Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Ereignisbasierte Architektur für Quantenanwendungen

Stefan Basaric

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Daniel Vietz, M.Sc.
Beginn am:	22. April 2021
Beendet am:	22. Oktober 2021

Kurzfassung

Im Vergleich zu herkömmlichen Rechnern können mithilfe von Quantencomputern zum ersten Mal komplexe Probleme mit akzeptablen Berechnungszeiten gelöst werden. Diese werden heutzutage durch eine Vielzahl von öffentlichen Cloud-Diensten wie IBM Quantum, Amazon Braket oder Azure Quantum registrierten Nutzern verfügbar gemacht. Um ihre Experimente auf Quantencomputern durchführen zu können, müssen Nutzer Quantenschaltungen schreiben und an die von den Cloud-Diensten bereitgestellten Schnittstellen schicken. Die Quantenschaltungen kommen dabei zunächst in eine Warteschlange, bevor sie schließlich auf dem Quantencomputer ausgeführt werden. Das hat zur Folge, dass die Ausführung im Vergleich zur reinen Berechnungszeit auf dem Quantencomputer sehr lange dauern kann. Die aktuell verfügbaren Cloud-Dienste bieten derzeit keine Möglichkeit, die Quantenanwendungen ihrer Nutzer zu hosten und sie beim Eintritt von Ereignissen automatisch auszuführen. In dieser Arbeit wird ein Konzept für eine ereignisbasierte Architektur vorgestellt, welches die automatisierte Ausführung von Quantenanwendungen beim Eintritt von beliebigen Ereignissen ermöglicht. Zusätzlich wird ein anhand des Konzepts umgesetzter Prototyp präsentiert, welcher mithilfe von IBM Quantum und OpenWhisk die ereignisbasierte Ausführung von Quantenanwendungen trotz einiger Limitationen ermöglicht.

Inhaltsverzeichnis

1 Einleitung **11**

2 Verwandte Arbeiten **13**

3 Grundlagen **15**
3.1 Function-as-Service 15
3.2 OpenWhisk 15
3.3 IBM Quantum 22

4 Konzept **25**
4.1 Verwaltung von FaaS-Diensten 25
4.2 Verwaltung von Quantenanwendungen 26
4.3 Emittieren von Ereignissen 28
4.4 Ausführung von Quantenanwendungen 30
4.5 Verwaltung von Ausführungsergebnissen 31
4.6 Aufbau von Quantenanwendungen 33

5 Implementierung **35**
5.1 Architekturübersicht 35
5.2 Genutzte Technologien 36
5.3 Struktur des QuantumService 37
5.4 Anbindung genutzter Plattformen 39
5.5 Integration von Quantenanwendungen 42
5.6 Einbindung von Ereignissen 50
5.7 Grafische Benutzeroberfläche 55

6 Diskussion **59**
6.1 Umsetzung der Problemstellung 59
6.2 Limitationen 60
6.3 Alternativkonzepte 61

7 Zusammanfassung und Fazit **65**

Literaturverzeichnis **67**

Abbildungsverzeichnis

3.1 Programmiermodell von OpenWhisk 16
3.2 Prinzip des Hook-Pattern 21
3.3 Prinzip des Polling-Pattern 21
3.4 Prinzip des Connections-Pattern 22

4.1 Skizze einer ereignisbasierten Architektur 25
4.2 Anwendungsfalldiagramm des Systems 26
4.3 Registrierung von FaaS-Diensten 27
4.4 Verwaltung von Quantenanwendungen 28
4.5 Funktionsweise der EventBridge 29
4.6 Integration von Ereignis-Feeds durch Polling 29
4.7 Integration von Ereignis-Feeds durch Push-Nachrichten 30
4.8 Ereignisbasierte Ausführung von Quantenanwendungen 31
4.9 Beschaffung der Ausführungs-ID 31
4.10 Beschaffung der Ausführungsergebnisse 32
4.11 Polling-Prozess für Ergebnisse von Quantencomputern 32

5.1 Ereignisbasierte OpenWhisk-Architektur 36
5.2 Klassendiagramm des QuantumService 38
5.3 Verwaltung des Access-Tokens 41
5.4 Anmeldungsprozess einer Quantenanwendung 45
5.5 Feed-Registrierung einer Quantenanwendung 46
5.6 Generierung von ScriptExecutions 47
5.7 Zyklus des ScriptExecutionChecker 49
5.8 Zyklus des JobChecker 50
5.9 Erstellung eines QueueSizeEventTrigger 52
5.10 Generierungsprozess von Warteschlangenereignissen 53
5.11 Sidebar-Menü 56
5.12 Tabellenansicht der angemeldeten Quantenanwendungen 56
5.13 Detailansicht einer angemeldeten Quantenanwendungen 57

Verzeichnis der Listings

3.1 Beispiel einer Action 17
3.2 Beispiel eines Trigger 18
3.3 Beispiel einer Rule 19
3.4 Beispiel einer Activation 19

5.1 Initiale IBMQ-Properties 40
5.2 Beispiel eines OpenWhiskService-Body 42
5.3 Funktion einer beispielhaften Quantenanwendung 42
5.4 Beispiel der initial erstellten ScriptExecution 48
5.5 Beispiel eines initial erstellten Job-Objekts 49
5.6 Beispiel eines EventTrigger-Bodys 51
5.7 Ereignisobjekt eines beispielhaften Warteschlangenereignisses 54
5.8 Ereignisobjekt eines beispielaften Ausführungsergebnis-Ereignisses 55

1 Einleitung

Mit der Entdeckung der Quantenmechanik eröffneten sich neue Möglichkeiten im Bereich des Computing. Der Grundstein des auf ihr basierenden Quantum Computing wurde bereits in den 80er Jahren gelegt, als ein auf der Quantenmechanik basiertes Modell einer Turingmaschine vorgeschlagen wurde [Ben80]. Kurze Zeit später wurde das Potenzial von Quantencomputern erkannt, spezielle Probleme deutlich schneller als traditionelle Rechner zu lösen [Fey82].

Heutzutage haben alle gängigen Cloud-Anbieter wie Microsoft, Amazon oder IBM ihr Angebot um auf Quantum Computing spezialisierte Dienste wie IBM Quantum[1], Amazon Braket[2] oder Azure Quantum[3] erweitert. Sie bieten registrierten Nutzern den Zugriff zu verschiedensten Quantencomputern. Um mit diesen zu interagieren, können Nutzer verschiedene QC-spezifische Programmiersprachen und SDKs wie Qiskit[4], OpenQasm[5], D-Wave-Ocean[6] oder Microsoft QDK[7] verwenden, um Quantenschaltungen zu generieren und sie für die Ausführung auf einem Quantencomputer an die Schnittstellen des jeweiligen Cloud-Dienstes zu schicken.

Damit die einzelnen, öffentlich zugänglichen Quantencomputer die Ausführungsanfragen aller Nutzer bearbeiten können, setzen Cloud-Anbieter Warteschlangen ein. Diese speichern alle ankommenden Ausführungsanfragen in der Reihenfolge ihrer Ankunft, damit sie vom jeweiligen Quantencomputer eine nach der anderen abgearbeitet werden können. Während die Nutzung von Warteschlangen die Bedienung von Tausenden parallel ankommenden Ausführungsanfragen ermöglicht, führt sie ebenfalls zu unausweichlichen Wartezeiten, welche im Vergleich zur eigentlichen Rechenzeit auf dem Quantencomputer sehr groß ausfallen können. In Systemen, bei denen mehrere Quantenanwendungen miteinander kommunizieren, ist das besonders problematisch, da die Quantenanwendungen ständig auf die Ergebnisse anderer aktiv warten müssen. In solchen Fällen würde sich die ereignisbasierte Ausführung der einzelnen Quantenanwendungen anbieten. Zum aktuellen Zeitpunkt bietet jedoch keiner der verfügbaren Cloud-Anbieter seinen Nutzern die Möglichkeit, ihre Quantenanwendungen zu hosten und bei beliebigen Ereignis-Feeds zu registrieren, damit sie beim Eintritt eines Ereignisses automatisch ausgeführt werden.

Um dieses Problem zu adressieren, wird im Rahmen dieser Masterarbeit ein Konzept für eine Systemarchitektur vorgestellt, welche Quantenanwendungen beim Eintritt von Ereignissen automatisch ausführt. Dafür ermöglicht das System seinen Nutzern, ihre Quantenanwendungen über Function-as-a-Service bereitzustellen und sie bei verfügbaren Ereignis-Feeds zu registrieren. Da in den meisten Fällen nicht alle Ereignisse eines Feeds interessant sind, können Nutzer die Feeds ihrer

[1] https://quantum-computing.ibm.com/
[2] https://aws.amazon.com/de/braket/
[3] https://azure.microsoft.com/de-de/services/quantum/
[4] https://qiskit.org/
[5] https://github.com/Qiskit/openqasm
[6] https://docs.ocean.dwavesys.com/en/stable/
[7] https://azure.microsoft.com/de-de/resources/development-kit/quantum-computing/

Quantenanwendungen individuell konfigurieren. Für die Integration der Ereignis-Feeds werden externe Ereignisquellen ins System eingebunden, welche intern auftretende Ereignisse emittieren. Beim Eintritt eines Ereignisses nutzt das System seine Inhalte, um Eingabeparameter zu generieren mit welchen alle passenden Quantenanwendungen ausgeführt werden. Für die Validierung des Konzepts kommt ein Prototyp zum Einsatz, welcher IBM Quantum als Ereignisquelle nutzt, um über OpenWhisk[8] bereitgestellte Quantenanwendungen auf IBMs Quantencomputern auszuführen.

Der Rest dieser Arbeit ist wie folgt aufgebaut: In Kapitel 2 werden Arbeiten vorgestellt, in denen sich mit verwandten Themen auseinandergesetzt wurde. Das darauf folgende Kapitel 3 beschreibt Plattformen, welche für die Implementierung des in dieser Arbeit entwickelten Prototyps genutzt wurden. In Kapitel 4 wird ein Konzept für die Umsetzung einer ereignisbasierten Architektur für die Ausführung von Quantenanwendungen vorgestellt. Im Anschluss wird in Kapitel 5 die Implementierung eines Prototyps präsentiert, welcher die ereignisbasierte Ausführung von Quantenanwendungen mittels IBM Quantum und OpenWhisk ermöglicht. Zum Schluss werden in Kapitel 6 das entwickelte Konzept sowie der damit umgesetzte Prototyp diskutiert, bevor in Kapitel 7 die wichtigsten Punkte der Arbeit zusammengefasst werden und ein kurzer Ausblick auf künftige Arbeiten gegeben wird.

[8] https://openwhisk.apache.org/

2 Verwandte Arbeiten

Durch die steigende Popularität des Quantum Computing und der immer leistungsfähiger werdenden Quantencomputer wächst der Drang, diese in bereits existierende Systeme für die Berechnung komplexer Aufgaben zu integrieren.

Zu diesem Zweck stellte J. Rojo in seiner Arbeit [RVB+21] einen *Quantum-Microservice* vor, welcher die Vorteile des Quantum Computing über öffentlich erreichbare Schnittstellen anderen Services zur Verfügung stellt. Dieser Quantum-Microservice ist in Python[1] und Flusk[2] geschrieben. Im Vergleich zu üblichen Microservices verwendet er jedoch spezielle Bibliotheken und SDKs, die es ihm ermöglichen, mit Quantencomputern von Amazon-Braket[3] zu kommunizieren, um das Problem des Handlungsreisenden[4] effizient zu lösen. Hierfür bietet der Quantum-Microservice Endpunkte an, welche mit verschiedenen URL-Parametern aufgerufen werden können. Das ermöglicht externen Services den vom Quantum-Microservice implementierten Quantenalgorithmus mit individuellen Eingabeparametern auszuführen und zugleich den Quantencomputer zu wählen, auf welchem dieser ausgeführt werden soll.

Im Vergleich dazu hat M.Grossi in seiner Arbeit [GCA+21] ein Framework vorgestellt, in welchem die Quantenanwendungen nicht als Microservices bereitgestellt werden, sondern als Funktionen über eine Function-as-a-Service Plattform. Diese ist ein Teil einer größeren Architektur von mehreren Komponenten. In ihr kommt ein Frontend zum Einsatz, welches Nutzern über ein Backend ermöglicht, die bereitgestellten und als Funktionen geschriebenen Quantenanwendungen mit beliebigen Eingabeparametern auszuführen. Diese Funktionen sind dafür zuständig, die beim Aufruf übergebenen Eingabeparameter zu nutzen, um eine Quantenschaltung zu generieren und für die Ausführung an einen von IBM zur Verfügung gestellten Quantencomputer zu schicken, bevor sie die in der Antwort enthaltene Referenz der langlaufenden Ausführung über Apache Kafka[5] an ein Backend weiterleiten, welches diese zum Abfragen des Ergebnisses nutzt. Sobald das Ergebnis des Quantencomputers durch wiederholtes Abfragen empfangen wurde, wird es über Apache Kafka an das mit dem Frontend verbundene Backend weitergeleitet, damit das Ergebnis dem Nutzer präsentiert werden kann.

Mit einer Lösung, welche die Bereitstellung und Ausführung von Quantenanwendungen vereinfachen soll, beschäftigt sich auch der Cloud-Anbieter IBM. Dieser entwickelt hierfür eine hauseigene Architektur, welche als *Qiskit Runtime* bezeichnet wird [IBMe][IBMf]. Über eine REST-API [IBMg] ermöglicht die Qiskit Runtime den bei IBM Quantum registrierten Nutzern ihre mit Qiskit

[1] https://www.python.org/
[2] https://flask.palletsprojects.com/en/2.0.x/
[3] https://aws.amazon.com/de/braket/
[4] https://qiskit.org/documentation/tutorials/optimization/6_examples_max_cut_and_tsp.html#Traveling-Salesman-Problem
[5] https://kafka.apache.org/

geschriebenen Quantenanwendungen bereitzustellen, zu verwalten und auf IBMs Quantencomputern auszuführen. Da sich die Qiskit Runtime aktuell im Beta-Stadium befindet, ist sowohl ihr Funktionsumfang als auch ihre API nicht endgültig festgelegt. In der aktuellen Version kann die API für die Durchführung der gängigen CRUD-Operationen auf Quantenanwendungen genutzt werden. Dazu gehört die durch Eingabeparameter individualisierte Ausführung der bereitgestellten Quantenanwendungen.

Die vorgestellten Arbeiten bieten unterschiedliche Lösungen für die Bereitstellung und Ausführung von Quantenanwendungen, welche mittels öffentlich zugänglicher Schnittstellen in andere Systeme integriert werden können. Der Beitrag dieser Arbeit ist die Erweiterung der bereits umgesetzten Funktionalitäten um eine ereignisbasierte Ausführung. Dabei sollen die bereitgestellten Quantenanwendungen bei Ereignis-Feeds von beliebigen Ereignisquellen registriert werden können, um beim Eintritt entsprechender Ereignisse automatisch mit den im Ereignis enthaltenen Informationen ausgeführt zu werden.

3 Grundlagen

Dieses Kapitel beinhaltet die Beschreibung und Funktionsweise aller relevanten Plattformen, welche für die Implementierung des in dieser Arbeit entwickelten Prototyps verwendet wurden.

3.1 Function-as-Service

Mit Functions-as-a-Service (FaaS) werden spezielle Typen von Cloud-Diensten beschrieben, welche Entwicklern eine Plattform für die Bereitstellung kleiner, zustandsloser Funktionen bieten [IBMc]. Die Konfiguration und Wartung der darunterliegenden Infrastruktur wird dabei vom Cloud-Anbieter übernommen, sodass sich Entwickler voll und ganz auf das Schreiben ihres Codes konzentrieren können. Im Vergleich zu anderen Cloud-Angeboten wie Platform-as-a-Service werden über FaaS bereitgestellte Funktionen nicht dauerhaft ausgeführt. Stattdessen wird ihre Ausführung durch den Eintritt von beliebigen Ereignissen, wie beispielsweise einer Nachricht in einer Messaging-Warteschlange oder eines neuen Eintrags in einer Datenbanktabelle ausgelöst. Die ausgeführten Funktionen liefern Ergebnisse, welche entweder von externen Systemen direkt verarbeitet oder als Eingabe für andere über FaaS bereitgestellte Funktionen genutzt werden können.

3.2 OpenWhisk

OpenWhisk ist eine quelloffene, unter der Apache-Lizenz entwickelte FaaS-Plattform für die ereignisbasierte Ausführung von Codeschnipseln, welche in Form von kleinen Funktionen bereitgestellt werden [Apaf]. Die Funktionen werden dabei in beliebigen Programmiersprachen geschrieben und bei beliebigen externen Ereignis-Feeds registriert, damit beim Eintritt eines Ereignisses deren Ausführung automatisch ausgelöst werden kann. Um OpenWhisk ins eigene System zu integrieren, muss zunächst ein OpenWhisk-Server eingerichtet werden. Hierfür stehen den Entwicklern unterschiedliche Möglichkeiten zur Verfügung [Apae]. So lässt sich OpenWhisk als ein eigenständiger Server starten, welcher als eine Java-Anwendung ausgeführt wird. Zusätzlich kann OpenWhisk auf einem Kubernetes Cluster brereitgestellt werden [Apai]. Neben der selbstständigen Bereitstellung gibt es auch öffentliche Cloud-Anbieter, welche OpenWhisk als Lösung ihres FaaS-Angebots verwenden. Ein Beispiel dafür sind die IBM Cloud Functions[1], welche intern auf OpenWhisk setzen [IBMb], um ihren Nutzern zu ermöglichen, eigene Funktionen zu verwalten und auszuführen.

[1] https://cloud.ibm.com/functions/

3.2.1 Funktionsweise

Das von OpenWhisk umgesetzte Programmiermodell ist in Abbildung 3.1 dargestellt [Apaf]. Mithilfe von OpenWhisk können Entwickler ihre Funktionen als sogenannte *Actions* bereitstellen, welche zunächst keine Abhängigkeiten zu Ereignissen besitzen [Apaa]. Für die Einbindung von Ereignissen müssen sogenannte *Trigger* erstellt werden, welche von externen Ereignisquellen explizit aufgerufen werden müssen, sobald ein entsprechendes Ereignis eintritt [Apaj]. Die Verknüpfung zwischen einem Trigger *T* und einer Action *A* wird durch die Erstellung einer *Rule* aufgestellt, welche bei jedem Aufruf von T dafür sorgt, dass A ausgeführt wird. So können Actions durch mehrere Trigger ausgeführt werden, während ein Trigger mehrere Actions ausführen kann. Durch den Einsatz von Rules, Actions und Triggern gibt es eine lose Kopplung zwischen den Funktionen und Ereignissen, da diese unabhängig voneinander existieren können. Welche Funktionen beim Eintritt von welchen Ereignissen ausgeführt werden müssen, wird erst zur Laufzeit ermittelt. Nach dem Aufruf eines Triggers oder nach einer abgeschlossenen Ausführung einer Action speichert OpenWhisk eine sogenannte *Activation* in einer von OpenWhisk verwalteten Datenbank. Diese enthält alle nötigen Informationen über den Aufruf eines Triggers bzw. die Ausführung einer Action. Alle bisher genannten OpenWhisk-Entitäten werden jeweils einem sogenannten *Namespace* zugeordnet [Apah].

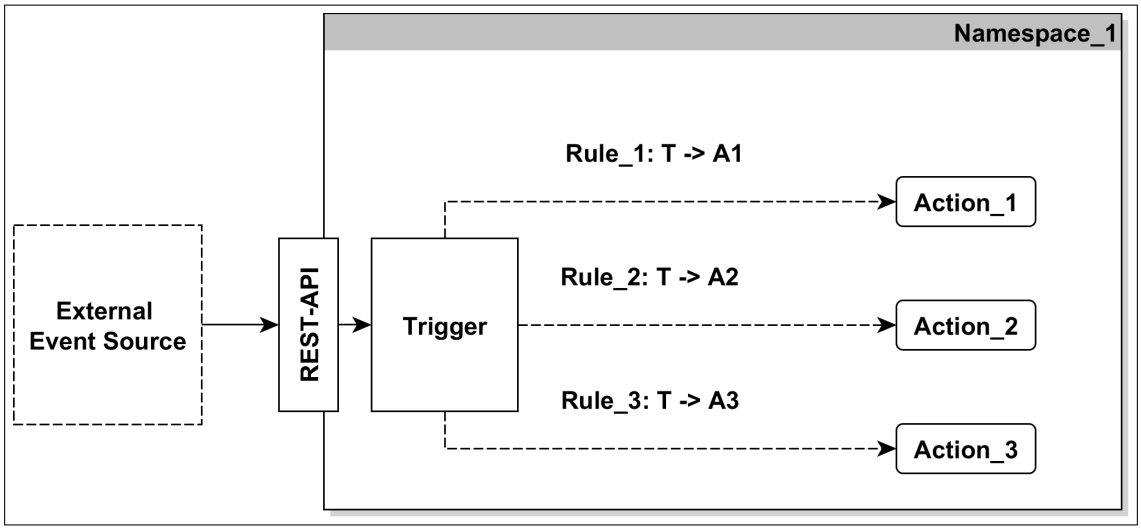

Abbildung 3.1: Programmiermodell von OpenWhisk

Die Verwaltung aller OpenWhisk-Entitäten wird durch eine umfangreiche CLI und REST-API ermöglicht. Diese können genutzt werden, um die Durchführung aller Grundoperationen auf allen Objekten durchzuführen [Apaf][Apak]. So können beispielsweise jederzeit und von jedem beliebigen Rechner aus Actions, Rules und Trigger erstellt, bearbeitet und gelöscht werden. In den folgenden Abschnitten werden die bereits erwähnten OpenWhisk-Entitäten im Detail beschrieben.

```
{
    "name": "MyAction",
    "namespace": "MyNamespace",
    "exec": {
        "code": "base64EncodedFunction"
        "kind": "blackbox",
        "image": "urlToDockerImage"
    }
}
```

Listing 3.1: Beispiel einer Action

3.2.2 Namespace

OpenWhisk verwendet für die Gruppierung aller Objekte sogenannte Namespaces [Apah]. Jedes von OpenWhisk verwaltete Objekt gehört immer zu genau einem Namespace. Nutzer einer OpenWhisk-Plattform können sich beliebig viele Namespaces erstellen und mit einer Basic-Autorisierung absichern, um unerwünschten Zugriff auf deren Inhalte zu verhindern. Dabei besitzt jeder Namespace einen eindeutigen Namen, mit welchem er identifiziert werden kann. Jedes von OpenWhisk verwaltete Objekt kann mittels seines Namens innerhalb eines Namespaces eindeutig bestimmt werden. Dadurch hat jedes Objekt eine eindeutige URL, welche mittels HTTP-Anfragen für die Durchführung der gängigen CRUD-Operationen aufgerufen werden kann [Apak].

3.2.3 Action

Actions repräsentieren die über OpenWhisk bereitgestellten Funktionen [Apaa]. Sie setzen sich aus dem auszuführenden, als Base64-String verschlüsselten Quellcode und weiteren Metadaten, welche für die Ausführung des Quellcodes relevant sind, zusammen. In Listing 3.1 ist eine mit den wichtigsten Daten befüllte Action zu sehen. Zu diesen gehören beispielsweise der Action-Name, der Namespace-Name und die zu verwendende Laufzeitumgebung. OpenWhisk kann standardmäßig auf eine Vielzahl an Laufzeitumgebungen zugreifen, welche die Ausführung von Funktionen gängiger Programmiersprachen ermöglichen. Zu diesen Laufzeitumgebungen gehören beispielsweise .Net Core 2.2, Java 8, Node.js v14 oder Python 3. Darüber hinaus ermöglicht OpenWhisk die Verwendung nutzergenerierter Laufzeitumgebungen, die als Docker-Images auf *DockerHub*[2] veröffentlicht sind [Apad]. Diese Laufzeitumgebungen können beliebige Zusatzbibliotheken oder Frameworks enthalten, welche für die Ausführung einer Action mit speziellen Anforderungen benötigt werden. Benutzerdefinierte Laufzeitumgebungen werden von OpenWhisk als *blackbox* bezeichnet. Wird bei der Erstellung einer Action eine blackbox-Laufzeitumgebung gewählt, muss zusätzlich der Name des auf DockerHub veröffentlichten Image angegeben werden. OpenWhisk kann diesen nutzen, um das Image zu laden und für die Ausführung dazugehöriger Actions einen Docker-Container zu starten, um den Code auszuführen.

[2] https://hub.docker.com/

```
{
    "name": "MyTrigger",
    "namespace": "MyNamespace",
    "rules": {
        "MyNamespace/MyRule": {
            "action": {
                "name": "MyAction",
                "path": "MyNamespace"
            },
            "status": "active"
        }
    },
}
```

Listing 3.2: Beispiel eines Trigger

3.2.4 Trigger

Ein Trigger repräsentiert einen Kanal für eine Klasse von Ereignissen [Apaj]. Er besitzt einen individuellen REST-Endpunkt, welcher von Ereignisquellen beim Eintritt eines Ereignisses eines speziellen Ereignis-Feeds mit zum Ereignis gehörenden Eingabeparametern aufgerufen werden kann. Ein Beispiel eines Triggers ist in Listing 3.2 zu sehen. Wie eine Action besitzt ein Trigger einen Namen und gehört zu einem Namespace. Des weiteren hat jeder Trigger direkten Zugriff auf alle mit ihm verknüpften Rules. Wird der Trigger über seinen Endpunkt von einer Ereignisquelle aufgerufen, kann er die Rules nutzen, um die entsprechenden Actions mit den von der Ereignisquelle übergebenen Eingabeparametern auszuführen.

3.2.5 Rule

Eine Rule repräsentiert die Verknüpfung zwischen einer Action und einem Trigger [Apaj]. In Listing 3.3 ist ein Beispiel einer Rule abgebildet. Die Hauptbestandteile einer Rule sind ihr Name, der Namespace-Name und ein Aktivitätsstatus. Darüber hinaus enthalten Rules die eindeutigen Namen der beiden Objekte, die sie verknüpfen. Diese setzen sich aus den Namespace- und den Objekt-Namen zusammen. Im Beispiel wird der eindeutige Name durch die Felder *name* und *path* repräsentiert.

3.2.6 Activation

Eine Activation repräsentiert einen Aufruf eines Triggers bzw. eine abgeschlossene Ausführung einer Action. Eine durch den Aufruf eines Triggers generierte Activation ist in Listing 3.4 abgebildet. Activations enthalten alle ausführungsrelevanten Informationen wie eine Activation-ID, den Namespace, den Name der ausgeführten Action bzw. des aufgerufenen Triggers und weitere Ausführungsdetails wie Zeitstempel, Statuscodes und Logs. Während die Actions Informationen der Quellcodeausführung loggen, enthalten die Trigger-Logs Informationen über alle ausgeführten Actions. Das Ergebnis der Trigger-Activations enthält alle von der Ereignisquelle übergebenen Eingabeparameter, während Action-Activations das Ergebnis-Feld nutzen, um den Rückgabewert

```
{
    "name": "MyRule",
    "namespace": "MyNamespace",
    "status": "active",
    "trigger": {
        "name": "MyTrigger",
        "path": "MyNamespace"
    }
    "action": {
        "name": "MyAction",
        "path": "MyNamespace"
    }
}
```

Listing 3.3: Beispiel einer Rule

```
{
    "namespace": "MyNamespace",
    "name": "MyTrigger",
    "activationId": "triggerActivationId",
    "start": 1632504664820,
    "end": 0,
    "duration": 0,
    "statusCode": 0,
    "response": {
        "status": "success",
        "statusCode": 0,
        "success": true,
        "result": {
            "inputKey_1": "inputValue_1",
            "inputKey_2": inputValue_2
        }
    },
    "logs": [
        "{\"statusCode\":0,\"success\":true,\"activationId\":\"actionActivationId\",\"rule\":\
"MyNamespace/MyRule\",\"action\":\"MyNamespace/MyAction\"}"
    ],
}
```

Listing 3.4: Beispiel einer Activation

der ausgeführten Action zu speichern. Die Activation-IDs werden direkt beim Aufruf von Triggern bzw. bei der Ausführung von Actions generiert und dem Anfragesteller zurückgegeben. Wird beispielsweise ein Trigger von einer Ereignisquelle über die REST-API aufgerufen, antwortet OpenWhisk auf die Anfrage mit der Activation-ID des aufgerufenen Triggers. Die Ereignisquelle kann diese ID nutzen, um über die Logs der Trigger-Activation die Activation-IDs der asynchron ausgeführten Funktionen auszulesen.

3.2.7 Ereignissteuerung

OpenWhisk ermöglicht Nutzern die Ereignis-Feeds kompatibler Ereignisquellen zu steuern [Apag]. Bei diesen Ereignisquellen handelt es sich um externe Services, die beim Eintritt von Ereignissen die Endpunkte der Trigger per POST-Anfragen aufrufen können und spezielle Schnittstellen für die Steuerung ihrer Feeds anbieten. Für die Feed-Steuerung können sogenannte *Feed-Actions* eingesetzt werden. Eine Feed-Action ist eine spezielle Art von Action, welche mit den Schnittstellen des externen Service kommuniziert, um beispielsweise einen Trigger für Ereignis-Feeds zu registrieren oder ihn von diesen abzumelden. Um die Steuerung von Feeds zu ermöglichen, müssen Feed-Actions spezielle Eingabeparameter akzeptieren. Über diese Parameter kann unter anderem eingestellt werden, welche Operation auf dem Feed durchgeführt werden soll, welche Ereignisse dieser Feed filtern soll und wie der externe Service die Trigger-URL aufrufen soll. Die Feed-Action wird unter den Metadaten des Triggers gespeichert, damit sein Feed jederzeit angepasst werden kann. So kann sie beispielsweise vor dem Löschen eines Triggers ausgeführt werden, um diesen vom Ereignis-Feed abzumelden. Zur Steuerung von Feeds können die drei Patterns *Hooks*, *Polling* und *Connections* genutzt werden.

3.2.8 Hooks

Abbildung 3.2 zeigt, wie das Hooks-Pattern genutzt werden kann, falls externe Ereignisquellen die Nutzung von Webhooks unterstützen [Apag]. Solche Ereignisquellen bieten Schnittstellen an, welche anderen Systemen ermöglichen Webhooks einzustellen bzw. zu konfigurieren. Die Ereignisquelle kann die dabei registrierten URLs nutzen, um die einzelnen Services zu benachrichtigen, sobald ein internes Ereignis eintritt. Um einen Trigger bei einer Ereignisquelle zu registrieren, kann OpenWhisk eine Feed-Action verwenden. Bei der Registrierung kann die Trigger-URL in der Ereignisquelle eingetragen werden, damit diese beim Eintritt von Ereignissen eine POST-Anfrage stellen kann, um den Trigger auszulösen und die mit diesem verbundenen Actions auszuführen. Ein Service, welcher Webhooks unterstützt ist Github[3]. OpenWhisk könnte beispielsweise die GitHub-Webhooks[4] nutzen, um einen Webhook zu konfigurieren, welcher einen Trigger auslöst, sobald sich der Stand eines bestimmten Branch in einem Repository ändert [Git].

3.2.9 Polling

Das Polling-Pattern wird genutzt, falls eine Ereignisquelle keine Webhooks unterstützt und das Hooks-Pattern somit nicht in Frage kommt [Apag]. Das Prinzip des Polling-Patterns ist in Abbildung 3.3 dargestellt. Beim Polling-Pattern kommt ein Feed zum Einsatz, welcher Trigger in regelmäßigen Abständen auslöst. Dieser Feed kann manuell in Form eines externen Services programmiert werden, welcher über eine spezielle Feed-Action gesteuert wird. Alternativ bietet OpenWhisk einen vorprogrammierten, internen Alarm-Feed mit entsprechenden Feed-Actions an, welcher alle dafür registrierten Trigger in konfigurierbaren Abständen auslösen kann [Apac]. Wird das Polling-Pattern genutzt, sind die einzelnen Actions dafür zuständig, die Ereignisquellen auf Ereignisinformationen abzufragen. Die Actions laden dafür zunächst die Daten einer Ereignisquelle und untersuchen sie

[3] https://github.com/

[4] https://docs.github.com/en/developers/webhooks-and-events/webhooks/about-webhooks

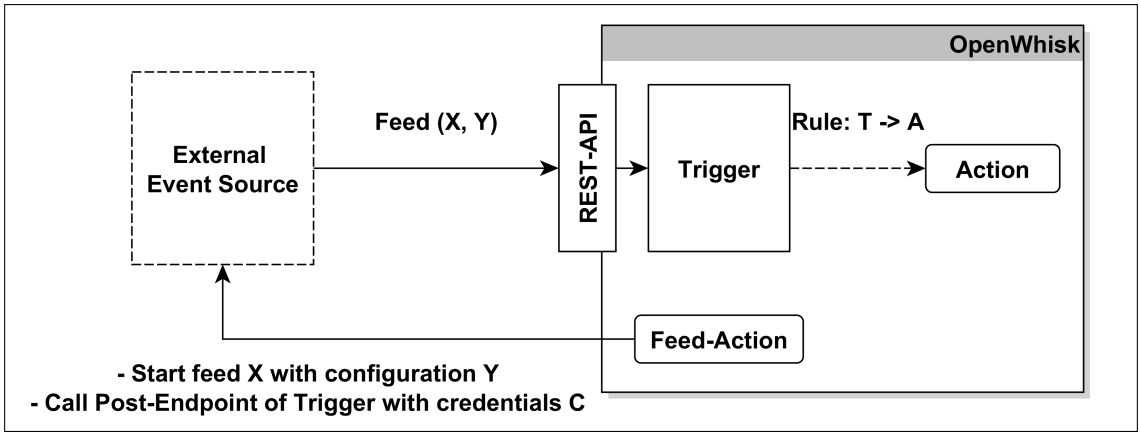

Abbildung 3.2: Prinzip des Hook-Pattern

anschließend auf Ereignisse. Wird bei der Untersuchung der Daten ein Ereignis festgestellt, kann die eigentliche Funktionslogik mit den im Ereignis enthaltenen Informationen ausgeführt werden. Das Polling-Pattern ist nicht für Ereignisse zu empfehlen, die mit hohen Frequenzen eintreten. In solchen Fällen müsste das Polling-Intervall sehr gering eingestellt werden, was dazu führen würde, dass alle Actions sehr oft bzw. ständig ausgeführt werden. Darüber hinaus fordert die Verwendung des Polling-Patterns, dass alle Actions zusätzlichen Quellcode enthalten, welcher die nötigen Ereignisquellen auf Ereignisse untersucht, was mit zusätzlichem Aufwand für die Entwickler der Actions verbunden ist.

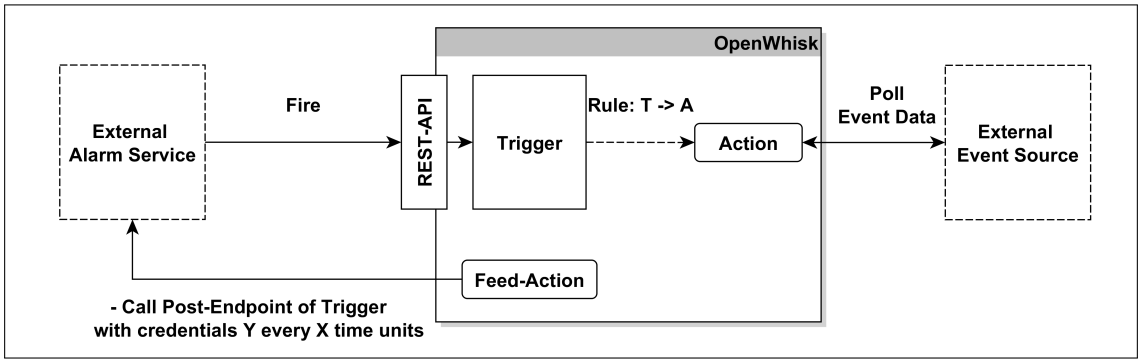

Abbildung 3.3: Prinzip des Polling-Pattern

3.2.10 Connections

Das Connections-Pattern ist eine Kombination der Polling- und Hook-Patterns [Apag]. Seine Funktionsweise wird in Abbildung 3.4 verdeutlicht. Für seine Umsetzung kommt ein externer Service zum Einsatz, welcher durchgehend mit einer Ereignisquelle verbunden ist und deren Daten abfragt. Gleichzeitig bietet er mehrere Schnittstellen zur Verwaltung von Webhooks an, damit registrierte Trigger ausgelöst werden können, sobald ein Ereignis in den von der Ereignisquelle abgefragten Daten festgestellt wird. OpenWhisk bezeichnet diese als Proxy zwischen der Ereignisquelle und OpenWhisk fungierenden Services als *Provider-Services*. Das Connections-Pattern wird immer

genutzt, wenn das Hooks-Pattern nicht infrage kommt und Ereignisse mit hohen Frequenzen eintreten. Zusätzlich wird die Entwicklung der einzelnen Actions erleichtert, da diese nicht wie beim Polling-Pattern mit speziellem Quellcode ausgestattet werden müssen, welcher die Ereignisquellen abfragt und dessen Daten auf Ereignisse untersucht.

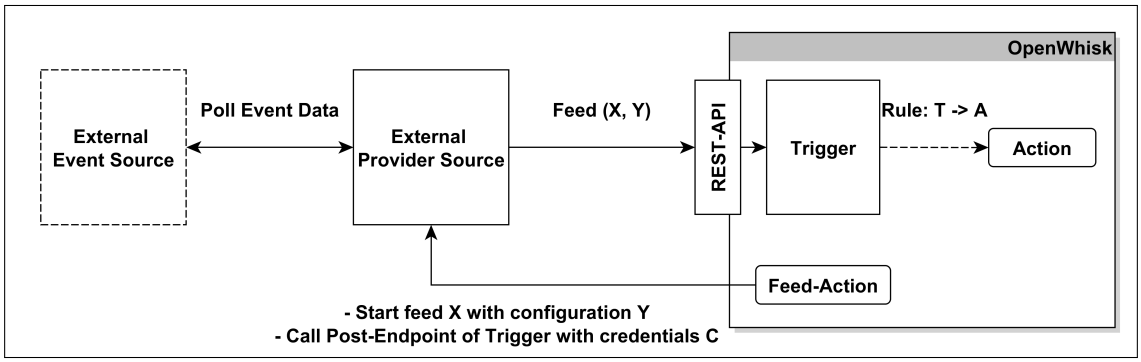

Abbildung 3.4: Prinzip des Connections-Pattern

3.3 IBM Quantum

IBM Quantum ist eine von IBM zur Verfügung gestellte Cloud-Plattform, welche den öffentlichen Zugang zu IBMs Quantum Computing Diensten ermöglicht. Mit IBM Quantum können registrierte Nutzer verfügbare Quantencomputer und Simulatoren für die Durchführung von Experimenten verwenden. Alle über IBM Quantum zur Verfügung gestellten Quantencomputer und Simulatoren gehören sogenannten *Hubs* an. Bei den Hubs handelt es sich um regionale Zentren, welche ihren Mitgliedern den Zugang zu Quantentechnologie bieten [IBMa]. Innerhalb der Hubs existieren Gruppen, welche wiederum unterschiedliche Projekte enthalten, in denen sich schlussendlich die einzelnen Quantencomputer bzw. Simulatoren befinden. Unterschiedliche IBM Quantum Accounts haben Zugang zu unterschiedlichen Hubs, Gruppen und Projekten. Premium-Nutzer haben beispielsweise Zugang zu einer größeren Anzahl von Hubs und somit zu mehr Quantencomputern [IBMd].

Für die Kommunikation mit den von IBM zur Verfügung gestellten Quantencomputern und Simulatoren kann die Programmiersprache OpenQASM[5], das Python-SDK Qiskit[6] oder eine über den Web-Browser erreichbare grafische Benutzeroberfläche verwendet werden. Durch deren Nutzung können Quantenschaltungen generiert und für die Ausführung an beliebige Quantencomputer abgeschickt werden. Da es sich hierbei um öffentliche Quantencomputer handelt, werden die Ausführungsanfragen einzelner Nutzer, welche von IBM als *Jobs* bezeichnet werden, nicht sofort bearbeitet. Stattdessen werden Jobs in eine spezielle, zum jeweiligen Quantencomputer gehörende Warteschlange gelegt, wo sie der Reihe nach abgearbeitet werden. IBM Quantum generiert bei der Erstellung eines Jobs automatisch eine Job-ID, welche für die eindeutige Identifikation dieses Jobs genutzt wird. Nutzer können diese Job-IDs verwenden, um jederzeit den aktuellen Status ihrer Jobs abzufragen oder sich die Ergebnisse nach erfolgreichem Abschluss der Ausführung zu

[5] https://github.com/Qiskit/openqasm
[6] https://qiskit.org/

beschaffen. Hierfür kann die von IBM Quantum angebotene REST-API[7] verwendet werden, welche allen registrierten Nutzern zur Verfügung steht. Mithilfe dieser können nicht nur Job-Informationen ausgelesen werden, sondern auch weitere Daten der IBM Quantum Dienste, wie beispielsweise Informationen zu den aktuell verfügbaren Quantencomputern und deren Warteschlangen.

[7] https://api.quantum-computing.ibm.com/v2/

4 Konzept

Das zu entwickelnde System soll Nutzern ermöglichen, ihre Quantenanwendungen über einen FaaS-Dienst bereitzustellen und sie für Ereignis-Feeds zu registrieren, damit sie durch den Eintritt von Ereignissen auf einem über die Cloud verfügbaren Quantencomputer ausgeführt werden. In Abbildung 4.1 ist eine grobe Skizze der umzusetzenden Systemarchitektur abgebildet.

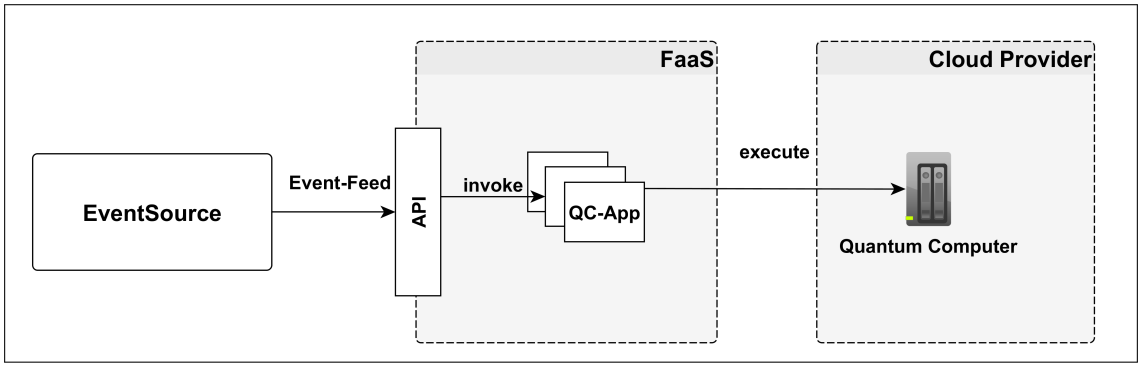

Abbildung 4.1: Skizze einer ereignisbasierten Architektur

Für die Umsetzung des skizzierten Systems müssen unabhängig von Architektur und genutzter Technologie einige essenzielle Funktionen implementiert werden. Abbildung 4.2 zeigt ein Anwendungsfalldiagramm, das Anforderungen und Erwartungen beschreibt, welche von Akteuren an das System gestellt werden. Die erste wichtige Funktion, die vom System umgesetzt werden muss, ist die Verwaltung von Quantenanwendungen. Hierbei soll Akteuren ermöglicht werden, diese zum System hinzuzufügen und sie bei verfügbaren Ereignis-Feeds zu registrieren. Darüber hinaus muss das System in der Lage sein, jede im System angemeldete Quantenanwendung über einen FaaS-Dienst auszuführen. Für die Umsetzung der ereignisbasierten Ausführung müssen externe Ereignisquellen in das System eingebunden werden, sodass mithilfe derer Ereignis-Feeds die Ausführung der dafür registrierten Quantenanwendungen automatisch ausgelöst wird. In den folgenden Abschnitten werden Konzepte zum Lösen aller nötigen Funktionen im Detail beschrieben.

4.1 Verwaltung von FaaS-Diensten

Damit Nutzer mehr Freiheiten bei der Wahl der für die Ausführung der Quantenanwendungen zuständigen FaaS-Dienste haben, soll das System in der Lage sein, diese zur Laufzeit zu verwalten. In Abbildung 4.3 ist der Registrierungsprozess von FaaS-Diensten beliebiger Anbieter skizziert. Bevor Nutzer den gewünschten FaaS-Dienst im System registrieren können, müssen sie diesen entweder auf einem Rechner selbstständig hosten oder ein existierendes Angebot eines Cloud-Anbieters nutzen. Dabei müssen sie einen Account erstellen, welcher ihnen die Interaktion mit dem jeweiligen

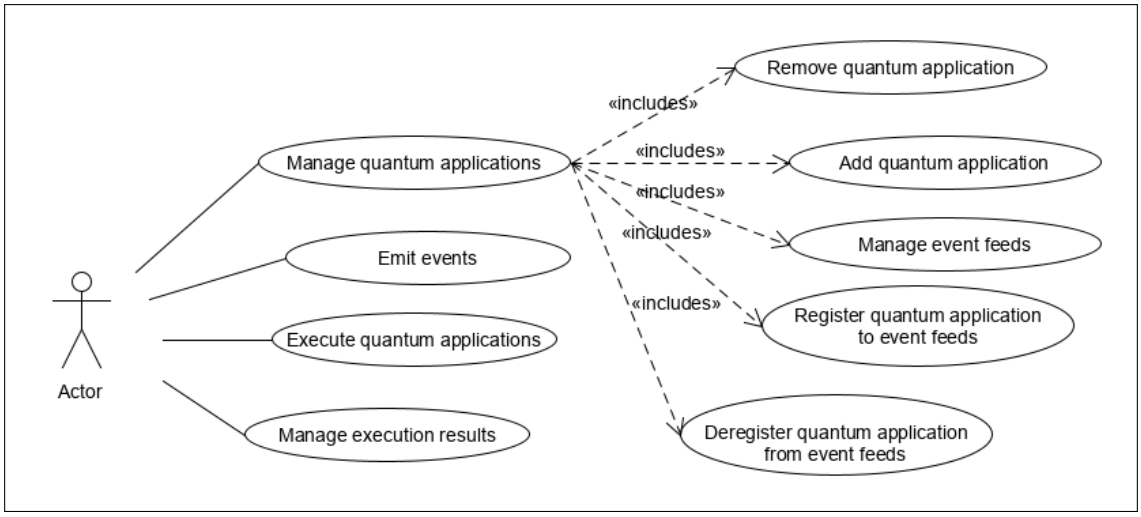

Abbildung 4.2: Anwendungsfalldiagramm des Systems

Dienst erlaubt. Im Anschluss kann der Dienst über eine dafür vorgesehene Schnittstelle registriert werden, indem die für die Kommunikation notwendigen Dienst- und Account-Daten dem System übergeben werden. Dieses speichert die Daten in einer Datenbank, damit der Dienst später für die Bereitstellung und Ausführung von Quantenanwendungen verwendet werden kann. Auf diese Weise kann jeder Nutzer seinen privaten FaaS-Dienst für die ereignisbasierte Ausführung seiner Quantenanwendungen nutzen. Damit die Registrierung eines Dienstes möglich ist, muss dieser vom System unterstützt werden. Dafür muss zum Dienst passender Client-Code existieren, welcher mit seinen Schnittstellen kommunizieren kann, um alle notwendigen Operationen wie das Bereitstellen, Löschen und Ausführen von Funktionen zu ermöglichen. Wenn das System beispielsweise nur OpenWhisk unterstützt, kann kein privat gehosteter OpenFaaS[1]-Server als Dienst registriert werden. Neben der Registrierung muss das System den Nutzern auch die Löschung der registrierten Dienste ermöglichen. Hierfür kann eine weitere Schnittstelle eingesetzt werden, deren Aufruf den Dienst aus der Datenbank löscht und bei Bedarf alle bereitgestellten Quantenanwendungen bzw. Funktionen entfernt.

4.2 Verwaltung von Quantenanwendungen

Damit Quantenanwendungen von einem FaaS-Dienst ausgeführt werden können, müssen sie zunächst über diesen bereitgestellt werden. Stellt ein Nutzer seine Quantenanwendungen auf direktem Wege über einen FaaS-Dienst bereit, kann das System aufgrund der Unwissenheit über ihre Existenz diese nicht ausführen. Der Nutzer müsste sich in diesem Fall zusätzlichen Aufwand machen und die bereitgestellte Quantenanwendung im System referenzieren. Um das Problem zu adressieren, werden alle Quantenanwendungen nur über eine spezielle, zum System gehörende Komponente angemeldet.

[1] https://www.openfaas.com/

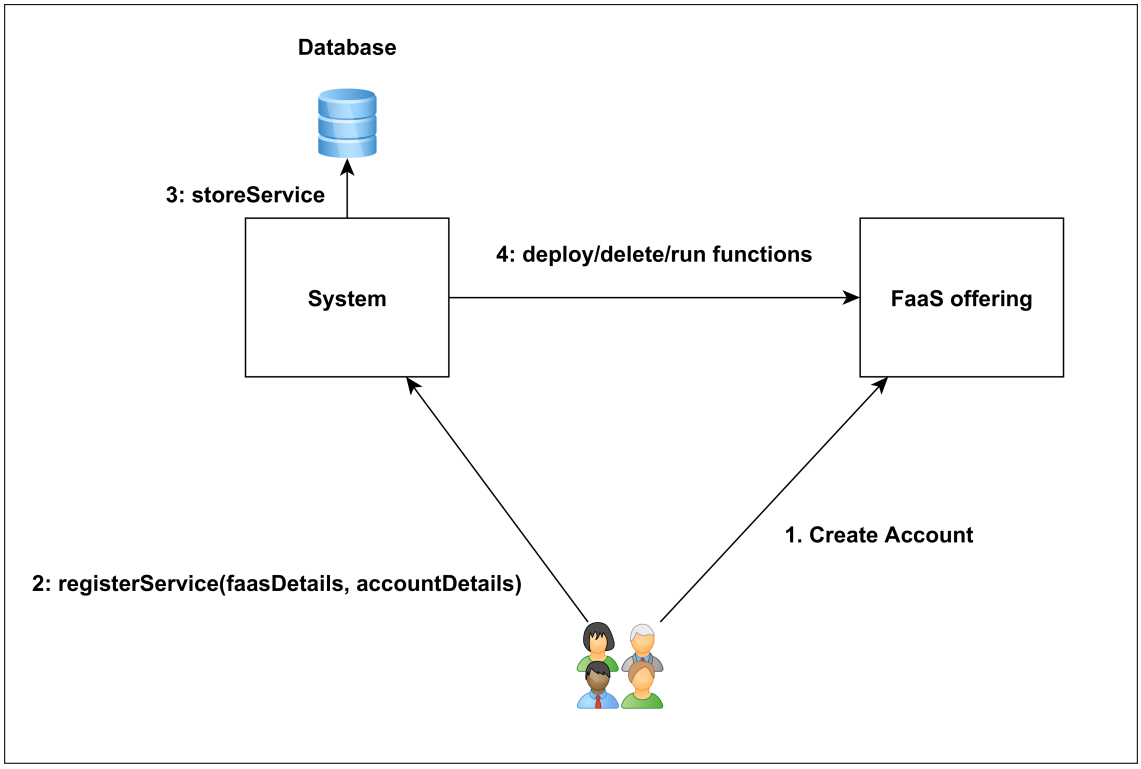

Abbildung 4.3: Registrierung von FaaS-Diensten

Abbildung 4.4 zeigt wie Nutzer ihre Quantenanwendungen durch die *QC-App-Management*-Komponente verwalten können. Hierfür kommen öffentlich zugängliche Schnittstellen zum Einsatz, welche von Nutzern oder externen Services für die Durchführung unterschiedlicher Operationen verwendet werden können. Dabei werden die Änderungen, welche bei der Durchführung der Operationen entstehen, stets in einer Datenbank gespeichert. Im Fall von Erstell- und Löschoperationen wird zusätzlich eine entsprechende Anfrage an den gewünschten, im System registrierten FaaS-Dienst gestellt, damit die Quantenanwendung für die künftige Ausführung bereitgestellt bzw. entfernt wird.

Damit das System mit den Quantenanwendungen arbeiten kann, werden zusätzlich zu ihrem Quellcode weitere Daten wie IDs, Namen, Ergebnisse, der zu nutzende FaaS-Dienst, Referenzen zu bereitgestellten Funktionen und sogenannte Ereignis-Trigger benötigt. Diese können zusammen mit der Quantenanwendung in die gleiche oder durch den Einsatz von Primär- und Fremdschlüsseln in separate Datenbanktabellen gespeichert werden. Daten wie IDs und Namen ermöglichen dem System die einzelnen Quantenanwendungen voneinander zu unterscheiden. Im Vergleich dazu wird der genutzte FaaS-Dienst und die Referenz zur bereitgestellten Funktion benötigt, damit das System mit dieser im Falle eines Ereigniseintritts interagieren kann.

Für die ereignisbasierte Ausführung müssen alle im System angemeldeten Quantenanwendungen bei verfügbaren Ereignis-Feeds registriert werden können. Da Quantenanwendungen üblicherweise nicht beim Eintritt aller Ereignisse ausgeführt werden sollen, kommen die erwähnten Ereignis-Trigger zum Einsatz. Mit diesen können die über Feeds gelieferten Ereignisse anhand ihrer Eigenschaften gefiltert werden. Nutzer können Ereignis-Trigger über spezielle Schnittstellen

erstellen, um die vorhandenen Feeds nach ihren wünschen zu individualisieren. Dadurch werden „virtuelle“ Versionen der existierenden Feeds erstellt, welche die individuellen Anforderungen der einzelnen Quantenanwendungen erfüllen können. Um Quantenanwendungen bei einem Ereignis-Feed zu registrieren, müssen diese über eine dafür vorgesehene Schnittstelle mit dem gewünschten Ereignis-Trigger verknüpft werden. Ist ein Feed nicht mehr erwünscht, kann die Verknüpfung gelöst oder der Ereignis-Trigger gelöscht werden.

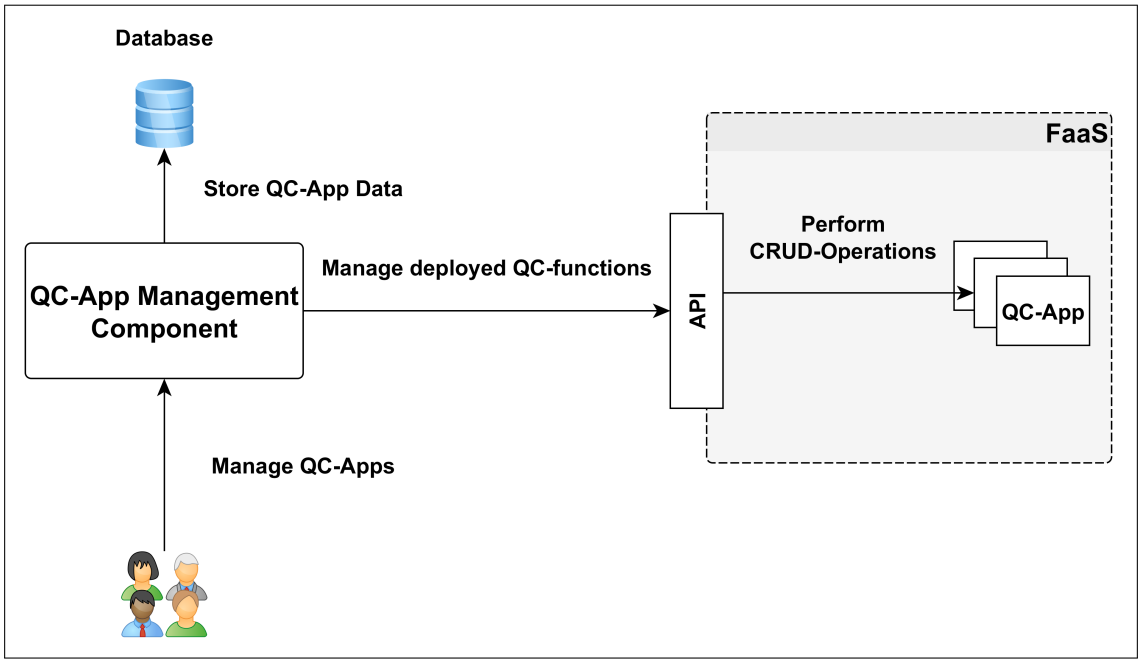

Abbildung 4.4: Verwaltung von Quantenanwendungen

4.3 Emittieren von Ereignissen

Da nicht alle Systeme, welche als Ereignisquelle eingesetzt werden können in der Lage sind, von sich aus Ereignisse zu emittieren, kommt eine Systemkomponente zum Einsatz, welche die Schnittstellen dieser nutzt, um nach dem bereits vorgestellten Polling-Pattern (siehe Abschnitt 3.2.9), die Ereignisdaten zu beschaffen und anhand derer Inhalte die Ausführung der Quantenanwendungen vorzubereiten. Zusätzlich bietet die Komponente Push-fähigen Ereignisquellen die nötigen Schnittstellen an, damit diese das System über den Eintritt von internen Ereignissen benachrichtigen können. Abbildung 4.5 zeigt die als *EventBridge* bezeichnete Komponente. Die EventBridge ist für die Integration von Ereignis-Feeds zuständig. Dabei überführt sie alle ankommenden Ereignisse in ein vom System verstandenes Ereignisobjekt, welches für die Ausführung von Quantenanwendungen genutzt werden kann. Sie erfüllt somit zusätzlich die Funktion eines *MessageTranslator*[2].

Im Aktivitätsdiagramm der Abbildung 4.6 wird dargestellt, wie mithilfe der EventBridge-Komponente eine Ereignisquelle integriert werden kann, welche nicht in der Lage ist selbstständig Ereignisse zu emittieren. Dazu muss die EventBridge-Komponente mit den von der Ereignisquelle

[2]https://www.enterpriseintegrationpatterns.com/MessageTranslator.html

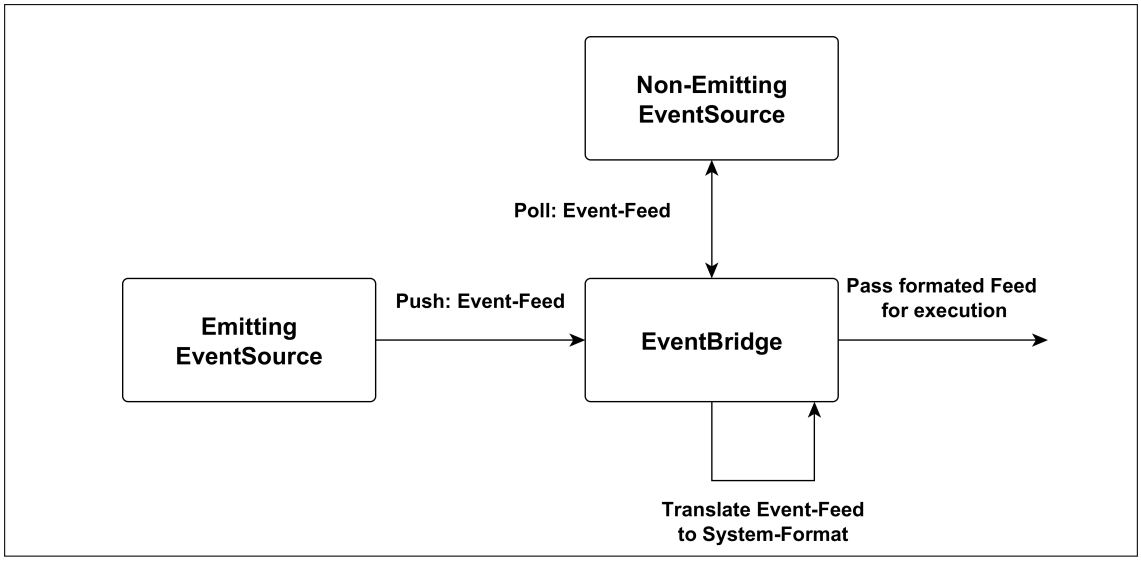

Abbildung 4.5: Funktionsweise der EventBridge

zur Verfügung gestellten Schnittstellen kommunizieren. Damit die EventBridge mit diesen interagieren kann, benötigt sie üblicherweise für die Kommunikation notwendige Daten, welche zuerst beschafft werden müssen. Zu diesen Daten gehören beispielsweise Authentifizierungstoken, welche den Aufruf von geschützten Schnittstellen möglich machen oder Identifikationsnummern, welche den gezielten Abruf von Ressourcen ermöglichen. Statische Daten wie Authentifizierungstoken werden in Umgebungsvariablen gespeichert und von der EventBridge bei Bedarf genutzt. Im Vergleich dazu werden Daten, welche zur Laufzeit generiert oder verändert werden, aus einer Datenbank ausgelesen. Nachdem die für die Anfrage notwendigen Daten beschafft wurden, kann mit den Schnittstellen der Ereignisquelle interagiert werden. Die dabei beschafften Daten können im Anschluss in das bereits erwähnte Ereignisobjekt umgewandelt und für die Ausführung von Quantenanwendungen an die dafür zuständige Komponente weitergereicht werden.

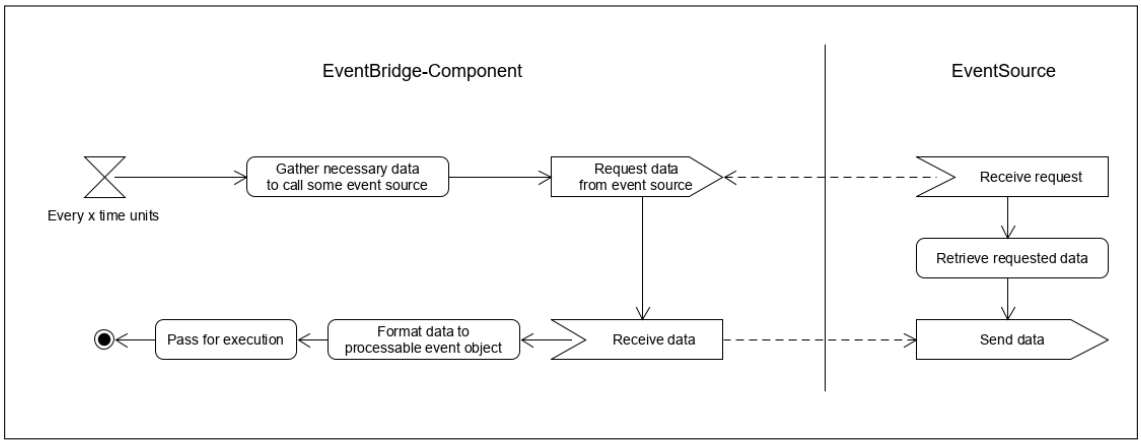

Abbildung 4.6: Integration von Ereignis-Feeds durch Polling

Wie in Abbildung 4.6 zu sehen ist, wird der Polling-Prozess über einen Timer in regelmäßigen Abständen ausgeführt. Dieser Abstand sollte immer passend zu den Daten ausgewählt werden, welche von der externen Ereignisquelle abgefragt werden. Werden beispielsweise Ausführungsergebnisse einer Quantenanwendung abgefragt, ist es nicht sinnvoll kurze Polling-Intervalle zu verwenden, da die Ausführung von Quantenanwendungen auf öffentlichen Quantencomputern durch die Nutzung von Warteschlangen mehrere Stunden in Anspruch nehmen kann. Deswegen ist es sinnvoll, verschiedene Schnittstellen in parallel laufenden Prozessen abzufragen und dabei unterschiedlich konfigurierte Timer einzusetzen.

Der beschriebene Umweg über Polling-Prozesse wird nicht benötigt, falls eine Ereignisquelle selbstständig Ereignisse emittieren kann. Abbildung 4.7 zeigt ein Aktivitätsdiagramm, welches beschreibt wie Push-fähige Ereignisquellen integriert werden können. Die externen Ereignisquellen benachrichtigen dabei die EventBridge-Komponente durch den Aufruf der von ihr zur Verfügung gestellten Schnittstellen. Dafür können beispielsweise HTTP-Schnittstellen oder Messaging-Warteschlangen verwendet werden. Wie beim Polling-Prozess wandelt die EventBridge die ankommenden Ereignisdaten bei Bedarf in ein Ereignisobjekt um, welches für die Ausführung der Quantenanwendungen weitergeleitet wird. Sollte eine Push-fähige Ereignisquelle nicht direkt über die Existenz der EventBridge Bescheid wissen, kann diese bei der Ereignisquelle mittels des bereits vorgestellten Hook-Pattern (siehe Abschnitt 3.2.8) für die nötigen Ereignis-Feeds registriert werden.

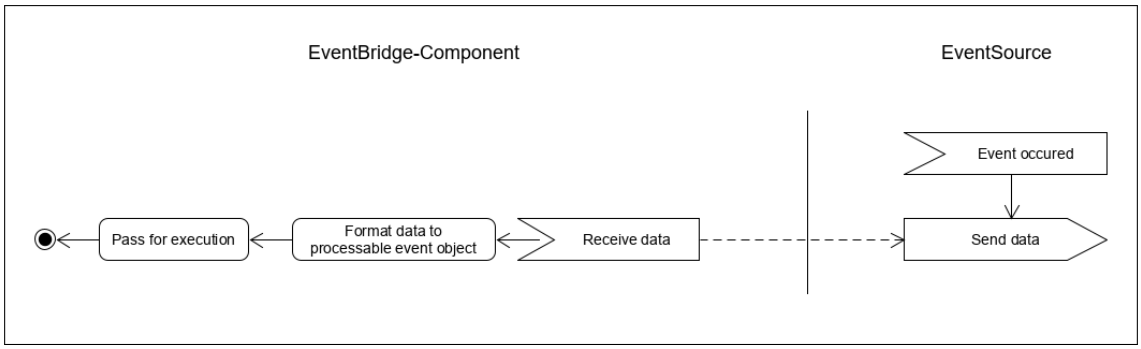

Abbildung 4.7: Integration von Ereignis-Feeds durch Push-Nachrichten

4.4 Ausführung von Quantenanwendungen

Da Quantenanwendungen nur mithilfe spezieller Programmiersprachen und SDKs geschrieben werden können, ist deren Ausführung nur in speziellen Laufzeitumgebungen möglich. Wie bereits erwähnt, erwarten IBMs Quantencomputer beispielsweise speziell formatierte Eingaben, welche nur mithilfe der Programmiersprache OpenQASM oder des Qiskit-SDKs erzeugt werden können. Das in dieser Arbeit entwickelte Konzept beschränkt sich somit auf FaaS-Dienste von Anbietern, welche in der Lage sind, die von Quantenanwendungen benötigten Laufzeiten für die Ausführung der bereitgestellten Funktionen zu verwenden.

Für die ereignisbasierte Ausführung der im System angemeldeten Quantenanwendungen müssen die über Feeds beschafften Ereignisse zum Einsatz kommen. In Abbildung 4.8 ist zu sehen, wie die von der EventBridge erzeugten Ereignisobjekte genutzt werden können, um die passenden Quantenanwendungen auf einem FaaS-Dienst auszuführen. Im ersten Schritt wird mithilfe des

Ereignisobjekts die Datenbank abgefragt. Diese vergleicht dabei die Eigenschaften des Ereignisobjekts mit den existierenden Ereignis-Triggern. Dabei werden die Quantenanwendungen aller Ereignis-Trigger zurückgegeben, die den Eigenschaften des Ereignisobjekts entsprechen. Mithilfe der in den Quantenanwendungen gespeicherten Referenzen können Anfragen an den richtigen FaaS-Dienst geschickt werden, um die korrekten Funktionen in passenden Laufzeitumgebungen auszuführen. Die auszuführenden Funktionen bekommen dabei den Inhalt des Ereignisobjekts als Eingabeparameter übergeben.

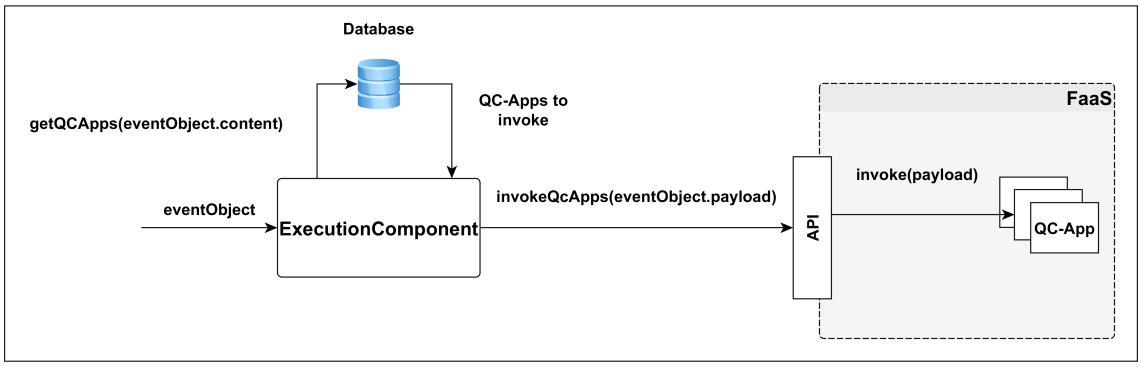

Abbildung 4.8: Ereignisbasierte Ausführung von Quantenanwendungen

4.5 Verwaltung von Ausführungsergebnissen

Bei der Ausführung einer Quantenanwendung werden mithilfe der übergebenen Eingabeparameter sogenannte Quantenschaltungen generiert, welche im Anschluss auf einem Quantencomputer ausgeführt werden. Für die Ausführung stellt die Quantenanwendung eine Anfrage an eine Schnittstelle eines Cloud-Dienstes, welcher Quantencomputer der Öffentlichkeit zur Verfügung stellt. Da öffentliche Quantencomputer auf Warteschlangen setzen, kann die Ausführung einer Quantenschaltung viel Zeit in Anspruch nehmen, sodass die angefragten Schnittstellen nicht direkt mit dem Ergebnis antworten können. Stattdessen antworten sie mit Daten wie beispielsweise einer ID, welche die laufende Ausführung identifiziert. Abbildung 4.9 zeigt skizzenhaft, wie das System diese Ausführungs-IDs beschaffen kann.

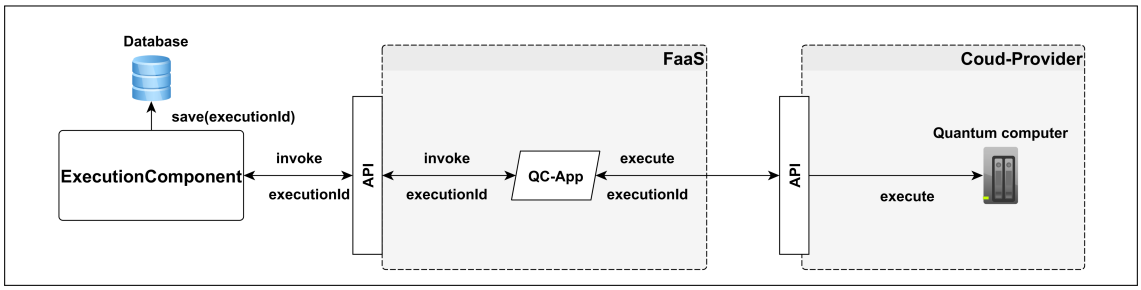

Abbildung 4.9: Beschaffung der Ausführungs-ID

Bei der Beschaffung der Ausführungs-IDs ist die über einen FaaS-Dienst bereitgestellte Quantenanwendung dafür zuständig, nach der Interaktion mit dem QC-Dienst eines Cloud-Anbieters die in der Antwort enthaltene Ausführungs-ID als Rückgabewert der Systemkomponente zu übergeben, welche sie aufgerufen hat. So kann das System die vom QC-Dienst generierte Ausführungs-ID für die Statusverfolgung und Beschaffung des Ergebnisses als Objekt in der Datenbank abspeichern.

Abbildung 4.10 zeigt die für die Beschaffung der Ergebnisse zuständige *ResultPolling*-Komponente, welche mithilfe der gespeicherten IDs die Schnittstellen des QC-Dienstes abfragt, um den Status der laufenden Ausführung zu verfolgen und das Ergebnis zu beschaffen. Der grobe Ablauf des von ihr durchgeführten Prozesses ist in Abbildung 4.11 zu sehen. Wie bei den Ereignisdaten mancher Ereignisquellen müssen auch die Ausführungsstatus in regelmäßigen Abständen abgefragt werden. Dafür werden zunächst alle Ausführungs-IDs von laufenden Ausführungen aus der Datenbank ausgelesen. Diese IDs werden für den Aufruf der Schnittstellen des QC-Dienstes genutzt, welche den aktuellen Status der entsprechenden Ausführungen liefern. Sollte die Ausführung des Quantencomputers abgeschlossen und das Ergebnis verfügbar sein, wird das Ausführungs-Objekt um das Ergebnis erweitert und als abgeschlossen markiert, damit der Polling-Prozess für diese Ausführung nicht weiter durchgeführt wird. Die Ergebnisse der Quantencomputer können im Anschluss von Nutzern oder externen Services über spezielle Schnittstellen ausgelesen und untersucht oder zum emittieren von neuen Ereignissen genutzt werden.

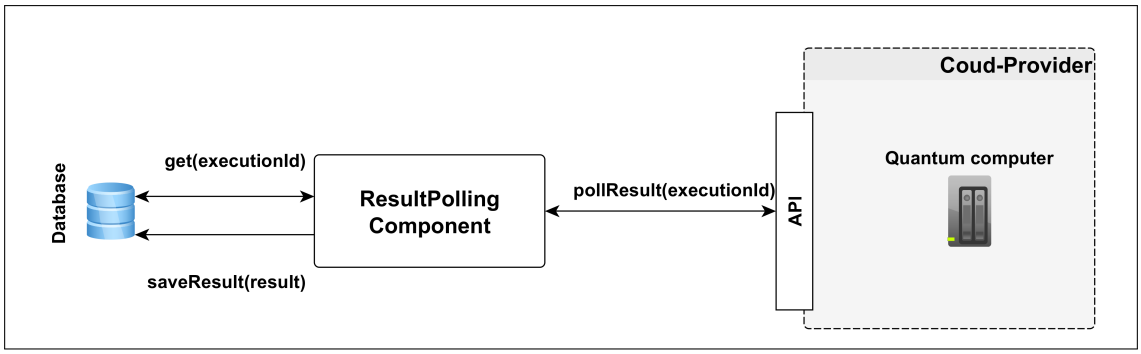

Abbildung 4.10: Beschaffung der Ausführungsergebnisse

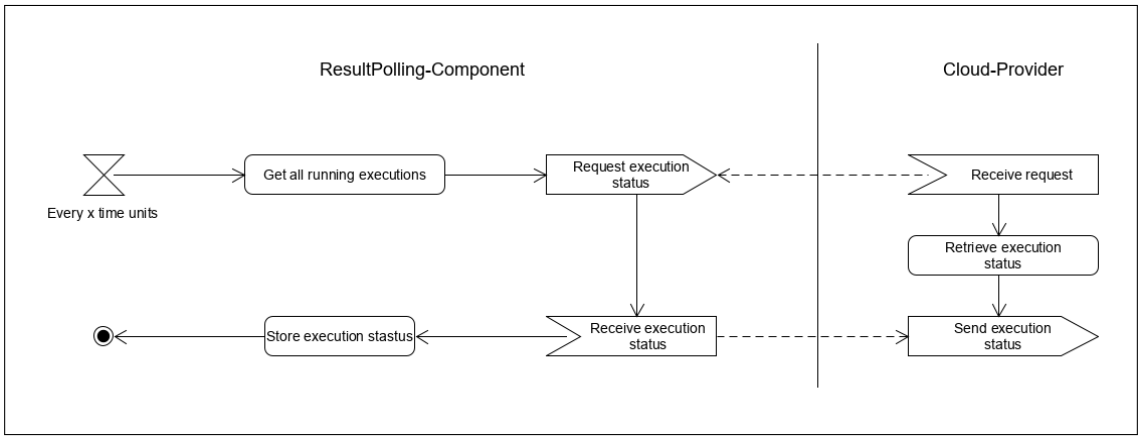

Abbildung 4.11: Polling-Prozess für Ergebnisse von Quantencomputern

4.6 Aufbau von Quantenanwendungen

Da das konzipierte System die Bereitstellung und Ausführung aller im System angemeldeten Quantenanwendungen mittels FaaS umsetzt, müssen sie entsprechend als Funktionen geschrieben werden. Damit die bereitgestellten Funktionen vom System verwendet werden können, muss ihre Struktur einer vom System vorgegebenen Vorlage folgen.

Alle Quantenanwendungen müssen in der Lage sein, die vom System übergebenen Eingabeparameter erkennen und verwenden zu können. Diese Eingabeparameter sind dabei vom Ereignis abhängig, welches die Ausführung der Quantenanwendung auslöst. Zusätzlich kann es Eingabeparameter geben, deren Nutzung durch das System vorgegeben wird. Ein Beispiel dafür wäre ein Authentifizierung-Token, welcher für den Aufruf der Schnittstellen des QC-Dienstes benötigt wird. Der Entwickler einer Quantenanwendung muss der Vorlage der Eingabeparameter nicht immer vollständig folgen. Für einige der Parameter lassen sich Standardwerte einführen, sodass die Quantenanwendung auch bei deren Abwesenheit funktionieren kann. Ebenfalls können einige Eingabeparameter hartkodiert werden, sodass die ankommenden Eingabeparameter dadurch „überschrieben“ werden.

Wie im letzten Abschnitt bereits erwähnt, wird bei der Ausführung einer Quantenanwendung eine ID generiert, mit welcher die Ergebnisse des Quantencomputers beschafft werden können, sobald sie verfügbar sind. Das konzipierte System erwartet diese über den Rückgabewert der Quantenanwendung. Das bedeutet, dass die Quantenanwendungen ihre Ausgabeparameter nach einer Vorlage strukturieren müssen, damit das System diese verarbeiten kann. Da ohne die Ausführungs-ID keine Ergebnisse beschafft werden können, ist die korrekte Struktur der Ausgabe für die fehlerfreie Operation des Systems kritisch.

5 Implementierung

In diesem Kapitel wird der Aufbau und die Funktionsweise eines Prototyps erläutert, welcher für die Bereitstellung und Ausführung von Quantenanwendungen OpenWhisk (siehe Abschnitt 3.2) und die von IBM Quantum (siehe Abschnitt 3.3) zur Verfügung gestellten Quantencomputer und Schnittstellen verwendet.

5.1 Architekturübersicht

In Abbildung 5.1 ist die Architektur des in dieser Arbeit implementierten Prototyps dargestellt. Der sogenannte *QuantumService*[1] ist hierbei die zentrale Komponente, die allen Nutzern und externen Services eine REST-API für die Verwaltung von Quantenanwendungen und Ereignis-Feeds zur Verfügung stellt. Alle dafür benötigten Daten werden in einer externen *MySQL*[2] Datenbank abgespeichert. Für die Ausführung und Bereitstellung der im System registrierten Quantenanwendungen können beliebige auf OpenWhisk-basierende FaaS-Dienste im System registriert und verwendet werden. Damit die Verwaltung, Konfiguration und Bereitstellung von Quantenanwendungen und Ereignis-Feeds möglichst simpel gestaltet werden kann, werden alle auf ihnen durchgeführten Operationen über den QuantumService durchgeführt. Dieser spiegelt die Operationen auf den entsprechenden OpenWhisk-Diensten, sodass alle benötigten Actions (siehe Abschnitt 3.2.3), Trigger (siehe Abschnitt 3.2.4) und Rules (siehe Abschnitt 3.2.5) automatisch erstellt, gelöscht oder ausgeführt bzw. ausgelöst werden. Für die Einbindung der Ereignisse werden QC-spezifische Daten genutzt, welche über die öffentlichen Schnittstellen[3] von IBM Quantum ausgelesen werden. Da IBM Quantum zum jetzigen Zeitpunkt nicht das selbstständige Emittieren von intern auftretenden Ereignissen unterstützt, müssen die Ereignis-Feeds mittels des bereits erwähnten Connection-Patterns (siehe Abschnitt 3.2.10) umgesetzt werden. Dafür fragt der QuantumService wiederholt IBM Quantums Schnittstellen ab und löst anhand der beschafften Ereignisdaten die passenden Trigger auf den registrierten OpenWhisk-Diensten aus. Das sorgt für die Ausführung der mit den Triggern verknüpften Actions, welche mittels Qiskit die für die Ausführung benötigten Quantenschaltungen generieren und an IBMs Quantencomputer schicken. Für die künftige Integration von Push-fähigen Ereignisquellen bietet der QuantumService zusätzlich einen speziellen HTTP-Endpunkt und eine MQ[4]-Warteschlange an, welche für das Empfangen der Ereignis-Feeds verwendet werden können. Der QuantumService bündelt damit die Funktionen aller im Konzept (siehe Kapitel 4) vorgestellten Komponenten in einem Service. Zusätzlich

[1] https://github.com/LHommeDeBat/QuantumServiceFaas
[2] https://www.mysql.com/de/
[3] https://api.quantum-computing.ibm.com/v2/
[4] https://www.ibm.com/de-de/products/mq

zum QuantumService kommt ein Frontend mit dem Namen *QuantumServiceUI*[5] zum Einsatz. Dieses ermöglicht Nutzern eine einfache Bedienung des Systems mittels einer über den Browser erreichbaren grafischen Benutzeroberfläche.

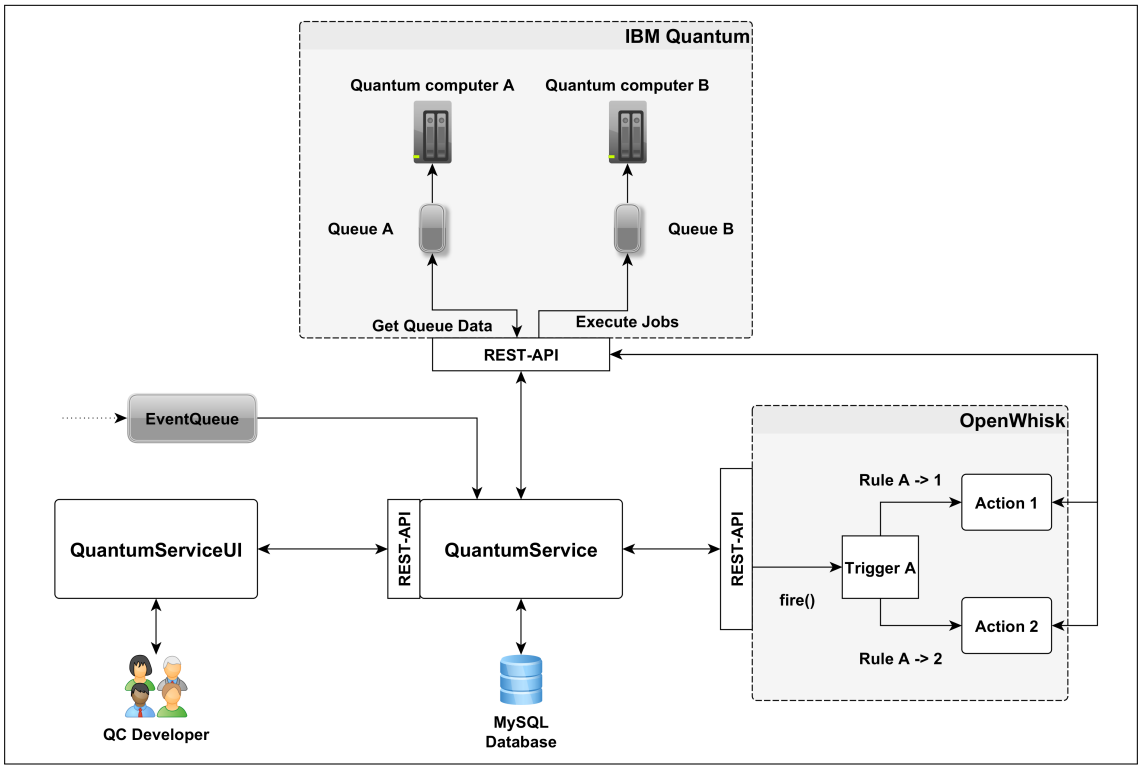

Abbildung 5.1: Ereignisbasierte OpenWhisk-Architektur

5.2 Genutzte Technologien

Der QuantumService wurde mithilfe des Frameworks *Spring Boot*[6] und der Programmiersprache *Java*[7] implementiert. Mit Spring Boot können Backend-Services schnell und einfach geschrieben und bereitgestellt werden. Die von Spring Boot zur Verfügung gestellten *Starter*[8] erleichtern zusätzlich das Abhängigkeitsmanagement, sodass nötige Bibliotheken mühelos zum Projekt hinzugefügt werden können. Im Vergleich zum QuantumService wurde die QuantumServiceUI mithilfe des Frameworks *Angular*[9] umgesetzt. Bei Angular handelt es sich um ein *TypeScript*[10]-basiertes Framework, welches die Entwicklung von Frontend-Webanwendungen vereinfacht und dadurch beschleunigt.

[5] https://github.com/LHommeDeBat/QuantumServiceFaasUI
[6] https://spring.io/projects/spring-boot
[7] https://www.java.com/de/
[8] https://github.com/spring-projects/spring-boot/tree/main/spring-boot-project/spring-boot-starters
[9] https://angular.io/
[10] https://www.typescriptlang.org/

Für die Implementierung der nötigen Systemfunktionen wird die Speicherung von verschiedensten Daten benötigt. Da bei den vom System verwalteten Daten die Konsistenz und Integrität eine wichtige Rolle spielen, wurde für die Speicherung eine relationale MySQL-Datenbank gewählt. Die Kommunikation zwischen Service und Datenbank erfolgt mittels *Hibernate*[11] und *JPA*[12], welche dank der bereits erwähnten Spring-Starter einfach zum Projekt hinzugefügt und konfiguriert werden können. Bei der Implementierung wurde dabei stets darauf geachtet, dass keine exklusiven MySQL-Funktionen genutzt wurden, sodass MySQL durch andere relationale Datenbankverwaltungssysteme ersetzt werden kann, welche SQL unterstützen.

Für den Einsatz von Messaging wird die von IBM zur Verfügung gestellte Messaging-Lösung MQ verwendet. IBMs MQ kann manuell über Docker-Container[13] bereitgestellt und genutzt werden, aber auch über den von IBM zur Verfügung gestellten Cloud-Dienst[14] gemietet werden. Die Verbindung zur MQ-Instanz erfolgt über einen von IBM entwickelten Spring-Starter[15].

5.3 Struktur des QuantumService

Für die Umsetzung seiner Funktionalitäten verwaltet der in dieser Arbeit implementierte QuantumService eine Vielzahl an unterschiedlichen Entitäten, welche dem Klassendiagramm der Abbildung 5.2 entnommen werden können. Dazu gehören die sogenannten *OpenWhiskServices*, *EventTrigger* und *QuantumApplications* sowie die dazugehörenden *Jobs* und *ScriptExecutions*. Die folgenden Abschnitte enthalten eine detaillierte Beschreibung der einzelnen Entitäten.

5.3.1 OpenWhiskService

Ein OpenWhiskService referenziert den Namespace (siehe Abschnitt 3.2.2) eines OpenWhisk-Dienstes. Er enthält Informationen wie die Heimat-URL, den Namen des Namespaces und die für die Basic-Authentifizierung benötigten Anmeldeinformationen. Diese Daten ermöglichen dem QuantumService alle nötigen Operationen auf dem Namespace des gegebenen OpenWhisk-Dienstes durchzuführen. So kann der QuantumService beispielsweise Trigger oder Actions erstellen bzw. löschen und diese durch die Erstellung von Rules miteinander in Verbindung bringen. Die Verwendung von OpenWhiskServices ermöglicht dem QuantumService die parallele Nutzung unterschiedlicher OpenWhisk-Dienste und Namespaces. So kann beispielsweise jeder Nutzer seinen eigenen OpenWhisk-Server über die Infrastruktur eines Cloud-Anbieters hosten und im QuantumService als OpenWhiskService registrieren. Neben der Bereitstellung eigener Server können auch die von Cloud-Anbietern zur Verfügung gestellten FaaS-Angebote, welche auf OpenWhisk basieren, im QuantumService registriert und verwendet werden.

[11] https://hibernate.org/
[12] https://spring.io/projects/spring-data-jpa
[13] https://hub.docker.com/r/ibmcom/mq
[14] https://cloud.ibm.com/catalog/services/mq
[15] https://github.com/ibm-messaging/mq-jms-spring

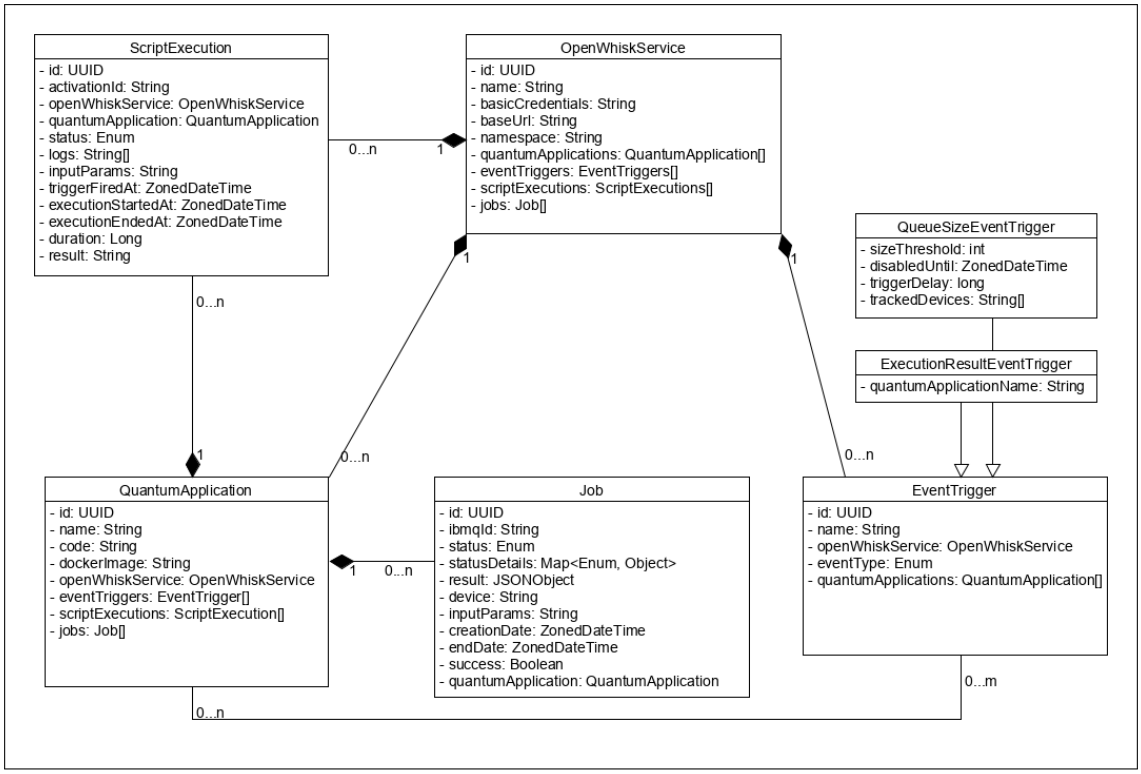

Abbildung 5.2: Klassendiagramm des QuantumService

5.3.2 QuantumApplication

Von Nutzern geschriebene Quantenanwendungen werden im QuantumService durch QuantumApplications repräsentiert. Eine QuantumApplication gehört immer zu genau einer Action eines OpenWhiskServices. Die Verknüpfung der beiden Objekte wird durch die Nutzung des identischen Namens erreicht, welcher sowohl im QuantumService als auch beim OpenWhiskService eindeutig ist. Eine QuantumApplication besteht aus einem eindeutigen Namen, dem Quellcode und dem Namen eines auf Dockerhub veröffentlichten Docker-Images. Dieses Docker-Image muss dabei eine Python-Laufzeitumgebung mit installiertem Qiskit enthalten, damit Quantenanwendungen auf einem von IBMs Quantencomputern ausgeführt werden können. Um durch Ereignisse ausgeführt zu werden, lassen sich QuantumApplications mit beliebig vielen EventTriggern verknüpfen. Zuletzt gehören zu einer QuantumApplication beliebig viele ScriptExecutions und Jobs, welche bei der Ausführung einer Quantenanwendung erzeugt werden.

5.3.3 ScriptExecution

Eine ScriptExecution ist eine für den QuantumService angepasste Repräsentation einer Activation (siehe Abschnitt 3.2.6), welche bei der Ausführung einer Action erzeugt wird und dementsprechend immer zu einer QuantumApplication und einem OpenWhiskService gehört. Die Verknüpfung zu einer Activation wird über eine durch den OpenWhiskService erzeugte *activationId* hergestellt. Eine ScriptExecution speichert Informationen, welche bei der Ausführung einer Quantenanwendung

entstehen. Zu diesen Informationen gehören ein Status, Ausführungslogs, ausführungsbezogene Zeitstempel, genutzte Eingabeparameter sowie das Ausführungsergebnis, welches nicht mit dem Ergebnis der Ausführung eines Jobs auf einem Quantencomputer verwechselt werden darf. Sollte die Ausführung einer Quantenanwendung fehlschlagen, werden die Fehlermeldungen in die Ausführungslogs gespeichert. Diese sollen Entwicklern helfen der Fehlerursache auf den Grund zu gehen.

5.3.4 Job

Bei einem Job handelt es sich um eine Repräsentation des gleichnamigen IBM Quantum Objekts, welches in Abschnitt 3.3 bereits erwähnt wurde und im System als *IBMQ-Job* bezeichnet wird. Als IBMQ-Job bezeichnet IBM Quantum die Ausführung einer Quantenschaltung auf einem ihrer Quantencomputer. Deswegen gehören die vom QuantumService verwalteten Jobs immer zu genau einer QuantumApplication. Wie bei einer ScriptExecution wird über eine spezielle *ibmqId* die Verbindung zum externen IBMQ-Job hergestellt. Im Vergleich zu einem IBMQ-Job enthält das vom QuantumService verwaltete Job-Objekt jedoch nur Informationen, welche für den QuantumService relevant sind. Zu diesen Informationen gehören ausführungsbezogene Zeitstempel, Statusdetails und das vom Quantencomputer erzeugte Ergebnis. Darüber hinaus besitzen die Jobs des QuantumService auch exklusive Informationen, wie die für die Ausführung genutzten Eingabeparameter oder den Namen des genutzten Quantencomputers.

5.3.5 EventTrigger

Ein EventTrigger repräsentiert immer genau einen Trigger eines im QuantumService registrierten OpenWhiskServices. Wie bei den QuantumApplications und Actions wird die Verknüpfung zwischen Trigger und EventTrigger durch die Nutzung des gleichen Namens erreicht. Bei einem EventTrigger handelt es sich demnach wie bei einem Trigger um ein Objekt, welches zu einem bestimmten Ereignis-Feed gehört und einen eindeutigen Namen besitzt. Im Vergleich zu einem Trigger gibt es EventTrigger unterschiedlicher Arten. Jede Art von EventTrigger hört dabei auf spezielle Feeds, welche zusätzlich über spezielle Eigenschaften individuell konfiguriert werden können. So reagieren *QueueSizeEventTrigger* nur auf Warteschlangengröße-basierende Ereignisse, während *ExecutionResultEventTrigger* nur durch Ergebnisse bereits ausgeführter Quantenanwendungen ausgelöst werden. Zum Schluss gibt es die gewöhnlichen, nicht weiter konfigurierbaren EventTrigger. Diese haben keine speziellen Eigenschaften und unterscheiden sich daher nur in ihrem Namen. Die gewöhnlichen EventTrigger haben im QuantumService keine große Bedeutung. Sie können jedoch von externen Systemen für die Umsetzung neuer Ereignis-Feeds genutzt werden, ohne den Quellcode des QuantumService anpassen zu müssen.

5.4 Anbindung genutzter Plattformen

In den folgenden Abschnitten wird beschrieben, wie die Interaktion zwischen QuantumService und OpenWhisk bzw. IBM Quantum ermöglicht wird.

```
{
    "apiHost": "https://api.quantum-computing.ibm.com/v2",
    "apiToken": "someIbmqApiToken",
    "accessToken": null,
    "tokenExpiry": null
}
```

Listing 5.1: Initiale IBMQ-Properties

5.4.1 Anbindung von IBM Quantum

IBM Quantum erfüllt im System eine Vielzahl von essenziellen Funktionen. Zum einen ist es die Ereignisquelle, welche Ereignis-Feeds für die Ausführung von Quantenanwendungen liefert. Zum anderen ist es der Cloud-Dienst für Quantencomputer, auf welchen die von Quantenanwendungen generierten Quantenschaltungen ausgeführt und deren Ergebnisse beschafft werden. Für die Umsetzung dieser Funktionen müssen der QuantumService und die einzelnen Quantenanwendungen mit IBM Quantum interagieren können. Um das zu ermöglichen, muss ein Account angelegt und der für die Interaktion benötigte API-Token generiert werden.

Der für den QuantumService generierte API-Token wird zu den Umgebungsvariablen hinzugefügt und von dort aus beim Start der Anwendung in ein *Spring-Bean*[16] übertragen, welches in allen Klassen und Methoden des QuantumServices injiziert und verwendet werden kann. Das initial erstellte Spring-Bean, welches als *IBMQProperties* bezeichnet wird, ist in Listing 5.1 abgebildet. Neben dem *apiToken* beinhalten die initialen IBMQProperties ein *apiHost*-Feld, welches die Heimat-URL der IBM Quantum API enthält. Dieses wird wie das apiToken-Feld aus den Umgebungsvariablen in die IBMQProperties übertragen.

Während mit Qiskit geschriebene Quantenanwendungen für die Ausführung von Quantenschaltungen auf Quantencomputern lediglich den API-Token benötigen, kommt der QuantumService nicht allein mit diesem aus. Er benötigt für die Kommunikation mit IBM Quantums REST-API einen zusätzlichen, kurzlebigen Access-Token, welcher mithilfe des API-Tokens generiert und im *accessToken*-Feld gespeichert wird. Er hat eine Lebenszeit von etwa 20 Minuten, sodass der Zeitpunkt zu welchem dieser abläuft zusätzlich vermerkt werden muss. Der Zeitstempel des Ablaufdatums wird hierfür im *tokenExpiry*-Feld gespeichert.

Für die Generierung und Erneuerung des Access-Tokens wird die aspektorientierte Programmierung[17] verwendet. Hierfür kommt ein *TokenChecker*-Aspekt zum Einsatz, welcher vor der Ausführung einer Methode, welche mit der IBM Quantum API interagiert, den aktuellen Access-Token überprüft und bei Bedarf auch erneuert. Der vom TokenChecker durchgeführte Prozess wird in Abbildung 5.3 mithilfe eines Sequenzdiagramms verdeutlicht. Der TokenChecker generiert zunächst den aktuellen Zeitstempel. Sollte das accessToken-Feld leer sein oder der tokenExpiry-Zeitstempel in der Vergangenheit liegen, so wird über eine spezielle Schnittstelle der IBM Quantum API ein neuer Access-Token mittels des API-Tokens generiert. Der in der Antwort enthaltene Access-Token wird im accessToken-Feld der IBMQProperties gespeichert, während der tokenExpiry-Zeitstempel um 15 Minuten in die Zukunft gestellt wird. Der tokenExpiry-Zeitstempel wird hierbei bewusst

[16] https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-definition
[17] https://docs.spring.io/spring-framework/docs/2.5.x/reference/aop.html

auf 15 Minuten abgerundet, damit sichergestellt ist, dass der accessToken-Wert immer aktuell ist. Die nun aktualisierten IBMQProperties werden automatisch in der Methode genutzt, welche im Anschluss die Endpunkte der IBM Quantum API abfragt.

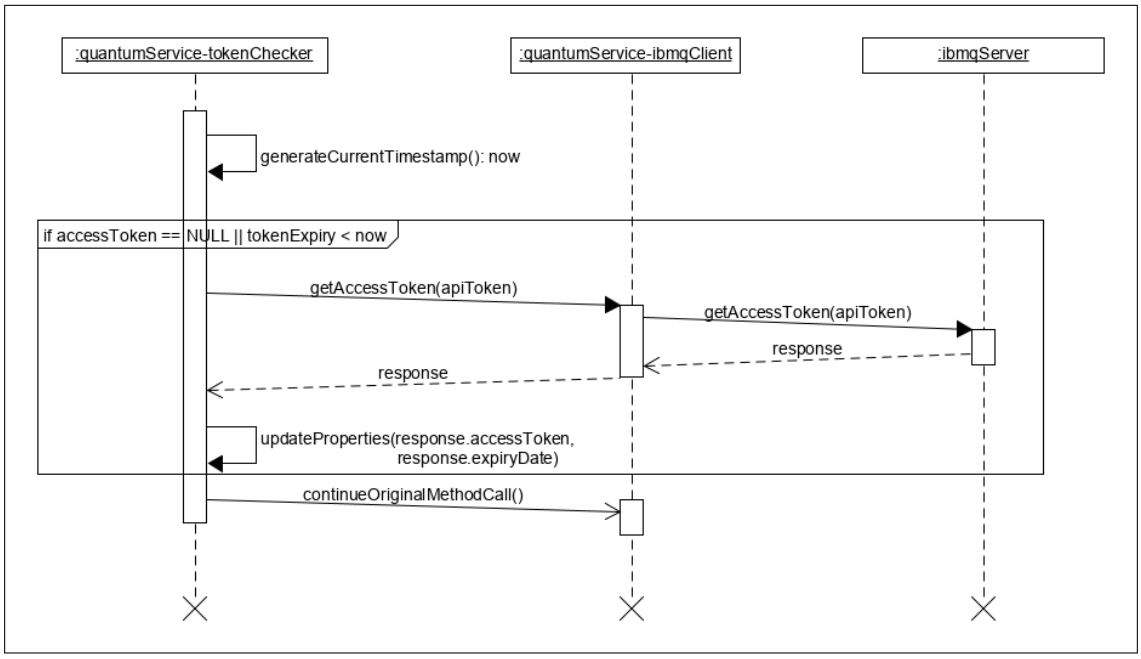

Abbildung 5.3: Verwaltung des Access-Tokens

5.4.2 Anbindung von OpenWhisk

Um Quantenanwendungen im System anmelden zu können, muss zunächst ein mit OpenWhisk laufender FaaS-Dienst registriert werden, auf welchem diese bereitgestellt und später durch Ereignisse ausgeführt werden können. Wie in Abschnitt 5.3.1 bereits erwähnt, können OpenWhisk-Dienste durch die Nutzung von OpenWhiskServices dynamisch zum System hinzugefügt oder entfernt werden.

Für die Verwaltung von OpenWhiskServices stellt der QuantumService seinen Akteuren spezielle Schnittstellen zu Verfügung, welche über HTTP-Anfragen angesprochen werden können. Um einen OpenWhiskService zu erstellen, wird eine passende POST-Anfrage an den QuantumService gestellt. Die für die Erstellung des OpenWhiskServices nötigen Daten werden hierbei im Body der Nachricht übergeben, welcher nach der im Listing 5.2 abgebildeten Vorlage strukturiert sein muss. Beim Empfang der Anfrage wird der Inhalt des Bodys zunächst validiert, bevor die *basicCredentials* als ein Base64-String verschlüsselt werden. Aus den Daten wird im Anschluss ein OpenWhiskService-Objekt erzeugt, welches in der Datenbank abgespeichert wird.

Im Vergleich zur Erstellung, erfolgt die Löschung eines OpenWhiskServices mithilfe einer DELETE-Anfrage, welche seinen eindeutigen Namen enthält. Der QuantumService kann diesen Namen nutzen, um den passenden OpenWhiskService in der Datenbank zu finden und zu löschen. Bei der Löschung eines OpenWhiskServices wird die Löschoperation an die zu ihm gehörenden Entitäten

```
{
    "name": "MyOpenWhisk",
    "basicCredentials": "username:password",
    "baseUrl": "http://localhost:3233/api/v1",
    "namespace": "MyNamespace"
}
```

Listing 5.2: Beispiel eines OpenWhiskService-Body

```
from qiskit import IBMQ, transpile
from qiskit.circuit.random import random_circuit

def main(params):
    try:
        provider = IBMQ.enable_account(params['apiToken'])
        backend = provider.get_backend(params['device'])
        qx = random_circuit(num_qubits=5, depth=4)
        transpiled = transpile(qx, backend=backend)
        job = backend.run(transpiled)
    finally:
        IBMQ.disable_account()

    return {
        "jobId": job.job_id()
    }
```

Listing 5.3: Funktion einer beispielhaften Quantenanwendung

kaskadiert, sodass auch diese gelöscht werden. Neben den Erstell- und Löschoperationen können für die Anzeige von verfügbaren OpenWhiskServices zusätzlich verschiedene GET-Anfragen genutzt werden.

5.5 Integration von Quantenanwendungen

In den folgenden Abschnitten wird beschrieben, wie von Nutzern geschriebene Quantenanwendungen in das entwickelte System integriert werden können.

5.5.1 Anforderungen an die Code-Struktur

Damit die bei einem OpenWhiskService als Actions bereitgestellten Quantenanwendungen mit dem QuantumService interagieren können, muss wie in Abschnitt 4.6 bereits erwähnt, ihr Quellcode nach einer bestimmten Vorlage geschrieben werden. Um den Entwicklern das Schreiben ihrer Quantenanwendungen zu erleichtern, wurde eine einfach zu befolgende Vorlage entwickelt, welche möglichst wenige Anforderungen an den geschriebenen Quellcode setzt. Listing 5.3 zeigt eine beispielhafte Quantenanwendung, welche dieser Vorlage folgt.

Die beim QuantumService angemeldeten Quantenanwendungen müssen alle als Funktionen geschrieben werden, welche dem von OpenWhisk vorausgesetzten Format folgen. Von OpenWhisk ausgeführte Funktionen müssen alle „main“ genannt werden und einen einzigen Eingabeparameter annehmen, welcher wie der Rückgabewert aus einer beliebigen Menge an Schlüssel-Wert-Paaren besteht [Apab]. Um Quantenanwendungen in den Workflow des QuantumServices integrieren zu können, muss die entwickelte Vorlage das von OpenWhisk vorgegebene Format erweitern.

Der vom QuantumService übergebene Eingabeparameter besitzt bei jedem Aufruf mindestens zwei Felder mit den Schlüsseln *device* und *apiToken*. Diese können von der Funktion genutzt werden, um sich bei IBM Quantum zu authentifizieren und die generierte Quantenschaltung auf dem im device-Feld angegebenen Quantencomputer auszuführen. Falls eine Funktion den device-Parameter nicht nutzen will, muss sie dessen Wert selbstständig beschaffen. So könnte sie beispielsweise HTTP-Anfragen durchführen, um diesen von externen Quellen zu laden. Alternativ kann der Wert als Konstante in der Funktion hartkodiert werden. Im Vergleich dazu ist die Funktion verpflichtet den über Eingabeparameter übergebenen API-Token zu verwenden, da die Ausführung sonst über einen IBM Quantum Account durchgeführt wird, welcher nicht zum QuantumService gehört. Dadurch wäre dieser nicht in der Lage, den Status von laufenden Ausführungen abzufragen oder Ergebnisse zu beschaffen.

Während die apiToken- und device-Werte bei jedem Aufruf vorhanden sind, gibt es zusätzliche Felder die nur beim Eintritt von bestimmten Ereignissen übergeben werden. So wird beim Eintritt von ergebnisbasierten Ereignissen jeder dafür registrierten Quantenanwendung das Ergebnis einer anderen im als JSON-String formatierten *result*-Feld übergeben. Die Aufgabe der Entwickler von Quantenanwendungen ist es, beim Schreiben ihrer Funktionen immer zu beachten, dass sie alle Eingaben der Ereignis-Feeds verarbeiten können, zu denen sie registriert sind.

Wie in der im Listing 5.3 abgebildeten Beispielfunktion zu sehen ist, muss die Quantenanwendung nach ihrer Ausführung die Job-ID zurückgeben, welche von IBM Quantum beim Einreichen einer Ausführungsanfrage generiert wird. Diese ID wird im QuantumService für das Beschaffen des Ausführungsergebnisses benötigt. Bei der Rückgabe der Job-ID müssen Entwickler immer darauf achten, dass ihre Funktionen ein Objekt zurückgeben, welches ein Schlüssel-Wert-Paar mit dem Schlüssel *jobId* besitzt.

Da alle im System angemeldeten Quantenanwendungen den gleichen, vom QuantumService zur Verfügung gestellten API-Token verwenden, um sich mit IBM Quantum zu verbinden und Ausführungsanfragen zu stellen, muss darauf geachtet werden, dass die dabei gestartete Session zum Schluss geschlossen wird. Deswegen muss der Funktionscode in einen *try-finally* Block geschrieben werden, damit die Schließung der Session in allen Fällen stattfindet. Die Schließung von Sessions ist wichtig, da OpenWhisk die Docker-Container nach der Ausführung einer Funktion eine Zeit lang warm laufen lässt und alle Funktionen im gleichen Python-Prozess ausführt. Schließt eine Quantenanwendung die Session nicht, wird die nächste Funktion, welche im gleichen Docker-Container ausgeführt werden kann, versuchen, eine neue Session mit dem gleichen API-Token zu starten. Dabei wird Qiskit einen Fehler werfen, da eine Session mit dem gleichen Token bereits läuft und die Ausführung der Quantenanwendung wird deswegen fehlschlagen.

5.5.2 Bereitstellung nötiger Laufzeitumgebungen

Da OpenWhisk keine Laufzeitumgebungen anbietet, um mit Qiskit geschriebene Quantenanwendungen auszuführen, müssen wie in Abschnitt 3.2.3 erwähnt, nutzergenerierte Docker-Images für die Bereitstellung nötiger Laufzeiten zum Einsatz kommen. Diese müssen vor der Anmeldung und Bereitstellung neuer Quantenanwendungen erstellt und auf Dockerhub veröffentlicht werden, damit die einzelnen OpenWhiskServices in der Lage sind, diese auszuführen. Um Nutzern diesen Prozess zu ersparen, bietet der QuantumService ein Docker-Image an, welches standardmäßig den OpenWhiskServices für die Ausführung der bereitgestellten Quantenanwendungen übergeben wird. Dieses Image sorgt dafür, dass der für die Ausführung gestartete Docker-Container mit einer Python-Laufzeit und vorinstalliertem Qiskit-SDK ausgestattet ist. Sollten die Quantenanwendungen von Nutzern weitere Bibliotheken in ihrer Laufzeit benötigen, haben sie weiterhin die Möglichkeit, bei der Anmeldung ihrer Quantenanwendungen eigene Docker-Images zu verwenden.

5.5.3 Anmeldung und Bereitstellung

Abbildung 5.4 zeigt ein Sequenzdiagramm, auf welchem abgebildet ist, wie Quantenanwendungen beim QuantumService angemeldet und über einen OpenWhiskService bereitgestellt werden. Für die Anmeldung einer Quantenanwendung wird eine Multipart-POST-Anfrage an den QuantumService gestellt. Die Anfrage besteht dabei aus den Teilen *file*, *actionName*, *dockerImage* und *owServiceName*. Beim file handelt es sich um eine Python-Datei, welche den Code der Quantenanwendung enthält. Beim Empfang werden die einzelnen Teile vom QuantumService validiert, bevor aus ihnen ein QuantumApplication-Objekt generiert wird. Dafür wird die Python-Datei für die spätere Nutzung beim OpenWhisk-Dienst in einen Base64-String kodiert und im code-Feld der QuantumApplication gespeichert. Der owServiceName wird hingegen genutzt, um das dazugehörende OpenWhiskService-Objekt aus der Datenbank zu laden und in die QuantumApplication zu setzen. Im Vergleich zu den anderen Werten ist das dockerImage-Feld optional, da bei seiner Abwesenheit das im letzten Abschnitt erwähnte Docker-Image als Standardwert zum Einsatz kommt. Nachdem das QuantumApplication-Objekt mit allen Daten befüllt wurde, wird es in die Datenbank eingetragen. Zum Schluss wird aus der QuantumApplication ein Action-Objekt generiert, welches im Body einer PUT-Anfrage an den OpenWhiskService mithilfe der im OpenWhiskService-Objekt enthaltenen Informationen geschickt wird. Dadurch wird beim OpenWhiskService eine Action bereitgestellt, welche ab diesem Zeitpunkt in der Laufzeitumgebung des übergebenen Docker-Images ausgeführt werden kann. Dieser komplette Prozess wird in einer Transaktion ausgeführt, sodass der QuantumService und der involvierte OpenWhiskService jederzeit synchronisiert bleiben. Nach der erfolgreichen Anmeldung können die QuantumApplications durch unterschiedliche GET-Anfragen ausgelesen werden.

5.5.4 Registrierung für Ereignis-Feeds

Im System angemeldete Quantenanwendungen können für beliebige Ereignis-Feeds registriert werden. Der Registrierungsprozess ist in Abbildung 5.5 in Form eines Sequenzdiagramms dargestellt. Die Registrierung für Ereignis-Feeds erfolgt durch das Verknüpfen einer QuantumApplication mit einem existierenden EventTrigger. Hierfür bietet der QuantumService einen speziellen Endpunkt an, welcher mit einer POST-Anfrage aufgerufen werden kann. Die einzigen für die Verknüpfung notwendigen Daten sind die eindeutigen Namen der QuantumApplication und des EventTriggers. Der

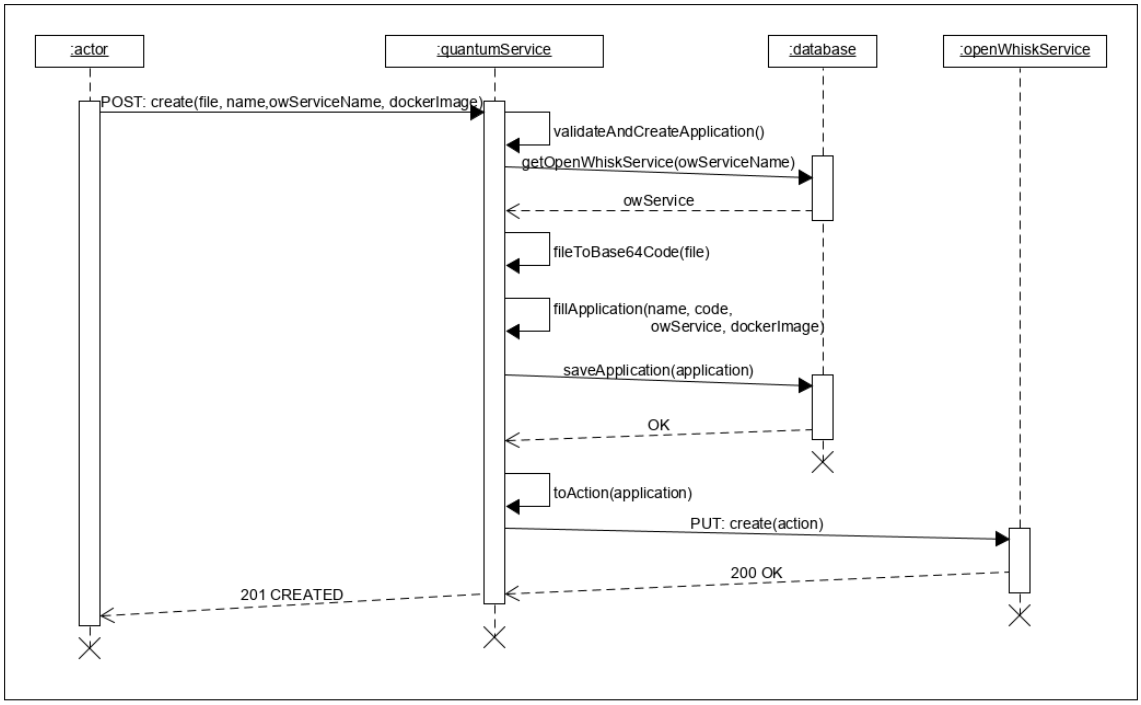

Abbildung 5.4: Anmeldungsprozess einer Quantenanwendung

QuantumService nutzt die beiden Namen, um die passenden Objekte aus der Datenbank auszulesen. Für die Verknüpfung der beiden Objekte wird die QuantumApplication in das EventTrigger-Objekt gesetzt, bevor dieses wieder in der Datenbank abgespeichert wird. Zum Schluss wird eine Rule beim OpenWhiskService erstellt, welche die zur QuantumApplication gehörende Action mit dem zum EventTrigger gehörendem Trigger verknüpft. Der Name der erstellten Rule setzt sich dabei aus den Namen der beiden Objekte zusammen, welche zusätzlich mit einem Bindestrich getrennt werden. Haben beispielsweise die QuantumApplication und der EventTrigger die Namen „HelloAction“ bzw. „HelloTrigger“, wird eine Rule mit den Namen „HelloAction-HelloTrigger“ erstellt. Wie bei der Anmeldung von Quantenanwendungen wird hier auf Transaktionen gesetzt, damit OpenWhiskService und QuantumService stets synchronisiert bleiben.

5.5.5 Deregistrierung und Löschung

Um eine QuantumApplication von einem Ereignis-Feed zu trennen, muss die Verknüpfung zwischen ihr und dem entsprechenden EventTrigger gelöst werden. Dafür bietet der QuantumService einen speziellen DELETE-Endpunkt an, welcher mithilfe der Namen der beiden involvierten Objekte aufgerufen werden kann. Wird der Endpunkt aufgerufen, nutzt der QuantumService die beiden Namen, um die QuantumApplication aus dem EventTrigger-Objekt zu entfernen. Zusätzlich wird in einer Transaktion die nun nicht mehr benötigte Rule mit einer separaten DELETE-Anfrage vom OpenWhiskService entfernt.

Quantenanwendungen lassen sich nicht nur von Feeds abmelden, sondern auch vollständig aus dem System nehmen. Dafür wird eine DELETE-Anfrage mit dem eindeutigen Namen der Quantenanwendung an den QuantumService gestellt. Dieser nutzt den Namen, um das richtige QuantumApplication-

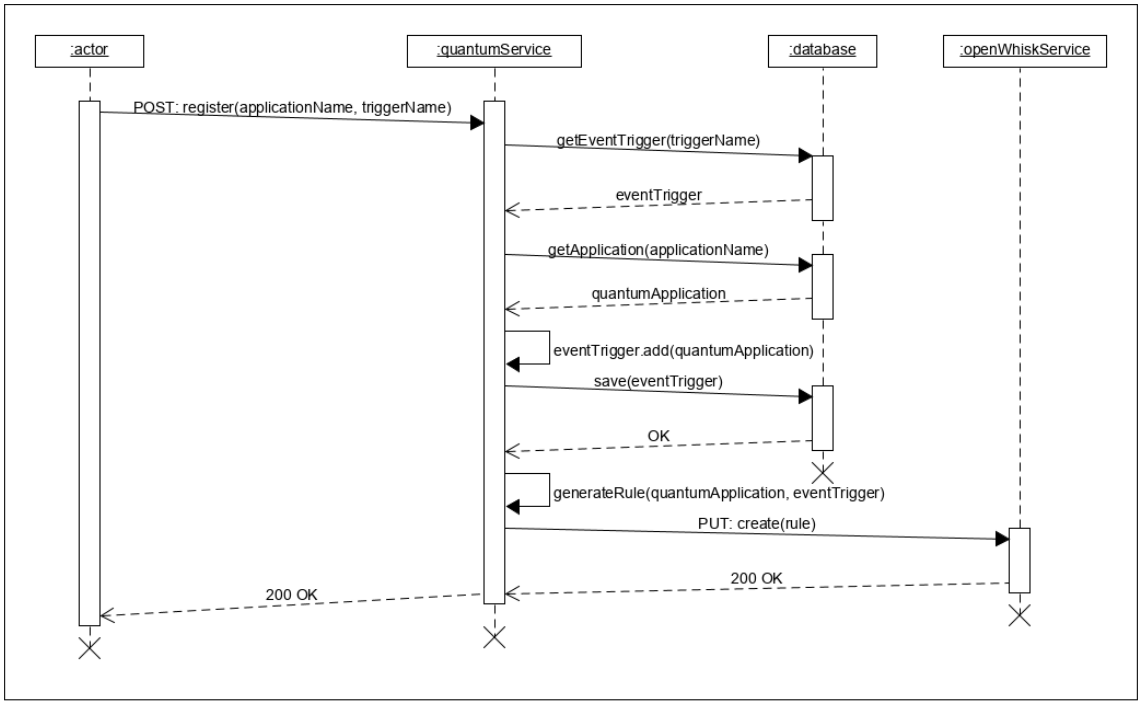

Abbildung 5.5: Feed-Registrierung einer Quantenanwendung

Objekt aus der Datenbank zu laden. Bevor eine Quantenanwendung gelöscht werden darf, muss sie von allen Feeds abgemeldet werden, wie im letzten Absatz bereits erklärt wurde. Darüber hinaus muss diese Löschoperation auch vom OpenWhiskService durchgeführt werden, damit die zur Quantenanwendung gehörende Action ebenfalls gelöscht wird. Hierfür wird in der gleichen Transaktion eine DELETE-Anfrage mit den eindeutigen Namen der Quantenanwendung an den OpenWhiskService gestellt. Zum Schluss wird die QuantumApplication aus der Datenbank entfernt. Bei der Löschung einer QuantumApplication werden auch alle zu ihr gehörenden Objekte aus der Datenbank gelöscht. In diesem Fall würde die Löschoperation an die zur gelöschten QuantumApplication gehörenden Jobs und ScriptExecutions kaskadieren. Im Gegensatz zu den Actions und Triggern bietet OpenWhisk keine Schnittstellen für die Löschung von Activations an. Sie werden von OpenWhisk nach einer bestimmten Zeit automatisch gelöscht, sodass sich der QuantumService im Bezug auf ScriptExecutions nicht mit den OpenWhiskServices synchronisieren kann und auch nicht muss.

5.5.6 Ergebnisbeschaffung

Für die ereignisbasierte Ausführung der Quantenanwendungen wird ein zum Ereignis passender Trigger ausgelöst, welcher alle mit ihm verknüpften Actions ausführt. Dabei antwortet OpenWhisk mit einer Activation-ID, welche die Trigger-Activation eindeutig identifiziert. Abbildung 5.6 zeigt wie mithilfe der Trigger-Activation passende ScriptExecutions generiert werden. Nach der Auslösung eines Triggers wird in der vom OpenWhiskService enthaltenen Antwort die Activation-ID für die Beschaffung des Activation-Objekts genutzt. Wie in Abschnitt 3.2.6 bereits gezeigt wurde, enthalten die Logs der Trigger-Activation die Activation-IDs und Namen aller ausgeführten Actions,

welche eine nach der anderen in ScriptExecutions umgewandelt werden. Dafür wird zunächst der Action-Name genutzt, um die dazugehörende QuantumApplication aus der Datenbank auszulesen. Im Anschluss wird aus der Activation-ID der ausgeführten Action, der QuantumApplication, des für die Ausführung genutzten OpenWhiskServices und den in der Trigger-Activation enthaltenen Eingabeparametern eine initiale ScriptExecution mit dem Status *RUNNING* generiert, welche in der Datenbank abgespeichert wird. Ein Beispiel einer initialen ScriptExecution ist in Listing 5.4 zu sehen. Da der ScriptExecution zunächst viele wichtige Daten fehlen, müssen diese nachträglich durch wiederholtes Abfragen des vom OpenWhiskService zur Verfügung gestellten Endpunkts beschafft werden.

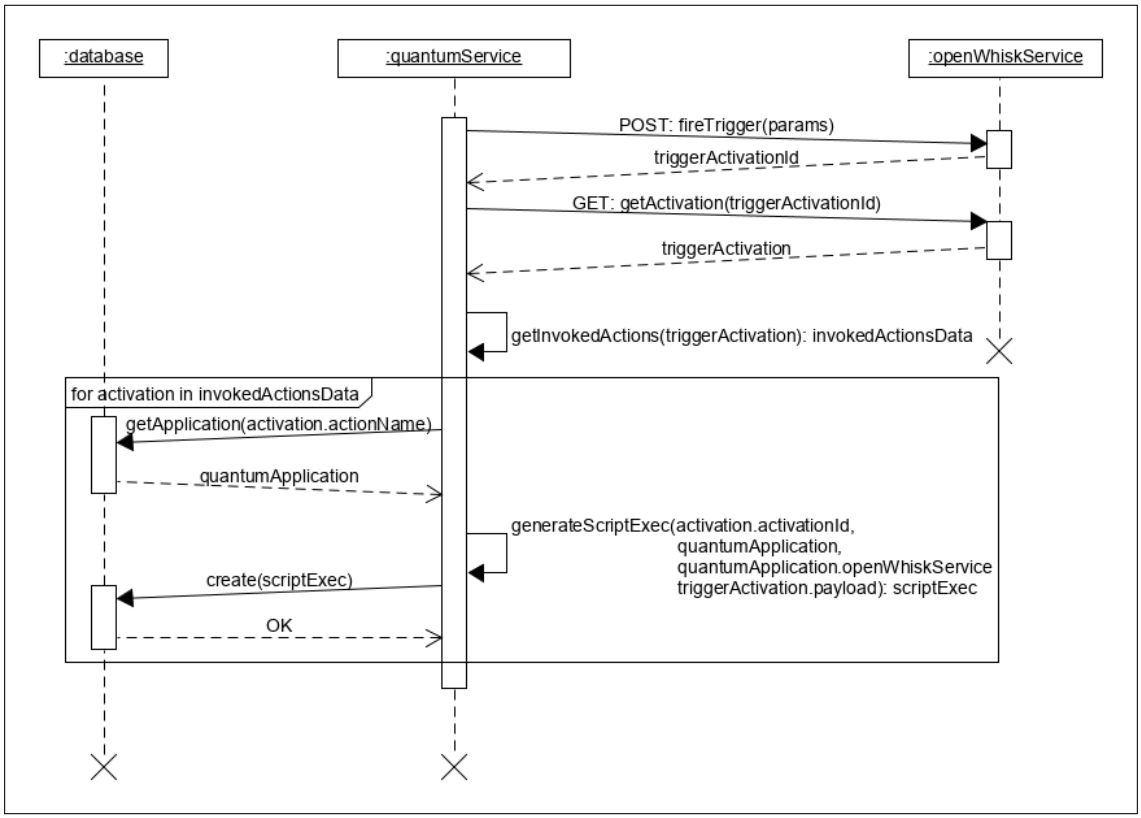

Abbildung 5.6: Generierung von ScriptExecutions

Die dafür zuständige QuantumService-Komponente wird als *ScriptExecutionChecker* bezeichnet. Der ScriptExecutionChecker arbeitet nach einem strikten Zeitplan und führt seine Operationen in regelmäßigen Abständen in einem im Hintergrund laufenden Prozess automatisch aus. Die Abstände, in welchen die vom ScriptExecutionChecker durchgeführten Operationen ausgeführt werden, lassen sich vor dem Start des QuantumService beliebig einstellen. Jede vom ScriptExecutionChecker durchgeführte Ausführung wird als *Zyklus* bezeichnet.

Abbildung 5.7 zeigt einen Zyklus des ScriptExecutionChecker. Im ersten Schritt lädt der ScriptExecutionChecker alle ScriptExecutions mit dem Status *RUNNING* aus der Datenbank. Diese werden im Anschluss eine nach der anderen bearbeitet. Hierfür wird zunächst mittels der gespeicherten Activation-ID eine Anfrage an den OpenWhiskService gestellt. Ist die Ausführung der Action nicht abgeschlossen, endet die Anfrage in einem Statuscode *404*, sodass die Bearbeitung der ScriptExe-

```
{
    "activationId": "externalOpenWhiskServiceActivationId",
    "status": "RUNNING",
    "quantumApplication": {
        "name": "QuantumApplicationTest",
        ...
    },
    "openWhiskService": {
        "name": "MyOpenWhiskService",
        ...
    },
    "inputParams": "{
        "apiToken": "*****",
        "device": "someIbmqDevice"
    }"
    "executionStartedAt": null,
    "executionEndedAt": null,
    "duration": null,
    "logs": [],
    "result": null,
}
```

Listing 5.4: Beispiel der initial erstellten ScriptExecution

cution in diesem Zyklus ausfällt. Anderenfalls antwortet der ScriptExecutionService mit der zur ausgeführten Action gehörenden Activation, welche alle fehlenden ScriptExecution-Informationen wie das Ergebnis enthält. Diese werden in die ScriptExecution übertragen, bevor ihr Status auf *COMPLETED* geändert und sie in der Datenbank abgespeichert wird.

Wie bereits erwähnt, enthält das ScriptExecution-Ergebnis nicht das Ergebnis des Quantencomputers, sondern lediglich eine Job-ID, mit welcher die laufende Job-Ausführung eindeutig identifiziert wird. Für die Beschaffung des Ergebnisses wird aus der Job-ID ein initiales Job-Objekt (siehe Listing 5.5) erstellt und in der Datenbank abgespeichert. Dabei werden notwendige Informationen aus der ScriptExecution übernommen, wie beispielsweise die Eingabeparameter, der für die Ausführung verwendete Quantencomputer und die ausgeführte Quantenanwendung. Wie bei den ScriptExecutions werden die fehlenden Daten durch wiederholtes Abfragen über eine *JobChecker*-Komponente beschafft.

Der Verlauf eines JobChecker-Zyklus ist in Abbildung 5.8 zu sehen. Der JobChecker beginnt seinen Zyklus mit dem Auslesen aller Jobs, die nicht den COMPLETED-Status erreicht haben. Diese arbeitet er der Reihe nach ab, indem er den Status des IBMQ-Jobs mithilfe der Job-ID über einen dafür vorgesehenen IBM Quantum Endpunkt abfragt. Die in der Antwort enthaltenen Daten des gelieferten IBMQ-Jobs werden in das vom QuantumService genutzte Job-Format umgewandelt und in das Job-Objekt übertragen. Da der Job-Endpunkt von IBM Quantum nicht die Ergebnisse abgeschlossener IBMQ-Jobs liefert, müssen diese über einen separaten Endpunkt abgerufen werden, sobald ein Job den COMPLETED-Status erreicht. Dieser Antwortet auf Anfragen mit einer Download-URL, welche das Ergebnis im JSON-Format liefert. Zum Schluss wird das beschaffte Ergebnis in das Job-Objekt gesetzt und der mit Daten befüllte Job wird wieder in der Datenbank abgespeichert.

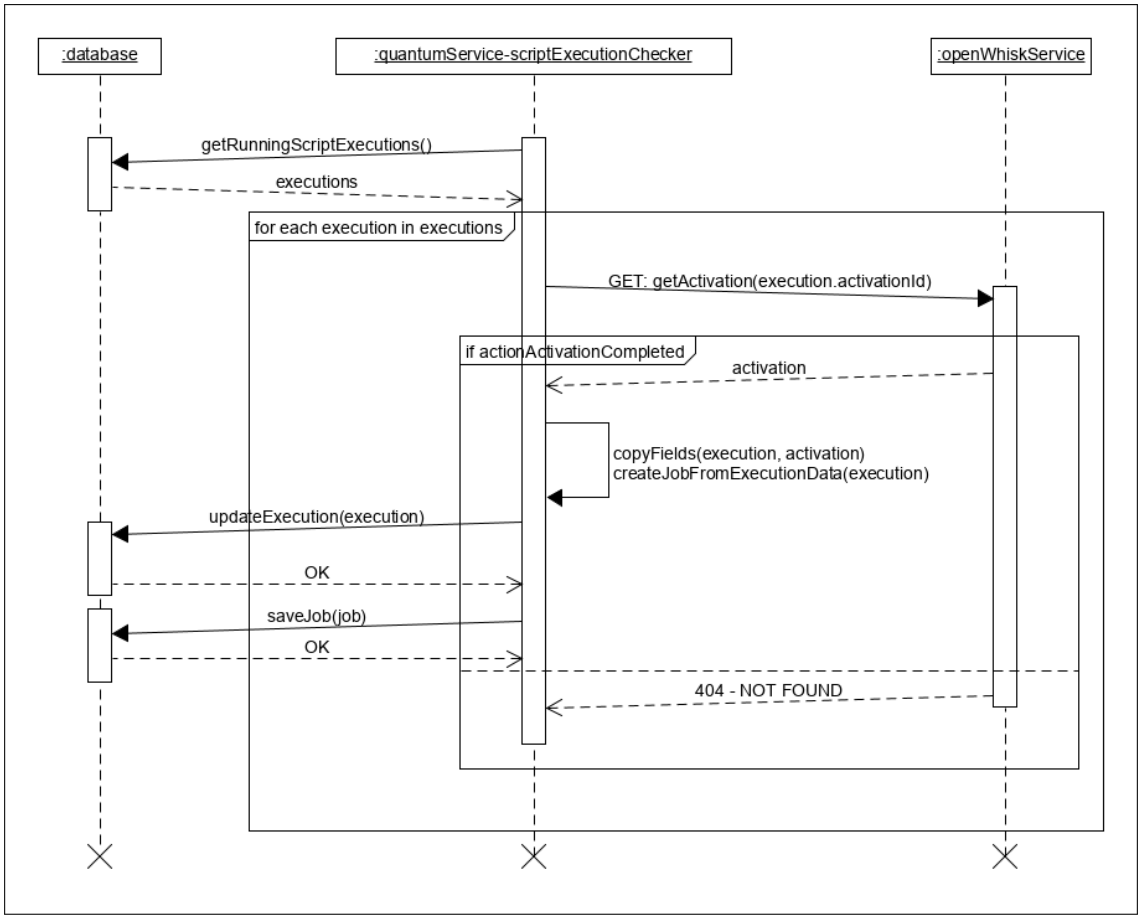

Abbildung 5.7: Zyklus des ScriptExecutionChecker

```
{
    "ibmqId": "externalIbmqJobId",
    "status": "CREATING",
    "device": "someIbmqDevice",
    "statusDetails: {},
    "quantumApplication": {
        "name": "QuantumApplicationTest",
        ...
    },
    "inputParams": "{
        "apiToken": "*****",
        "device": "someIbmqDevice"
    }",
    "success": null
    "creationDate": null,
    "endDate": null,
    "result": null,
}
```

Listing 5.5: Beispiel eines initial erstellten Job-Objekts

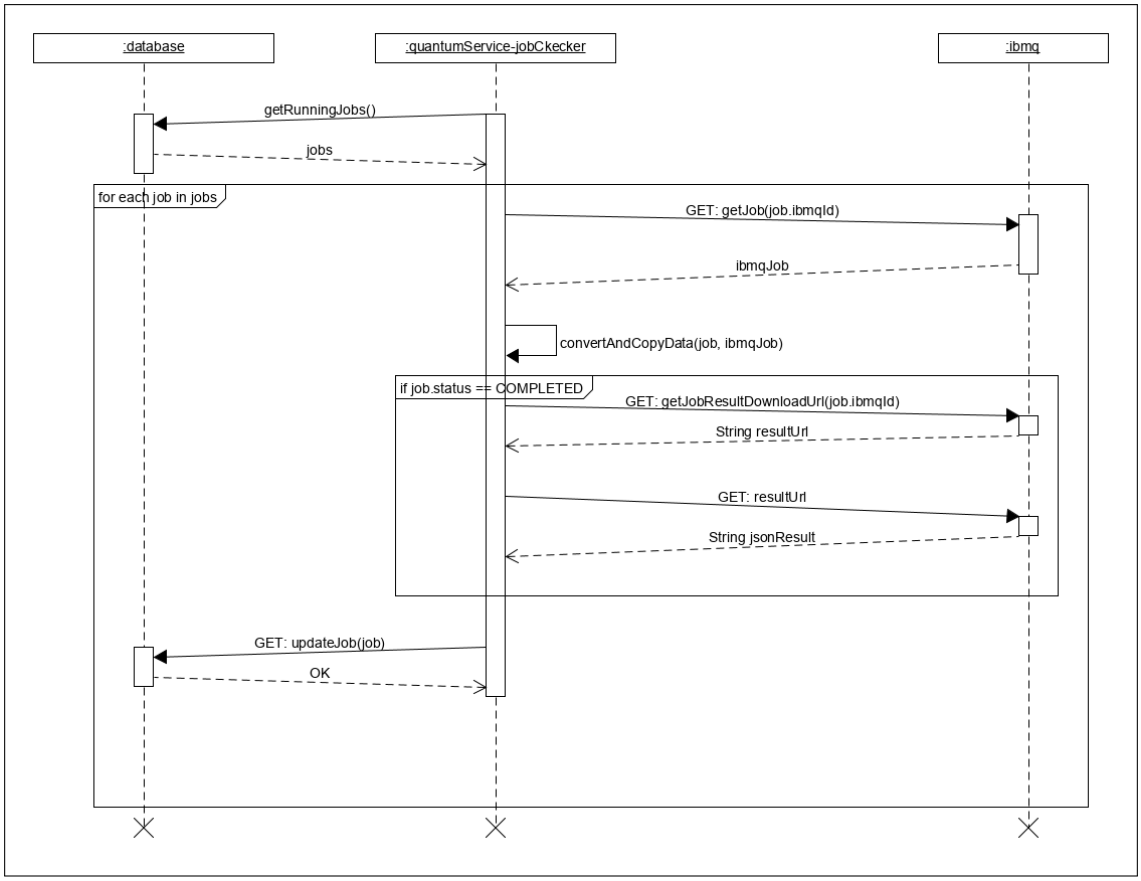

Abbildung 5.8: Zyklus des JobChecker

5.6 Einbindung von Ereignissen

In den folgenden Abschnitten wird im Detail beschrieben, wie der QuantumService die ereignisbasierte Ausführung ermöglicht.

5.6.1 Steuerung und Konfiguration von Ereignis-Feeds

OpenWhisk setzt für die Steuerung und Konfiguration von Feeds standardmäßig auf sogenannte Feed-Actions (siehe Abschnitt 3.2.7), welche die von Nutzern manuell erstellten Trigger bei externen Services mit beliebiger Konfiguration registrieren. Diese Services können anschließend beim Eintritt von Ereignissen die registrierten Trigger auslösen. Im entwickelten System wird jedoch auf die manuelle Erstellung von Triggern und die Nutzung der Feed-Actions vollständig verzichtet. Stattdessen werden Trigger automatisch vom QuantumService erstellt, sobald ein neuer EventTrigger (siehe Abschnitt 5.3.5) zum System hinzugefügt wird. Da der QuantumService die Erstellung des Triggers bei einem als OpenWhiskService registrierten OpenWhisk-Dienst selbst übernimmt, weiß er automatisch über seine Existenz Bescheid, sodass eine zusätzliche Registrierung durch eine Feed-Action nicht nötig ist. Darüber hinaus gibt es verschiedene Arten von EventTriggern, welche zusätzlich konfiguriert werden können, sodass auch die zweite Funktion der Feed-Actions vom

```
{
    "name": "HelloTrigger",
    "eventType": "QUEUE_SIZE",
    "sizeThreshold": 25
    "trackedDevices": [
        "quantumComputerX",
        "quantumSimulatorY"
    ],
    "triggerDelay": 60 // in minutes
}
```

Listing 5.6: Beispiel eines EventTrigger-Bodys

QuantumService übernommen wird, was deren Nutzung obsolet macht. Der QuantumService kann somit über die EventTrigger den Feed eines Triggers konfigurieren und diesen auch beim Eintritt von passenden Ereignissen auslösen. Wie der QuantumService entscheidet, welche (Event)Trigger beim Eintritt von welchen Ereignissen ausgelöst werden, wird in den nächsten Abschnitten gezeigt.

In Abbildung 5.9 wird der Erstellungsprozess eines EventTriggers mithilfe eines Sequenzdiagramms dargestellt. Für die Erstellung wird eine POST-Anfrage an den QuantumService gestellt. Der Name des zu verwendenden OpenWhiskServices wird dabei als URL-Parameter übergeben, während alle EventTrigger-Informationen über den Body der Anfrage verschickt werden. Eine beispielhafte JSON-Repräsentation des Bodys ist in Listing 5.6 abgebildet. Beim Empfang der Anfrage deserialisiert der QuantumService den JSON-Body anhand des angegebenen *eventType* in das passende EventTrigger-Objekt. Die im Listing 5.6 angegebene JSON würde beispielsweise in ein QueueSizeEventTrigger-Objekt deserialisiert werden. Im Anschluss wird der im URL-Parameter übergebene *owServiceName* genutzt, um das entsprechende OpenWhiskService-Objekt aus der Datenbank auszulesen und in das EventTrigger-Objekt zu setzen. Das mit Daten befüllte EventTrigger-Objekt wird im Anschluss in der Datenbank abgespeichert. Um den Erstellungsprozess abschließen zu können, muss der EventTrigger zusätzlich beim OpenWhiskService in Form eines Triggers erstellt werden. Dazu wird das EventTrigger-Objekt in ein von OpenWhisk akzeptiertes Trigger-Objekt umgewandelt. Zum Schluss wird eine PUT-Anfrage an den OpenWhiskService abgeschickt, mit welcher die Erstellung des Triggers durchgeführt wird. Wie bei der Anmeldung von QuantumApplications wird der Prozess in einer Transaktion durchgeführt, sodass die involvierten Systeme konsistent gehalten werden. Erfolgreich erstellte EventTrigger können nach ihrer Erstellung über die vom QuantumService angebotenen GET-Endpunkte ausgelesen werden.

Sollte die durch einen EventTrigger gesteuerte Feed-Konfiguration nicht mehr benötigt werden, kann für die Löschung eine entsprechende DELETE-Anfrage an den dafür vorgesehenen Endpunkt des QuantumService gestellt werden. Die Löschung erfolgt über den eindeutigen Namen des EventTriggers, welcher als Variable über die Anfrage-URL übergeben wird. Wie bei der Löschung von QuantumApplications muss die Verknüpfung zu jeder mit dem EventTrigger verknüpften QuantumApplicaiton gelöst werden. Dafür werden alle nötigen Rules beim entsprechenden OpenWhiskService gelöscht, bevor der beim OpenWhiskService existierende und zum EventTrigger gehörende Trigger gelöscht wird.

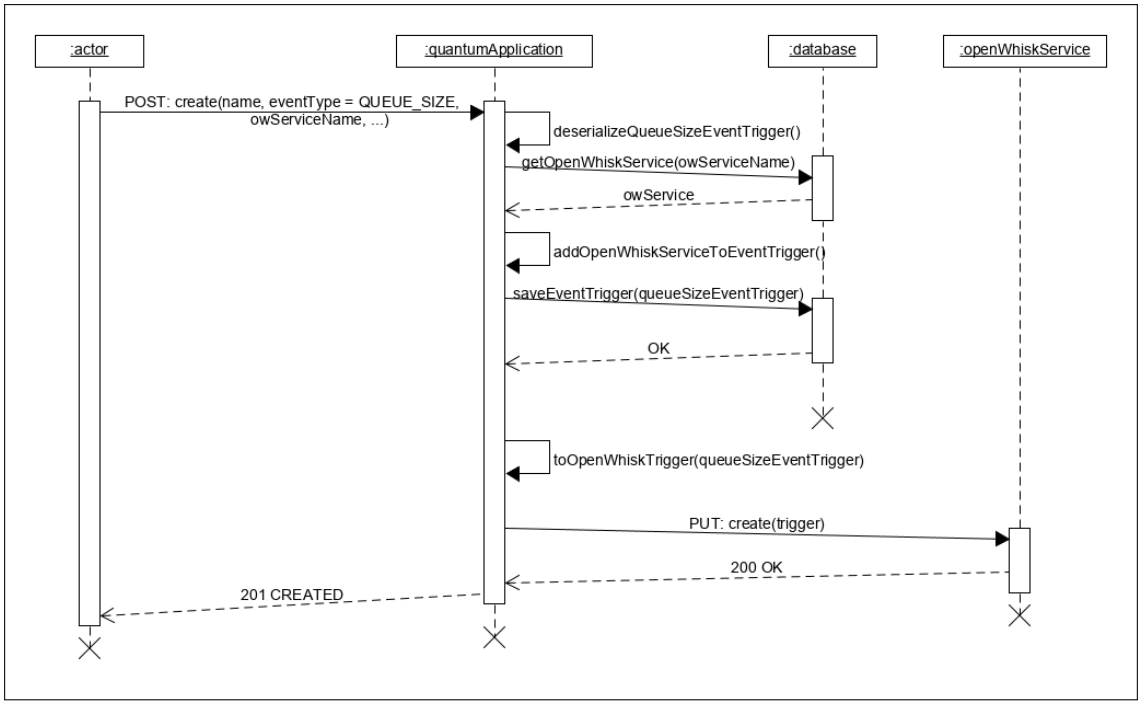

Abbildung 5.9: Erstellung eines QueueSizeEventTrigger

5.6.2 Warteschlangen-Feed

Der Warteschlangen-Feed liefert in regelmäßigen Abständen die Warteschlangengröße aller auf IBM Quantum verfügbarer Quantencomputer. Dabei werden QueueSizeEventTrigger ausgelöst und die mit ihnen verknüpften QuantumApplications ausgeführt. Um den Feed für jede Quantum-Application individuell zu gestalten, können QueueSizeEventTrigger unterschiedlich konfiguriert werden. Hierfür werden die exklusiven QueueSizeEventTrigger-Felder *trackedDevices*, *triggerDelay*, *disabledUntil* und *sizeThreshold* verwendet. Mit der sizeThreshold und den trackedDevices können Nutzer einstellen, wie leer die Warteschlangen welcher Quantencomputer sein müssen, damit der QueueSizeEventTrigger ausgelöst wird. Zusätzlich können Nutzer den Mindestabstand zwischen zwei Auslösungen mit den Feldern disableUntil und triggerDelay steuern. Beim disableUntil-Wert handelt es sich um einen Zeitstempel, welcher verhindert, dass ein QueueSizeEventTrigger vor einem bestimmten Zeitpunkt ausgeführt wird. Bei der Erstellung des QueueSizeEventTriggers wird dieser automatisch mit dem Erstellzeitpunkt initiiert. Der von Nutzern angegebene und optionale triggerDelay sorgt hingegen dafür, dass der disableUntil-Zeitstempel nach jeder Auslösung nach hinten verschoben wird. Ist der triggerDelay-Wert nicht verfügbar, wird der QueueSizeEventTrigger nach der ersten Auslösung gelöscht. Die QuantumService-Komponente, welche für die Umsetzung des Warteschlangen-Feeds zuständig ist, wird als *QueueSizeChecker* bezeichnet. Der QueueSizeChecker ist für das wiederholte Abfragen der Warteschlangen-Endpunkte aller verfügbaren Quantencomputer zuständig. Er funktioniert nach demselben Prinzip wie die bereits vorgestellten Job- und ScriptExecutionChecker (siehe Abschnitt 5.5.6).

Die grafische Darstellung eines QueueSizeChecker-Zyklus ist in Abbildung 5.10 zu sehen. Für das Auslösen von QueueSizeEventTriggern nutzt der QueueSizeChecker im ersten Schritt eine spezielle Schnittstelle von IBM Quantum, um alle verfügbaren Hubs abzufragen, da diese, wie

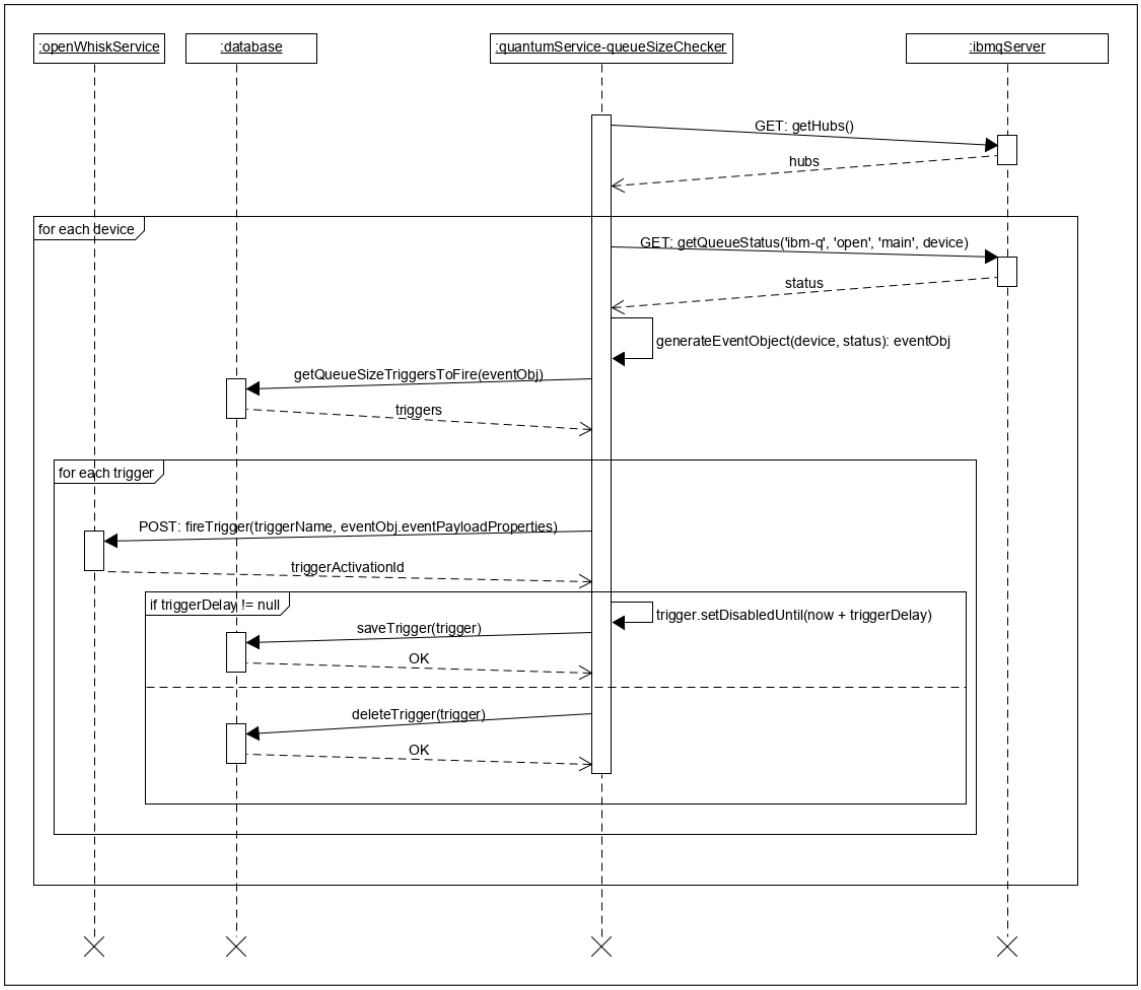

Abbildung 5.10: Generierungsprozess von Warteschlangenereignissen

in Abschnitt 3.3 bereits erwähnt, die verfügbaren Quantencomputer enthalten. Im zweiten Schritt werden die Quantencomputer dieser Hubs und der darin enthaltenen Gruppen und Projekte durchgegangen. In Abbildung 5.10 ist dieser Prozess etwas vereinfacht, da der in dieser Arbeit genutzte IBM Quantum Account nur den Zugriff auf die Hub *ibm-q*, die Gruppe *main* und das Projekt *open* hat. Im Anschluss wird die Warteschlange jedes Quantencomputers über einen dafür vorgesehenen Endpunkt abgefragt. Nach Erhalt der Warteschlangen-Informationen wird aus der Warteschlangengröße, dem Quantencomputer-Namen und den in den IBMQProperties enthaltenen API-Token ein Ereignisobjekt generiert. Ein Beispiel eines zum Warteschlangen-Feed gehörenden Ereignisobjekts ist in Listing 5.7 abgebildet. Der *eventType* wird dabei für die Unterscheidung der unterschiedlichen Ereignis-Arten genutzt. Die *additionalProperties* enthalten zum eventType gehörende Eigenschaften. So enthalten *QUEUE_SIZE*-Ereignisse beispielsweise die aktuelle Warteschlangengröße eines Quantencomputers. Zuletzt besitzt das Ereignisobjekt die als *eventPayloadProperties* bezeichneten Schlüssel-Wert-Paare, welche für die Ausführung der Quantenanwendungen als Eingabeparameter genutzt werden. Bei Warteschlangen-Ereignissen beinhalten die Eingabeparameter standardmäßig nur den API-Token und den Namen des zu verwendenden Quantencomputers.

```
{
    "eventType": "QUEUE_SIZE",
    "additionalProperties": {
        "queueSize": 5
    },
    "eventPayloadProperties": {
        "apiToken": "someApiToken",
        "device": "someIbmqDevice"
    }
}
```

Listing 5.7: Ereignisobjekt eines beispielhaften Warteschlangenereignisses

Die Inhalte des Ereignisobjekts werden zusammen mit dem aktuellen Zeitstempel für die Durchführung einer SQL-Abfrage genutzt, um alle zum Ereignis passenden QueueSizeEventTrigger aus der Datenbank auszulesen. Diese werden im nächsten Schritt durchgegangen und für jeden wird eine POST-Anfrage an den verantwortlichen OpenWhiskService abgeschickt, damit der gleichnamige Trigger ausgelöst wird. Die POST-Anfrage nutzt dabei die triggerPayloadProperties des Ereignisobjekts als Body, damit die mit dem Trigger verknüpften Actions mit den nötigen Eingabeparametern ausgeführt werden. Als Antwort auf die POST-Anfrage erhält der QuantumService ein Objekt zurück, welches die vom OpenWhiskService generierte Activation-ID des ausgelösten Triggers enthält. Wie diese Activation-ID genutzt wird, um das Ergebnis der Ausführung zu beschaffen, wurde bereits in Abschnitt 5.5.6 im Detail erklärt. Falls möglich, wird zum Schluss der disabledUntil-Zeitstempel des QueueSizeEventTriggers mithilfe seines optionalen triggerDelay-Werts aktualisiert, sonst wird der einmalig auszulösende QueueSizeEventTrigger gelöscht.

5.6.3 Ausführungsergebnis-Feed

Der Ausführungsergebnis-Feed liefert Ergebnisse von erfolgreich ausgeführten Quantenanwendungen. Dabei werden ExecutionResultEventTrigger ausgelöst, welche die mit ihnen verknüpften Quantenanwendungen ausführen. Für die individuelle Gestaltung des Feeds werden ExecutionResultEventTrigger mithilfe des *executedApplicationName*-Felds konfiguriert. Dieses Feld gibt den eindeutigen Namen der Quantenanwendung an, deren Ergebnisse den ExecutionResultEventTrigger auslösen sollen. Im Vergleich zum Warteschlangen-Feed, wird der Ausführungsergebnis-Feed nicht durch eine spezielle Komponente umgesetzt. Er entsteht stattdessen im Normalbetrieb des Systems, bei welchem Quantenanwendungen entweder manuell oder durch Ereignisse auf IBMs Quantencomputern ausgeführt werden.

Das Auslösen der ExecutionResultEventTrigger setzt standardmäßig immer am Ende einer Job-Iteration des JobChecker-Zyklus (siehe Abbildung 5.8) an, in welcher der betrachtete Job den COMPLETED-Status erreicht hat. Dabei wird der Job für die Generierung eines Ereignisobjekts genutzt, welches wie das im Listing 5.8 abgebildete Beispiel strukturiert ist. Wie beim Warteschlangen-Feed wird dieses Objekt für die Durchführung einer SQL-Abfrage genutzt, um alle passenden ExecutionResultEventTrigger aus der Datenbank auszulesen. Im Anschluss werden die verwandten Trigger der entsprechenden OpenWhiskServices mittels einer POST-Anfrage

```
{
    "eventType": "EXECUTION_RESULT",
    "additionalProperties": {
        "quantumApplicationName": "executedQuantumApplicationName"
    },
    "triggerPayloadProperties": {
        "apiToken": "someApiToken",
        "device": "someIbmqDevice"
        "result": "{...}"
    }
}
```

Listing 5.8: Ereignisobjekt eines beispielaften Ausführungsergebnis-Ereignisses

ausgelöst. Dabei werden alle verknüpften QuantumApplications bzw. Actions mit den triggerPayloadProperties ausgeführt, welche hierbei noch den zusätzlichen, als JSON-String formatierten result-Eingabeparameter besitzen.

5.7 Grafische Benutzeroberfläche

Um Nutzern eine möglichst einfache Bedienung des Systems zu ermöglichen, kommt ein als QuantumServiceUI bezeichneter Frontend-Prototyp zum Einsatz. Dieser setzt sich aus Tabellenansichten zusammen, welche mithilfe einer Sidebar (siehe Abbildung 5.11) navigiert werden können. Der aktuelle Prototyp besitzt vier Tabellenansichten für QuantumApplications, EventTrigger, Jobs und OpenWhiskServices. Die Tabellenansichten ermöglichen Nutzern alle unterstützten Operationen auf der jeweiligen Collection oder den darin enthaltenen Elementen durchzuführen.

Abbildung 5.11 zeigt die Tabellenansicht aller im System angemeldeten Quantenanwendungen. In den Zeilen der Tabelle werden verfügbare Quantenanwendungen aufgelistet (2). Den einzelnen Spalten werden dabei Datensätze der jeweiligen Quantenanwendungen zugeordnet. Die letzte Spalte ist für Operationen reserviert, welche auf den verfügbaren Quantenanwendungen ausgeführt werden können. Aktuell unterstützte Operationen sind die manuelle Ausführung (3), der Download des Quellcodes (4) und die Löschung des Elements (5). Für die manuelle Ausführung werden über einen Dialog alle nötigen Eingabeparameter ausgewählt, bevor eine Anfrage an den QuantumService geschickt wird, damit dieser die Ausführung der beim OpenWhiskService bereitgestellten Action auslöst. Bei der Download-Funktion wird der Quellcode der Quantenanwendung als Python-Datei auf die lokale Maschine heruntergeladen. Mithilfe der Löschen-Funktion kann die ausgewählte Quantenanwendung im QuantumService und beim OpenWhiskService gelöscht werden. Zuletzt ist es auch möglich neue Quantenanwednungen zu erstellen (1). Dafür wird ein Dialog geöffnet, welcher Nutzern erlaubt, eine Python-Datei hochzuladen und sie zusammen mit den anderen wichtigen Daten an den QuantumService zu schicken, um ihre Quantenanwendung im System anzumelden und beim OpenWhiskService für die künftige Ausführung bereitzustellen. Die Tabellenansichten anderer Entitäten sind identisch strukturiert und ermöglichen die Ausführung ähnlicher Operationen, sodass sie an dieser Stelle nicht im Detail erklärt werden.

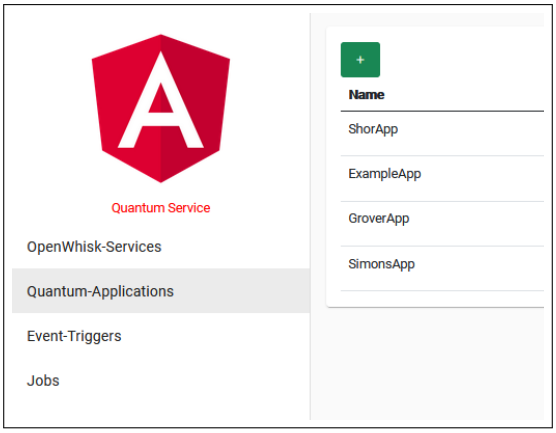

Abbildung 5.11: Sidebar-Menü

Abbildung 5.12: Tabellenansicht der angemeldeten Quantenanwendungen

Entitäten wie beispielsweise die QuantumApplications haben viele Daten, welche in einer Tabellenansicht nicht vollständig abgebildet werden können. Für solche Daten kommt eine Detailansicht zum Einsatz, welche durch den Klick auf ein in der Tabellenansicht enthaltenes Element aufgeklappt wird. Die Detailansicht einer Quatenanwendung ist in Abbildung 5.13 zu sehen. Neben simplen Informationen (1) können in dieser Ansicht ebenfalls komplexere Daten wie der Quellcode (2) und die aktuell registrierten EventTrigger (6) angezeigt werden. Zusätzlich zu den in der Tabellenansicht vorgestellten Operationen (3) können in der Detailansicht die Verknüpfungen zu verfügbaren EventTriggern verwaltet werden. Beim Hinzufügen (4) eines neuen EventTriggers wird ein Dialog geöffnet, in welchem Nutzer die ausgewählte Quantenanwendung mit allen verfügbaren EventTriggern verknüpfen können. Alternativ kann auch eine bereits vorhandene Verknüpfung zu einem EventTrigger gelöst werden (5).

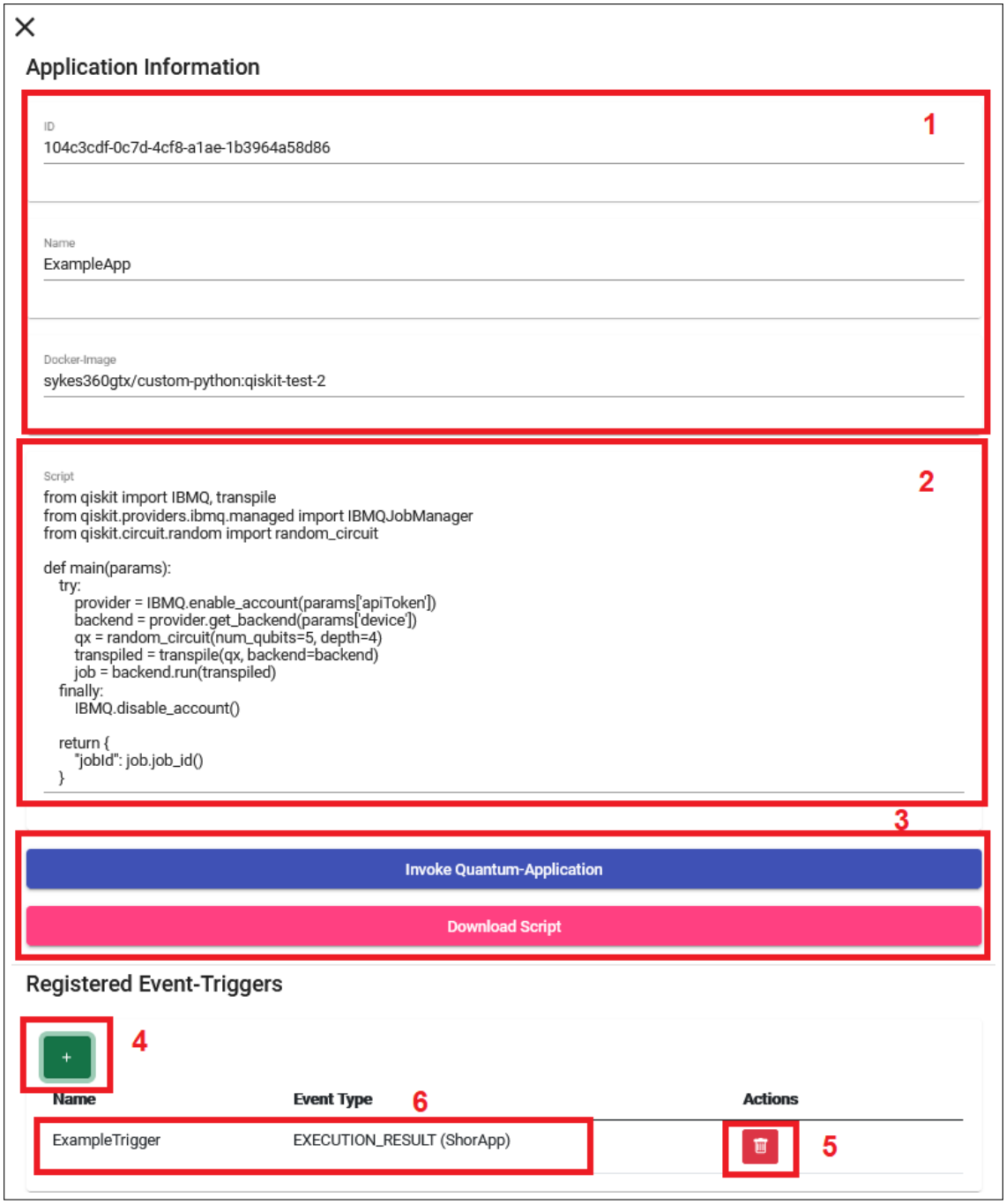

Abbildung 5.13: Detailansicht einer angemeldeten Quantenanwendungen

6 Diskussion

In diesem Kapitel werden die Ergebnisse, Limitationen und mögliche Alternativen des in dieser Arbeit entwickelten Konzepts und des damit umgesetzten Prototyps diskutiert.

6.1 Umsetzung der Problemstellung

Zur Lösung der Problemstellung muss eine ereignisbasierte Architektur umgesetzt werden, welche die Bereitstellung und Ausführung von Quantenanwendungen auf von Cloud-Diensten angebotenen Quantencomputern ermöglicht. Dabei sollen die einzelnen Quantenanwendungen beim Eintritt von beliebigen Ereignissen mit individuellen Eingabeparametern automatisch ausgeführt werden können.

Für die Bereitstellung und Ausführung von Quantenanwendungen bietet unser Konzept allen Nutzern die Möglichkeit beliebige FaaS-Dienste, welche die nötigen Laufzeiten unterstützen, im System zu registrieren und über diese ihre Quantenanwendungen für die Ausführung bereitzustellen. Da sich FaaS auf die Ausführung von Funktionen spezialisiert, können die bereitgestellten Quantenanwendungen jederzeit mit beliebigen Eingabeparametern ausgeführt werden. Die Umsetzbarkeit dieser Funktionalität wurde mithilfe des entwickelten Prototyps gezeigt. Dieser ermöglicht seinen Nutzern, privat gehostete oder öffentliche, auf OpenWhisk basierende FaaS-Dienste im System zu registrieren, um auf diesen Quantenanwendungen auszuführen. Deren Ausführung wird durch die Nutzung von Docker-Images ermöglicht, welche beliebige Laufzeiten enthalten, sodass auch die mit speziellen Bibliotheken oder SDKs (z.B. Qiskit) ausgestatteten Python-Funktionen erfolgreich ausgeführt werden können. In diesem Punkt ist der implementierte Prototyp beliebig erweiterbar, sodass FaaS-Anbieter jenseits von OpenWhisk für die Bereitstellung und Ausführung von Quantenanwendungen ins System integriert werden können.

Quantenanwendungen lassen sich nur auf speziellen Quantencomputern ausführen, welche von einigen Cloud-Anbietern über spezielle Dienste registrierten Nutzern zur Verfügung gestellt werden. Mit dem erarbeiteten Konzept können solche Dienste ins System integriert werden, sodass alle bereitgestellten Quantenanwendungen deren Quantencomputer nutzen können. Zur Demonstration wurde der von IBM angebotene Dienst „IBM Quantum“ in den entwickelten Prototyp integriert. Dafür wurde dieser mit einem IBM Quantum Account ausgestattet, dessen exklusiver API-Token allen im System angemeldeten Quantenanwendungen über Eingabeparameter übergeben werden kann, damit sie IBMs Quantencomputer und Simulatoren ansprechen können. Da derselbe IBM Quantum Account sowohl vom System als auch von allen Quantenanwendungen verwendet wird, können die Ergebnisse aller Ausführungen über die dafür vorgesehenen Schnittstellen des QC-Dienstes ausgelesen werden. Wie bei den FaaS-Anbietern ist der Prototyp auch hier erweiterbar, sodass weitere auf Quantum Computing spezialisierte Cloud-Dienste mit wenig Aufwand integriert werden können.

Damit die Quantenanwendungen durch den Eintritt von Ereignissen mit individuellen Eingabeparametern ausgeführt werden können, ermöglicht das entwickelte Konzept die Integration von Ereignis-Feeds beliebiger externer Ereignisquellen. Dabei kann mithilfe von EventTriggern (siehe Abschnitt 5.3.5) jeder vom System unterstützte Feed für jede Quantenanwendung individualisiert werden, sodass nur Ereignisse mit gewünschten Eigenschaften die Ausführung der Quantenanwendungen mit ereignisspezifischen Eingabeparametern auslösen. Zu Demonstrationszwecken wurde für den Prototyp IBM Quantum zusätzlich als externe Ereignisquelle ins System integriert, damit Quantenanwendungen durch Ereignisse eines Warteschlangen- oder Ausführungsergebnis-Feeds automatisch ausgeführt werden können.

6.2 Limitationen

Im vorherigen Abschnitt wurde mittels des in Kapitel 5 vorgestellten Prototyps gezeigt, dass das entwickelte Konzept die ereignisbasierte Ausführung von Quantenanwendungen ermöglicht. Dennoch gibt es einige Einschränkungen die durch das Konzept und die im Prototyp eingesetzten Technologien zustande kommen.

Der entwickelte Prototyp ist für seine Funktionalität auf die externen Ereignisquellen, QC- und FaaS-Dienste angewiesen. Aktualisieren diese ihre Schnittstellen auf eine neue Version, funktioniert der Prototyp unter Umständen nicht mehr, bis sein Code an die neue Schnittstellenversion angepasst wurde. Ebenfalls kann durch Netzwerkpartitionierung oder Ausfälle die Funktion des Prototyps vorübergehend beeinträchtigt werden. Zuletzt bringt auch die Nutzung von FaaS-Diensten ihre Einschränkungen mit sich. Diese haben üblicherweise strikte Beschränkungen im Bezug auf Ausführungsdauer und Speicherverbrauch, welche von allen Quantenanwendungen eingehalten werden müssen.

Eine weitere Einschränkung bilden die speziellen Strukturanforderungen (siehe Abschnitt 5.5.1) für Quantenanwendungen. Für die Umsetzung des Konzepts, muss der Quellcode aller von Nutzern bereitgestellten Quantenanwendungen einer bestimmten Strukturvorlage folgen. Werden diese beim Schreiben der Funktionen nicht eingehalten oder unterläuft dem Entwickler der Quantenanwendung ein Fehler, ist die Funktionalität des Systems beeinträchtigt.

Ein mit den Strukturanforderungen verwandtes Problem des aktuellen Prototyps ist die Nutzung eines servicespezifischen API-Tokens für die Interaktion mit IBM Quantum. Im aktuellen Prototyp müssen alle Quantenanwendungen den API-Token desselben IBM Quantum Accounts nutzen. Dieser wird den Quantenanwendungen immer als Eingabeparameter übergeben. Da dieser Token von jeder Quantenanwendung über beispielsweise HTTP an jeden beliebigen Ort weitergeleitet werden kann, ist es keine Option Accounts bzw. Token von QC-Diensten zu verwenden, die mit Kosten verbunden sind.

Der implementierte Prototyp setzt alle im Konzept definierten Komponenten in einem einzigen QuantumService um. Daraus folgt, dass der gleiche Service sowohl die Verwaltung von Quantum-Applications (siehe 5.3.2), OpenWhiskServices (siehe Abschnitt 5.3.1) und EventTriggern (siehe Abschnitt 5.3.5) als auch die Umsetzung der Ereignis-Feeds übernimmt. Wird der Service horizontal skaliert, würde er alle durch Polling integrierte Ereignis-Feeds mehrfach umsetzen. So würde der Prototyp beispielsweise IBM Quantums Warteschlangeninformationen aller Quantencomputer mit

jeder Service-Instanz separat abfragen. Deswegen ist für die aktuelle Version des QuantumService nur eine vertikale Skalierung sinnvoll. Um die horizontale Skalierung zu ermöglichen, müssen die Komponenten auf mehrere Services aufgeteilt werden.

Das in dieser Arbeit entwickelte Konzept achtet nicht auf die Mandantenfähigkeit des Systems. Das bedeutet, dass dieses nicht in der Lage ist, zwischen verschiedenen Nutzern zu unterscheiden und entsprechend alle Nutzer gleich behandelt. Die fehlende Mandantenfähigkeit kann zu verschiedenen Problemen im Bezug auf Security und Datenschutz führen. Registriert beispielsweise ein Nutzer im implementierten Prototyp einen privat gehosteten OpenWhisk-Server als OpenWhiskService, wird dieser für alle anderen Nutzer sichtbar sein. Sie können somit auf die sensitiven Daten wie URL und Anmeldeinformationen des Servers zugreifen und ihn für die Ausführung eigener Quantenanwendungen verwenden. Ein ähnliches Problem entsteht mit den im System angemeldeten Quantenanwendungen. Hier können beispielsweise Nutzer auf die Details fremder Quantenanwendungen zugreifen und potenziell sensitiven Quellcode oder Ausführungsergebnisse auslesen. Darüber hinaus können Nutzer alle in der Datenbank abgespeicherten Objekte manipulieren, sodass sie beispielsweise existierende Quantenanwendungen anderer Nutzer löschen oder ihre Feeds verändern können.

6.3 Alternativkonzepte

In den folgenden Abschnitten werden Alternativen zur Umsetzung einer ereignisbasierten Architektur vorgestellt und mit dem aktuellen Konzept und dem damit entwickelten Prototyp verglichen.

6.3.1 Alternative Ergebnisbeschaffung

Das in dieser Arbeit entwickelte Konzept schlägt vor, für die Beschaffung der Ereignisse die als Funktionen umgesetzten Quantenanwendungen so zu schreiben, dass diese die Ausführungs- bzw. Job-IDs als Rückgabewert an eine Systemkomponente weiterleiten, damit das Ergebnis durch wiederholtes Abfragen der Schnittstellen des QC-Dienstes beschafft werden kann. Dadurch müssen alle Quantenanwendungen einen Rückgabewert enthalten, welcher von der entsprechenden Systemkomponente verarbeitet werden kann. Die in Kapitel 2 vorgestellte Arbeit [GCA+21] nutzt für die Bereitstellung und Ausführung der Quantenanwendungen ebenfalls OpenWhisk und IBM Quantum. Im Vergleich zum in dieser Arbeit implementierten Prototyp werden die von den Quantenanwendungen generierten Job-IDs nicht durch den Rückgabewert im Action- bzw. Activation-Ergebnis gespeichert, sondern als eine Nachricht mittels Kafka an eine Systemkomponente weitergeleitet, welche das Ergebnis mittels der ID beschafft. Das hat zur Folge, dass die Job-IDs nicht durch eine Komponente wie den ScriptExecutionChecker (siehe Abschnitt 5.5.6) beschafft werden müssen, bevor das Ergebnis-Polling gestartet werden kann.

Das alternative Verfahren vereinfacht zwar den Beschaffungsprozess der Ausführungsergebnisse, stellt jedoch größere Anforderungen an die von Nutzern bereitgestellten Quantenanwendungen. So müssen diese um Code erweitert werden, welcher sich mit Kafka verbindet, um die Job-ID als Nachricht zu verschicken. Dafür müssen den Quantenanwendungen neben den Ereignis-spezifischen Daten zusätzlich alle nötigen Informationen für die Verbindung zu Kafka übergeben werden, welche im Messaging-Code korrekt eingesetzt werden müssen. Dies führt dazu, dass Entwickler sich nicht vollständig auf das Schreiben ihres QC-Codes konzentrieren können.

6.3.2 Alternative Bereitstellung der Quantenanwendungen

Unser Konzept beschränkt sich für die Bereitstellung und Ausführung der Quantenanwendungen auf FaaS. In Kapitel 2 wurden als Alternative sogenannte Quantum-Microservices [RVB+21] vorgestellt, welche über Schnittstellen Quantenalgorithmen auf Quantencomputern ausführen können. Ein alternatives Konzept für eine ereignisbasierte Architektur könnte Quantum-Microservices statt FaaS nutzen. Anstatt FaaS-Dienste von Anbietern im System zu registrieren, könnten Nutzer ihre Quantum-Microservices hosten und die einzelnen Quantenanwendungen als POST-Endpunkte implementieren, welche den Body als Eingabeparameter nutzen und die Ausführungs-ID zur späteren Ergebnisbeschaffung als Antwort zurückgeben. Diese Endpunkte könnten sie mit zusätzlichen Informationen wie beispielsweise Namen und den nötigen Details zur Authentifizierung als Quantenanwendungen im System registrieren und sie mit gewünschten EventTriggern verknüpfen. Beim Eintritt von Ereignissen könnte das System anhand der EventTrigger die entsprechenden Quantenanwendungen ausführen, indem es die registrierten POST-Endpunkte mit den Ereignisdaten und den nötigen API-Token aufruft und die in der Antwort enthaltene Ausführungs-ID zur Ergebnisbeschaffung in der Datenbank speichert. Auf diese Weise könnte die aktuelle Funktion des Prototyps mit Quantum-Microservices umgesetzt werden. Die Nutzung der Quantum-Microservices entfernt FaaS-Einschränkungen wie die beschränkte Ausführungsdauer und Speichernutzung, erhöht jedoch den Aufwand den Nutzer aufbringen müssen, um ihre Quantenanwendungen für die ereignisbasierte Ausführung bereitzustellen. So müssen Nutzer nicht nur ihre Quantenanwendungen schreiben, sondern auch einen permanent laufenden Microservice entwickeln, den sie selbstständig mit einer passenden Laufzeitumgebung hosten und warten müssen. Im Fall des aktuellen Prototyps können diese Aufgaben vollständig vom Dienst des FaaS-Anbieters übernommen werden, sodass sich Nutzer nur um das Schreiben ihrer Quantenanwendungen kümmern müssen.

Im Rahmen dieser Arbeit wurde zusätzlich ein weiterer Prototyp entwickelt, welcher für die Ausführung und Bereitstellung der Quantenanwendungen ein Dateisystem und die Python-CLI verwendet. Er besitzt im Vergleich zum in Kapitel 5 vorgestellten Prototyp zwei Java-Services, welche über Messaging-Warteschlangen miteinander verbunden sind. Der *QuantumServiceMessaging*[1] ist dabei für die Verwaltung der QuantumApplications (siehe Abschnitt 5.3.2) und EventTrigger (siehe Abschnitt 5.3.5) zuständig. Wie beim bereits vorgestellten Prototyp werden diese über HTTP-Schnittstellen im System angemeldet. Anstatt die dabei erstellte QuantumApplication als Action (siehe Abschnitt 3.2.3) über einen OpenWhiskService (siehe Abschnitt 5.3.1) bereitzustellen, wird die als Funktion umgesetzte Python-Datei im Dateisystem abgespeichert, während eine Referenz zur Datei im QuantumApplication-Objekt abgelegt wird. Im Vergleich dazu ist der sogenannte *EventSourceMessaging*[2] für die Beschaffung der Ereignisdaten zuständig. Diese schickt er als Ereignisobjekt (siehe Listing 5.8) über eine spezielle Ereignis-Warteschlange an den QuantumServiceMessaging, welcher dieses nutzt, um die entsprechenden EventTrigger aus der Datenbank zu laden. Im Anschluss setzt er die Python-CLI ein, um jede zum EventTrigger gehörende Quantenanwendungen auszuführen. Dafür verwendet er die im QuantumApplication-Objekt gespeicherte Referenz zu der im Dateisystem abgespeicherten Datei. Nach der erfolgreichen Ausführung einer Quantenanwendung liest der QuantumServiceMessaging die dabei generierte Job-ID aus dem Rückgabewert der Funktion, damit anhand dieser ein initialer Job (siehe Listing 5.5) für die Ergebnisbeschaffung erstellt werden kann. Alle anderen Funktionen werden ähnlich wie

[1] https://github.com/LHommeDeBat/QuantumServiceMessaging
[2] https://github.com/LHommeDeBat/EventSourceMessaging

vom FaaS-Prototyp umgesetzt. Damit auch dieser Prototyp möglichst einfach bedient werden kann, wurde ein an den Prototyp angepasstes Frontend[3] implementiert. Die Nutzung des Dateisystems und der Python-CLI bringt einige Vor- und Nachteile mit sich. Zu den Vorteilen gehört der Wegfall der Synchronisation mit den externen, als OpenWhiskService bezeichneten OpenWhisk-Diensten. Des weiteren fallen FaaS-Einschränkungen wie beschränkte Ausführungszeiten weg. Da für jede Anwendung eine neue Python-CLI und somit ein neuer Python-Prozess geöffnet wird, müssen sich die Nutzer nicht um die Schließung der IBM Quantum Sessions kümmern, wie es im aktuellen FaaS-Prototyp der Fall ist (siehe Abschnitt 5.5.1). Ein Nachteil dieser Architektur ist, dass auf der Maschine, auf welcher der QuantumServiceMessaging läuft, zusätzlich eine passende Python-Laufzeit mit Qiskit installiert sein muss. Diese lässt sich nicht für jede Quantenanwendung individuell wählen, wie es bei den von OpenWhisk genutzten Docker-Images der Fall ist. Das führt dazu, dass alle Quantenanwendungen so angepasst werden müssen, dass sie nur Bibliotheken verwenden, welche in der vorgegebenen Laufzeit verfügbar sind. Alternativ kann die Laufzeitumgebung ständig manuell angepasst werden, sodass sie jederzeit die Bedürfnisse aller neuen Quantenanwendungen erfüllt. Da der QuantumServiceMessaging in Java geschrieben ist und zusätzlich eine Python-Laufzeitumgebung mit Qiskit und ein Dateisystem benötigt, kann dieser nur auf einer Infrastruktur bereitgestellt werden, sodass im Vergleich zur Nutzung von FaaS viel weniger Verantwortung an die Cloud-Anbieter übertragen werden kann. Das Dateisystem sorgt zusätzlich für Probleme bei der horizontalen Skalierung, sodass der QuantumServiceMessaging nur vertikal skaliert werden kann. Um dieses Problem zu adressieren, könnten die für die Ausführung nötigen Dateien beispielsweise nur temporär im Dateisystem erstellt und nach der Ausführung gelöscht werden.

6.3.3 Alternativer Einsatz von IBM Quantum Accounts

Wie in Abschnitt 6.2 bereits erwähnt, nutzt der aktuelle Prototyp nur einen IBM Quantum Account für alle im System angemeldeten Quantenanwendungen. Das erleichtert die Ergebnisbeschaffung, da alle bei IBM Quantum laufenden Ausführungen und Ergebnisse zum gleichen Account gehören und somit mit dem gleichen API-Token ausgelesen werden können. Eine alternative Vorgehensweise wäre, dass Nutzer den API-Token im Funktionscode hartkodieren und ihn zusammen mit der Job-ID im Rückgabewert dem System übergeben. Auf diese Weise könnte das System die privaten API-Token der einzelnen Quantenanwendungen nutzen, um auf die Ergebnisse der entsprechenden Accounts zugreifen zu können. Das hätte den Vorteil, dass Nutzer mit Accounts, welche Zugriff zu mehr Hubs und somit Quantencomputern haben, diese für die Ausführung ihrer Quantenanwendungen verwenden können. Die Nutzung mehrerer API-Tokens würde jedoch accountspezifischen Ereignis-Feeds wie beispielsweise dem Warteschlangen-Feed (siehe Abschnitt 5.6.2) Probleme bereiten. Dieser liefert in kurzen Abständen die Warteschlangengrößen aller verfügbaren Quantencomputer, wofür ein IBM Quantum Account bzw. API-Token benötigt wird. Würde jede Quantenanwendung einen spezifischen Account bzw. Token haben, welcher Zugriff auf unterschiedliche Quantencomputer hat, könnte dieser Feed nur umgesetzt werden, wenn die Warteschlangengrößen aller verfügbaren Quantencomputer für jede Quantenanwendung separat abgefragt werden würden, was bei vielen Quantenanwendungen und kurzen Abfrageabständen schwer realisierbar wäre.

[3] https://github.com/LHommeDeBat/QuantumServiceUI

7 Zusammanfassung und Fazit

Im Rahmen dieser Arbeit wurde ein Konzept für eine Softwarearchitektur entwickelt, welche durch den Eintritt von Ereignissen die automatisierte Ausführung von Quantenanwendungen ermöglicht, die über unterstützte FaaS-Dienste bereitgestellt werden können. Das erarbeitete Konzept wurde dabei für die Umsetzung eines Prototyps genutzt, welcher für das Emittieren von Ereignissen IBM Quantum als Ereignisquelle nutzt, um von Nutzern bereitgestellte Quantenanwendungen mithilfe von OpenWhisk auszuführen. Hierfür wurde die als QuantumService bezeichnete Anwendung implementiert, welche die Verwaltung von Quantenanwendungen, Ausführungsergebnissen, Ereignis-Feeds und OpenWhisk-Diensten über eine REST-API ermöglicht. Damit bietet das System seinen Nutzern die Flexibilität, ihre Quantenanwendungen sowohl über privat gehostete als auch öffentliche OpenWhisk-Dienste wie die IBM Cloud Functions auszuführen. Die Anmeldung neuer Quantenanwendungen erfolgt durch den Upload von mit Qiskit geschriebenen, einer Vorlage folgenden Python-Funktionen, welche automatisch beim ausgewählten OpenWhisk-Dienst für die künftige Ausführung in einer passenden Laufzeitumgebung bereitgestellt werden. Um die ereignisbasierte Ausführung der Quantenanwendungen zu ermöglichen, können diese mit konfigurierbaren EventTriggern verknüpft werden, welche beim Eintritt eines passenden Ereignisses ausgelöst werden. Für die Umsetzung der Ereignis-Feeds wurden Warteschlangen- und Ausführungsergebnis-Ereignisse definiert, deren Eintritt durch die von IBM Quantum zur Verfügung gestellten Schnittstellen geprüft werden kann. Damit das implementierte System einfach zu bedienen ist, wurde zusätzlich ein Frontend implementiert, welches Nutzern eine über den Webbrowser erreichbare grafische Benutzeroberfläche bietet.

Der Einsatz des Prototyps hat gezeigt, dass das in dieser Arbeit entwickelte Konzept trotz einiger Einschränkungen genutzt werden kann, um die Bereitstellung und ereignisbasierte Ausführung von Quantenanwendungen zu ermöglichen. Durch die Nutzung von FaaS wird sichergestellt, dass Quantenanwendungen nicht durchgehend ausgeführt werden, sondern nur wenn ein entsprechendes Ereignis eintritt. Mithilfe der Ereignisse des Ausführungsergebnis-Feeds können Workflows aus vielen unterschiedlichen Quantenanwendungen automatisiert ausgeführt werden, ohne das einzelne Quantenanwendungen auf die Ergebnisse anderer aktiv warten müssen. Für eine detailliertere Untersuchung des Prototyps könnte eine qualitative Studie mit Entwicklern von Quantenanwendungen durchgeführt werden. Mit dieser könnte genauer geprüft werden, welche weiteren Anforderungen das System erfüllen muss, welche existierenden Funktionen angepasst oder erweitert werden müssen und welche existierenden Funktionen nicht benötigt werden.

Ausblick

Für die Verbesserung des in dieser Arbeit entwickelten Konzepts und des damit umgesetzten Prototyps müssen die bereits erwähnten Einschränkungen, sofern möglich, behoben bzw. minimiert werden. Dafür könnte das entwickelte Konzept mandantenfähig gemacht werden. Hierfür muss

die Architektur mit einem Authentifizierungsserver und einer Nutzerverwaltung erweitert werden. Ebenfalls kann der Prototyp in mehrere Services aufgeteilt werden, welche bei Bedarf über Messaging-Warteschlangen kommunizieren, sodass sich die einzelnen Services besser horizontal skalieren lassen.

Zusätzlich zum Beheben seiner Einschränkungen kann die Funktionalität des Prototyps erweitert werden. Dafür können weitere Ereignisquellen ins System integriert werden, damit Quantenanwendungen für weitere nützliche Ereignis-Feeds registriert werden können. Um die Auswahl von verfügbaren Quantencomputern zu vergrößern, können weitere auf Quantum Computing spezialisierte Cloud-Dienste wie Amazon-Braket eingebunden werden. Zuletzt kann die Auswahl von unterstützten FaaS-Diensten erweitert werden, sodass Nutzer für die Ausführung ihrer Quantenanwendungen nicht nur auf privat gehostete OpenWhisk-Server oder die IBM Cloud Functions angewiesen sind.

Literaturverzeichnis

[Apaa] Apache Software Foundation. *Actions*. https://github.com/apache/openwhisk/blob/master/docs/actions.md (zitiert auf S. 16, 17).

[Apab] Apache Software Foundation. *Actions - The basics*. https://github.com/apache/openwhisk/blob/master/docs/actions.md#the-basics (zitiert auf S. 43).

[Apac] Apache Software Foundation. *Apache OpenWhisk Alarm Package*. https://github.com/apache/openwhisk-package-alarms (zitiert auf S. 20).

[Apad] Apache Software Foundation. *Creating and invoking Docker actions*. https://github.com/apache/openwhisk/blob/master/docs/actions-docker.md (zitiert auf S. 17).

[Apae] Apache Software Foundation. *Deployment Options*. https://openwhisk.apache.org/documentation.html#openwhisk_deployment (zitiert auf S. 15).

[Apaf] Apache Software Foundation. *Documentation*. https://openwhisk.apache.org/documentation.html (zitiert auf S. 15, 16).

[Apag] Apache Software Foundation. *Feeds*. https://github.com/apache/openwhisk/blob/master/docs/feeds.md (zitiert auf S. 20, 21).

[Apah] Apache Software Foundation. *Namespaces and Packages*. https://github.com/apache/openwhisk/blob/master/docs/reference.md#namespaces-and-packages (zitiert auf S. 16, 17).

[Apai] Apache Software Foundation. *OpenWhisk Deployment on Kubernetes*. https://github.com/apache/openwhisk-deploy-kube/blob/master/README.md (zitiert auf S. 15).

[Apaj] Apache Software Foundation. *Triggers and Rules*. https://github.com/apache/openwhisk/blob/master/docs/triggers_rules.md (zitiert auf S. 16, 18).

[Apak] Apache Software Foundation. *Using REST APIs with OpenWhisk*. https://github.com/apache/openwhisk/blob/master/docs/rest_api.md (zitiert auf S. 16, 17).

[Ben80] P. Benioff. „The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines“. In: *Journal of Statistical Physics* 22.5 (Mai 1980), S. 563–591. ISSN: 1572-9613. DOI: 10.1007/BF01011339. URL: https://doi.org/10.1007/BF01011339 (zitiert auf S. 11).

[Fey82] R. P. Feynman. „Simulating physics with computers“. In: *International journal of theoretical physics* 21.6/7 (1982), S. 467–488 (zitiert auf S. 11).

[GCA+21] M. Grossi, L. Crippa, A. Aita, G. Bartoli, V. Sammarco, E. Picca, N. Said, F. Tramonto, F. Mattei. „A Serverless Cloud Integration For Quantum Computing“. In: *arXiv preprint arXiv:2107.02007* (2021) (zitiert auf S. 13, 61).

[Git] GitHub. *GitHub event types - PushEvent*. https://docs.github.com/en/developers/webhooks-and-events/events/github-event-types#pushevent (zitiert auf S. 20).

[IBMa] IBM. *Become an IBM Quantum Hub*. https://www.ibm.com/quantum-computing/network/members/ (zitiert auf S. 22).

[IBMb] IBM. *Cloud Functions*. https://www.ibm.com/de-de/cloud/functions (zitiert auf S. 15).

[IBMc] IBM. *FaaS (Function-as-a-Service)*. https://www.ibm.com/cloud/learn/faas (zitiert auf S. 15).

[IBMd] IBM. *IBM Quantum Services*. https://www.ibm.com/quantum-computing/services/ (zitiert auf S. 22).

[IBMe] IBM. *Qiskit Runtime*. https://quantum-computing.ibm.com/lab/docs/iql/runtime/ (zitiert auf S. 13).

[IBMf] IBM. *Qiskit Runtime*. https://github.com/Qiskit-Partners/qiskit-runtime (zitiert auf S. 13).

[IBMg] IBM. *Qiskit Runtime API*. https://qiskit.org/documentation/partners/qiskit_runtime/tutorials/API_direct.html (zitiert auf S. 13).

[RVB+21] J. Rojo, D. Valencia, J. Berrocal, E. Moguel, J. Garcia-Alonso, J. M. M. Rodriguez. „Trials and Tribulations of Developing Hybrid Quantum-Classical Microservices Systems“. In: *arXiv preprint arXiv:2105.04421* (2021) (zitiert auf S. 13, 62).

Alle URLs wurden zuletzt am 22. 10. 2021 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift