine komplizierten Richtlinien gefordert. Dies erlaubt es den Service-Provider, mit geringerem Aufwand, seinen Service Broker auch in andere Cloud-Marketplaces zu integrieren. Der letzte Nachteil für einen Einstieg in den Microsoft Azure Marketplace ist die Einarbeitung in die sehr umfangreiche Weboberfläche, die Richtlinien sowie die API-Voraussetzungen des Cloud-Marketplaces.

Die reine Betrachtung der Architektur, Funktionalitäten, Stabilität und Qualität des Microsoft Azure Marketplaces zeigt einen modernen, widerstandsfähigen und hochverfügbaren Cloud-Marketplace. Dies lässt sich auf einen sorgfältig erstellten Entwurf von Microsoft zurückzuführen. Über die interne Architektur des Cloud-Marketplaces lassen sich jedoch nur Vermutungen erstellen, da dieser nicht Open Source ist. Dadurch lässt sich zum Schluss nur noch die Dokumentation zum Onboarding (Registrierung) sowie des Lebenszyklus von SaaS-Produkten untersuchen und anschließend mit der vorgestellten Cloud-Marketplace vergleichen.

Vergleich des Onboardings sowie des Lebenszyklus von SaaS-Produkten

Der Microsoft Azure Marketplace stellt eine ausführliche schriftliche Dokumentation für das Onboarding sowie für die verschiedenen Workflows des SaaS-Lebenszyklus von SaaS-Produkten zur Verfügung. Zusätzlich bietet der Microsoft Azure Marketplace Videotutorials für den Onboarding-Prozess sowie für die verschiedenen Workflows des SaaS-Lebenszyklus [Microsoft 2022a]. Die vom Microsoft Azure Marketplaces erwähnten **SaaS-Angebote** lassen sich mit dem in dieser Arbeit definierten Begriff **SaaS-Produkt** gleichstellen. Im Folgenden wird, anhand einer Schrittanleitung für die Erstellung eines SaaS-Angebotes, die Gemeinsamkeiten und Unterschiede zwischen der vorgestellten Lösung untersucht. Diese Schrittanleitung gliedert sich in acht Schritte [Microsoft 2022a]:

1. Planen eines SaaS-Angebots für den kommerziellen Marketplace
2. Planen eines SaaS-Angebots für Tests und Entwicklung
3. Azure Active Directory und transaktionsfähige SaaS-Angebote im kommerziellen Marketplace
4. Volumenabrechnung für SaaS mit dem Messungsdienst für den kommerziellen Marketplace
5. Erstellen eines SaaS-Angebots
6. Testen eines SaaS-Angebots
7. Verkaufen eines SaaS-Angebots über Microsoft
8. Testen und Veröffentlichen eines SaaS-Angebots im kommerziellen Marketplace

1. Planen eines SaaS-Angebots für den kommerziellen Marketplace

Der erste Schritt widmet sich den verschiedenen Optionen und Anforderungen zum Veröffentlichen von SaaS-Angeboten für den kommerziellen Marketplace von Microsoft. Unterschiedlich für den Service-Provider ist es, dass der Service-Provider zum Authentifizieren von Käufern sowohl *„Microsoft-Konten“* als auch *„Azure Active Directory“* verwenden muss. Zudem muss der Service-Provider für sein SaaS-Produkt eine *„Landingpage“* entwickeln, die nahtloses Anmelden und Onboarding für Kunden bietet, die das SaaS-Angebot erworben haben. Zur Integration des Angebots in den Microsoft Azure Marketplace muss der Service-Provider die *„SaaS-Fulfillment-APIs“* verwenden. Zusätzlich muss für die Kommunikation mit dem Microsoft Azure Marketplace ein *„Webhooker“* implementiert werden. Die einzelnen technischen Optionen und Anforderungen werden in den jeweiligen weiteren Schritten erläutert.

2. Planen eines SaaS-Angebots für Tests und Entwicklung

Um in einer, von dem Produktivsystem getrennten Umgebung zu entwickeln, wird die Erstellung eines separaten Angebots für Test- und Entwicklung empfohlen. So ist es dem Service-Provider möglich, das SaaS-Produkte früh im Entwicklungsprozess in dem Microsoft Azure Marketplace zu integrieren. Dies ermöglicht eine Nutzung der automatischen Tests und Validierungen des Microsoft Azure Marketplaces. Unterschiedlich für den Service-Provider ist es, dass dem Service-Provider für eine Test- und Entwicklungsumgebung ein zusätzliches Registrieren eines SaaS-Produktes angeboten und empfohlen wird.

3. Azure Active Directory und transaktionsfähige SaaS-Angebote im kommerziellen Marketplace

In Abbildung 7.2 ist die Nutzung des Microsoft Azure Active Directorys dargestellt, aus dem sich einige Unterschiede zum vorgestellten Cloud-Marketplace ableiten lassen.

Der Service-Provider muss eine Landingpage erstellen, die den Service-Consumern, nach einer erfolgreichen Buchung eines Abonnements, aufrufen kann. Der Service-Consumer übergibt der Landingpage dabei einen *Buchungs-ID-Token*, welcher in einer URI verpackt, per E-Mail versendet wurde. Der Service-Provider bietet dem Service-Consumer eine Anmeldung über die Azure Active Directory an und lässt sich von dieser einen *Access-Token* für die Kommunikation mit der Microsoft

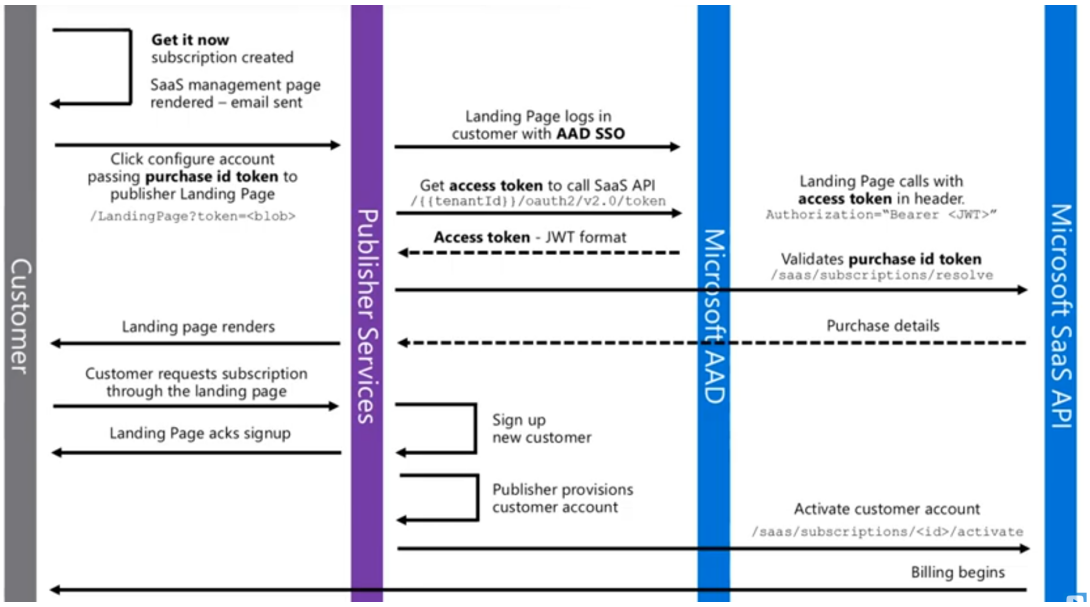


Abbildung 7.2: Microsoft Azure: Sequenzdiagramm für das Provisionieren [Microsoft 2022a]

SaaS API geben. Über diese Kommunikation kann der Service-Provider die Gültigkeit des *Buchungs-ID-Token* prüfen. Eine erfolgreiche Prüfung des *Buchungs-ID-Token* lässt die Landingpage für Service-Consumer erscheinen. Auf dieser Landingpage ist es nun möglich, Nutzer zu erstellen, welche der Service-Provider für die Nutzungserfassung bei der Microsoft SaaS API aktiviert.

4. Volumenabrechnung für SaaS mit dem Messungsdienst für den kommerziellen Marketplace

Die Volumenabrechnung und die Übertragung der Nutzungsdaten sind bei dem Microsoft Azure Marketplace ähnlich umgesetzt wie bei dem vorgestellten Cloud-Marketplace. Wie in Abbildung 7.3 dargestellt, ist es ebenfalls möglich, verschiedene einmalige und dynamische Kosten zu kombinieren.

Zusätzlich erlaubt der Microsoft Azure Marketplace eine monatliche kostenfreie Anzahl an Nutzungseinheiten, wie beispielsweise 2000 freie E-Mails pro Monat.

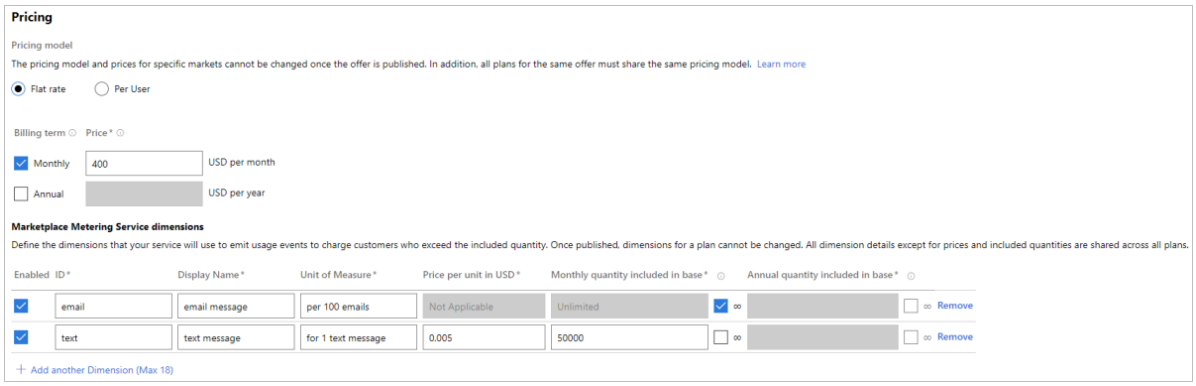


Abbildung 7.3: Microsoft Azure: Beispielangebot für getaktete Abrechnung und Preisgestaltung [Microsoft 2022a]

Zusätzlich lassen sich Pläne als Einheitspreis für SaaS-Produkt oder pro Nutzer erstellen. Eine monatliche oder jährliche Abrechnung der Kosten lassen sich ebenfalls definieren. Des Weiteren bietet der Microsoft Azure Marketplace die Funktion eines kostenfreien ersten Monats.

5. Erstellen eines SaaS-Angebots

Für die Erstellung eines SaaS-Angebots muss der Service-Provider, zusätzlich zum SaaS, eine Landingpage und einen Webhooker implementieren, welche in einer beliebigen Infrastruktur des Service-Providers bereitgestellt werden. Die Aufgaben und Notwendigkeit der Landingpage wurde im Schritt 3 der Schrittanleitung erläutert. Der Webhooker ist ein REST-Service, der POST-Aufrufe des Microsoft Azure Marketplaces entgegennimmt. Diese REST-Aufrufe liefern dem Service-Provider Informationen zu Statusänderungen der SaaS-Instanzen.

In Abbildung 7.4 ist der Lebenszyklus der SaaS-Instanzen als Zustandsdiagramm dargestellt. Eine SaaS-Instanz startet im Status *„Bezahlt, aber nicht aktiviert“*. Eine Aktivierung der SaaS-Instanz durch den Service-Consumer überführt den Status der SaaS-Instanz in den Status *„Provisioniert“*. Von diesem Status kann die SaaS-Instanz in den Status *„Wird aktualisiert“*, *„Gesperrt“* oder *„Abbestellt“* wechseln. Gesperrt wird eine SaaS-Instanz, wenn der Service-Consumer die Bezahlung nicht korrekt tätigt, die Sperrung kann jedoch innerhalb von 30 Tagen wieder rückgängig gemacht werden. Nach den 30 Tagen wird die SaaS-Instanz jedoch abbestellt. Abbestellt kann die SaaS-Instanz auch jederzeit durch den Service-Consumer [Microsoft 2022a].

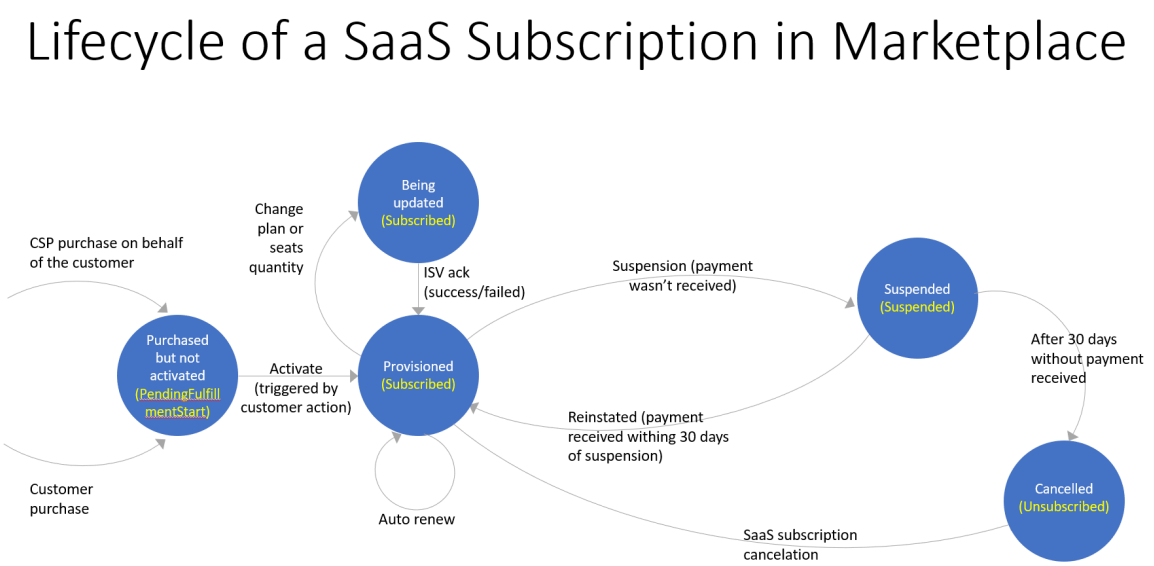


Abbildung 7.4: Microsoft Azure: Zustände eines SaaS-Abonnements [Microsoft 2022a]

Der größte Unterschied ist, dass der Microsoft Azure Marketplace alle Informationen des Service-Providers und des SaaS-Produktes bei der Registrierung abgefragt, validiert, prüft und eventuell zertifiziert. Es gibt die *„SaaS Offer Fulfillment API“*, welche der Microsoft Azure Marketplaces anbietet, die der Service-Provider bei seinem SaaS-Produkt nutzen muss, um die Erstellung und den Lebenszyklus der SaaS-Produkte zu steuern. Die *SaaS Offer Fulfillment API* bietet dem Service-Provider verschiedene Schnittstellen für Aktionen, wie beispielsweise das Auflisten aller bestehenden Abonnements, das Aktivieren von Abonnements, das Abbestellen von Abonnements oder verschiedene Anpassungen des SaaS-Produktes. Der Service-Provider ist demnach selbst für die Erstellung und Verwaltung von Service-Instanzen und Service-Accounts zuständig. Die einzige Kommunikation vom Microsoft Azure Marketplace zum Service-Provider sind die Events bei einer Statusänderung zum Webhooker, wie zum Beispiel das Sperren eines Abonnements, da der Service-Consumer nicht bezahlt hat.

6. Testen eines SaaS-Angebots

Der Microsoft Azure Marketplace bietet und erzwingt eine Reihe an automatischen Validierungen des SaaS-Produktes. Jede Validierungsprüfung muss abgeschlossen sein, bevor das Angebot zum nächsten Schritt im Veröffentlichungspro-

zess übergehen kann. Die automatischen Validierungen bestehen aus den folgenden fünf Schritten:

1. Einrichten des Flows zum Kauf eines Angebots
2. Validierung der Testversionsdaten
3. Bereitstellung der Testversion
4. Überprüfung der Leadverwaltung und Registrierung
5. Angebotsvalidierung

Vor der Veröffentlichung müssen SaaS-Produkte, die an den Microsoft Azure Marketplace übermittelt werden, zertifiziert werden. Die SaaS-Produkte werden strengen Tests unterzogen, die teils automatisch und teils manuell erfolgen [Microsoft 2022a].

7. Verkaufen eines SaaS-Angebots über Microsoft

Der Microsoft Azure Marketplace bietet dem Service-Provider eine Dokumentation über bewährte Marketingmethoden für SaaS-Produkte. Diese Dokumentation definiert empfohlene Vorgehensweisen für die Angebotserstellung, für eine marktfähige Präsentation des SaaS-Produktes sowie für technische Vorgehensweisen, um eine Kundengewinnung zu beschleunigen und um die UX zu verbessern.

8. Testen und Veröffentlichen eines SaaS-Angebots im kommerziellen Marketplace

Wenn das SaaS-Produkt für das Veröffentlichen bereit ist, wird dem Service-Provider eine E-Mail mit der Bitte, die Angebotsvorschau zu überprüfen und zu genehmigen, gesendet. Anschließend gibt bei der Angebotsübersicht, wie in Abbildung 7.5 dargestellt, die Schaltfläche **Live schalten** (Go live). Zusätzlich werden, je nach vorheriger Konfiguration, verschiedene Vorschaulinks für eine Überprüfung angeboten.

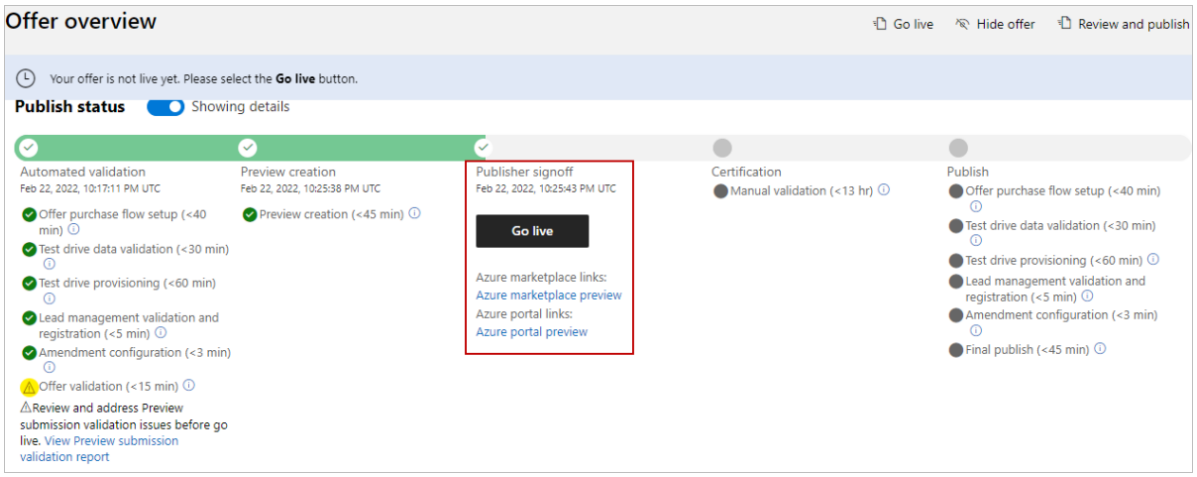


Abbildung 7.5: Microsoft Azure: Angebotsübersicht für ein SaaS-Angebot [Microsoft 2022a]

Bewertung des Onboarding-Prozesses

Der Microsoft Azure Marketplace besitzt einen feingranularen und klar definierten Onboarding-Prozess für SaaS-Produkte. Während dieses Prozesses werden verschiedene Informationen abgefragt und geprüft. Es müssen alle Validierungsprüfung abgeschlossen sein, bevor das SaaS-Produkt veröffentlicht werden darf. Nach dem Start des Veröffentlichungsprozesses wird zudem eine manuelle Validierung und Zertifizierung des SaaS-Produkt durchgeführt.

In der vorgestellten Lösung des Cloud-Marketplaces muss der Service-Provider keine Landingpage erstellen. Zudem übernimmt der Cloud-Marketplace im Vergleich mehr Aufgaben, wie die Nutzerverwaltung oder die Steuerung der Prozesse des Lebenszyklus. Außerdem ist bei dem Cloud-Marketplace ein schnelleres Onboarding von SaaS-Produkt möglich, da keine aufwendigen Validierungsprozesse und keine manuelle Zertifizierung der eigenen SaaS-Produkte vorgesehen ist. Des Weiteren ist es mit der vorgestellte Architektur möglich, schlankere SaaS-Produkte zu implementieren, da der Cloud-Marketplace mehr Aufgaben für den Service-Provider übernimmt und da der die Service Broker weniger Anforderungen erfüllen müssen, als bei dem Microsoft Azure Marketplace.

7.1.3 Evaluation der vorgestellten Architektur

Zusammenfassend sind die Architekturentscheidungen für die Umsetzung eines Cloud-Marketplaces, welche eine widerstandsfähige und hochverfügbare Softwarelösung ergeben soll, zu dem aktuellen Technologiestand richtig gewählt. Dem Cloud-Marketplace ist es so möglich, effizient zu skalieren sowie eine Ausfall- und eine Datensicherheit zu gewährleisten.

Ein weiterer großer Vorteil ist die Flexibilität und Portabilität der Service Broker. Der schlankere Standard der Service Broker ist leichtgewichtig, transparent und lässt sich problemlos für andere Cloud-Marketplaces oder Applikationen überführen. Zudem darf der Service Broker an einem beliebigen Ort eingesetzt werden. So ist auch ein *Scale to Zero* für den Service Broker in einer beliebigen Infrastruktur möglich.

Im Vergleich zum Microsoft Azure Marketplace stellt sich die vorgestellte Lösung für das Onboarding des SaaS-Produktes als effizienter heraus, da es keine aufwendigen automatischen und manuellen Validierungen oder Zertifizierungen notwendig sind. Zudem ist eine schnellere Entwicklung des SaaS-Produktes möglich, da der Cloud-Marketplace, im Vergleich zum Microsoft Azure Marketplace, weniger Anforderungen und Aufgaben dem Service-Provider überträgt.

Im funktionalem Umfang ist der Cloud-Marketplace jedoch noch ausbaufähig. Weitere Funktionen, neben dem Kerngeschäft des Cloud-Marketplaces, können nachgelagert hinzugefügt werden.

7.2 Technische Evaluation

Im Folgenden wird eine Reihe von Untersuchungen und Experimente durchgeführt, die die Erfüllung der Anforderungen prüft. Die Anforderungen beziehen sich dabei auf den Anforderungskataloges aus Kapitel 4. Besonders interessant für eine Untersuchung sind die nicht-funktionalen Anforderungen sowie die betrieblichen Anforderungen durch das LGLN.

Da die Qualitätsszenarien angeben, wie sich der Cloud-Marketplace verhalten muss, um die Qualitätsziele zu erfüllen, sind die Qualitätsszenarien aus Kapitel 5.8 hilfreich für die Untersuchung dieser Anforderungen. Folgende Szenarien werden untersucht:

- Skalierbarkeit von:
  - Microservices,
  - Datenbanken,
  - IAM,
  - NATS-Service.
- Ausfall- und Datensicherheit:
  - Ausfall einzelner oder mehrerer Microservices,
  - Ausfall einzelner oder mehrerer NATS-Services,
  - Ausfall einzelner oder mehrerer IAM-Instanzen,
  - Datensicherheit der Datenbanken sowie der NATS-Events.

7.2.1 Skalierbarkeit

Um eine stetige Antwortbereitschaft zu garantieren ist es notwendig, alle Bausteine des Systems vertikal und horizontal skalieren zu können. Dafür wird die Skalierbarkeit der Microservice, der Datenbanken, des IAM sowie des NATS-Services untersucht.

Skalierbarkeit der Microservices

Mit der Docker Compose-Konfiguration funktioniert das Skalieren der Microservices bereits problemlos, da die Microservices eine lose Kopplung zu den anderen Services besitzen. Zudem sind die einzelnen Microservices statuslos. Mit dem Befehl aus dem Quelltext 7.1 lässt sich beispielsweise der Registry Service auf fünf Instanzen skalieren.

```
1 docker-compose up -d --scale registry-service=5
```

Quelltext 7.1: Skalieren des Registry Service auf fünf Instanzen

Jedoch ist dafür eine kleine Anpassung der Docker Compose-Konfiguration notwendig. In Quelltext 7.2 ist dargestellt, dass der externe Port des Service nicht mehr nur einen möglichen Port enthalten darf. Nun ist es notwendig, eine Spanne verfügbarer Ports zur Verfügung zu stellen. In Quelltext 7.2 ist eine neue mögliche Docker Compose-Konfiguration für die Service Registry dargestellt. Es ist zusätzlich notwendig, in der Konfiguration keinen Containernamen zu vergeben, der *container_name*-Parameter muss demnach entfernt werden.

```
registry-service:
    build: ../registry
    ports:
      - "8181-8190:8081"
    networks:
      - keycloak-net
      - nats-net
      - registry-postgres-net
      - brokers-net
    depends_on:
      - registry-postgres
      - keycloak
      - nats1
    links:
      - keycloak
```

Quelltext 7.2: Neue Docker Compose-Konfiguration für die Registry Service

Zu dieser Skalierung ist es jedoch ratsam, einen Load-Balancer einzusetzen. Ein Load-Balancer dient der Verteilung von Anfragen auf mehrere Service-Instanzen.

Skalierungsmöglichkeiten der Datenbanken

Für ein reaktives System ist das Skalieren der Datenbanken notwendig. Ein Ausfall eines Datenbank-Services oder sogar der ganzen Datenbank würde die Funktionalität und Antwortbereitschaft des Cloud-Marketplace beeinträchtigen oder, im schlimmsten Fall, zu Datenverlust führen. Deshalb ist eine Replikation oder ein Clustering der verschiednenen Datenbanken notwendig.

Die Replikation, das Clustering und das Pooling von Verbindungen einer PostgreSQL-Datenbank ist möglich [PostgreSQL 2022], jedoch für die Verfügbarkeit, die Datensicherheit und die korrekte Anzahl von Replikationen zu sorgen, ist eine komplexe Aufgabe. Deshalb bietet es sich an, auf bestehende, hochverfügbare

und replizierte Datenbanklösungen oder Patterns zurückzugreifen. Dafür gibt es bei allen gängigen Public Cloud-Anbietern Datenbanklösungen, die verschiedene Sicherheiten für Replikation, Datensicherheit und Verfügbarkeiten besitzen.

Skalierungsmöglichkeiten des IAMs

Das Skalieren der Keycloak funktioniert analog zu dem Skalieren der Microservices. Keycloak bietet zusätzlich ein Clustering mehrerer Keycloak-Instanzen [Keycloak 2022b]. Auch eine praxistaugliche Lösung für ein Keycloak-Cluster mit Kubernetes existiert, welches nachgelagert, bei der Kubernetes-Migration, eingebaut werden kann [Keycloak 2022c].

Skalierungsmöglichkeiten des NATS-Services

Eine vertikale Skalierung des NATS-Services in der Docker Compose-Konfiguration ist im Kapitel 6.2 erläutert. Eine praxistaugliche Lösung für einen NATS-Cluster mit Kubernetes existiert ebenfalls, welches nachgelagert, bei der Kubernetes-Migration, eingebaut werden kann [NATS 2022b]. Dazu bietet NATS fertige Helm Charts an.

7.2.2 Ausfall- und Datensicherheit

Damit der Nutzer auch bei Ausfällen verschiedener Maschinen oder Instanzen keine funktionalen Ausfälle oder Datenverlust erleidet, wird der Ausfall einzelner oder mehrerer Microservices, NATS-Services, IAM-Instanzen und Datenbanken simuliert und untersucht.

Ausfallszenarien

Wenn die Microservices, wie in Kapitel 7.2.1 beschrieben, auf mehrere Instanzen skaliert sind, folgt aus einem Ausfall mehrerer Microservices kein Ausfall der Antwortbereitschaft des Cloud-Marketplaces. Jedoch nur so lange, wie mindestens eine Instanz von jedem Service aktiv bleibt. Zusätzlich sollte ein Load-Balancer die Anfragen auf die verschiedenen Service-Instanzen verteilen.

Fallen jedoch alle Instanzen eines Microservices aus, lassen sich die Verantwortlichkeiten des Microservices nicht mehr erfüllen. Deshalb sollten nicht nur genügend Instanzen erstellt werden, abgestürzte Instanzen sollten auch schnellstmöglich wieder bereitgestellt werden. Eine Lösung dieser Aufgabe erfolgt automatisch mit dem Umzug des Cloud-Marketplaces in einem Kubernetes-Cluster.

Dasselbe gilt für die NATS-Services und IAM-Instanzen der prototypischen Umsetzung. Die statuslosen Services benötigen für ihre Funktionsfähigkeit mindestens eine aktive Instanz.

Datensicherheit der NATS-Events und der Datenbanken

Die Persistierung der NATS-Events mit NATS JetStream im Dateiformat ist im Kapitel 6.2.2 erläutert. Die Persistierung der NATS-Events lässt sich, bei der Kubernetes-Migration, auch auf eine Cloud-Datenbank migrieren.

Diese Persistierung bleibt auch nach dem Ausfall der NATS-Instanzen erhalten. Für die Persistierung der Events einer Message-Queue ist immer ein NATS-Service zuständig. Fällt dieser Service aus, erfahren die anderen NATS-Services im Cluster dies durch die Heartbeats zwischen den Instanzen. Ein Heartbeat beschreibt die regelmäßige Kommunikation für den Status der Instanzen. Fällt der zuständige NATS-Service eines Streams aus, übernimmt direkt ein anderer NATS-Service als Master für diesen Stream. Dies sorgt für einen effizienten Failover der NATS-Services und Message-Queues.

Die Persistierung ist in dieser prototypischen Umsetzung jedoch nur mit Docker-Volumes gelöst. Das bedeutet, dass die Daten lediglich einmal auf dem Datenträger einer Maschine geschrieben sind. Fällt diese Maschine aus, würden die Daten der Datenbank verloren gehen. Deshalb ist für das Produktivsystem eine replizierte und hochverfügbare Datenbank unerlässlich.

7.2.3 Evaluation des prototypischen Umsetzung

Die prototypische Umsetzung für den Cloud-Marketplace zeigt, dass die gewählte Lösungsstrategie für den Entwurf des Cloud-Marketplace sich durch eine besondere Eignung zur Skalierbarkeit ausweist. Diese Skalierbarkeit bietet die Grundsteine für ein reaktives System, welches eine fortwährende Antwortbereitschaft

anstrebt. Jedoch ist für das Produktivsystem eine Migration der Datenbanken in eine replizierte und hochverfügbare Datenbanklösung erforderlich.

7.3 Evaluation der Zielsetzung

Anhand der prototypischen Umsetzung, der Experimente, der Untersuchungen und den gewonnenen Erkenntnissen wird im Folgenden die Erreichung der gestellten Anforderungen geprüft. Dazu wird entlang des Anforderungskataloges aus Kapitel 4 kurz erläutert, inwiefern die jeweilige Anforderung umgesetzt werden konnte.

7.3.1 Funktionale Anforderungen

Aus dem beiliegenden Speichermedium lässt sich die Software für den Cloud-Marketplace, eine Installationsanleitung und eine Postman-Datei für das Testen und Überprüfen der Schnittstellen des Cloud-Marketplaces finden. Im Anhang auf Seite 148, nach der Schnittstellen-Dokumentation, befindet sich ein Ausschnitt der REST-Aufrufe, die die funktionalen Anforderungen überprüfen.

1 a) Registrierung und Anmeldung von Service-Consumer und Service-Provider

Diese Anforderung ist vollständig erfüllt. Eine Anmeldung und Registrierung ist über die Keycloak-API möglich.

1 b) Registrierung von Service Broker und SaaS-Produkten

Diese Anforderung ist vollständig erfüllt (siehe die REST-API-Tests mit Postman im Anhang auf Seite 148).

1 c) Konfigurieren der eigenen Service Broker und SaaS-Produkte

Diese Anforderung ist vollständig erfüllt (siehe die REST-API-Tests mit Postman im Anhang auf Seite 148).

1 d) Aktivieren und Löschen von Service Brokern und SaaS-Produkte

Diese Anforderung ist vollständig erfüllt (siehe die REST-API-Tests mit Postman im Anhang auf Seite 148).

1 e) Verfügbare SaaS-Produkte anzeigen

Diese Anforderung ist vollständig erfüllt (siehe die REST-API-Tests mit Postman im Anhang auf Seite 148).

1 f) Verfügbare SaaS-Produkte provisionieren

Diese Anforderung ist vollständig erfüllt (siehe die REST-API-Tests mit Postman im Anhang auf Seite 148).

1 g) Status der Service-Provisionierung einsehen

Diese Anforderung ist vollständig erfüllt (siehe die REST-API-Tests mit Postman im Anhang auf Seite 148).

1 h) Eigene Service-Instanzen anzeigen und deprovisionieren

Diese Anforderung ist vollständig erfüllt (siehe die REST-API-Tests mit Postman im Anhang auf Seite 148).

1 i) Nutzungsdaten der Instanzen entgegennehmen

Diese Anforderung ist vollständig erfüllt (siehe die REST-API-Tests mit Postman im Anhang auf Seite 148).

1 j) Nutzungsdaten der eigenen Instanzen einsehen

Diese Anforderung ist vollständig erfüllt (siehe die REST-API-Tests mit Postman im Anhang auf Seite 148).

7.3.2 Nicht-funktionale Anforderungen

2 a) Skalierbarkeit

Diese Anforderung ist vollständig erfüllt. Das IAM, die MoM sowie die Microservices sind vertikal skalierbar. Auch die Datenbanken sind skalierbar oder lassen sich durch Cloud-Datenbanken austauschen.

2 b) Elastizität

Diese Anforderung ist erfüllt, ist jedoch ausbaufähig. Ein Scale-In und Scale-Out ist mit allen Services des Cloud-Marketplaces möglich, allerdings muss das Skalieren aktiv unternommen werden. In späterer Instanz der Entwicklung des Cloud-Marketplace sollte ein System für ein automatisches Skalieren der Service-Instanzen entworfen werden, um nicht eine unnötig hohe Anzahl an Instanzen bereit zu halten.

2 c) Zeitverhalten

Diese Anforderung ist vollständig erfüllt. Die Skalierbarkeit und die lose Kopplung der Microservices erlaubt es, alle Dienste auf eine Anzahl zu skalieren, dass eine große Menge an Anfragen bewältigt werden kann. Zusätzlich sorgt die asynchrone Nachrichtenverarbeitung für eine direkte Antwort zurück zum Frontend, da anschließend im Hintergrund die Prozesse, wie zum Beispiel das Provisionieren, asynchron erledigt werden.

2 d) Resilienz

Diese Anforderung ist vollständig erfüllt. Durch ausreichendes Skalieren und dank des Failovers des NATS-Clusters, ist der Cloud-Marketplace auch bei einem Ausfall einzelner Service-Instanzen reaktionsfähig.

2 e) Sicherheit

Diese Anforderung ist vollständig erfüllt. Die Software verfügt über eine wohldefinierte API. Zudem lassen sich Sicherheitsmaßnahmen zum Login und zur Registrierung in der Keycloak einstellen.

2 f) Wartbarkeit

Diese Anforderung ist vollständig erfüllt. Die Microservice-Architektur ist wartbar, da die Software in kleine modulare Softwarekomponenten gegliedert ist, welche sich getrennt warten und erweitern lassen.

7.3.3 Betriebliche Anforderungen durch das LGLN

3 a) Open Source Technologien

Diese Anforderung ist vollständig erfüllt. Mit NATS, Keycloak und PostgreSQL wurden Open Source-Technologien verwendet.

3 b) Lose Kopplung der Services

Diese Anforderung ist vollständig erfüllt. Die Lösungsstrategie für den Entwurf der Architektur setzt zentral auf Microservices mit loser Kopplung (siehe Kapitel 5.2).

3 c) Technologie-Stack des LGLN

Diese Anforderung ist vollständig erfüllt. Mit Keycloak und PostgreSQL wurden Technologien aus dem Technologie-Stack des LGLN verwendet. Weitere Technologie-Entscheidungen wurden ebenfalls mit dem LGLN abgestimmt.

3 d) Spezieller Datenschutz

Diese Anforderung ist vollständig erfüllt. Da alle Daten in einer selbst bereitgestellten Datenbank persistiert werden können, lassen sich die Datenschutzrichtlinien einhalten.

3 e) Cloudfähigkeit

Diese Anforderung ist vollständig erfüllt. Die einzelnen Docker-Container der Microservices lassen sich in eine Cloud migrieren.

3 f) Überwachung und Persistenz der Events

Diese Anforderung ist vollständig erfüllt (siehe Kapitel 7.2.2 und 6.2.2).

8 Fazit

Mit der vorliegenden Arbeit wurde ein Cloud-Marketplace anhand einer Cloud Native-Event-Driven Reactive Services-Architektur konzeptioniert und anschließend mittels einer prototypischen Umsetzung vorgestellt und evaluiert.

Anhand der Charakterisierung der Verantwortlichkeiten und Anforderung von Service-Provider und Service-Consumer können sich Anforderungen und Funktionsabläufe herleiten. Die Untersuchung des Lebenszyklus der Service-Instanzen ergibt eine nützliche Wissensbasis für die Erhebung der Funktionalitäten und Anforderungen. Basierend auf den erhobenen Anforderungen und Qualitätszielen lässt sich eine geeignete Microservice-Architektur erstellen. Diese erweist sich, durch ihre lose Kopplung und starker Kohäsion, einer guten Skalierbarkeit, welches zu einer guten Resilienz führt. Die Architekturentscheidungen des vorgestellten Cloud-Marketplaces sind zu dem aktuellen Technologiestand richtig gewählt, da es so möglich ist, effizient zu skalieren sowie eine Ausfall- und eine Datensicherheit zu gewährleisten.

Die Technologieentscheidungen sind im Hinblick einer guten Skalierbarkeit und cloudfähigen Bereitstellung gewählt. Zudem gewähren die eingesetzten modernen Open Source-Technologien eine große Flexibilität, Kosteneinsparungen und neueste technologische Innovationen.

Die Evaluation der prototypischen Umsetzung zeigt, dass die gewählte Lösungsstrategie für den Entwurf des Cloud-Marketplace sich durch eine besondere Eignung zur Skalierbarkeit ausweist. Diese Skalierbarkeit bietet die Grundsteine für ein reaktives System, welches eine fortwährende Antwortbereitschaft anstrebt. Im Vergleich zum Microsoft Azure Marketplace stellt sich die vorgestellte Lösung für das Onboarding des SaaS-Produktes als effizienter heraus, da keine aufwendigen Validierungsprozesse und keine manuelle Zertifizierung der eigenen SaaS-Produkte vorgesehen sind. Des Weiteren ist durch die Entwicklung der SaaS-Produkte schneller, da im Vergleich zum Microsoft Azure Marketplace, weniger Anforderungen und Aufgaben dem Service-Provider übertragen werden. Jedoch

ist die vorgestellte Lösung des Cloud-Marketplaces, im Vergleich zu bestehenden Umsetzung, im funktionalen Umfang ausbaufähig.

8.1 Ausblick

Da es sich bei dem in dieser Arbeit vorgestellten Architektur um eine Konzeptionierung und prototypische Umsetzung handelt, gibt es eine Vielzahl von Erweiterungs- und Verbesserungsmöglichkeiten. Zuerst müssen die technischen und funktionalen Anforderungen, welche im Rahmen dieser Arbeit nicht behandelt wurden, implementiert werden. Dazu gehört die Konzeptionierung und Implementierung einer Webseite sowie die Erhebung und Abrechnung der Kosten durch den Payment Service.

Weitere interessante Querschnittaufgaben des Cloud-Marketplaces ist ein Mitteilungs-Service, ein Tool für eine grafische Nutzungsüberwachung, automatische Validierungen und Tests und ein E-Mail-Service. Darüber hinaus gibt es, wie bei den führenden Public Cloud-Anbietern, zahlreiche Möglichkeiten für weitere Funktionen oder Produkte, um die UX der Nutzer zu steigern. Um den Cloud-Marketplace für externe Service-Provider attraktiv zu gestalten, bedarf es einer ausführlichen sowie gut verständlichen Dokumentation und Anleitungen für die Erstellung von SaaS-Produkten und wie diese in den Cloud-Marketplace integriert werden können. Dazu sind detailreiche Schrittanleitungen, Musterbeispiele oder Videoanleitungen gut geeignet. Zudem könnte dem Service-Provider ebenfalls mehr Funktionalitäten, wie beispielsweise eine Entwicklungsumgebung, verschiedene Entwicklungshilfen für die Erstellung der SaaS-Produkte oder eine Zertifizierung, geboten werden.

Als technische Erweiterungen bietet sich zuerst die Migration in eine Cloud und die Verwendung von Kubernetes an. Dies vereinfacht die Skalierung und bietet eine bessere Verfügbarkeit durch das automatische Bereitstellen einer gewünschten Anzahl an Instanzen. Anschließend sind vorausschauende als auch reaktive Skalierungsalgorithmen eine wichtige technische Erweiterung, um auf akute Änderungen der Input-Rate reagieren zu können. Des Weiteren ist das automatische Überprüfen und Ändern der Instanz-Status eine interessante Erweiterungen. So könnte der Cloud-Marketplace beispielsweise SaaS-Instanzen automatisch deaktivieren, wenn Zahlungen nicht getätigt wurden. Zuletzt ist der Umzug auf eine replizierte Cloud-Datenbank, welche verschiedene Sicherheiten für Replikation, Datensicherheit und Verfügbarkeiten besitzt, eine essenzielle Verbesserung.

Literatur

Amazon-AWS (2022a). *AWS Marketplace Documentation*. URL: https://docs.aws.amazon.com/marketplace/index.html (besucht am 22. 03. 2022).

— (2022b). *Using AWS Marketplace as a buyer*. URL: https://docs.aws.amazon.com/marketplace/latest/buyerguide/what-is-marketplace.html (besucht am 22. 03. 2022).

— (2022c). *Using AWS Marketplace as a seller*. URL: https://docs.aws.amazon.com/marketplace/latest/userguide/what-is-marketplace.html (besucht am 22. 03. 2022).

— (2022d). *Was sind Microservices?* URL: https://aws.amazon.com/de/microservices/ (besucht am 05. 07. 2022).

baeldung (2022). *Documenting a Spring REST API Using OpenAPI 3.0*. URL: https://www.baeldung.com/spring-rest-openapi-documentation (besucht am 14. 05. 2022).

Bonér, Jonas u. a. (2014). *Das Reaktive Manifest*. URL: https://www.reactivemanifesto.org (besucht am 01. 07. 2022).

Bruns, Prof. Dr. Ralf und Prof. Dr. Jürgen Dunkel (2010). *Event-Driven Architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*.

Bundesamt-für-Sicherheit-und-Informationstechnik (2022). *Cloud Computing Grundlagen BSI*. URL: https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Empfehlungen-nach-Angriffszielen/Cloud-Computing/Grundlagen/grundlagen_node.html (besucht am 15. 06. 2022).

Cloud-Native-Computing-Foundation (2022a). *Building sustainable ecosystems for cloud native software*. URL: https://www.cncf.io/ (besucht am 13. 04. 2022).

— (2022b). *Cloud Native Computing Foundation NATS*. URL: https://www.cncf.io/projects/nats/ (besucht am 02. 05. 2022).

Cloudogu (2022). *Was bedeutet „Skalierung“?* URL: https://cloudogu.com/de/glossar/skalierung/ (besucht am 09. 07. 2022).

Confluent (2022). *Event-Driven Microservices Architecture*. URL: https://www.confluent.io/resources/event-driven-microservices/ (besucht am 05. 07. 2022).

Docker-Inc. (2022). *Docker Homepage*. URL: https://www.docker.com/ (besucht am 12. 07. 2022).

Fraunhofer (2022). *Everything-as-a-Service (XaaS)*. URL: https://www.ipa.fraunhofer.de/de/Kompetenzen/kompetenzzentrum-digitale-werkzeuge-in-der-produktion/cloud-plattformen/everything-as-a-service--xaas.html (besucht am 27. 06. 2022).

Gohad, Atul, Karthikeyan Ponnalagu und Nanjangud C. Narendra (2012). *Model Driven Provisioning in Multi-tenant Clouds*.

group24 (2022). *Grafik für den Vergleich von On-site IT mit den verschiedenen Servicemodellen*. URL: https://www.group24.de/blog/wp-content/uploads/2022/03/on-site3.jpg (besucht am 27. 06. 2022).

Gunaratne, Sam u. a. (2022). *Open Service Broker API*. URL: https://www.openservicebrokerapi.org/ (besucht am 05. 03. 2022).

Hruschka, Dr. Peter u. a. (2022). *arc42 Methodik: Systematisch und prozess-agnostisch zu angemessenen Lösungen*. URL: https://www.arc42.de (besucht am 13. 07. 2022).

Keycloak (2022a). *Keycloak Admin REST API*. URL: https://www.keycloak.org/docs-api/18.0/rest-api/ (besucht am 15. 06. 2022).

— (2022b). *Keycloak Cluster Setup*. URL: https://www.keycloak.org/2019/05/keycloak-cluster-setup.html (besucht am 27. 07. 2022).

— (2022c). *Keycloak Kubernetes*. URL: https://www.keycloak.org/getting-started/getting-started-kube (besucht am 27. 07. 2022).

LGLN (10. März 2022). *geoPlattform Workshop*.

Microsoft (2022a). *Planen eines SaaS-Angebots für den kommerziellen Marketplace*. URL: https://docs.microsoft.com/de-de/azure/marketplace/plan-saas-offer (besucht am 30. 07. 2022).

— (2022b). *Willkommen beim kommerziellen Marketplace*. URL: https://docs.microsoft.com/de-de/azure/marketplace/ (besucht am 14. 07. 2022).

Möhring, Michael, Barbara Keller und Rainer Schmidt (2017). *CRM in der Public Cloud*. URL: https://link.springer.com/chapter/10.1007/978-3-658-19724-7_3 (besucht am 30. 06. 2022).

National-Institute-of-Standards-and-Technology (2022). *The NIST Definition of Cloud*. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (besucht am 25. 06. 2022).

NATS (2022a). *NATS Documentation*. URL: https://docs.nats.io/ (besucht am 02. 05. 2022).

— (2022b). *NATS Kubernetes*. URL: https://docs.nats.io/running-a-nats-service/nats-kubernetes (besucht am 16. 05. 2022).

nats-io (2022). *NATS - Java Client*. URL: https://github.com/nats-io/nats.java (besucht am 15. 05. 2022).

Open-Service-Broker-API (2022). *Open Service Broker API Specifikation*. URL: https://petstore.swagger.io/?url=https://raw.githubusercontent.com/openservicebrokerapi/servicebroker/master/openapi.yaml (besucht am 17. 07. 2022).

Plusserver (2022). *Was sind IT-Container?* URL: https://www.plusserver.com/blog/was-sind-it-container (besucht am 12. 07. 2022).

PostgreSQL (2022). *Replication, Clustering, and Connection Pooling*. URL: https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling (besucht am 25. 07. 2022).

Red-Hat (2022). *What are microservices?* URL: https://www.redhat.com/en/topics/microservices/what-are-microservices (besucht am 10. 07. 2022).

Researchgate (2022). *The NIST cloud computing definitions*. URL: https://www.researchgate.net/profile/Seyyed-Mohsen-Hashemi/publication/327882704/figure/fig1/AS:750274822012928@1555890865543/The-NIST-cloud-computing-definitions.ppm (besucht am 02. 07. 2022).

Salesforce (2022). *Was ist SaaS?* URL: https://www.salesforce.com/de/learning-centre/tech/saas/ (besucht am 06. 06. 2022).

Schäfer, Michael (2022). *Why Reactive: Reaktive Architekturen und ihre Geschichte*. URL: https://www.heise.de/hintergrund/Why-Reactive-Reaktive-Architekturen-und-ihre-Geschichte-4999096.html (besucht am 02. 07. 2022).

Schiborr, Marco u. a. (2021). *Event-Driven Reactive Services*.

Stark, Jens (2022). *Microsoft Azure im Langzeit-Test*. URL: https://www.computerworld.ch/business/betriebssysteme/microsoft-azure-im-langzeit-test-1347321.html (besucht am 14. 07. 2022).

Valdes, Alfonso (2012). *Multi tenant Architecture for a SaaS Application on AWS*. URL: https://www.clickittech.com/saas/multi-tenant-architecture/ (besucht am 22. 03. 2022).

Vinugayathri (2022). *5 Best Technologies To Build Microservices Architecture*. URL: https://www.clariontech.com/blog/5-best-technologies-to-build-microservices-architecture (besucht am 24. 07. 2022).

VMware-Inc. (2022a). *Spring Cloud Open Service Broker*. URL: https://docs.spring.io/spring-cloud-open-service-broker/docs/current/reference/index.html#customizing-the-service-broker-path (besucht am 14. 05. 2022).

— (2022b). *Spring Initializr*. URL: https://start.spring.io/ (besucht am 24. 07. 2022).

9 Anhang

Aus dem beiliegenden Speichermedium lässt sich der Anhang entnehmen. Dieser enthält den Quelltext, eine Installationsanleitung, die Postman API-Tests, die API-Dokumentation und eine Serviceübersicht für die Erstellung des Prototypen.

9.1 Installationsanleitung

1. Docker und Docker Compose installieren (URL: `https://docs.docker.com/compose/install/`).
2. Je nach installierter Docker-Version, eventuell Docker Desktop starten
3. In den Ordner **docker** navigieren
4. Ausführen des Befehls ***docker-compose build*** baut den Prototypen
5. Anschließend die Ausführung des Befehls ***docker-compose up -d*** startet den Prototypen

Anschließend kann die REST-API des Marketplaces getestet werden. Jedoch sind dafür Einstellungen im Keycloak notwendig. In 9.1.2 sind die notwendigen Einstellungen für Realm, Rollen und Benutzer in der Keycloak erläutert.

Für ein Beenden und Löschen der Marketplace-Services muss der Befehl ***docker-compose down*** in der gleichen Konsole ausgeführt werden aus.

9.1.1 Serviceübersicht

Serviceübersicht

Service	Location
Keycloak	keycloak:8090
Registry Service	registry-service:8081
Service Provisioning Service	provisioning-service:8082
Service Instanzen Service	instances-service:8083
Service Usage Service	usage-service:8084
Keycloak Database (PostgreSQL)	keycloak_db_postgres:5433
Instanzen Database (PostgreSQL)	instances-postgres:5434
Registry Database (PostgreSQL)	registry-postgres:5432
Usage Database (PostgreSQL)	usage-postgres:5435
NATS Server 1	nats1:4442
NATS Server 2	nats2:4423
NATS Server 3	nats3:4424
Service Broker: alpha-mail-service	broker1:8085
Service Broker: bravo-mail-service	broker2:8086
Service Broker: zulu-mail-service	broker3:8087

Abbildung 9.1: Serviceübersicht der prototypischen Umsetzung

9.1.2 Keycloak Einstellungen

1. Öffnen der Keycloak Admin Oberfläche (bspw: URL: `http://localhost:8080/auth/`)
2. Den Admin-Benutzernamen und Password lassen sich aus den Environment-Variablen aus der Docker Compose-Konfiguration entnehmen.
3. Zuerst muss ein Realm mit dem Namen ***Platform*** erstellt werden (oben links auf **add realm** klicken)
4. Anschließend muss unter dem Configurationsmenu: ***Client*** ein neuer Client für die Microservices, mit dem Namen ***springboot-keycloak***, erstellt werden.
5. Für die Benutzer müssen die Rollen ***consumer*** und ***producer*** erstellt werden.
6. Nun lassen sich die ersten Nutzer erstellen. Nur Nutzer mit der **producer** Rolle dürfen SaaS-Produkte im Marketplace registrieren.
7. Für REST-Aufrufe müssen Benutzer sich bei der Keycloak anmelden und den **Bearer-Token** in den HTTP-Header hinterlegen.
8. Dafür lässt dich die API-Token-URL der Keycloak nutzen (bspw: URL: `http://keycloak:8080/auth/realms/platform/protocol/openid-connect/token`)

Diese Realm-Einstellungen lassen sich anschließend auch exportieren und importieren, sodass bei einer neu angelegen Datenbank diese Einstellungen nicht ein weiteres mal durchgeführt werden müssen. Die Realm-Exportdatei für den Prototyp befindet sich im Anhang des digitalen Speichermediums.

9.2 Dockerfiles

9.2.1 Dockerfile: Service Registry

```
FROM maven:3.6.1-jdk-11-slim AS build
RUN mkdir -p /workspace
WORKDIR /workspace
COPY pom.xml /workspace
COPY src /workspace/src
COPY ./src/main/resources/application.yml .
RUN mvn -f pom.xml clean package -e

FROM openjdk:11
COPY --from=build /workspace/target/*.jar registry.jar
EXPOSE 8081
ENTRYPOINT ["java","-jar","registry.jar"]
```

Quelltext 9.1: Dockerfile: Service Registry

9.2.2 Dockerfile: Service Provisioning Service

```
FROM maven:3.6.1-jdk-11-slim AS build
RUN mkdir -p /workspace
WORKDIR /workspace
COPY pom.xml /workspace
COPY src /workspace/src
COPY ./src/main/resources/application.yml .
RUN mvn -f pom.xml clean package -e

FROM openjdk:11
COPY --from=build /workspace/target/*.jar provisioning.jar
EXPOSE 8082
ENTRYPOINT ["java","-jar","provisioning.jar"]
```

Quelltext 9.2: Dockerfile: Service Provisioning Service

9.2.3 Dockerfile: Service Usage Service

```
FROM maven:3.6.1-jdk-11-slim AS build
RUN mkdir -p /workspace
WORKDIR /workspace
COPY pom.xml /workspace
COPY src /workspace/src
COPY ./src/main/resources/application.yml .
RUN mvn -f pom.xml clean package -e

FROM openjdk:11
COPY --from=build /workspace/target/*.jar usage.jar
EXPOSE 8084
ENTRYPOINT ["java","-jar","usage.jar"]
```

Quelltext 9.3: Dockerfile: Service Usage Service

9.2.4 Dockerfile: Service Instanzen Service

```
FROM maven:3.6.1-jdk-11-slim AS build
RUN mkdir -p /workspace
WORKDIR /workspace
COPY pom.xml /workspace
COPY src /workspace/src
COPY ./src/main/resources/application.yml .
RUN mvn -f pom.xml clean package -e

FROM openjdk:11
COPY --from=build /workspace/target/*.jar instances.jar
EXPOSE 8083
ENTRYPOINT ["java","-jar","instances.jar"]
```

Quelltext 9.4: Dockerfile: Service Instanzen Service

9.3 Vollständige Docker Compose-Datei

```
version: '3'

services:
  registry-postgres:
    image: 'postgres:alpine'
    restart: unless-stopped
    container_name: registry-postgres
    volumes:
      - postgres-db:/var/lib/postgres
    environment:
      POSTGRES_PASSWORD: "registry-postgres-passwort"
      POSTGRES_USER: "postgres"
      POSTGRES_DB: "serviceBrokerDB"
    ports:
     - "5432:5432"
    expose:
      - "5432"
    networks:
      - registry-postgres-net

  instances-postgres:
    container_name: instances-postgres
    image: 'postgres:alpine'
    restart: unless-stopped
    volumes:
      - postgres-instances-db:/var/lib/postgres
    environment:
      POSTGRES_PASSWORD: "instances-postgres-passwort"
      POSTGRES_USER: "postgres"
      POSTGRES_DB: "instancesDB"
    ports:
      - "5434:5432"
    expose:
      - "5434"
    networks:
      - instances-postgres-net

  usage-postgres:
    container_name: usage-postgres
    image: 'postgres:alpine'
    restart: unless-stopped
    volumes:
      - usage-instances-db:/var/lib/postgres
    environment:
      POSTGRES_PASSWORD: "usage-postgres-passwort"
```

```
46      POSTGRES_USER: "postgres"
47      POSTGRES_DB: "usageDB"
48    ports:
49      - "5435:5432"
50    expose:
51      - "5435"
52    networks:
53      - usage-postgres-net
54
55  keycloak_db_postgres:
56    container_name: keycloak_db_postgres
57    image: 'postgres:alpine'
58    volumes:
59      - postgres_data:/var/lib/postgresql/data
60    restart: 'always'
61    ports:
62     - "5433:5432"
63    environment:
64      POSTGRES_USER: keycloak
65      POSTGRES_PASSWORD: keycloak-password-for-postgres
66      POSTGRES_DB: keycloak
67      POSTGRES_HOST: postgres
68    networks:
69      - keycloak-net
70
71
72  keycloak:
73    image: quay.io/keycloak/keycloak:legacy
74    container_name: keycloak
75    environment:
76      HTTP_ENABLED: "true"
77      DB_VENDOR: postgres
78      DB_ADDR: keycloak_db_postgres
79      DB_DATABASE: keycloak
80      DB_USER: keycloak
81      DB_PASSWORD: keycloak-password-for-postgres
82      KEYCLOAK_USER: admin
83      KEYCLOAK_PASSWORD: lgln
84      PROXY_ADDRESS_FORWARDING: "true"
85    depends_on:
86      - keycloak_db_postgres
87    ports:
88      - "8080:8080"
89      - "8443:8443"
90    networks:
91      - keycloak-net
92
```

```
93  nats1:
94    container_name: nats1
95    image: synadia/jsm:nightly
96    dns_search: example.net
97    entrypoint: /nats-server
98    command: --name NATS1 --cluster_name JSC --js --sd /data --
     cluster nats://0.0.0.0:4245 --routes nats://nats1:4245,nats://
     nats2:4245,nats://nats3:4245 -p 4222
99    networks:
100     - nats-net
101   ports:
102     - 4222:4222
103   volumes:
104     - ./jetstream-cluster/nats1:/data
105
106 nats2:
107   container_name: nats2
108   image: synadia/jsm:nightly
109   dns_search: example.net
110   entrypoint: /nats-server
111   command: --name NATS2 --cluster_name JSC --js --sd /data --
     cluster nats://0.0.0.0:4245 --routes nats://nats1:4245,nats://
     nats2:4245,nats://nats3:4245 -p 4222
112   networks:
113     - nats-net
114   ports:
115     - 4223:4222
116   volumes:
117     - ./jetstream-cluster/nats2:/data
118
119 nats3:
120   container_name: nats3
121   image: synadia/jsm:nightly
122   dns_search: example.net
123   entrypoint: /nats-server
124   command: --name NATS3 --cluster_name JSC --js --sd /data --
     cluster nats://0.0.0.0:4245 --routes nats://nats1:4245,nats://
     nats2:4245,nats://nats3:4245 -p 4222
125   networks:
126     - nats-net
127   ports:
128     - 4224:4222
129   volumes:
130     - ./jetstream-cluster/nats3:/data
131
132 registry-service:
133   build: ../registry
```

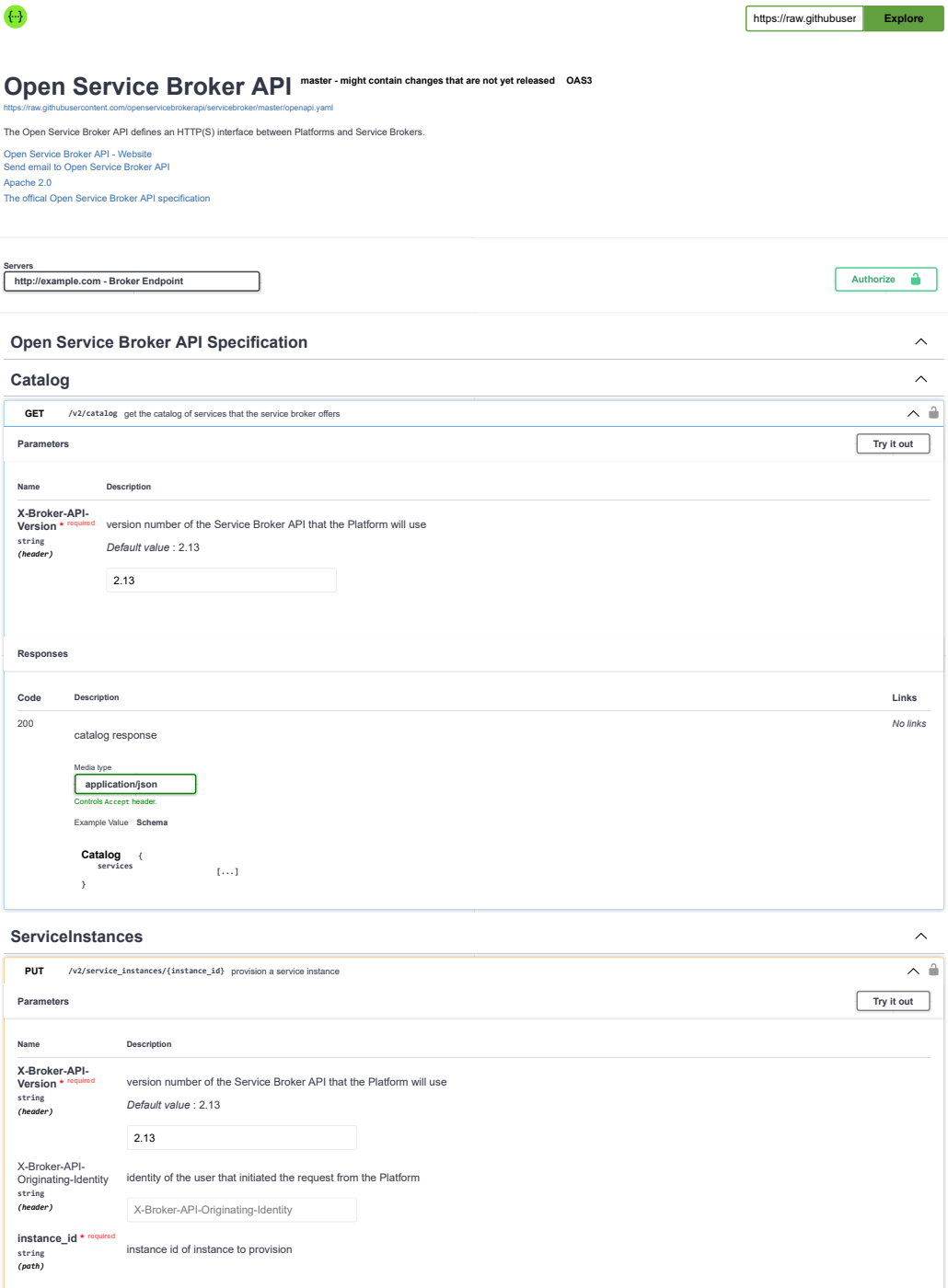
```
134     #container_name: registry
135     ports:
136       - "8181-8190:8081"
137     networks:
138       - keycloak-net
139       - nats-net
140       - registry-postgres-net
141       - brokers-net
142     depends_on:
143       - registry-postgres
144       - keycloak
145       - nats1
146     links:
147       - keycloak
148
149   instances-service:
150     build: ../instances
151     container_name: instances
152     ports:
153       - "8083:8083"
154     networks:
155       - keycloak-net
156       - nats-net
157       - instances-postgres-net
158       - brokers-net
159     depends_on:
160       - instances-postgres
161       - keycloak
162       - nats1
163       - provisioning-service
164     links:
165       - keycloak
166
167   provisioning-service:
168     build: ../provisioning
169     container_name: provisioning
170     ports:
171       - "8082:8082"
172     networks:
173       - keycloak-net
174       - nats-net
175       - brokers-net
176     depends_on:
177       - keycloak
178       - nats1
179       - registry-service
180     links:
```

```
181       - keycloak
182
183   usage-service:
184     build: ../usage
185     container_name: usage
186     ports:
187       - "8084:8084"
188     networks:
189       - usage-postgres-net
190       - nats-net
191       - brokers-net
192       - keycloak-net
193     depends_on:
194       - keycloak
195       - nats1
196       - registry-service
197       - usage-postgres
198     links:
199       - usage-postgres
200       - keycloak
201
202   broker1:
203     build: ../spring-cloud-open-service-broker_1
204     container_name: broker1
205     ports:
206       - "8085:8085"
207     networks:
208       - brokers-net
209
210   broker2:
211     build: ../spring-cloud-open-service-broker_2
212     container_name: broker2
213     ports:
214       - "8086:8086"
215     networks:
216       - brokers-net
217
218   broker3:
219     build: ../spring-cloud-open-service-broker_3
220     container_name: broker3
221     ports:
222       - "8087:8087"
223     networks:
224       - brokers-net
225
226 networks:
227   keycloak-net:
```

```
228   registry-postgres-net:
229     driver: bridge
230   nats-net:
231     driver: bridge
232   brokers-net:
233   instances-postgres-net:
234   usage-postgres-net:
235
236
237 volumes:
238   postgres_data:
239     driver: local
240   postgres-db:
241   postgres-instances-db:
242   usage-instances-db:
```

Quelltext 9.5: Vollständige Docker Compose-Datei

9.4 Open Service Broker API-Spezifikation



Name	Description
	instance_id
accepts_incomplete boolean *(query)*	asynchronous operations supported --

Request body required — application/json

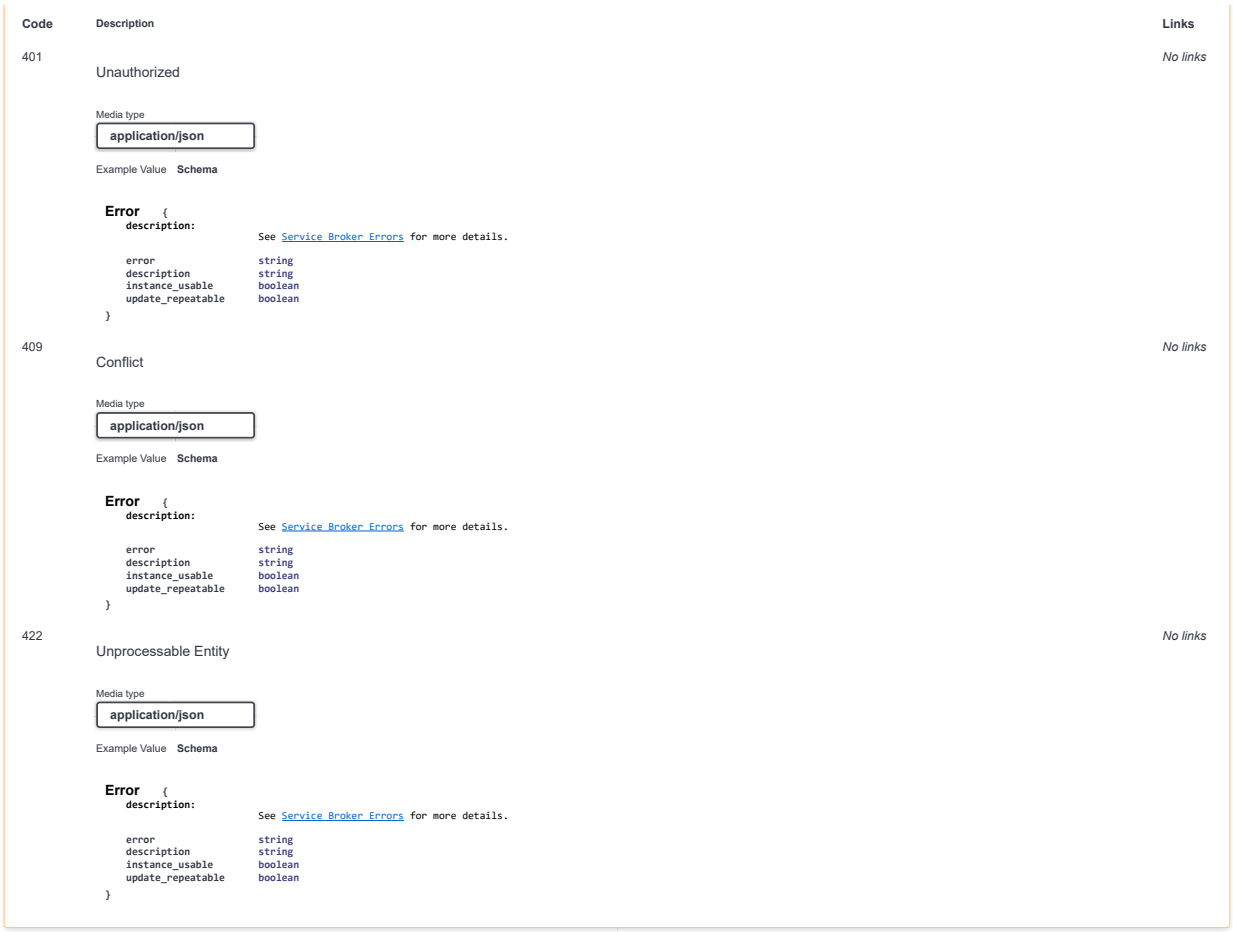
parameters for the requested service instance provision

Example Value **Schema**

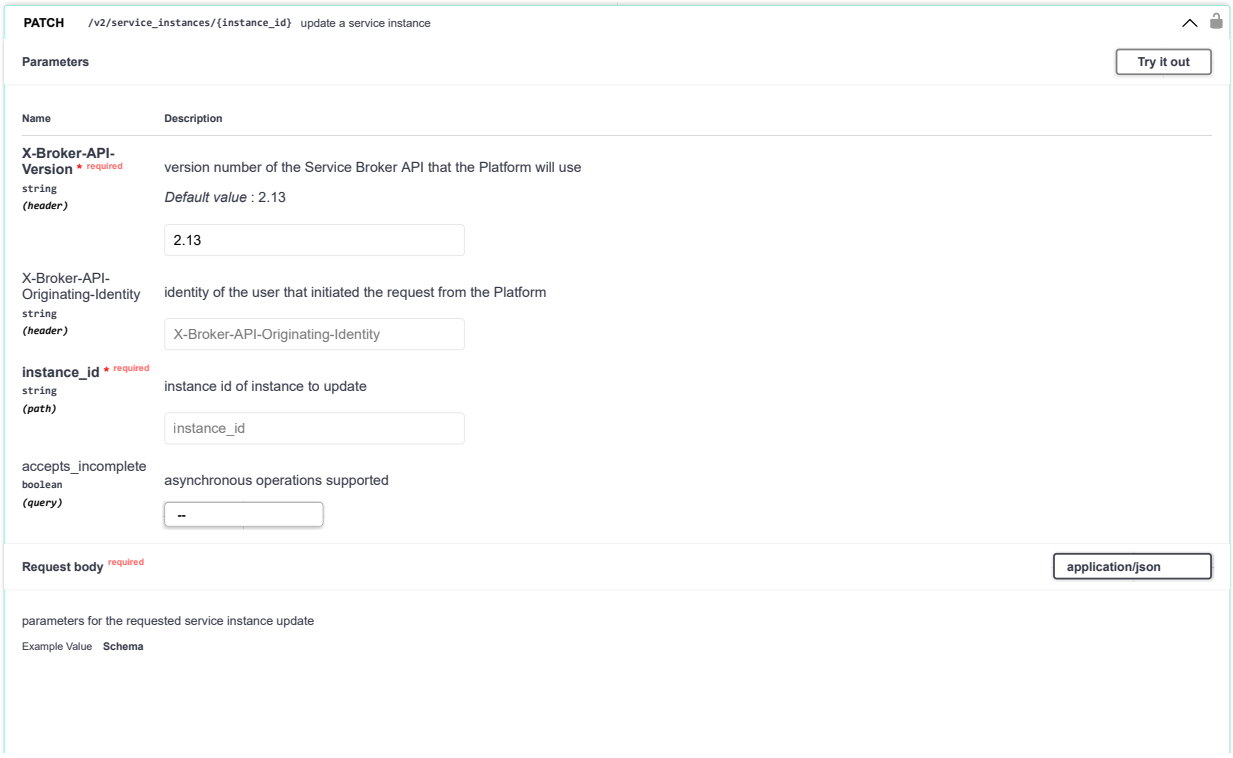
```
ServiceInstanceProvisionRequestBody {
    service_id*             string
    plan_id*                string
    context                 Context {...}
    ~~organization_guid*~~  string
                            deprecated: true
    ~~space_guid*~~         string
                            deprecated: true
    parameters              Object {...}
}
```

Responses

Code	Description	Links
200	OK Media type: application/json Controls Accept header. Example Value **Schema** **ServiceInstanceProvisionResponse** { dashboard_url string metadata **ServiceInstanceMetadata** {...} }	*No links*
201	Created Media type: application/json Example Value **Schema** **ServiceInstanceProvisionResponse** { dashboard_url string metadata **ServiceInstanceMetadata** {...} }	*No links*
202	Accepted Media type: application/json Example Value **Schema** **ServiceInstanceAsyncOperation** { dashboard_url string operation string metadata **ServiceInstanceMetadata** {...} }	*No links*
400	Bad Request Media type: application/json Example Value **Schema** **Error** { description: See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*



Code	Description	Links
401	Unauthorized Media type: application/json Example Value Schema Error { description: See Service Broker Errors for more details. error: string, description: string, instance_usable: boolean, update_repeatable: boolean }	No links
409	Conflict Media type: application/json Example Value Schema Error { description: See Service Broker Errors for more details. error: string, description: string, instance_usable: boolean, update_repeatable: boolean }	No links
422	Unprocessable Entity Media type: application/json Example Value Schema Error { description: See Service Broker Errors for more details. error: string, description: string, instance_usable: boolean, update_repeatable: boolean }	No links



PATCH /v2/service_instances/{instance_id} update a service instance

Parameters — Try it out

Name	Description
X-Broker-API-Version * required string (header)	version number of the Service Broker API that the Platform will use *Default value* : 2.13 2.13
X-Broker-API-Originating-Identity string (header)	identity of the user that initiated the request from the Platform X-Broker-API-Originating-Identity
instance_id * required string (path)	instance id of instance to update instance_id
accepts_incomplete boolean (query)	asynchronous operations supported --

Request body required — application/json

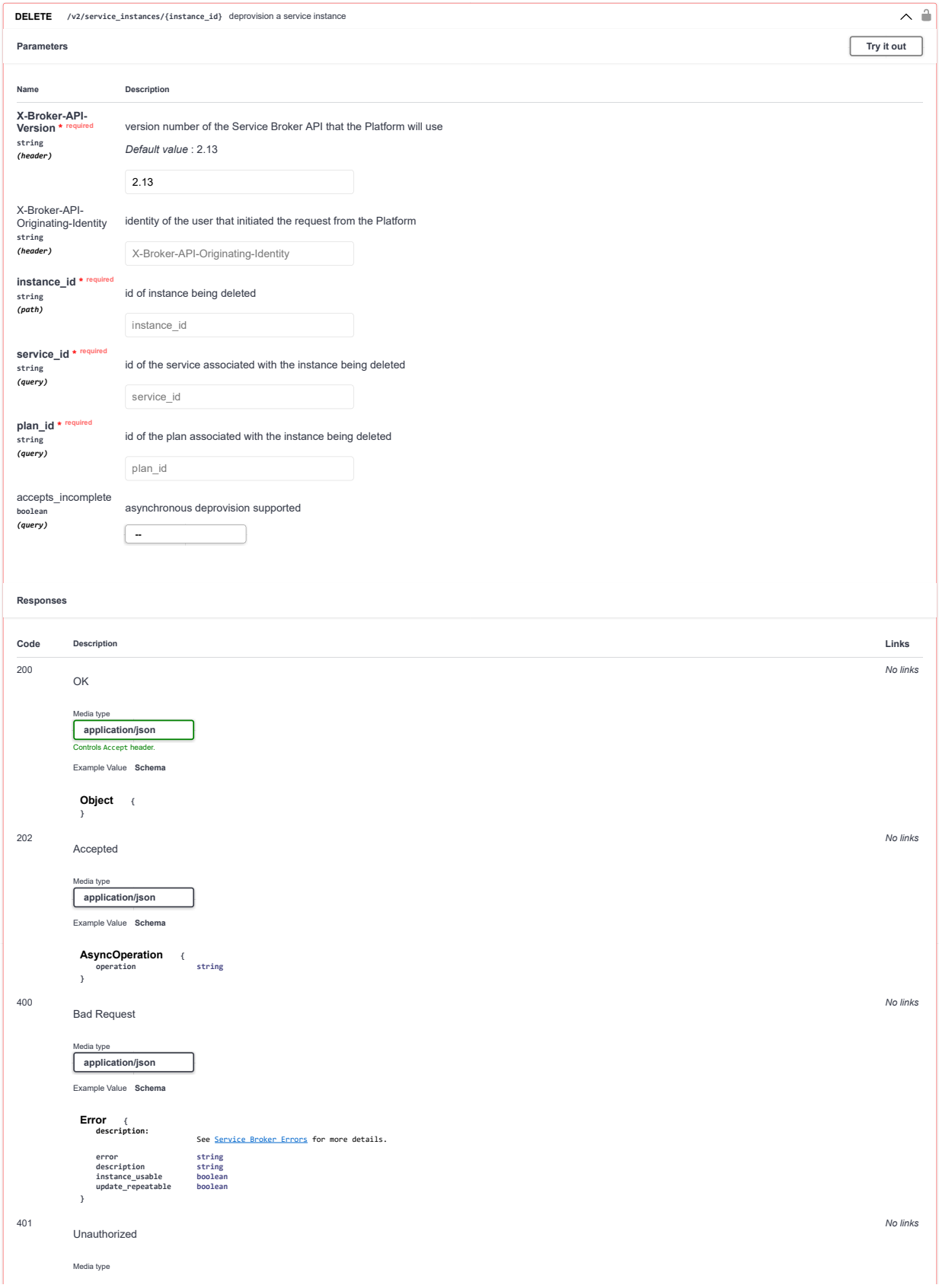
parameters for the requested service instance update

Example Value Schema

ServiceInstanceUpdateRequestBody {
context **Context** {...}
service_id* string
plan_id string
parameters **Object** {...}
previous_values **ServiceInstancePreviousValues** {...}
}

Responses

Code	Description	Links
200	OK Media type: application/json Controls Accept header. Example Value Schema **Object** { } }	*No links*
202	Accepted Media type: application/json Example Value Schema **ServiceInstanceAsyncOperation** { dashboard_url string operation string metadata **ServiceInstanceMetadata** {...} }	*No links*
400	Bad Request Media type: application/json Example Value Schema **Error** { description: See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*
401	Unauthorized Media type: application/json Example Value Schema **Error** { description: See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*
422	Unprocessable entity Media type: application/json Example Value Schema **Error** { description: See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*



DELETE `/v2/service_instances/{instance_id}` deprovision a service instance

Parameters — Try it out

Name	Description
X-Broker-API-Version * required `string` *(header)*	version number of the Service Broker API that the Platform will use *Default value* : 2.13 2.13
X-Broker-API-Originating-Identity `string` *(header)*	identity of the user that initiated the request from the Platform X-Broker-API-Originating-Identity
instance_id * required `string` *(path)*	id of instance being deleted instance_id
service_id * required `string` *(query)*	id of the service associated with the instance being deleted service_id
plan_id * required `string` *(query)*	id of the plan associated with the instance being deleted plan_id
accepts_incomplete `boolean` *(query)*	asynchronous deprovision supported --

Responses

Code	Description	Links
200	OK Media type: application/json Controls Accept header. Example Value Schema **Object** { }	*No links*
202	Accepted Media type: application/json Example Value Schema **AsyncOperation** { operation string }	*No links*
400	Bad Request Media type: application/json Example Value Schema **Error** { description: See Service Broker Errors for more details. error string, description string, instance_usable boolean, update_repeatable boolean }	*No links*
401	Unauthorized Media type	*No links*

Code	Description	Links
	application/json Example Value Schema **Error** { description: See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	
410	Gone Media type application/json **Example Value** Schema { "error": "string", "description": "string", "instance_usable": true, "update_repeatable": true }	*No links*
422	Unprocessable Entity Media type application/json Example Value Schema **Error** { description: See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*

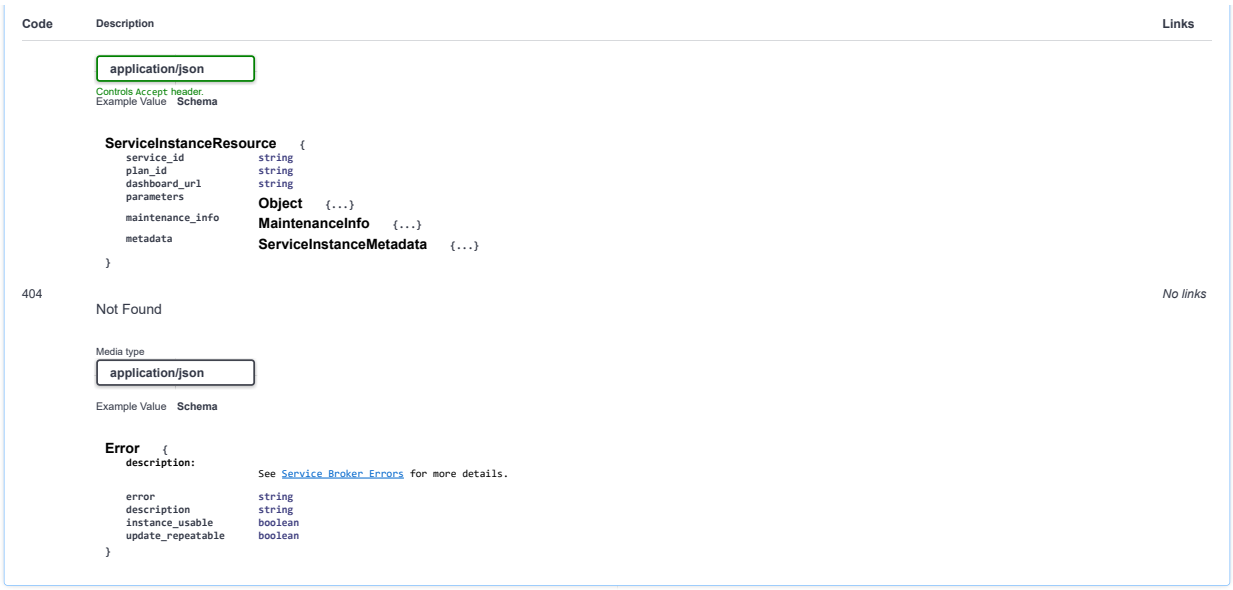
GET `/v2/service_instances/{instance_id}` get a service instance

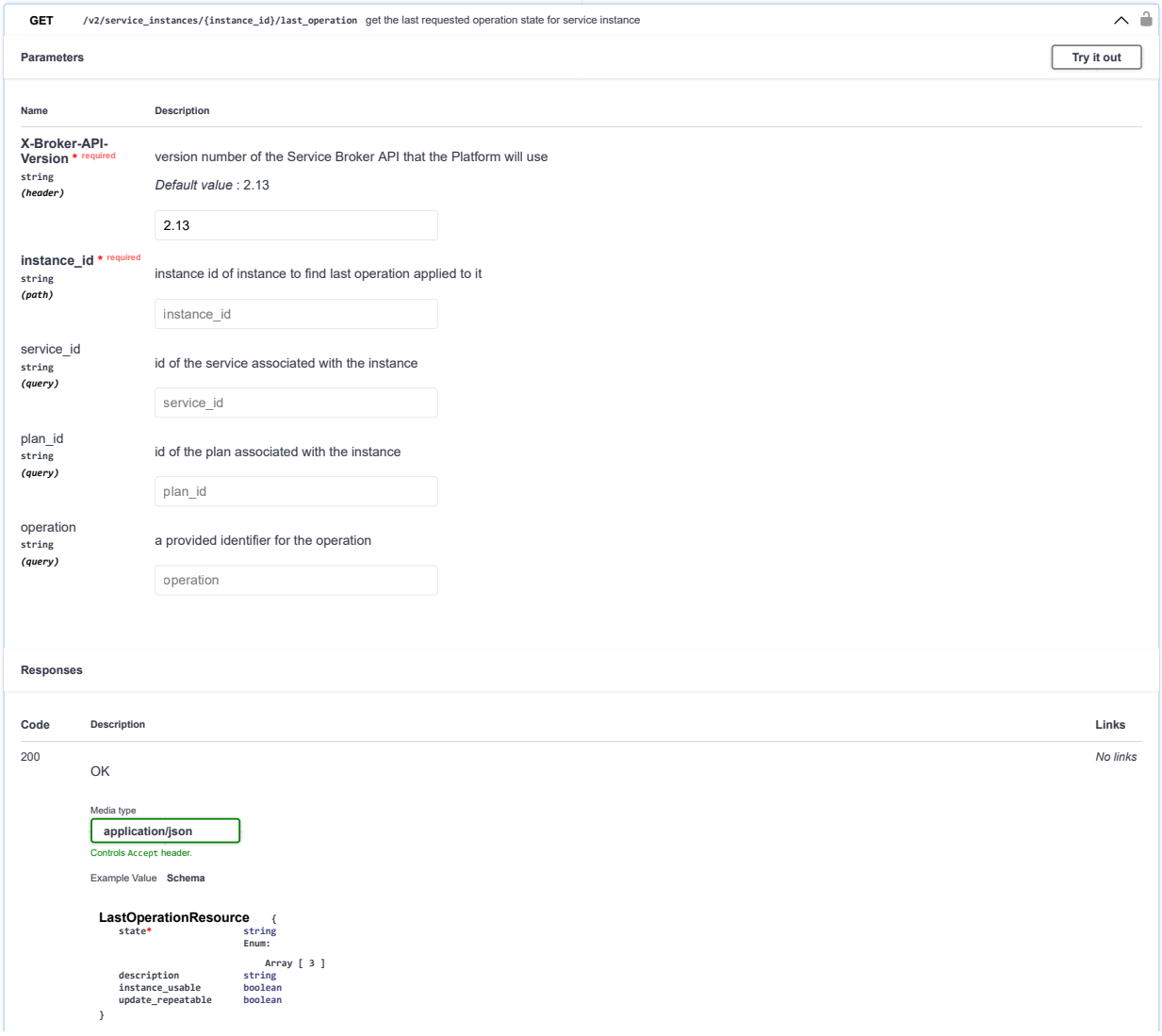
Parameters Try it out

Name	Description
X-Broker-API-Version * required string *(header)*	version number of the Service Broker API that the Platform will use *Default value* : 2.13 2.13
X-Broker-API-Originating-Identity string *(header)*	identity of the user that initiated the request from the Platform X-Broker-API-Originating-Identity
instance_id * required string *(path)*	instance id of instance to fetch instance_id
service_id string *(query)*	id of the service associated with the instance service_id
plan_id string *(query)*	id of the plan associated with the instance plan_id

Responses

Code	Description	Links
200	OK Media type	*No links*


Code
Description
Links
application/json
Controls Accept header.
Example Value Schema
ServiceInstanceResource {
service_id string
plan_id string
dashboard_url string
parameters Object {...}
maintenance_info MaintenanceInfo {...}
metadata ServiceInstanceMetadata {...}
}
404
Not Found
No links
Media type
application/json
Example Value Schema
Error {
description: See Service Broker Errors for more details.
error string
description string
instance_usable boolean
update_repeatable boolean
}


GET /v2/service_instances/{instance_id}/last_operation get the last requested operation state for service instance
Parameters
Try it out
Name
Description
X-Broker-API-Version * required
string
(header)
version number of the Service Broker API that the Platform will use
Default value : 2.13
2.13
instance_id * required
string
(path)
instance id of instance to find last operation applied to it
instance_id
service_id
string
(query)
id of the service associated with the instance
service_id
plan_id
string
(query)
id of the plan associated with the instance
plan_id
operation
string
(query)
a provided identifier for the operation
operation
Responses
Code
Description
Links
200
OK
No links
Media type
application/json
Controls Accept header.
Example Value Schema
LastOperationResource {
state* string
Enum:
Array [3]
description string
instance_usable boolean
update_repeatable boolean
}

Code	Description	Links
	Headers: **Name** / **Description** / **Type** `Retry-After` / `Indicates when to retry the request` / `string`	
400	Bad Request Media type: application/json Example Value **Schema** **Error** { description: See Service Broker Errors for more details. error: string, description: string, instance_usable: boolean, update_repeatable: boolean }	*No links*
401	Unauthorized Media type: application/json Example Value **Schema** **Error** { description: See Service Broker Errors for more details. error: string, description: string, instance_usable: boolean, update_repeatable: boolean }	*No links*
404	Not Found Media type: application/json Example Value **Schema** **Error** { description: See Service Broker Errors for more details. error: string, description: string, instance_usable: boolean, update_repeatable: boolean }	*No links*

ServiceBindings

GET `/v2/service_instances/{instance_id}/service_bindings/{binding_id}/last_operation` get the last requested operation state for service binding

Parameters — Try it out

Name	Description
X-Broker-API-Version * required string *(header)*	version number of the Service Broker API that the Platform will use *Default value* : 2.13 2.13
instance_id * required string *(path)*	instance id of instance to find last operation applied to it instance_id
binding_id * required string *(path)*	binding id of service binding to find last operation applied to it binding_id
service_id string *(query)*	id of the service associated with the instance service_id
plan_id string *(query)*	id of the plan associated with the instance plan_id
operation string	a provided identifier for the operation

Name	Description
(query)	operation

Responses

Code	Description	Links
200	OK Media type application/json Controls Accept header. **Example Value** Schema `{ "state": "in progress", "description": "string", "instance_usable": true, "update_repeatable": true }` **Headers:** Name: `Retry-After` — Description: `Indicates when to retry the request` — Type: `string`	*No links*
400	Bad Request Media type application/json Example Value **Schema** **Error** { description: See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*
401	Unauthorized Media type application/json Example Value **Schema** **Error** { description: See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*
404	Not Found Media type application/json Example Value **Schema** **Error** { description: See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*
410	Gone Media type application/json Example Value **Schema**	*No links*

Code	Description	Links

Error {
description: See Service Broker Errors for more details.
error string
description string
instance_usable boolean
update_repeatable boolean
}

PUT `/v2/service_instances/{instance_id}/service_bindings/{binding_id}` generate a service binding

Parameters Try it out

Name	Description
X-Broker-API-Version * required string *(header)*	version number of the Service Broker API that the Platform will use *Default value* : 2.13 2.13
X-Broker-API-Originating-Identity string *(header)*	identity of the user that initiated the request from the Platform X-Broker-API-Originating-Identity
instance_id * required string *(path)*	instance id of instance to create a binding on instance_id
binding_id * required string *(path)*	binding id of binding to create binding_id
accepts_incomplete boolean *(query)*	asynchronous operations supported --

Request body required — application/json

parameters for the requested service binding

Example Value **Schema**

ServiceBindingRequest {
context **Context** {...}
service_id* string
plan_id* string
~~app_guid~~ string *deprecated: true*
bind_resource **ServiceBindingResouceObject** {...}
parameters **Object** {...}
predecessor_binding_id string
}

Responses

Code	Description	Links
200	OK Media type: application/json Controls Accept header. Example Value **Schema** **ServiceBindingResponse** { metadata **ServiceBindingMetadata** {...} credentials **Object** {...} syslog_drain_url string route_service_url string volume_mounts [...] endpoints [...] }	*No links*
201	Created	*No links*

Code	Description	Links

Media type

application/json

Example Value **Schema**

ServiceBindingResponse {
- metadata: **ServiceBindingMetadata** {...}
- credentials: **Object** {...}
- syslog_drain_url: string
- route_service_url: string
- volume_mounts: [...]
- endpoints: [...]

}

202 — Accepted — *No links*

Media type

application/json

Example Value **Schema**

AsyncOperation {
- operation: string

}

400 — Bad Request — *No links*

Media type

application/json

Example Value **Schema**

Error {
- description: See Service Broker Errors for more details.
- error: string
- description: string
- instance_usable: boolean
- update_repeatable: boolean

}

401 — Unauthorized — *No links*

Media type

application/json

Example Value **Schema**

Error {
- description: See Service Broker Errors for more details.
- error: string
- description: string
- instance_usable: boolean
- update_repeatable: boolean

}

409 — Conflict — *No links*

Media type

application/json

Example Value **Schema**

Error {
- description: See Service Broker Errors for more details.
- error: string
- description: string
- instance_usable: boolean
- update_repeatable: boolean

}

422 — Unprocessable Entity — *No links*

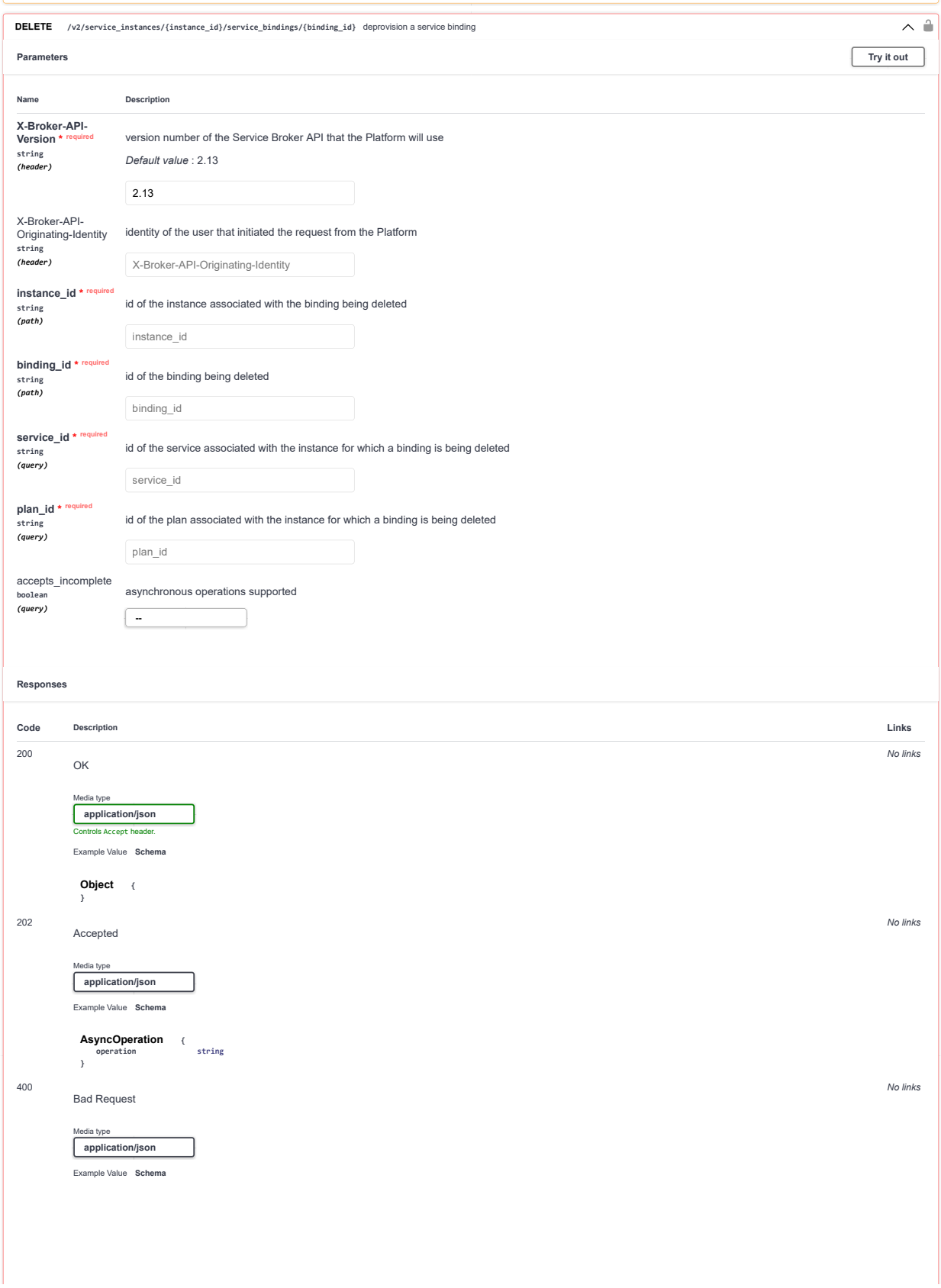
Media type

application/json

Example Value **Schema**

Error {
- description: See Service Broker Errors for more details.
- error: string
- description: string
- instance_usable: boolean
- update_repeatable: boolean

}



DELETE `/v2/service_instances/{instance_id}/service_bindings/{binding_id}` deprovision a service binding

Parameters — Try it out

Name	Description
X-Broker-API-Version * required string *(header)*	version number of the Service Broker API that the Platform will use *Default value* : 2.13 2.13
X-Broker-API-Originating-Identity string *(header)*	identity of the user that initiated the request from the Platform X-Broker-API-Originating-Identity
instance_id * required string *(path)*	id of the instance associated with the binding being deleted instance_id
binding_id * required string *(path)*	id of the binding being deleted binding_id
service_id * required string *(query)*	id of the service associated with the instance for which a binding is being deleted service_id
plan_id * required string *(query)*	id of the plan associated with the instance for which a binding is being deleted plan_id
accepts_incomplete boolean *(query)*	asynchronous operations supported --

Responses

Code	Description	Links
200	OK Media type: application/json Controls Accept header. Example Value Schema **Object** { }	*No links*
202	Accepted Media type: application/json Example Value Schema **AsyncOperation** { operation string }	*No links*
400	Bad Request Media type: application/json Example Value Schema	*No links*

Code	Description	Links
	Error { **description:** See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	
401	Unauthorized Media type application/json Example Value **Schema** **Error** { **description:** See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*
410	Gone Media type application/json Example Value **Schema** **Error** { **description:** See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*
422	Unprocessable Entity Media type application/json Example Value **Schema** **Error** { **description:** See Service Broker Errors for more details. error string description string instance_usable boolean update_repeatable boolean }	*No links*

GET `/v2/service_instances/{instance_id}/service_bindings/{binding_id}` get a service binding

Parameters Try it out

Name	Description
X-Broker-API-Version * required string *(header)*	version number of the Service Broker API that the Platform will use *Default value* : 2.13 2.13
X-Broker-API-Originating-Identity string *(header)*	identity of the user that initiated the request from the Platform X-Broker-API-Originating-Identity
instance_id * required string *(path)*	instance id of instance associated with the binding instance_id
binding_id * required string *(path)*	binding id of binding to fetch binding_id

Name	Description
service_id string *(query)*	id of the service associated with the instance service_id
plan_id string *(query)*	id of the plan associated with the instance plan_id

Responses

Code	Description	Links
200	OK Media type application/json Controls Accept header. Example Value Schema	*No links*

Schemas

ServiceBindingResource {
- metadata — **ServiceBindingMetadata** {...}
- credentials — **Object** {...}
- syslog_drain_url — string
- route_service_url — string
- volume_mounts — [...]
- endpoints

}

Catalog {
- services — [...]

}

Service {
- name* — string
- id* — string
- description* — string
- tags — [...]
- requires — [...]
- bindable* — boolean
- metadata — **Metadata** {...}
- dashboard_client — **DashboardClient** {...}
- binding_rotatable — boolean
- plan_updateable — boolean
- plans* — [...]

}

DashboardClient {
- id — string
- secret — string
- redirect_uri — string

}

Plan {
- id* — string
- name* — string
- description* — string
- metadata — **Metadata** {...}
- maintenance_info — **MaintenanceInfo** {...}
- free — boolean *default: true*
- bindable — boolean
- schemas — **Schemas** {...}
- maximum_polling_duration — integer
- plan_updateable — boolean
- binding_rotatable — boolean *default: false*

}

Schemas {
- service_instance — **ServiceInstanceSchema** {...}
- service_binding — **ServiceBindingSchema** {...}

}

ServiceInstanceSchema {
create {
parameters {...}
}
update {
parameters {...}
}
}

ServiceBindingSchema {
create {
parameters {...}
}
}

ServiceInstanceResource {
service_id string
plan_id string
dashboard_url string
parameters **Object** {...}
maintenance_info **MaintenanceInfo** {...}
metadata **ServiceInstanceMetadata** {...}
}

ServiceInstanceProvisionRequestBody {
service_id* string
plan_id* string
context **Context** {...}
~~organization_guid*~~ string
deprecated: true
~~space_guid*~~ string
deprecated: true
parameters **Object** {...}
}

ServiceInstanceProvisionResponse {
dashboard_url string
metadata **ServiceInstanceMetadata** {...}
}

ServiceInstanceAsyncOperation {
dashboard_url string
operation string
metadata **ServiceInstanceMetadata** {...}
}

ServiceInstanceMetadata {
labels {...}
attributes {...}
}

ServiceInstanceUpdateRequestBody {
context **Context** {...}
service_id* string
plan_id string
parameters **Object** {...}
previous_values **ServiceInstancePreviousValues** {...}
}

ServiceInstancePreviousValues {
~~service_id~~ string
deprecated: true
plan_id string
~~organization_id~~ string
deprecated: true
~~space_id~~ string
deprecated: true
}

AsyncOperation {
operation string
}

LastOperationResource {
- state* — string
  Enum:
  Array [3]
- description — string
- instance_usable — boolean
- update_repeatable — boolean

}

ServiceBindingResource {
- metadata — **ServiceBindingMetadata** {...}
- credentials — **Object** {...}
- syslog_drain_url — string
- route_service_url — string
- volume_mounts — [...]
- endpoints — [...]
- parameters — **Object** {...}

}

ServiceBindingRequest {
- context — **Context** {...}
- service_id* — string
- plan_id* — string
- ~~app_guid~~ — string
  deprecated: true
- bind_resource — **ServiceBindingResouceObject** {...}
- parameters — **Object** {...}
- predecessor_binding_id — string

}

ServiceBindingMetadata {
- expires_at — string
- renew_before — string

}

ServiceBindingResouceObject {
- app_guid — string
- route — string

}

ServiceBindingResponse {
- metadata — **ServiceBindingMetadata** {...}
- credentials — **Object** {...}
- syslog_drain_url — string
- route_service_url — string
- volume_mounts — [...]
- endpoints — [...]

}

ServiceBindingEndpoint {
- host* — string
- ports* — [...]
- protocol — string
  default: tcp
  Enum:
  Array [3]

}

ServiceBindingVolumeMount {
- driver* — string
- container_dir* — string
- mode* — string
  Enum:
  Array [2]
- device_type* — string
  Enum:
  Array [1]
- device* — **ServiceBindingVolumeMountDevice** {...}

}

ServiceBindingVolumeMountDevice {
- volume_id* — string
- mount_config — **Object** {...}

}

Context {
- description:
  See Context Conventions for more details.

}

Metadata {
description:
See Service Metadata Conventions for more details.

}

MaintenanceInfo {
version* string
description string
}

Object {
}

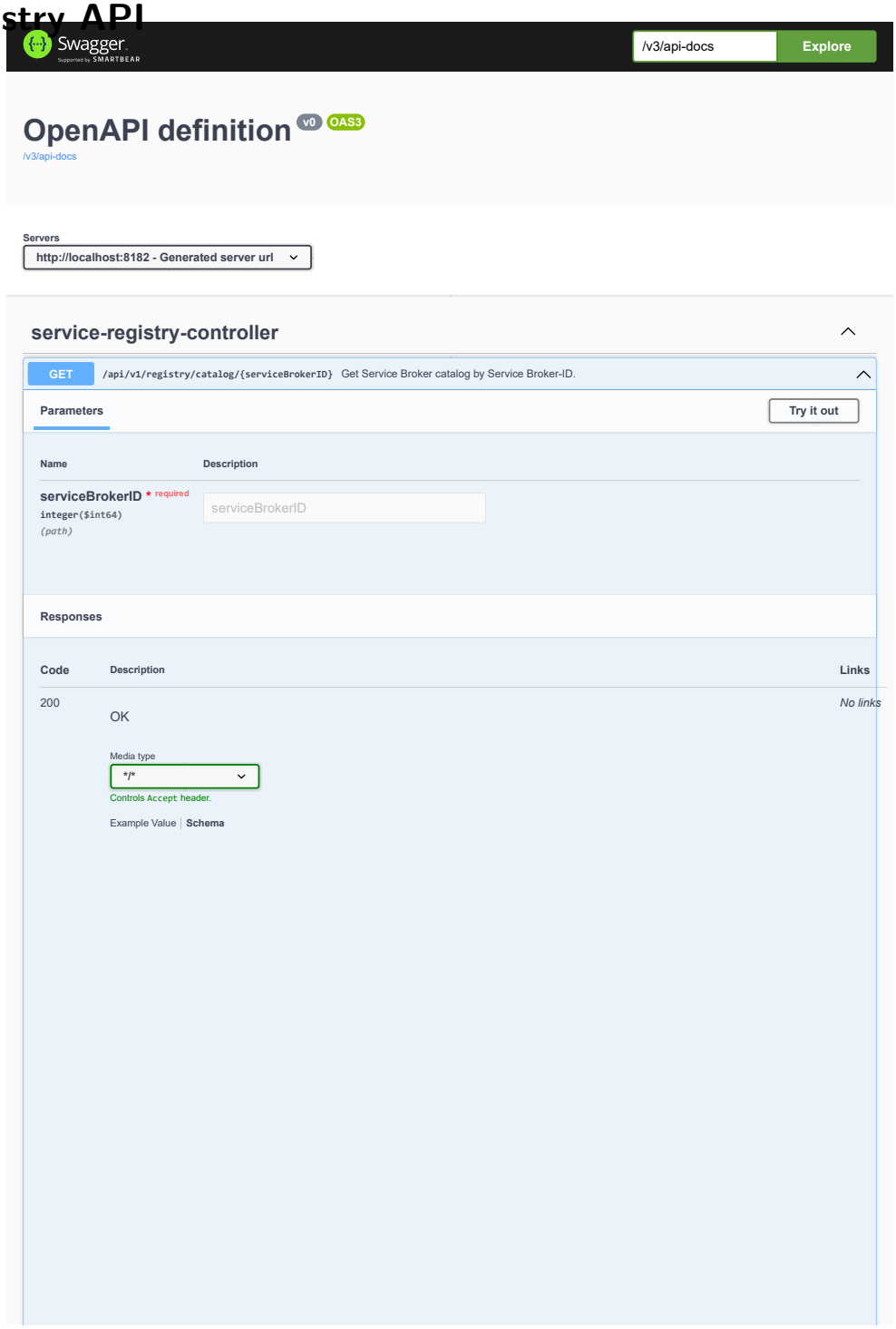
Error {
description:
See Service Broker Errors for more details.

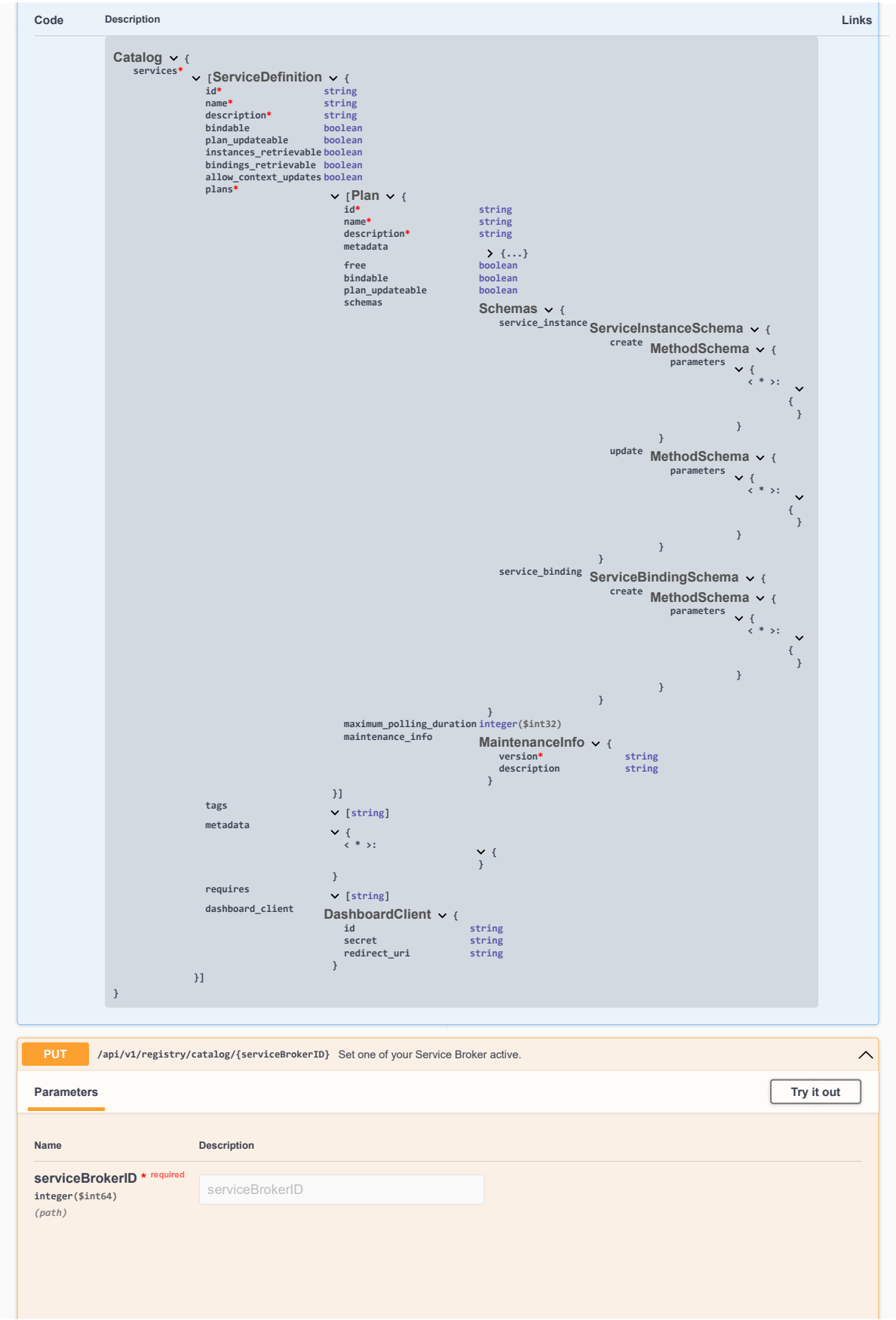
error string
description string
instance_usable boolean
update_repeatable boolean
}

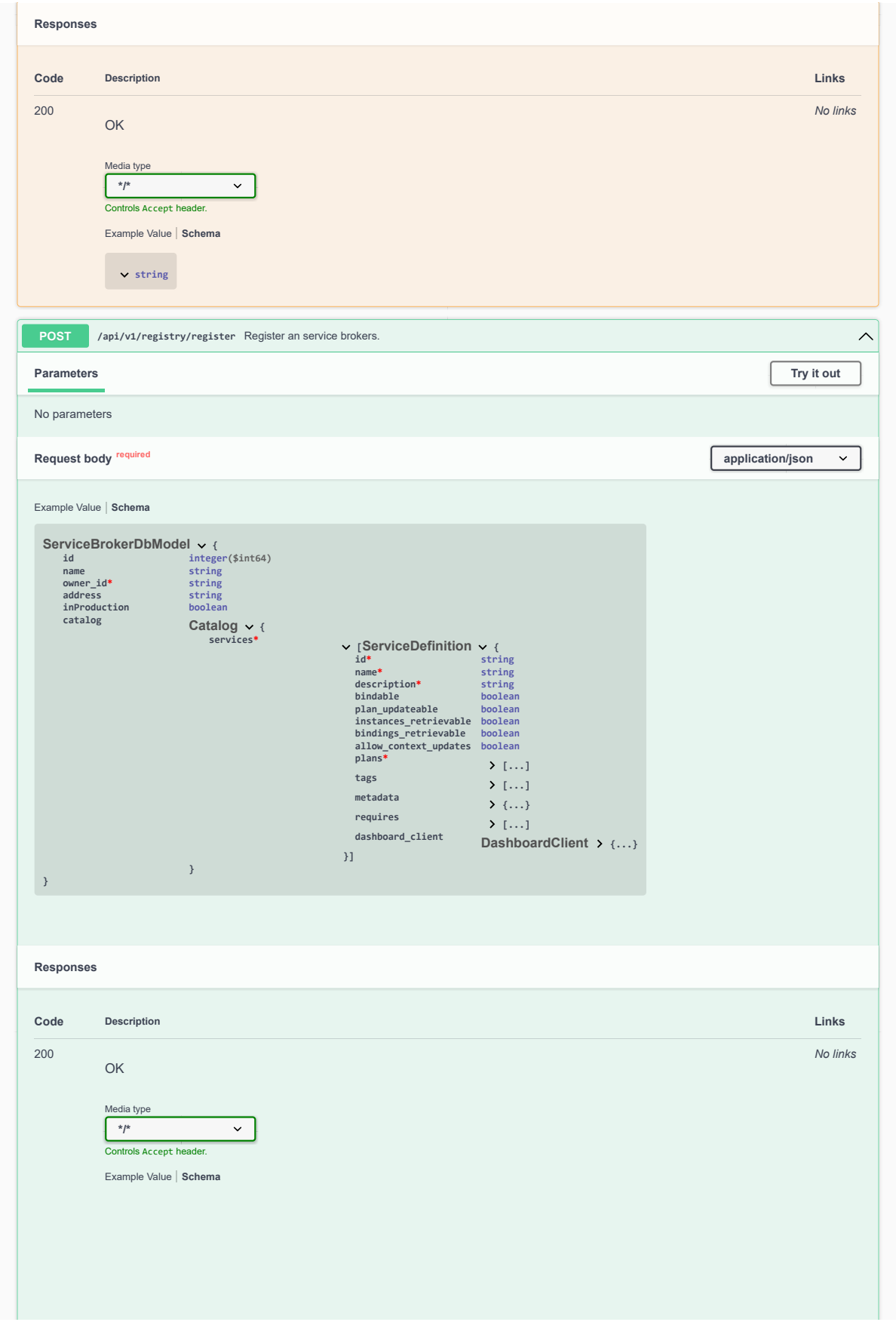
VALID {...}

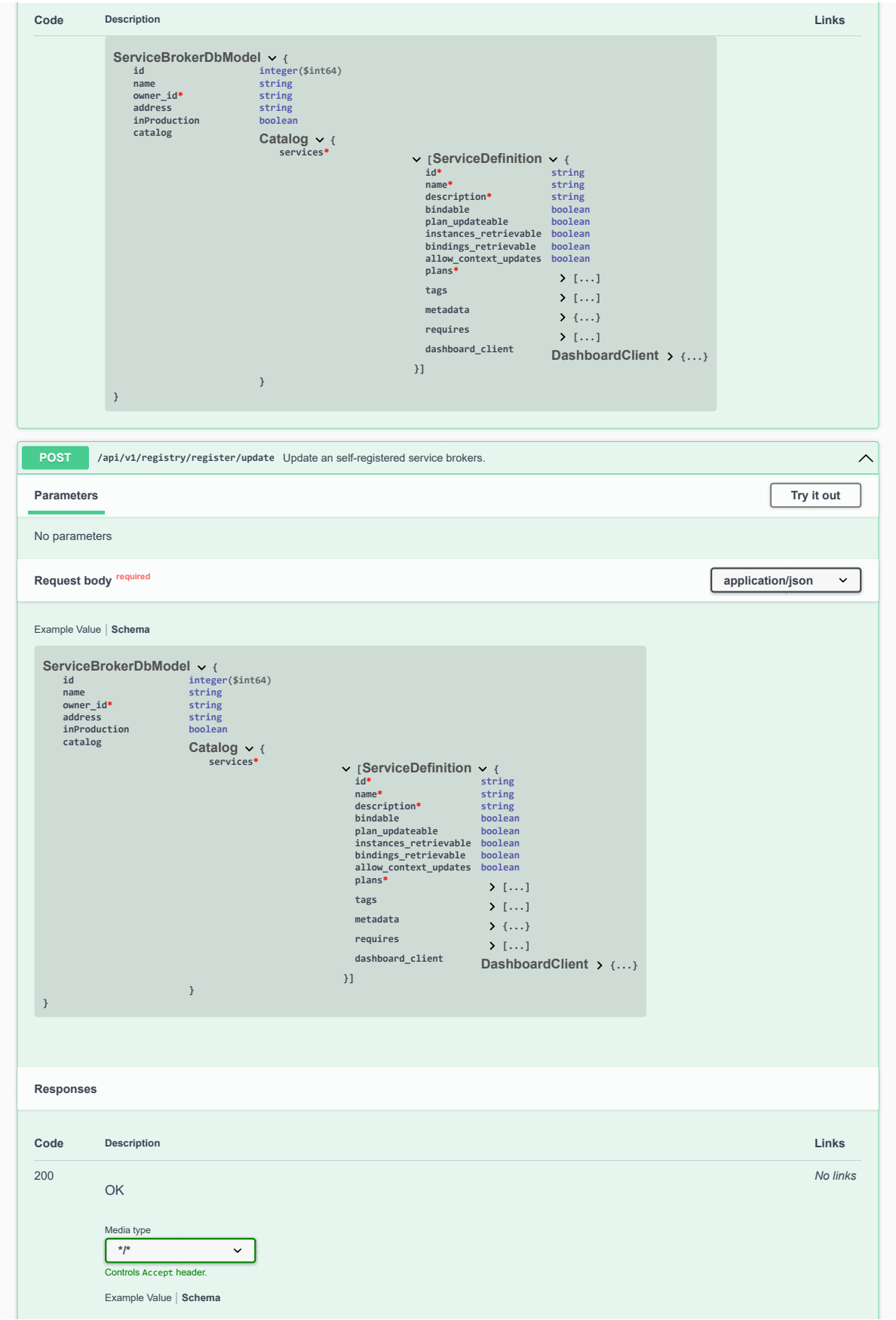
9.5 Marketplace REST-API-Dokumentation

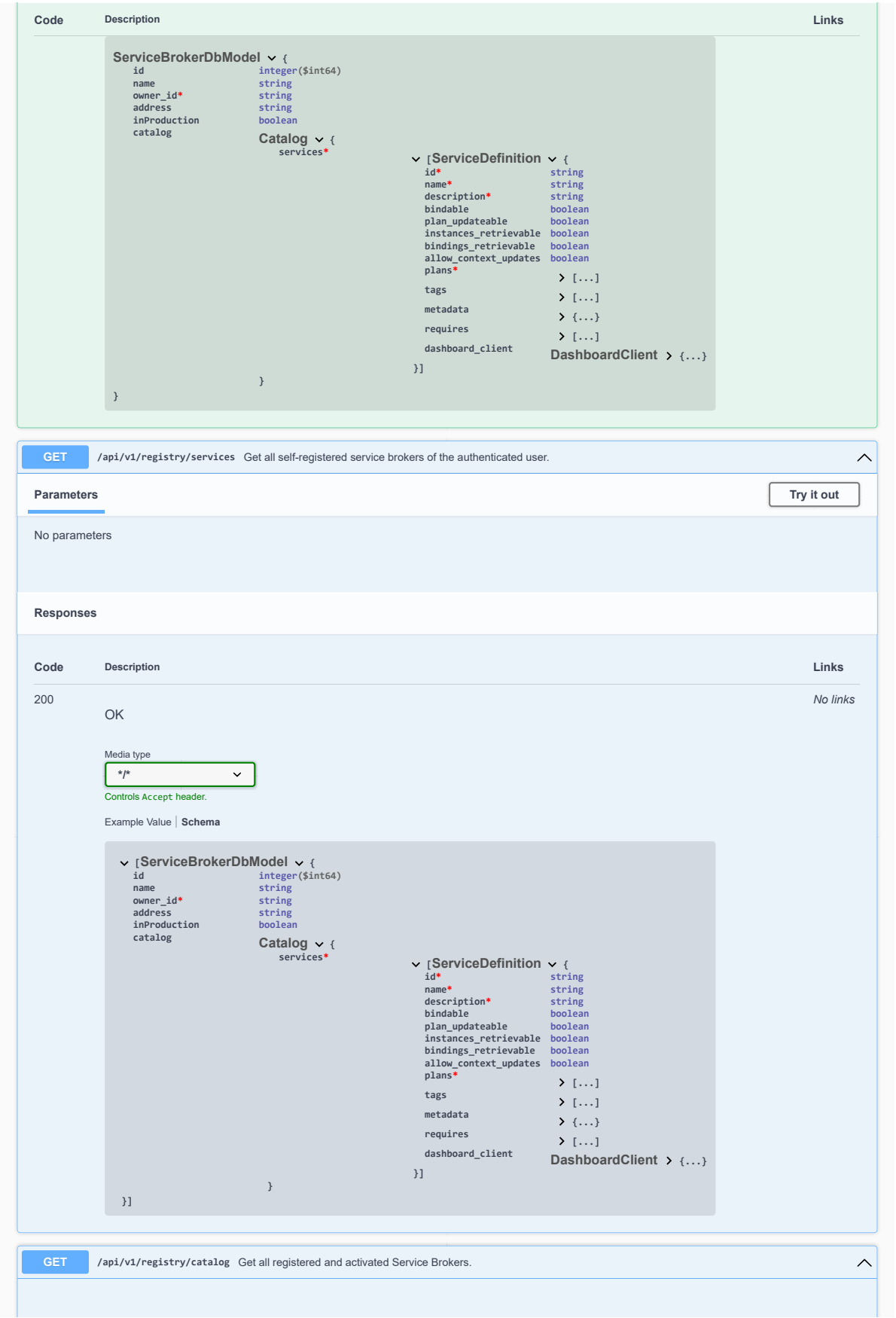
Registry API

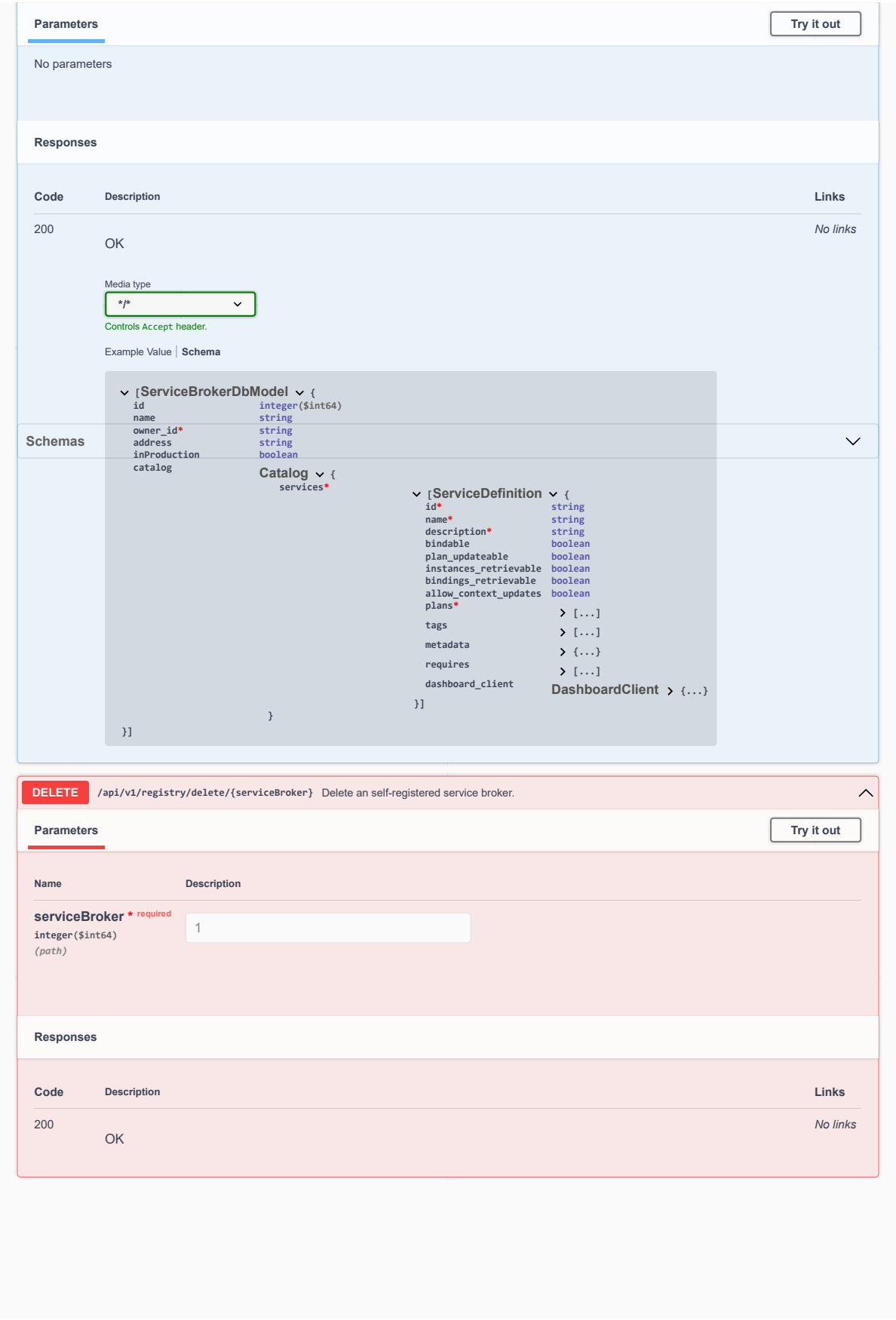


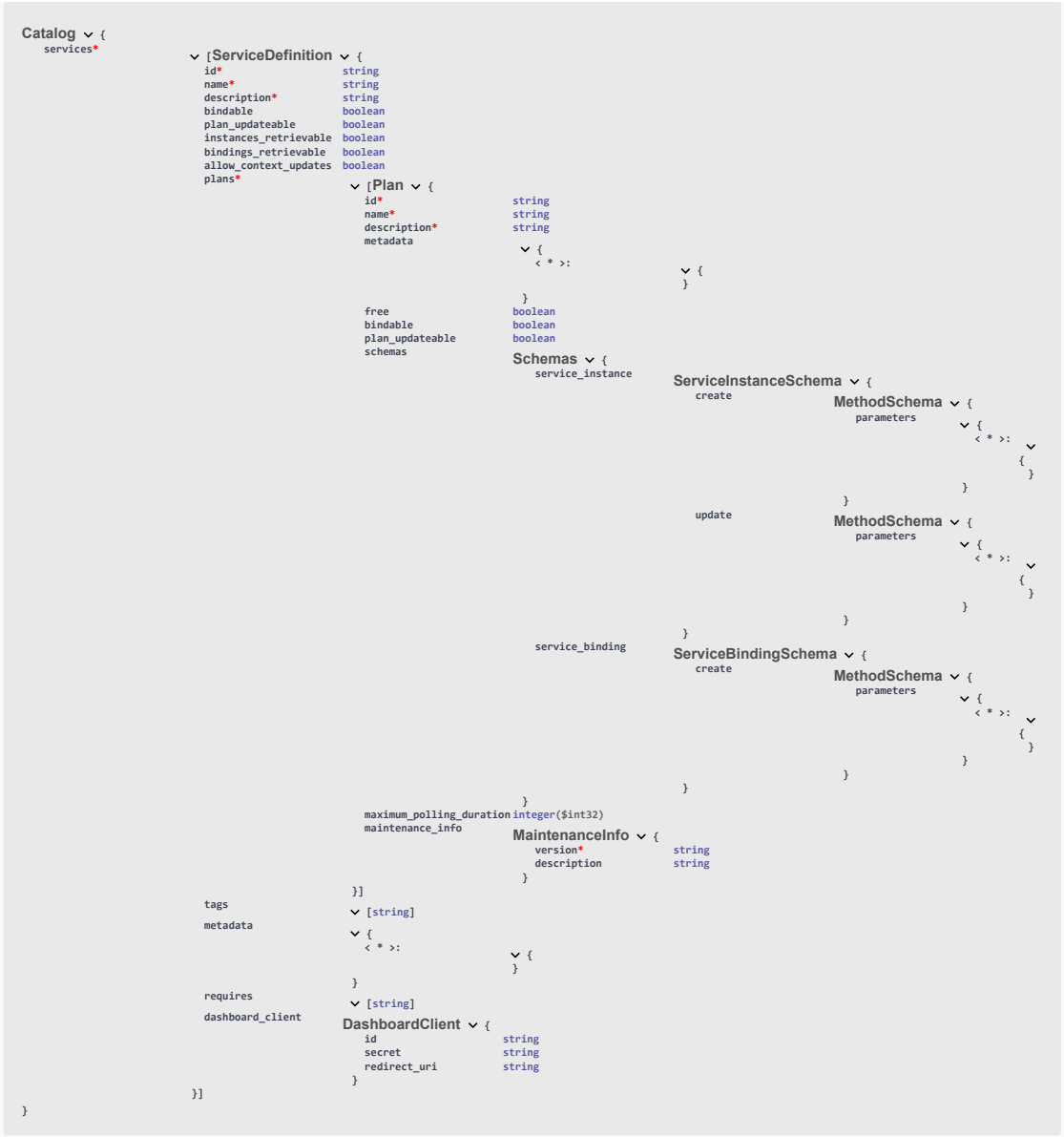

Code
Description
Links
Catalog { services* [ServiceDefinition { id* string name* string description* string bindable boolean plan_updateable boolean instances_retrievable boolean bindings_retrievable boolean allow_context_updates boolean plans* [Plan { id* string name* string description* string metadata {...} free boolean bindable boolean plan_updateable boolean schemas Schemas { service_instance ServiceInstanceSchema { create MethodSchema { parameters { < * >: { } } } update MethodSchema { parameters { < * >: { } } } } service_binding ServiceBindingSchema { create MethodSchema { parameters { < * >: { } } } } } maximum_polling_duration integer($int32) maintenance_info MaintenanceInfo { version* string description string } }] tags [string] metadata { < * >: { } } requires [string] dashboard_client DashboardClient { id string secret string redirect_uri string } }] }
PUT
/api/v1/registry/catalog/{serviceBrokerID}
Set one of your Service Broker active.
Parameters
Try it out
Name
Description
serviceBrokerID * required
integer($int64)
(path)
serviceBrokerID


Responses
Code
Description
Links
200
OK
No links
Media type
/
Controls Accept header.
Example Value | Schema
string
POST
/api/v1/registry/register
Register an service brokers.
Parameters
Try it out
No parameters
Request body required
application/json
Example Value | Schema
ServiceBrokerDbModel { id integer($int64) name string owner_id* string address string inProduction boolean catalog Catalog { services* [ServiceDefinition { id* string name* string description* string bindable boolean plan_updateable boolean instances_retrievable boolean bindings_retrievable boolean allow_context_updates boolean plans* [...] tags [...] metadata {...} requires [...] dashboard_client DashboardClient {...} }] } }
Responses
Code
Description
Links
200
OK
No links
Media type
/
Controls Accept header.
Example Value | Schema

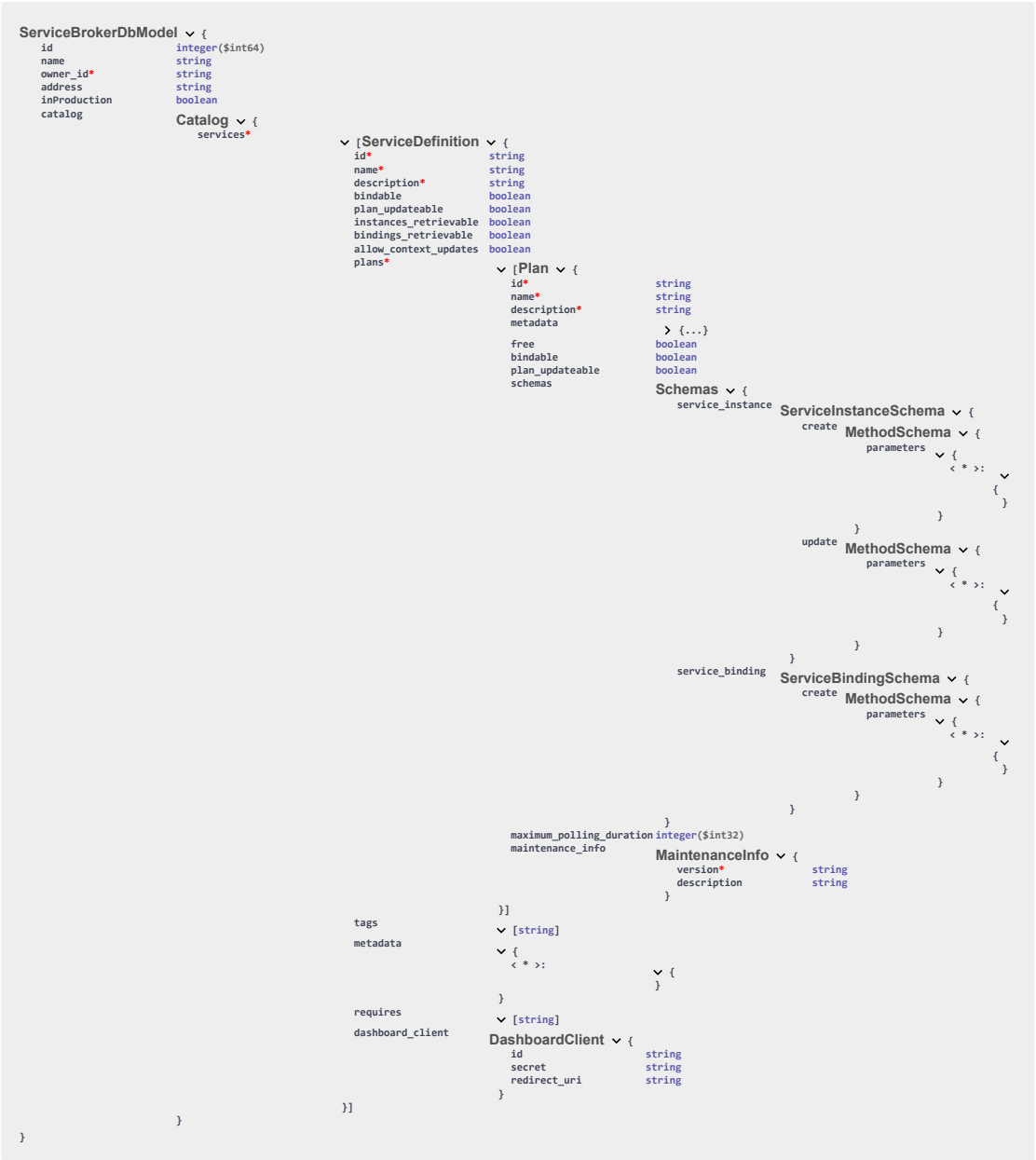

Code
Description
Links
ServiceBrokerDbModel { id integer($int64) name string owner_id* string address string inProduction boolean catalog Catalog { services* [ServiceDefinition { id* string name* string description* string bindable boolean plan_updateable boolean instances_retrievable boolean bindings_retrievable boolean allow_context_updates boolean plans* [...] tags [...] metadata {...} requires [...] dashboard_client DashboardClient {...} }] } }
POST
/api/v1/registry/register/update
Update an self-registered service brokers.
Parameters
Try it out
No parameters
Request body required
application/json
Example Value | Schema
ServiceBrokerDbModel { id integer($int64) name string owner_id* string address string inProduction boolean catalog Catalog { services* [ServiceDefinition { id* string name* string description* string bindable boolean plan_updateable boolean instances_retrievable boolean bindings_retrievable boolean allow_context_updates boolean plans* [...] tags [...] metadata {...} requires [...] dashboard_client DashboardClient {...} }] } }
Responses
Code
Description
Links
200
OK
No links
Media type
/
Controls Accept header.
Example Value | Schema

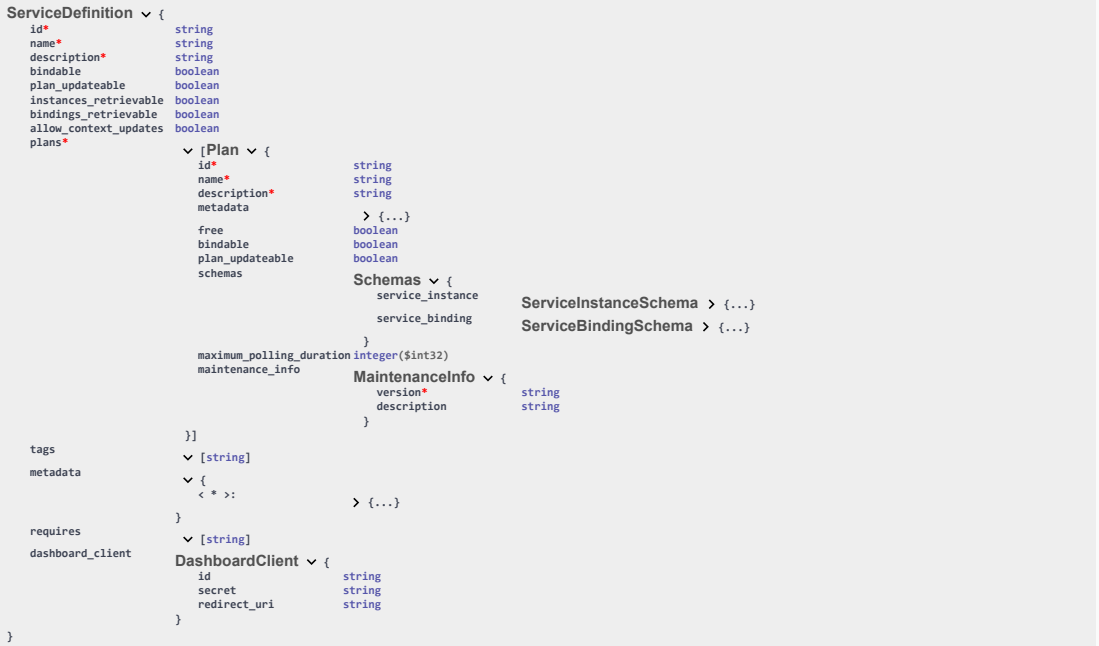

Code
Description
Links
ServiceBrokerDbModel { id integer($int64) name string owner_id* string address string inProduction boolean catalog Catalog { services* [ServiceDefinition { id* string name* string description* string bindable boolean plan_updateable boolean instances_retrievable boolean bindings_retrievable boolean allow_context_updates boolean plans* [...] tags [...] metadata {...} requires [...] dashboard_client DashboardClient {...} }] } }
GET
/api/v1/registry/services
Get all self-registered service brokers of the authenticated user.
Parameters
Try it out
No parameters
Responses
Code
Description
Links
200
OK
No links
Media type
/
Controls Accept header.
Example Value | Schema
[ServiceBrokerDbModel { id integer($int64) name string owner_id* string address string inProduction boolean catalog Catalog { services* [ServiceDefinition { id* string name* string description* string bindable boolean plan_updateable boolean instances_retrievable boolean bindings_retrievable boolean allow_context_updates boolean plans* [...] tags [...] metadata {...} requires [...] dashboard_client DashboardClient {...} }] } }]
GET
/api/v1/registry/catalog
Get all registered and activated Service Brokers.

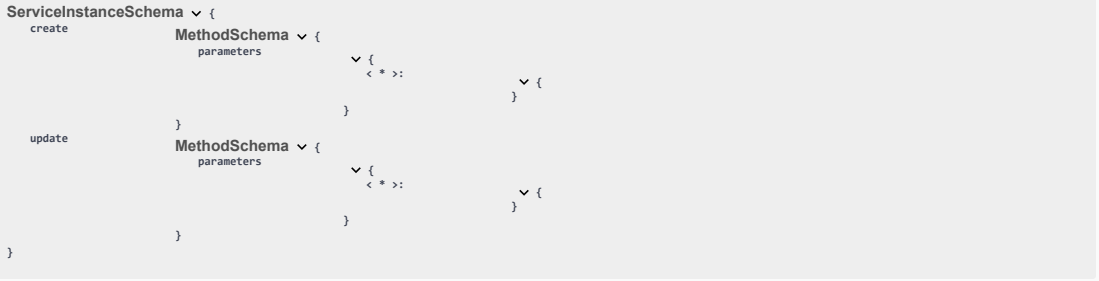

Parameters
Try it out
No parameters
Responses
Code
Description
Links
200
OK
No links
Media type
/
Controls Accept header.
Example Value | Schema
[ServiceBrokerDbModel { id integer($int64) name string owner_id* string address string inProduction boolean catalog Catalog { services* [ServiceDefinition { id* string name* string description* string bindable boolean plan_updateable boolean instances_retrievable boolean bindings_retrievable boolean allow_context_updates boolean plans* [...] tags [...] metadata {...} requires [...] dashboard_client DashboardClient {...} }] } }]
Schemas
DELETE
/api/v1/registry/delete/{serviceBroker}
Delete an self-registered service broker.
Parameters
Try it out
Name
Description
serviceBroker * required
integer($int64)
(path)
1
Responses
Code
Description
Links
200
OK
No links


Catalog { services* [ServiceDefinition { id* string name* string description* string bindable boolean plan_updateable boolean instances_retrievable boolean bindings_retrievable boolean allow_context_updates boolean plans* [Plan { id* string name* string description* string metadata { < * >: { } } free boolean bindable boolean plan_updateable boolean schemas Schemas { service_instance ServiceInstanceSchema { create MethodSchema { parameters { < * >: { } } } update MethodSchema { parameters { < * >: { } } } } service_binding ServiceBindingSchema { create MethodSchema { parameters { < * >: { } } } } } maximum_polling_duration integer($int32) maintenance_info MaintenanceInfo { version* string description string } }] tags [string] metadata { < * >: { } } requires [string] dashboard_client DashboardClient { id string secret string redirect_uri string } }] }
DashboardClient { id string secret string redirect_uri string }
MaintenanceInfo { version* string description string }
MethodSchema { parameters { < * >: { } } }

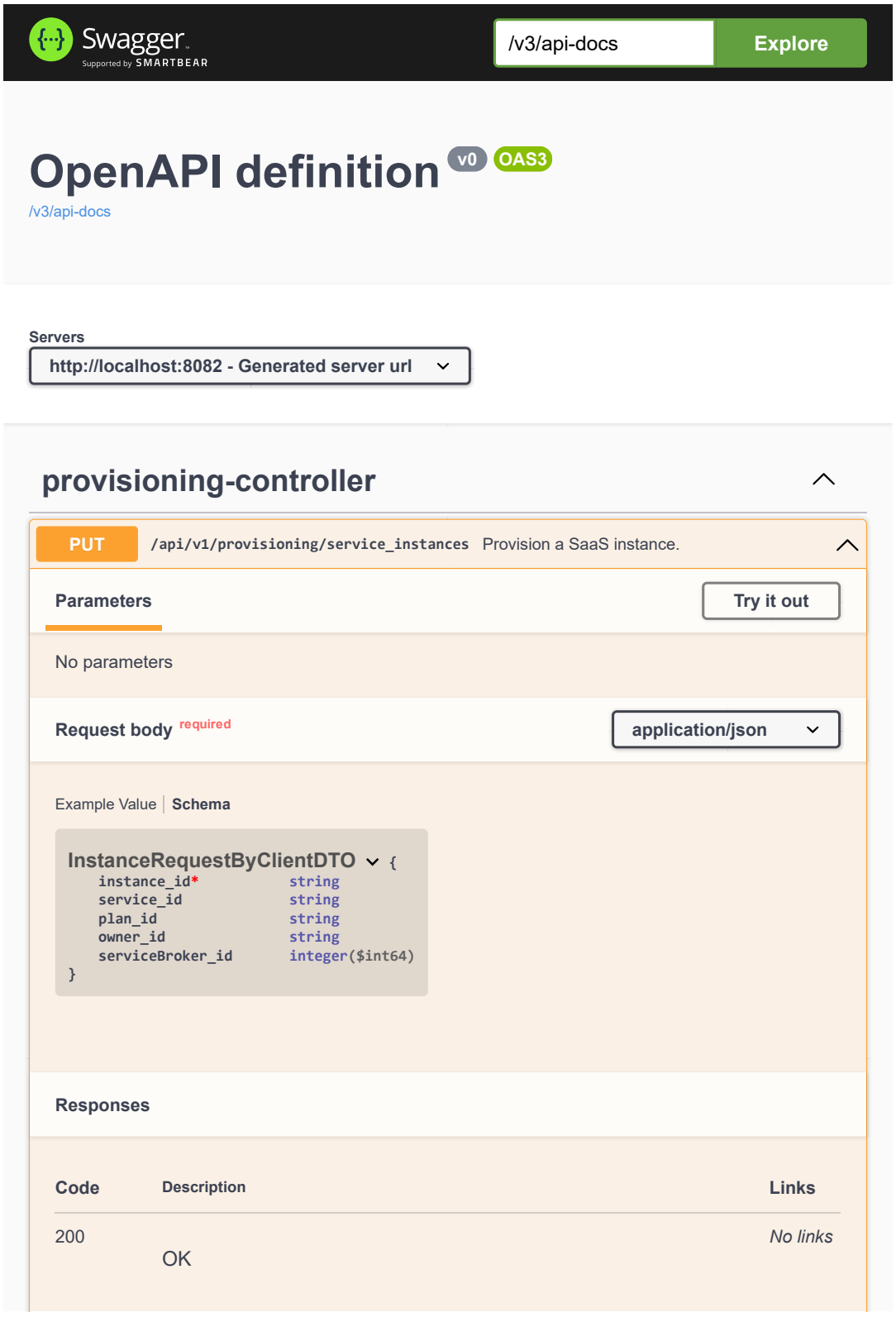

Plan { id* string name* string description* string metadata {...} free boolean bindable boolean plan_updateable boolean schemas Schemas { service_instance ServiceInstanceSchema { create MethodSchema { parameters { < * >: { } } } update MethodSchema { parameters { < * >: { } } } } service_binding ServiceBindingSchema { create MethodSchema { parameters { < * >: { } } } } } maximum_polling_duration integer($int32) maintenance_info MaintenanceInfo { version* string description string } }
Schemas { service_instance ServiceInstanceSchema { create MethodSchema { parameters { < * >: { } } } update MethodSchema { parameters { < * >: { } } } } service_binding ServiceBindingSchema { create MethodSchema { parameters { < * >: { } } } } }
ServiceBindingSchema { create MethodSchema { parameters { < * >: { } } } }


ServiceBrokerDbModel { id integer($int64) name string owner_id* string address string inProduction boolean catalog Catalog { services* [ServiceDefinition { id* string name* string description* string bindable boolean plan_updateable boolean instances_retrievable boolean bindings_retrievable boolean allow_context_updates boolean plans* [Plan { id* string name* string description* string metadata {...} free boolean bindable boolean plan_updateable boolean schemas Schemas { service_instance ServiceInstanceSchema { create MethodSchema { parameters { < * >: { } } } update MethodSchema { parameters { < * >: { } } } } service_binding ServiceBindingSchema { create MethodSchema { parameters { < * >: { } } } } } maximum_polling_duration integer($int32) maintenance_info MaintenanceInfo { version* string description string } }] tags [string] metadata { < * >: { } } requires [string] dashboard_client DashboardClient { id string secret string redirect_uri string } }] } }


ServiceDefinition { id* string name* string description* string bindable boolean plan_updateable boolean instances_retrievable boolean bindings_retrievable boolean allow_context_updates boolean plans* [Plan { id* string name* string description* string metadata {...} free boolean bindable boolean plan_updateable boolean schemas Schemas { service_instance ServiceInstanceSchema {...} service_binding ServiceBindingSchema {...} } maximum_polling_duration integer($int32) maintenance_info MaintenanceInfo { version* string description string } }] tags [string] metadata { < * >: {...} } requires [string] dashboard_client DashboardClient { id string secret string redirect_uri string } }


ServiceInstanceSchema { create MethodSchema { parameters { < * >: { } } } update MethodSchema { parameters { < * >: { } } } }

Service Provisioning Service API



Code	Description	Links

Media type

/

Controls `Accept` header.

Example Value | **Schema**

```
string
```

PUT `/api/v1/provisioning/binding` Get an instance binding for an existing SaaS instance of the authenticated user.

Parameters Try it out

No parameters

Request body required application/json

Example Value | **Schema**

InstanceBindingRequestDTO {

```
id                  integer($int64)
serviceBroker_id    integer($int64)
instance_id         string
service_id          string
plan_id             string
binding_id          string
}
```

Responses

Code	Description	Links
200	OK	*No links*

Media type

/

Controls `Accept` header.

Example Value | **Schema**

Code	Description	Links

```
InstanceBindingResponseDTO ⌄ {
    credentials          Credentials ⌄ {
                             password    string
                             uri         string
                             username    string
                         }
    description          string
}
```

GET `/api/v1/provisioning/service_instances/status/{instance_id}` Get the instance status from the service broker.

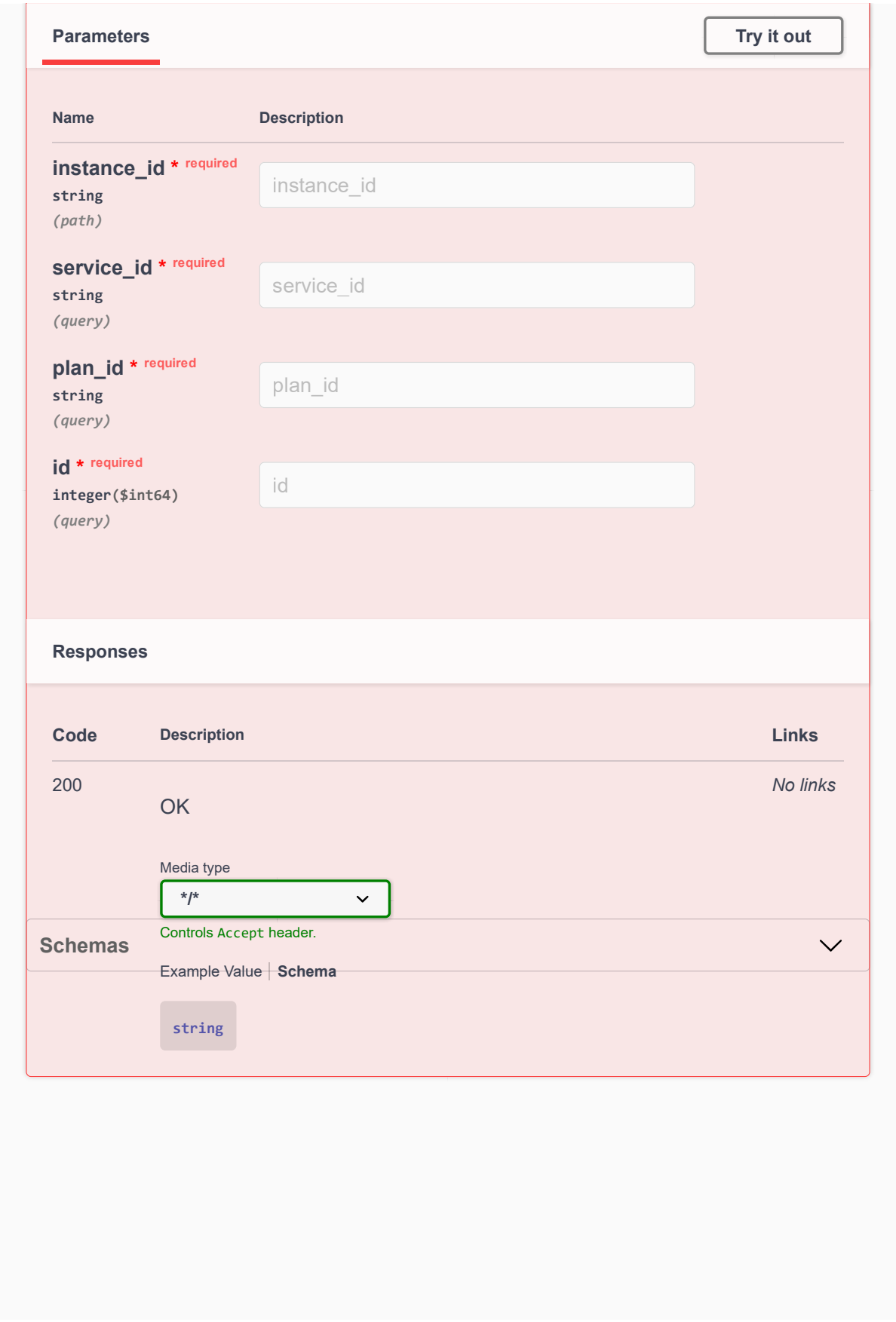
Parameters | **Try it out**

Name	Description
instance_id * required `integer($int64)` *(path)*	instance_id

Responses

Code	Description	Links
200	OK Media type */* Controls Accept header. Example Value | **Schema** `string` `Enum:` ⌄ `[ ACTIVE, FAILED, PROCESSING, DEPLOYING, DISABLED, DEPROVISIONING, DEPROVISIONED ]`	*No links*

DELETE `/api/v1/provisioning/service_instances/{instance_id}` Deprovision an existing SaaS instance of the authenticated user.


Parameters
Try it out
Name
Description
instance_id * required
string
(path)
instance_id
service_id * required
string
(query)
service_id
plan_id * required
string
(query)
plan_id
id * required
integer($int64)
(query)
id
Responses
Code
Description
Links
200
OK
No links
Media type
/
Controls Accept header.
Schemas
Example Value | Schema
string

Schemas

InstanceRequestByClientDTO {

instance_id*	string
service_id	string
plan_id	string
owner_id	string
serviceBroker_id	integer($int64)

}

InstanceBindingRequestDTO {

id	integer($int64)
serviceBroker_id	integer($int64)
instance_id	string
service_id	string
plan_id	string
binding_id	string

}

Credentials {

password	string
uri	string
username	string

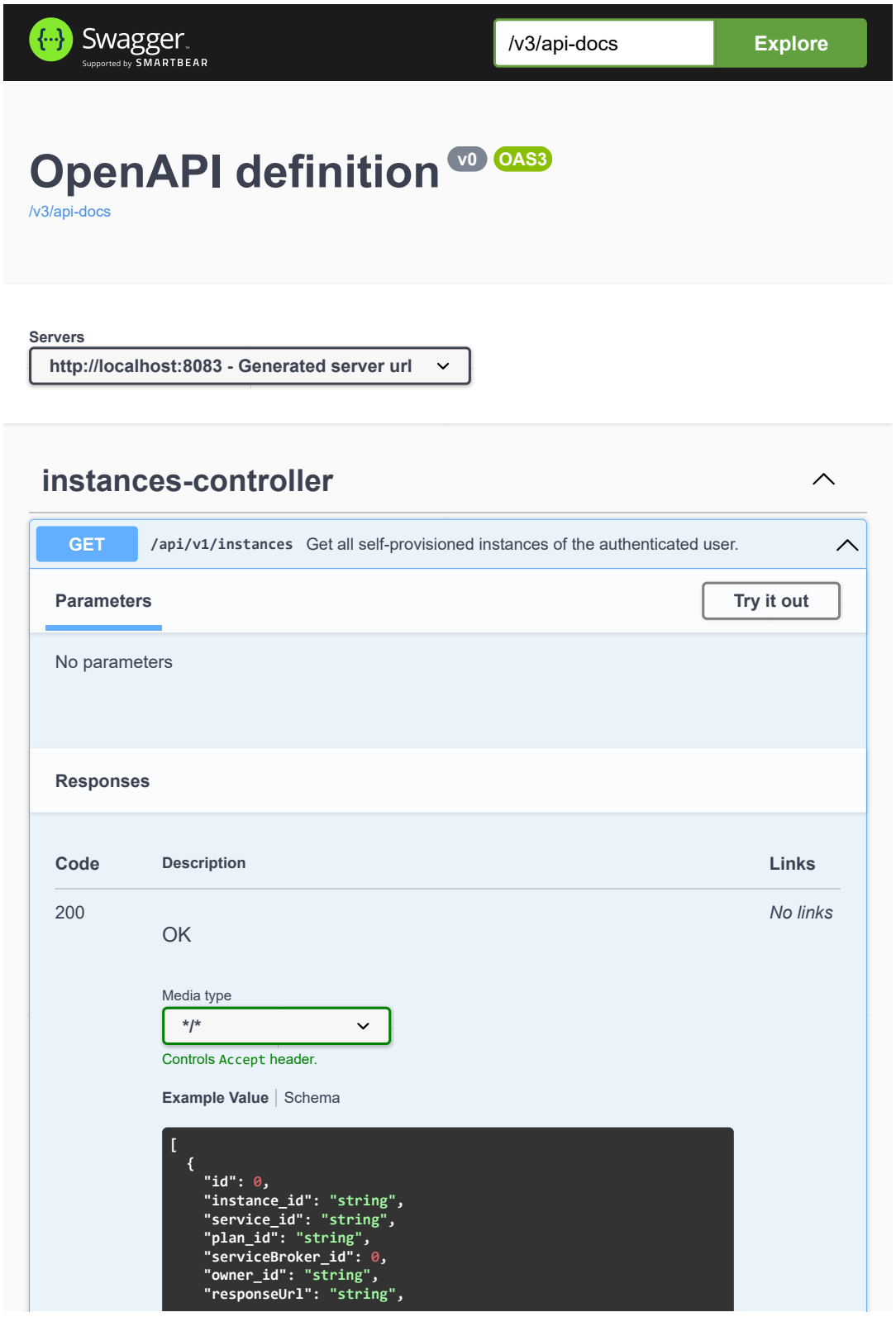
}

InstanceBindingResponseDTO {

credentials	**Credentials** {	
	password	string
	uri	string
	username	string
	}	
description	string	

}

Service Instanzen Service API



Code	Description	Links


```
    "createDate": "2022-08-07T13:02:20.799Z",
    "status": "ACTIVE",
    "enabled": true,
    "lastChanged": "2022-08-07T13:02:20.799Z"
  }
]
```

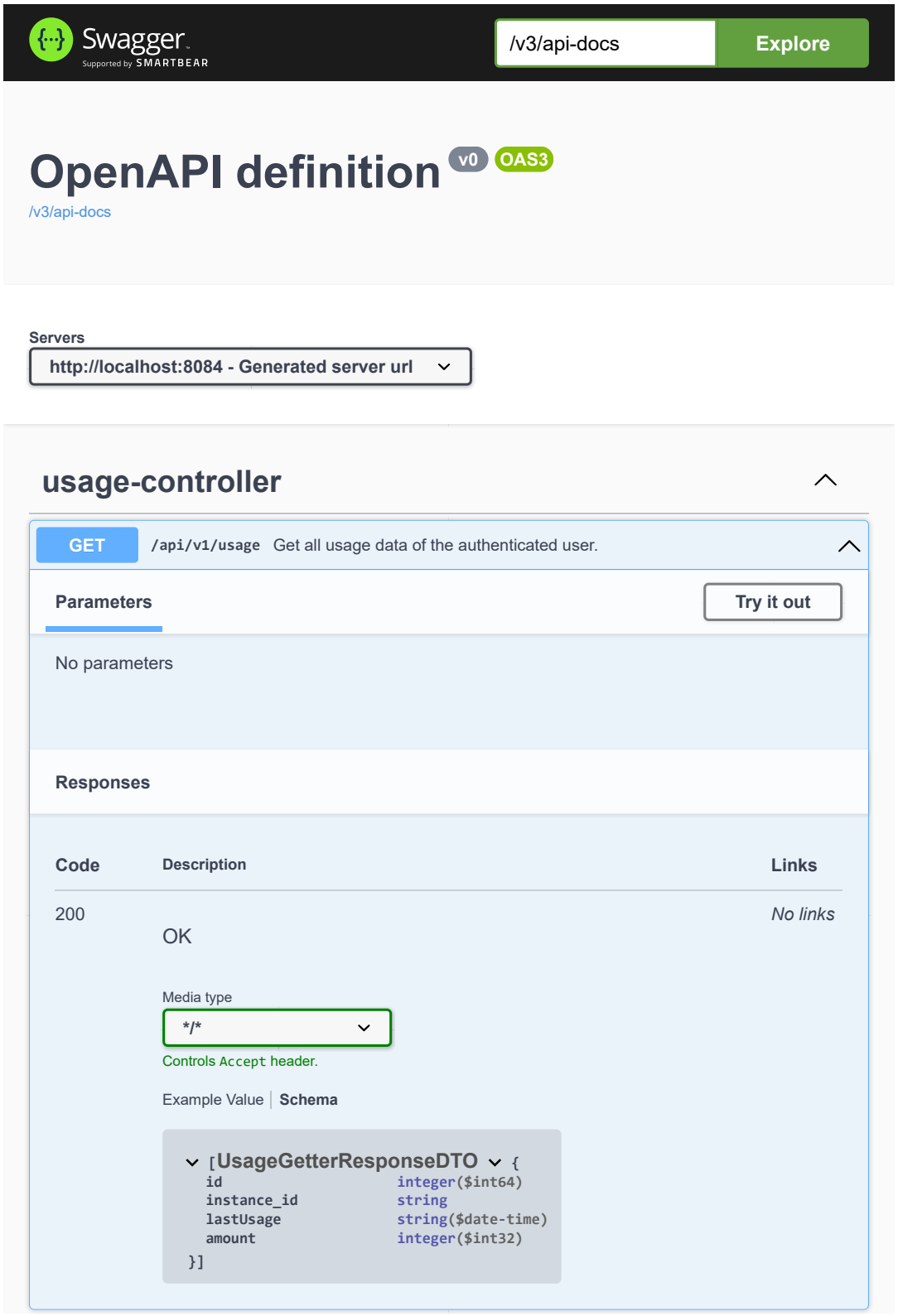

Schemas

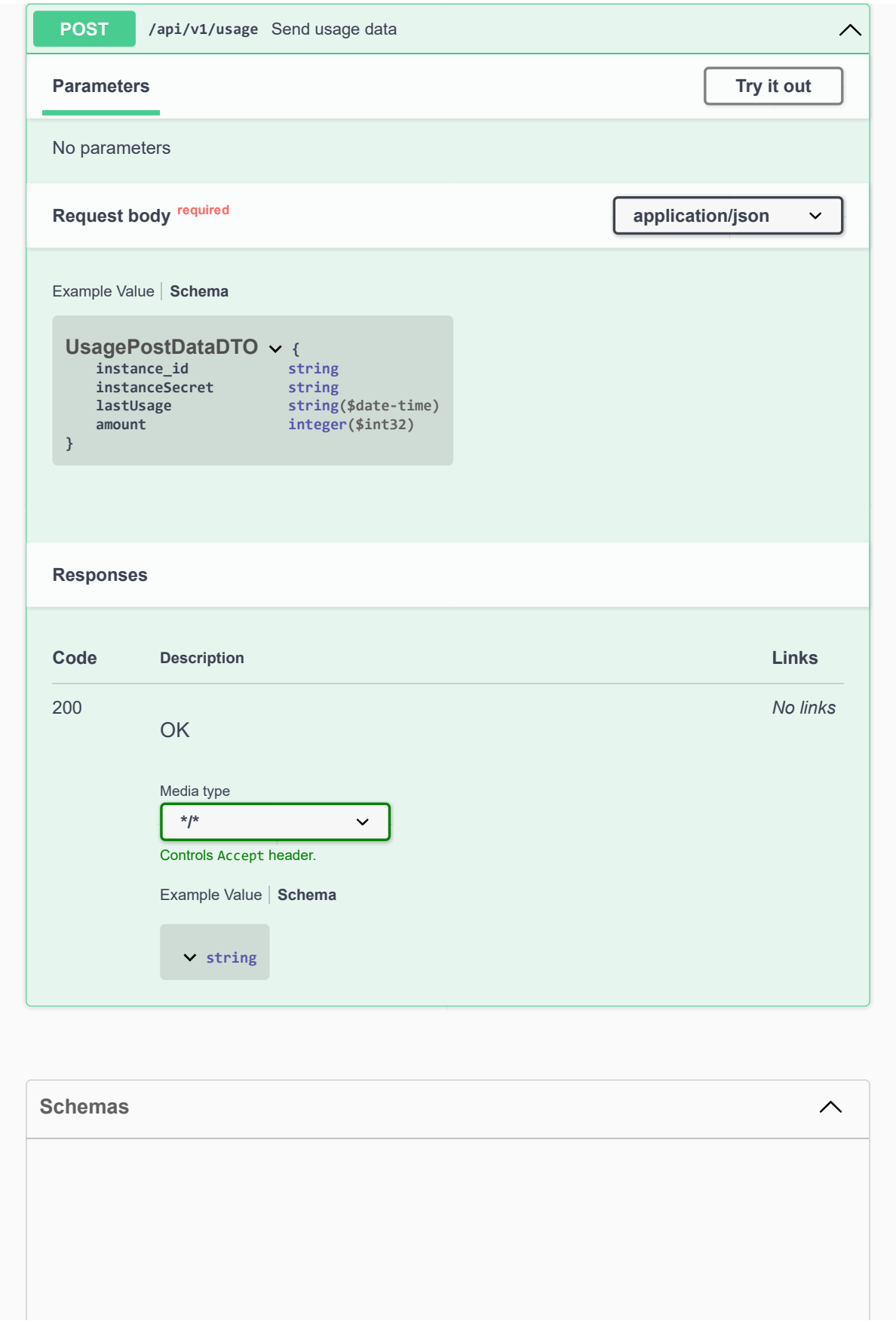
InstanceDbModel {

id	integer($int64)
instance_id*	string
service_id*	string
plan_id*	string
serviceBroker_id*	integer($int64)
owner_id*	string
responseUrl	string
createDate	string($date-time)
status	string Enum: [ACTIVE, FAILED, PROCESSING, DEPLOYING, DISABLED, DEPROVISIONING, DEPROVISIONED]
enabled	boolean
lastChanged	string($date-time)

}

Service Usage Service API




POST
/api/v1/usage
Send usage data
Parameters
Try it out
No parameters
Request body
required
application/json
Example Value
Schema
UsagePostDataDTO
{
instance_id
string
instanceSecret
string
lastUsage
string($date-time)
amount
integer($int32)
}
Responses
Code
Description
Links
200
OK
No links
Media type
/
Controls Accept header.
Example Value
Schema
string
Schemas

UsagePostDataDTO ⌄ {

```
instance_id          string
instanceSecret       string
lastUsage            string($date-time)
amount               integer($int32)
```

}

UsageGetterResponseDTO ⌄ {

```
id                   integer($int64)
instance_id          string
lastUsage            string($date-time)
amount               integer($int32)
```

}

9.6 Postman API-Tests

Aus dem beiliegenden Speichermedium lässt sich eine Exportdatei für die API-Tests entnehmen. Diese lässt sich in dem Tool **Postman** importieren. Postman ist eine API-Plattform für Entwickler zum Entwerfen, Erstellen, Testen und Iterieren ihrer APIs.
Im Folgenden befinden sich Ausschnitte der API-Tests.

Service Broker API-Tests

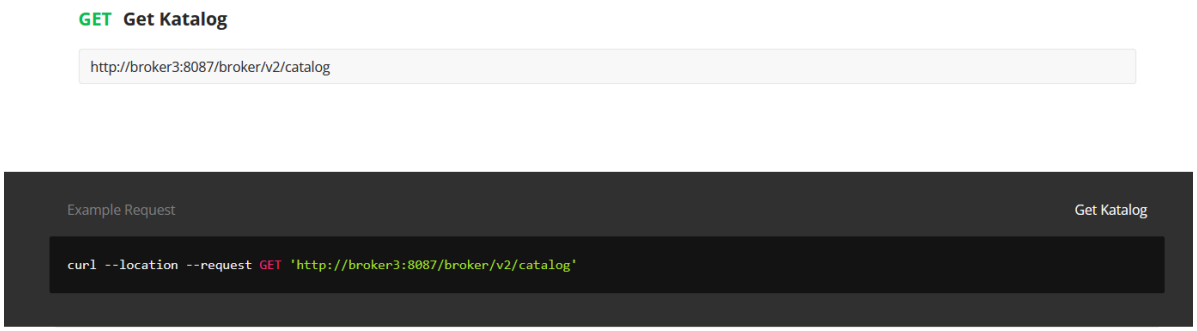


Abbildung 9.2: Service Broker API-Test: Get Katalog

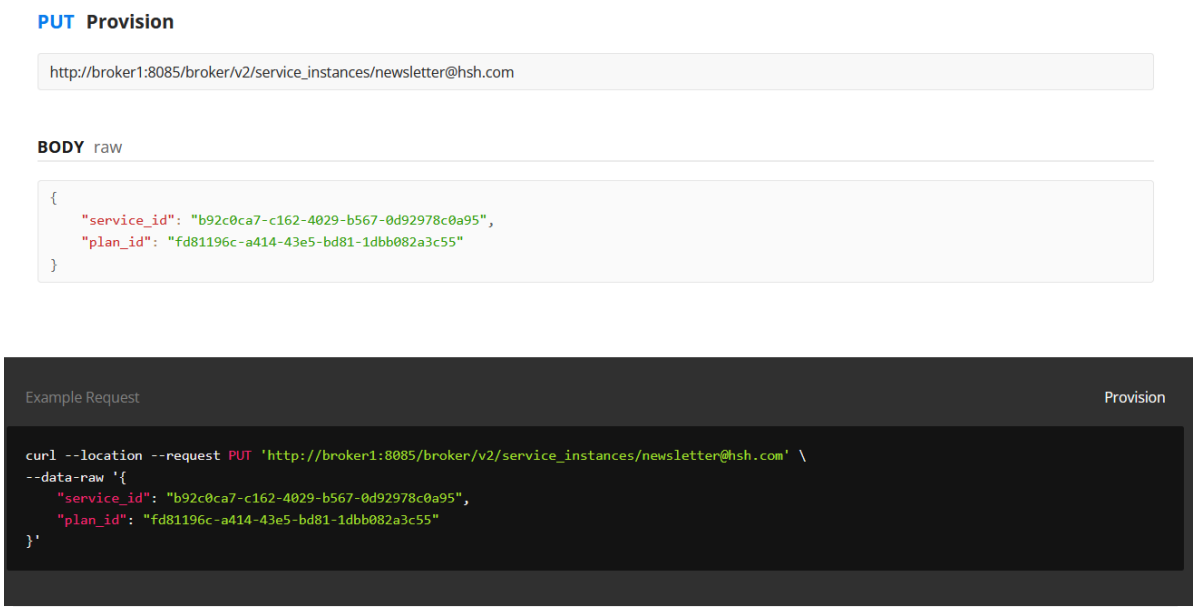


Abbildung 9.3: Service Broker API-Test: Provision

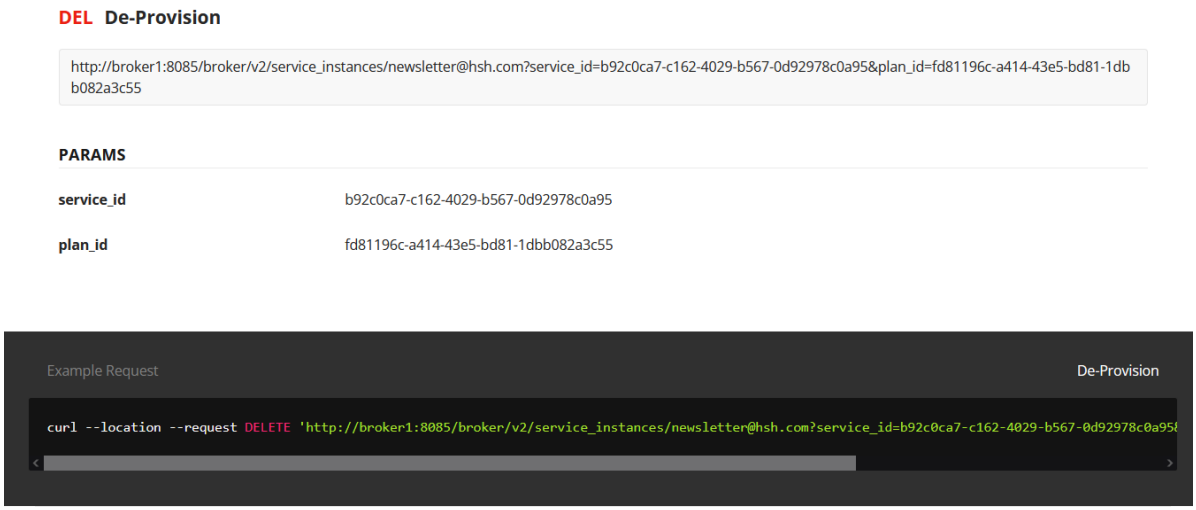


Abbildung 9.4: Service Broker API-Test: Deprovision

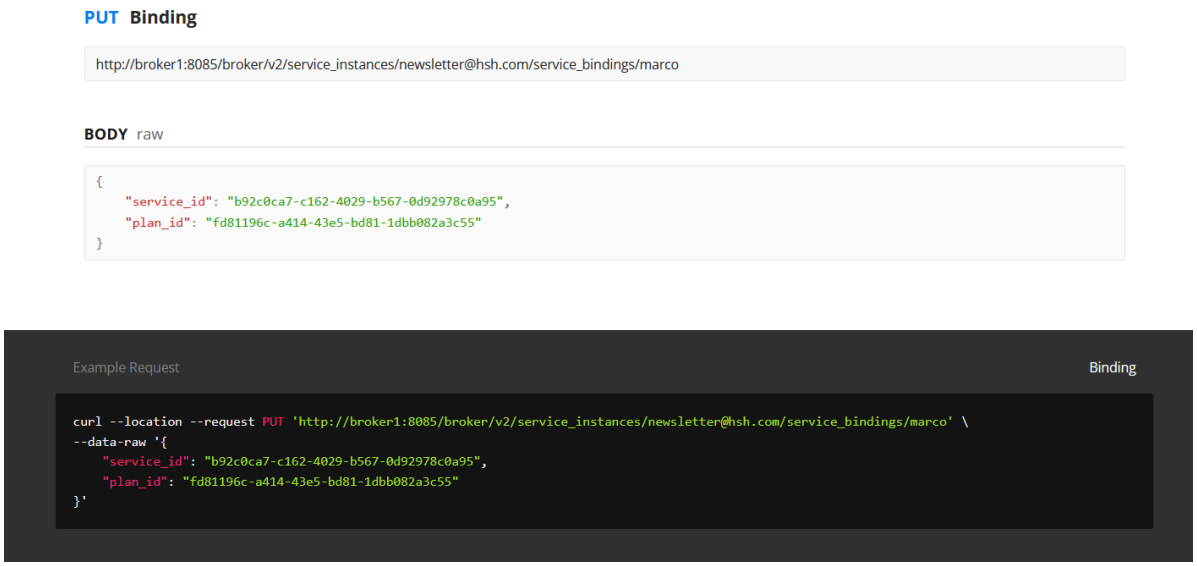


Abbildung 9.5: Service Broker API-Test: Binding

Service Registry API-Tests

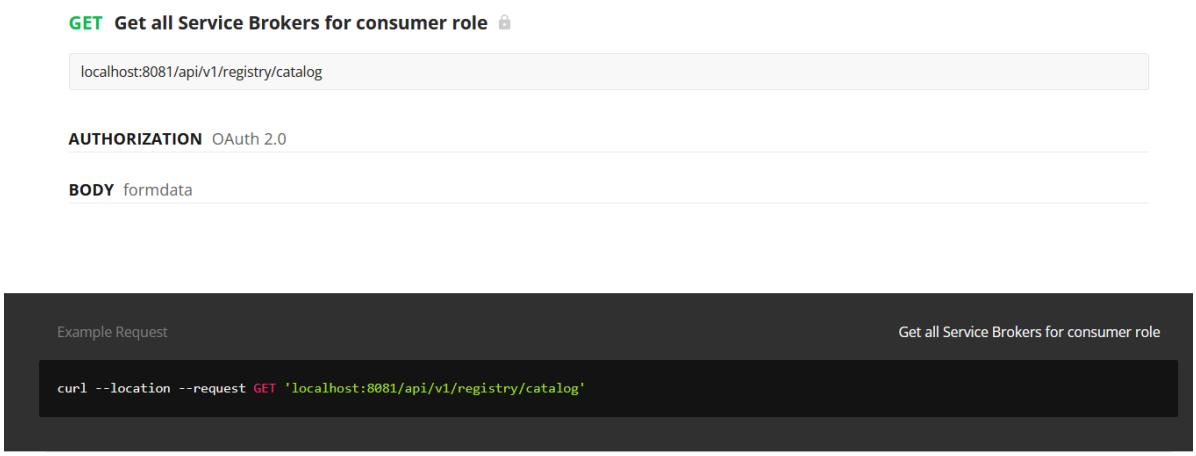


Abbildung 9.6: Service Registry API-Test: Get all Service Brokers

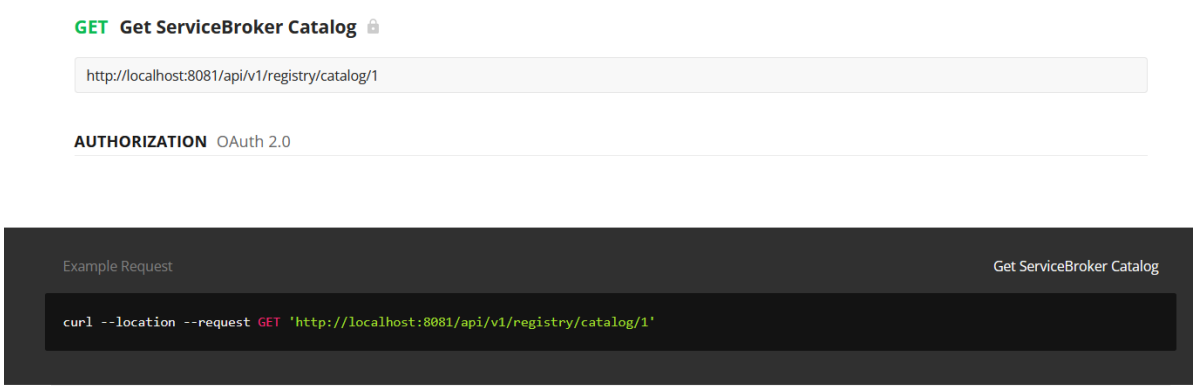


Abbildung 9.7: Service Registry API-Test: Get Service Broker Catalog

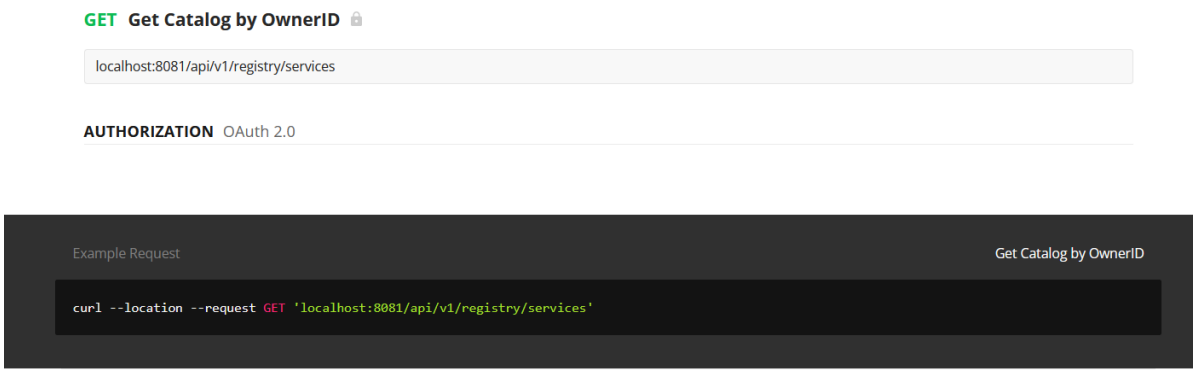


Abbildung 9.8: Service Registry API-Test: Get Catalog by OwnerID

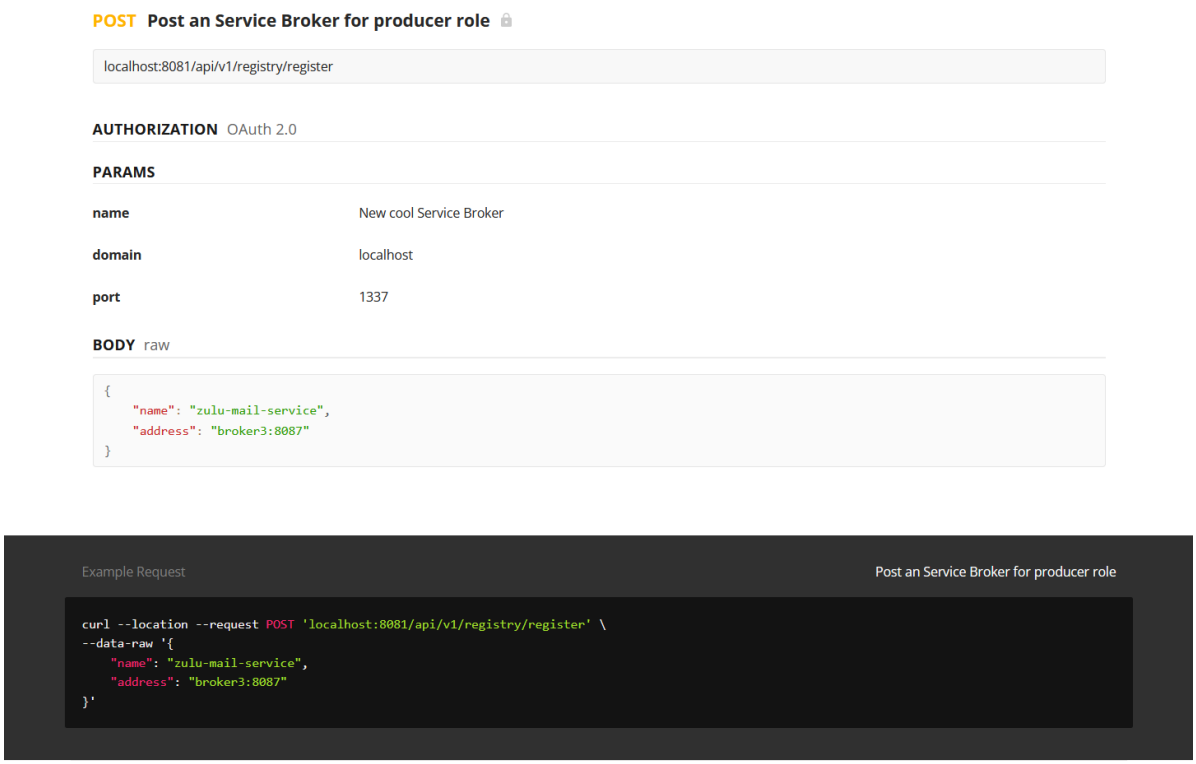


Abbildung 9.9: Service Registry API-Test: Post an Service Broker

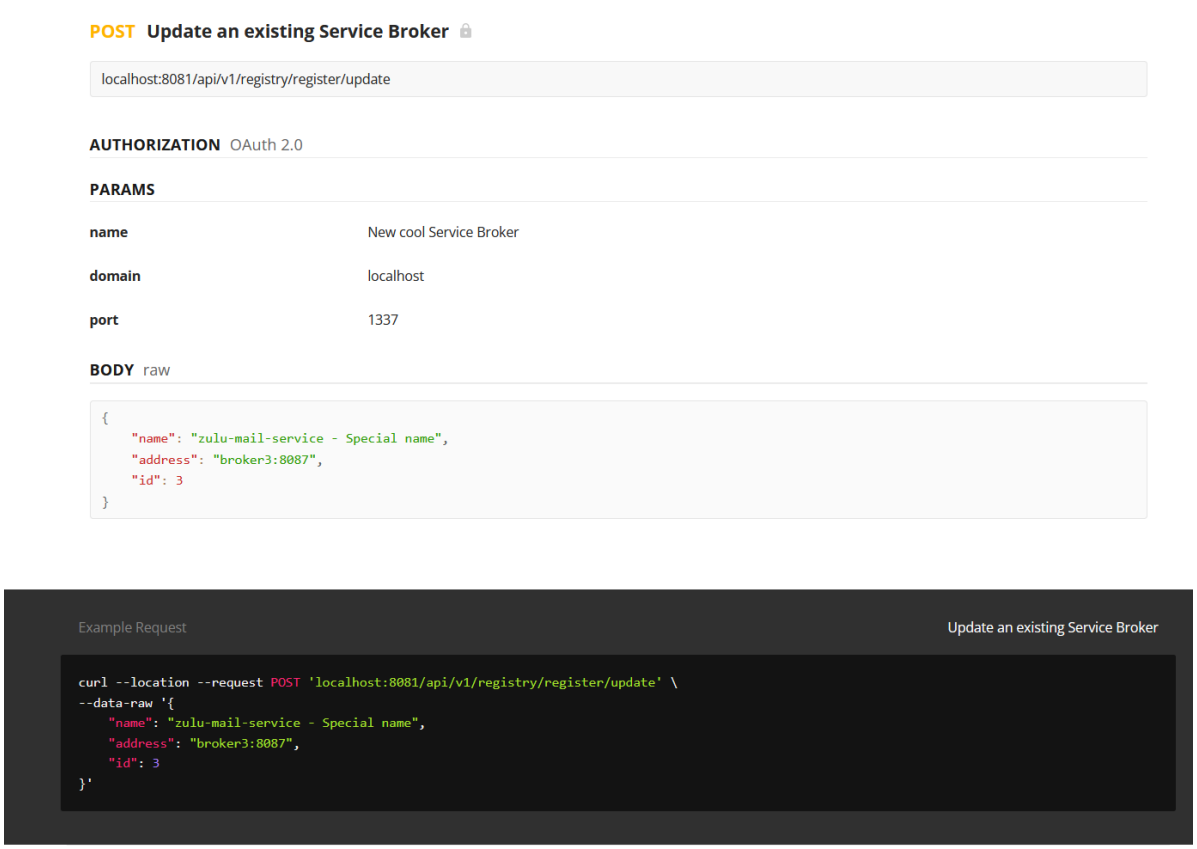


Abbildung 9.10: Service Registry API-Test: Update an existing Service Broker

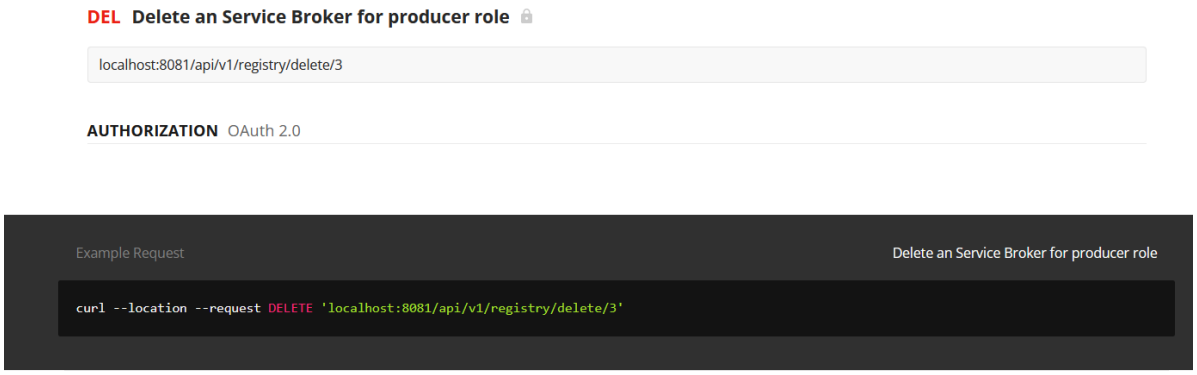


Abbildung 9.11: Service Registry API-Test: Delete an Service Broker

Abbildung 9.12: Service Registry API-Test: Activate an Service Broker

Service Provisioning Service API-Tests

PUT **Provision Service by Service Broker, Service Id and plan Id**

localhost:8082/api/v1/provisioning/service_instances

AUTHORIZATION OAuth 2.0

BODY raw


```
{
    "instance_id": "newsletter@hsh.com",
    "service_id": "b92c0ca7-c162-4029-b567-0d92978c0a95",
    "plan_id": "fd81196c-a414-43e5-bd81-1dbb082a3c55",
    "serviceBroker_id": 1
}
```


Example Request — Provision Service by Service Broker, Service Id and plan Id


```
curl --location --request PUT 'localhost:8082/api/v1/provisioning/service_instances' \
--data-raw '{
    "instance_id": "newsletter@hsh.com",
    "service_id": "b92c0ca7-c162-4029-b567-0d92978c0a95",
    "plan_id": "fd81196c-a414-43e5-bd81-1dbb082a3c55",
    "serviceBroker_id": 1
}'
```


Abbildung 9.13: Service Provisioning Service API-Test: Provision Service

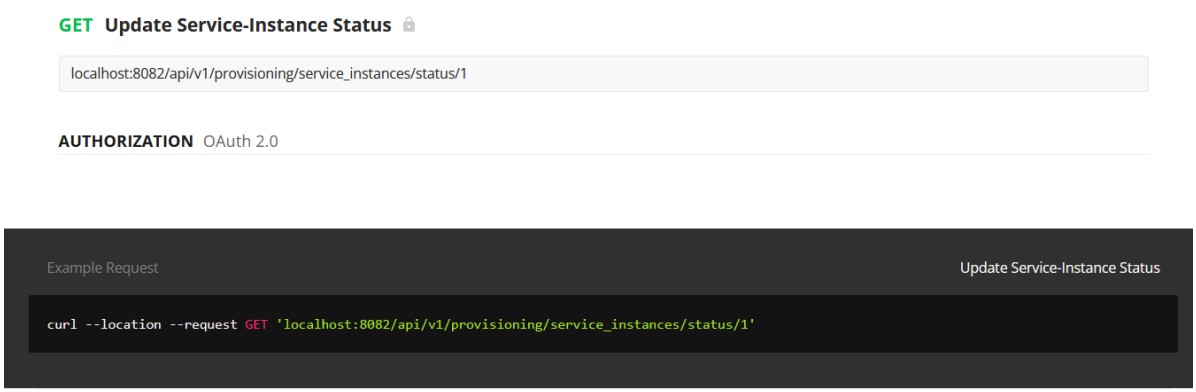


Abbildung 9.14: Service Provisioning Service API-Test: Update Service-Instance Status

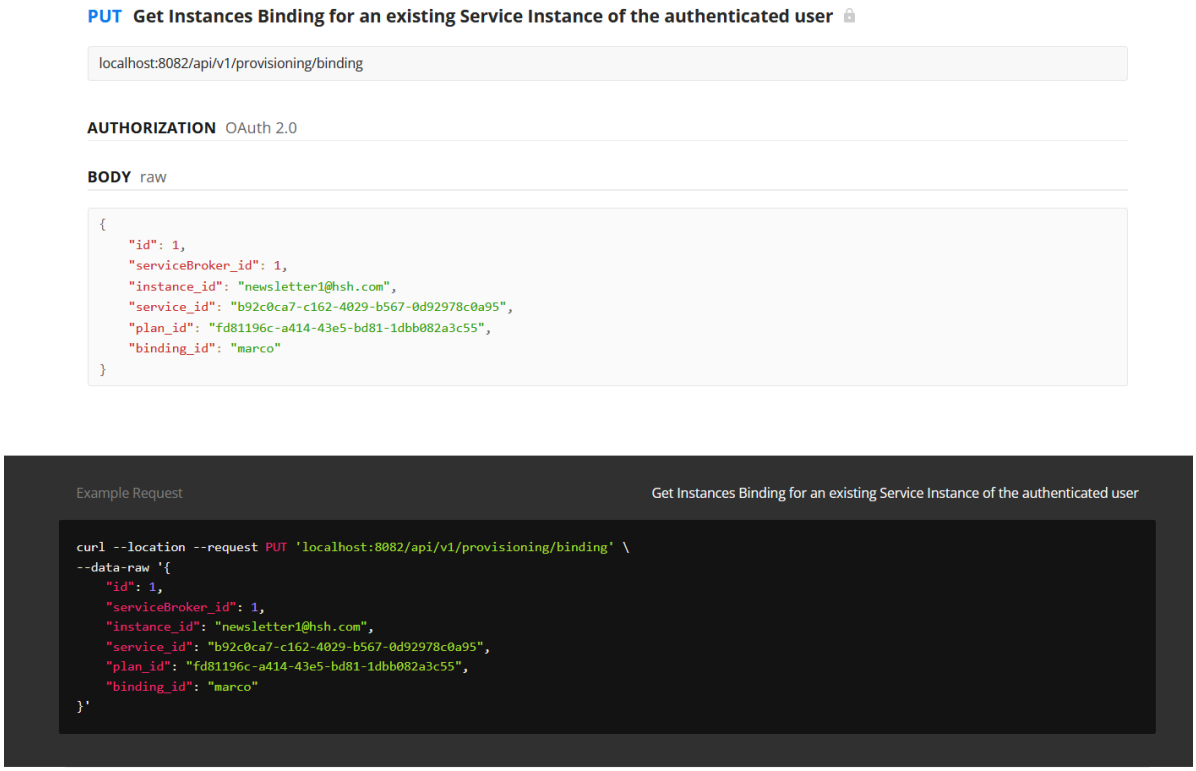


Abbildung 9.15: Service Provisioning Service API-Test: Get Instances-Binding

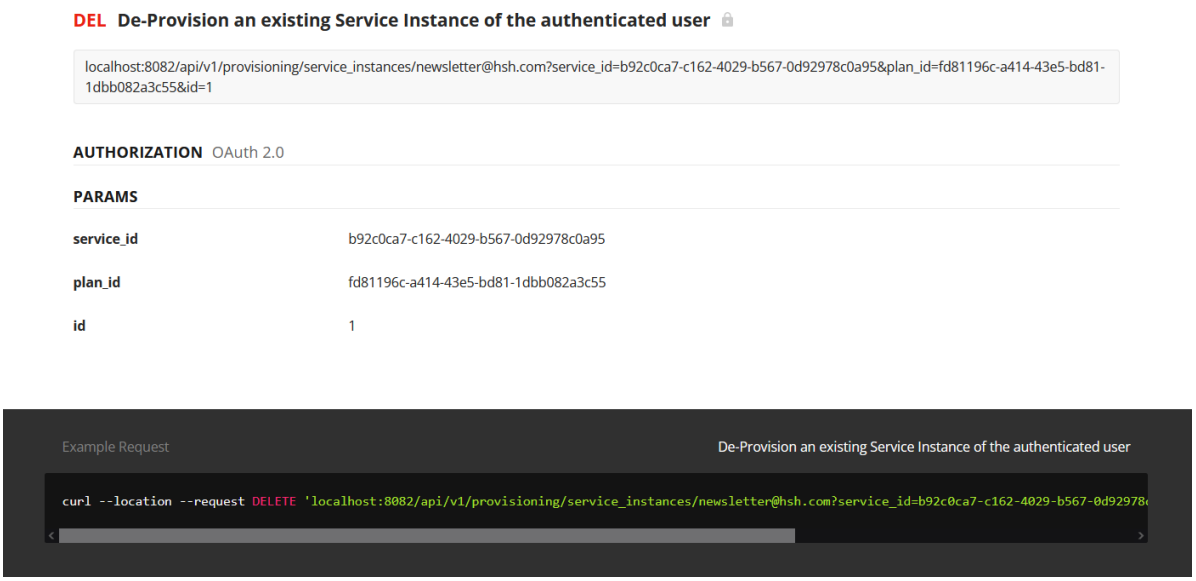


Abbildung 9.16: Service Provisioning Service API-Test: Deprovision an existing Service Instance

Service Instanzen Service API-Test

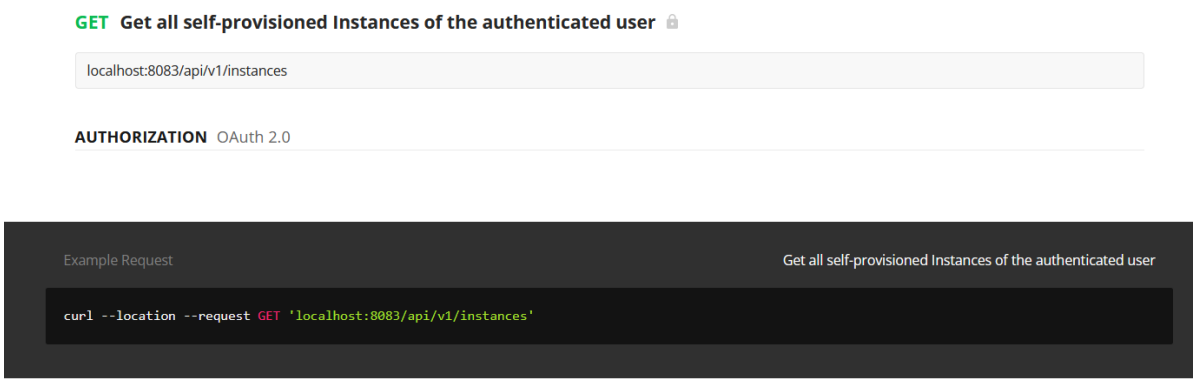


Abbildung 9.17: Service Instanzen Service API-Test: Get all self-provisioned Instances

Service Usage Service API-Tests

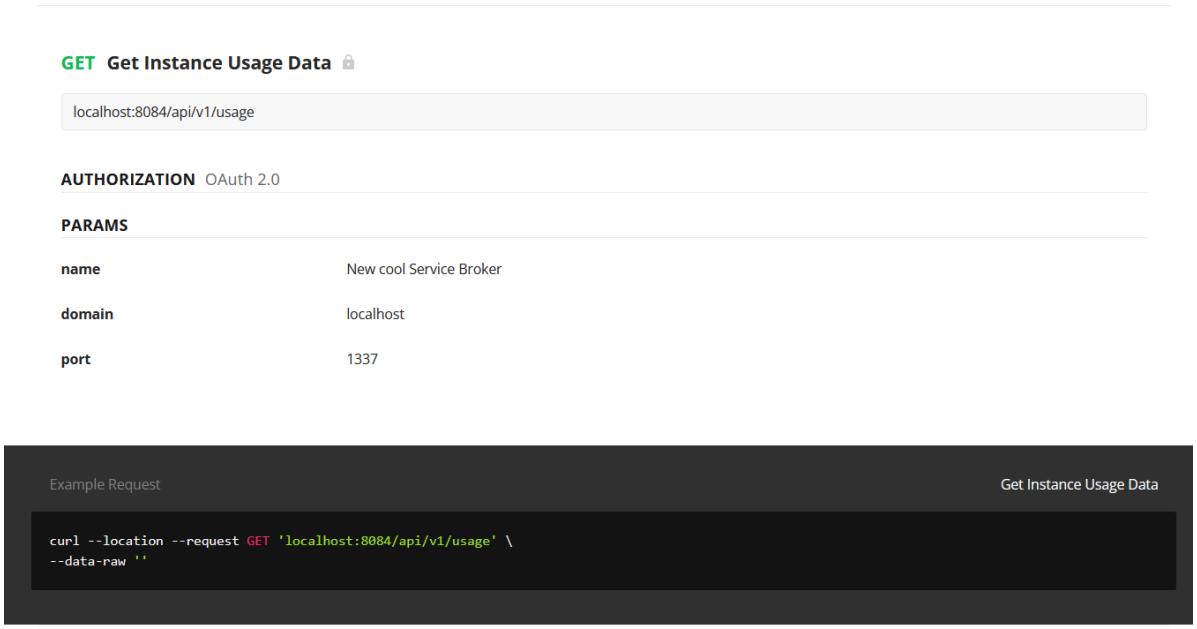


Abbildung 9.18: Service Usage Service API-Test: Get Instance Usage Data

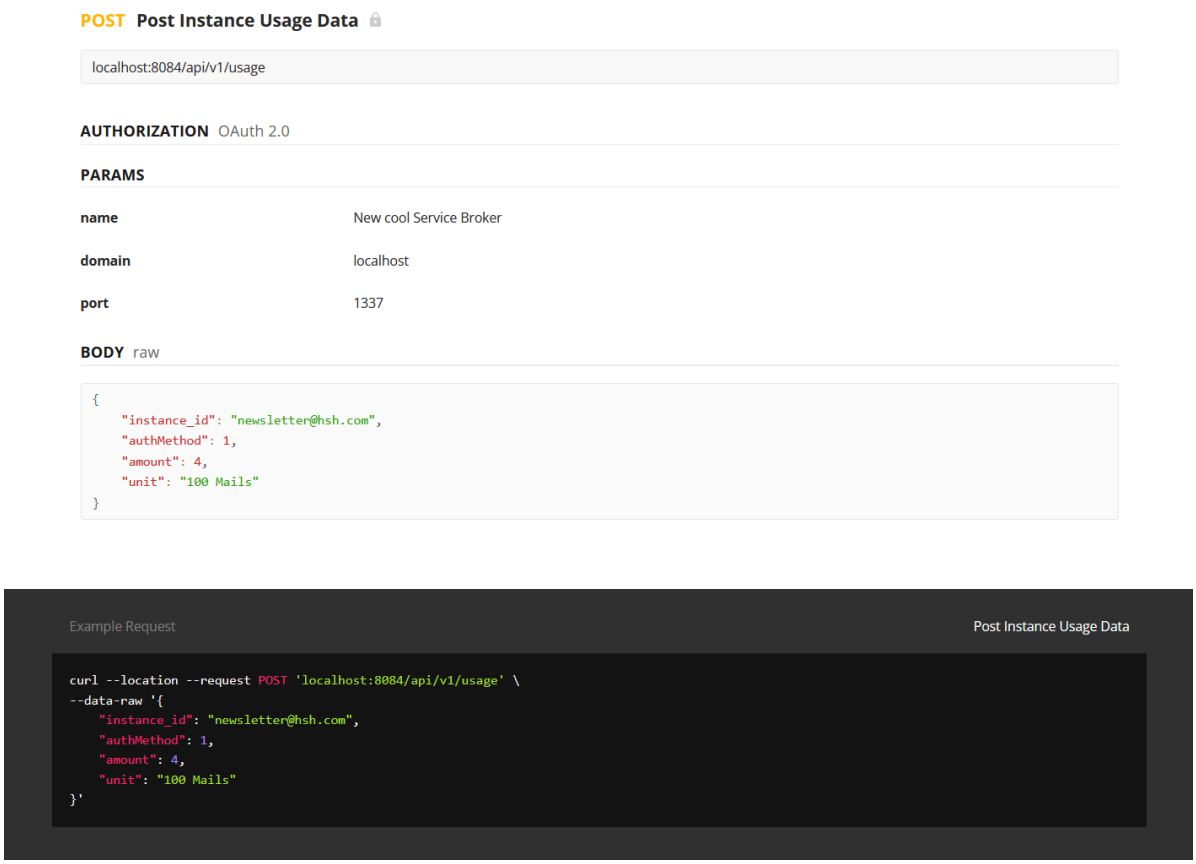


Abbildung 9.19: Service Usage Service API-Test: Post Instance Usage Data