Java™ magazin
Java | Architektur | Software-Innovation

Gemeinsam stark

Java, Angular und Kubernetes

Fullstack in modern

Ausgabe 5.2024
Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

entwickler.de

Architektur in der Praxis

Von Offline bis Echtzeit

In diesem Schwerpunkt liegt der Fokus darauf, am Beispiel einer interaktiven Webanwendung mit Echtzeitinteraktion typische Bausteine einer möglichen Umsetzung im Detail zu betrachten – Fullstack in modern: leichtgewichtig und leistungsfähig.

von Thomas Kruse

In den folgenden Artikeln wird, um ein umfassendes Verständnis der Zusammenhänge zu gewähren, auf die folgenden Aspekte, die immer wieder gebraucht werden, eingegangen:

- Betriebsumgebung
- Persistenz und Datenmodell
- Kommunikation zwischen Systemkomponenten
- Authentifizierung
- Backend
- Frontend

Das Anwendungsbeispiel ist eine Anwendung (**Abb. 1**), mit der auf einem Event, wie beispielsweise einer IT-Konferenz, der Austausch der Teilnehmer unterstützt und Gesprächspartner zusammengebracht werden sollen, die an ähnlichen Themen interessiert sind. Der Ablauf sieht vereinfacht so aus, dass jeder Teilnehmer sich an der Anwendung mit seinem Konferenzausweis (Badge) anmelden und dann auf spielerische Weise mit anderen Teilnehmern in Interaktion treten kann. Dazu wird der auf dem Badge angebrachte eindeutige Barcode als Identifikationsmerkmal genutzt. Wichtig ist in diesem Zusammenhang, dass die dabei durchgeführten Aktionen im zeitlichen Kontext ausgewertet werden müssen, um den zugrundeliegenden Nutzen durch das System zu unterstützen.

So kann man auch andere Teilnehmer – nachdem deren Einverständnis eingeholt wurde – scannen. Damit ist bereits ein erster Eisbrecher gegeben, um ins Gespräch zu kommen. Mit Gamification soll genau dieses Verhalten belohnt werden: Es gibt Punkte für Scans. Und damit auch die Herausforderungen: Duplikate erkennen, Bonuspunkte für besonders viele Scans in einem bestimmten Zeitraum, Pausierung des Spiels, während Vorträge stattfinden – hier sind der Fantasie keine Grenzen gesetzt. Und aus vergleichbaren Projekten ist auch bekannt, dass am besten flexible Iterationen ermöglicht werden sollten, um die Anwendung schrittweise weiterzuentwickeln und aus den gemachten Erfahrungen kurzfristig zu lernen. Auch das Feedback der Nutzer und Stakeholder sollte schnell in Form von neuen Ideen und Modifikationen umsetzbar sein. Optimalerweise schon während des Einsatzes auf einem Event. Es bedarf keiner besonderen Erwähnung, dass dabei nichts schiefgehen sollte, schließlich ist der Einsatzzeitraum typischerweise sehr kurz. Robustheit ist also ein wichtiges Kriterium. Dazu kommt, dass mit einer hohen Last zu rechnen ist, die noch dazu stoßweise auftritt.

Ein moderner Stack im Jahr 2024 muss nicht nur die funktionalen Anforderungen erfüllen, sondern auch nichtfunktionale. Dabei darf auf keinen Fall der Aspekt der Entwicklungsergonomie fehlen. Genauso wie Nutzer moderne und intuitive Oberflächen erwarten, ist es wichtig, Entwickler zu motivieren und ihr Engagement zu erhalten. Entsprechend sollten sich die eingesetzten Technologien leichtgewichtig und entsprechend dem Stand der Technik darstellen – kaum

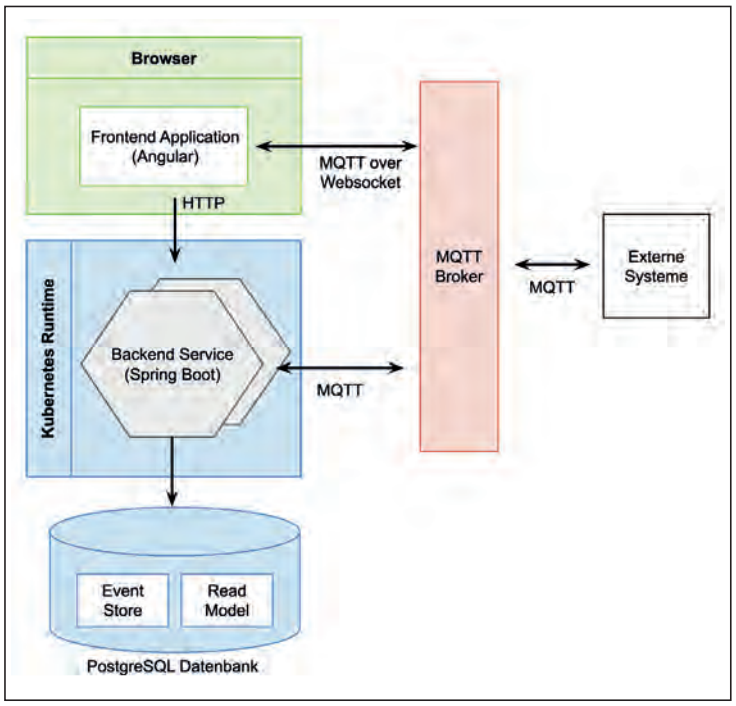

Abb. 1: Architekturübersicht der Beispielanwendung

jemand wird eine hohe Motivation entwickeln, wenn eine Anwendung heute noch mit jQuery oder JSF entwickelt werden soll.

Als Backend-Technologie wurde Spring Boot gewählt – jedoch als Event-getriebene Architektur und mit Event Sourcing für die Persistenz. Im nachfolgenden Beitrag „Datengeschichte schreiben“ wird nach einer Einführung in die zugrundeliegenden Ideen von Event Sourcing und den Gründen zur Wahl dieses Modells die Umsetzung in Spring Boot beschrieben. Dabei wird

auch darauf eingegangen, wie eine Persistenz mit einer relationalen Datenbank umgesetzt werden kann. Als konkrete Plattform wurde für das Frontend die Form einer clientseitigen Webanwendung gewählt, da dies eine extrem breite Unterstützung auf unterschiedlichsten Plattformen, leichtes Rollout der Anwendung und hohe Verfügbarkeit an Entwicklerressourcen bedeutet. Zudem ist es damit prinzipiell möglich, ohne serverseitigen Zustand auszukommen, was die Entwicklung der Anwendung erleichtert und gleichzeitig auf die Skalierbarkeit einzahlt. Alternativen dazu wären eine native Mobilanwendung oder eine serverseitig gerenderte Webanwendung.

Als konkrete Technologien verwenden wir Angular für das Frontend, da dieses Framework wesentliche konzeptionelle Elemente mitbringt, die anderen Libraries oder Frameworks wie React und Vue fehlen. Diese sind jedoch wichtig, um kosteneffizient zu entwickeln und die Anwendung langfristig effizient warten zu können. Zudem macht Angular einige Vorgaben, die zusammen mit etablierten Best Practices dafür sorgen, dass Entwickler schnell zwischen Projekten wechseln können – auch ein wichtiger Punkt, wenn es um (Kosten-)Effizienz geht. Dank TypeScript, starker Konzepte, ausgereiftem Tooling und langfristigem Support mit Weiterentwicklung durch das Angular-Team und Google ist Angular eine hervorragende Wahl sowohl für hohe Entwicklungsgeschwindigkeit als auch langfristige Sicherung des Investments. Der Beitrag „Echt jetzt!“ widmet sich der Kommunikation zwischen Backend-Komponenten und Frontend durch Verwendung von Messaging und Verarbeitung von Events für den Frontend-Zustand. Hier lohnt es sich, auch einen Blick auf die Parallelen zwischen dem Event Sourcing im Backend und den Konzepten für das Frontend zu werfen.

Als Betriebsumgebung wurde Kubernetes ausgewählt. Auch wenn Kubernetes mittlerweile als Standard etabliert und nicht mehr wegzudenken ist, so ist tiefergehende Erfahrung mit der Entwicklung von Anwendungen für Kubernetes und entsprechende Betriebserfahrung noch nicht überall in vollem Umfang vorhanden. Genau hier setzt der Beitrag „Tuning für Kubernetes“ an, indem er zahlreiche Tipps für die Nutzung von Kubernetes nicht nur für Einsteiger, sondern vor allem auch für fortgeschrittene Anwender und Teams liefert. Bei dem großen Umfang des Themas kann zwar nicht jeder Tipp mit vollständigen Beispielen veranschaulicht werden, es sei aber jedem Team, das Kubernetes als Betriebsplattform nutzt, angeraten, die einzelnen Punkte zum Anlass zu nehmen, sich mit dem jeweiligen Thema tiefergehend zu beschäftigen.

Die Anforderung einer Authentifizierung mit dem Konferenz-Badge wird genutzt, um einen Überblick über OAuth2, OpenID Connect und den Spring Authorization Server im Beitrag „Leichtgewichtig, anpassbar und standardkonform“ zu liefern. Damit lassen sich zusammen mit dem Spring Framework besondere und individuelle Anforderungen an die Authentifizierung mit verhältnismäßig geringem Aufwand umsetzen. Dabei wird weiterhin auf offene und bewährte Standards gesetzt und dennoch die Möglichkeit geschaffen, die Verfahren an die eigenen Anforderungen anzupassen. Auf diese Weise lassen sich auch weitere Integrationen in individuellen Systemumgebungen effizient bewältigen.

Zur Kommunikation wird auf Publish-Subscribe-Messaging gesetzt. Die Wahl fällt hier auf MQTT. Das Protokoll stammt ursprünglich aus dem IoT-Umfeld. Welche Eigenschaften MQTT für diesen wichtigen Bereich auszeichnen, der oft unsichtbar im Hintergrund wichtige Funktionen umsetzt, und wie MQTT funktioniert, wird in dem Beitrag „Postdienst für das Internet der Dinge“ behandelt. Ist der Einsatz von MQTT denn nicht lediglich auf die Kommunikation von Sensordaten beschränkt? Nein! Für welche breiten Einsatzzwecke MQTT geeignet ist, und wieso MQTT auch eine sehr gute und leichtgewichtige Wahl für Enterprise-Anwendungen sein kann, wird im Beitrag „Nur was für Maker und Bastler – oder?“ erläutert.

Die hier gewählten Optionen sind nicht zwangsläufig kanonische Entscheidungen oder die einzig sinnvollen Varianten. Es wurden vielmehr gezielt Kandidaten ausgewählt, die das Potenzial haben, bekannte Lösungsmuster herauszufordern und das Spektrum von möglichen Designs zu erweitern. Jeder Leser soll diesen Ansatz als Einladung auffassen, Alternativen zu betrachten, auch wenn ein „Das haben wir noch nie so gemacht“-Gedanke dabei aufkommt. Umso mehr ist das ein Grund, sich zumindest gedanklich auf neue Wege zu begeben und Architekturalternativen bewusst zu identifizieren und zu bewerten. Selbst wenn am Ende die Rückkehr zum Bekannten folgt, bleibt als Erkenntnisgewinn ein tieferes Verständnis für die Gründe der Entscheidung und die Stärken, Schwächen und Grenzen der gewählten Lösung.

Weitere Events mit unseren Schwerpunktautoren

- **Workshop**: „Kubernetes Workshop: How to get to Speed“ (DevOpsCon London, DevOpsCon Berlin) von Thomas Kruse
- **Workshop**: „Mastering CI/CD on Kubernetes“ (DevOpsCon London, DevOpsCon Berlin) von Thomas Kruse
- **Workshop**: „CI/CD mit Kubernetes in der Praxis“ (MAD Summit München) von Thomas Kruse

Thomas Kruse unterstützt bei der Trion Unternehmen als Architekt, Trainer und Entwickler für Projekte, die Java-Technologien einsetzen. Sein Fokus liegt auf Java-basierten Webanwendungen und Cloud- und Containertechnologien, speziell mit Kubernetes. In seiner Freizeit engagiert er sich für Open-Source-Projekte und organisiert die Java User Group und die Frontend-Freunde in Münster. Kontaktieren Sie ihn gerne für weitergehende Unterstützung zu den behandelten Themen.

www.trion.de **tk@trion.de**

Das (Industrial) Internet of Things wird zum Player der digitalen Transformation

„Nur was für Maker und Bastler“ – oder?

Das Internet of Things, kurz IoT, hat sich in den vergangenen Jahren weit über seine bescheidenen Anfänge – die oft mit Hobbybasteleien auf Geräten wie Arduino und dem Raspberry Pi in Verbindung gebracht werden – hinaus entwickelt. IoT repräsentiert in der Geschäftswelt bereits heute in vielen Bereichen ein wichtiges strategisches Feld und wird in naher Zukunft an Wichtigkeit und Umfang gewinnen. Eine wichtige IoT-Technologie ist dabei das Protokoll MQTT.

von Jens Deters

Ein Blick auf die Wachstumsprognosen dieses Markts allein für Deutschland unterstreicht diese Aussage eindrucksvoll: Laut dem Portal Statista sollen sich die Umsätze in den kommenden vier Jahren mit den Schwerpunkten Automotive IoT und industrielles IoT (Industrie 4.0) verdoppeln (**Abb. 1**). Auch die Steigerung der zu erwartenden Anzahl der IoT-Verbindungen ist demnach beachtlich. Hier fallen besonders der Consumer-Markt und der Bereich Industrial IoT auf (**Abb. 2**). Weltweit wird bis 2030 zudem nahezu eine Verdoppelung der Anzahl der mit IoT verbundenen Geräte erwartet (**Abb. 3**).

Warum jedes Unternehmen langfristig IoT einsetzen muss

Die „digitale Transformation“ bezeichnet den Prozess der Nutzung digitaler Technologien, um neue oder bestehende Geschäftsprozesse, die Unternehmenskultur und Kundenerfahrungen zu verändern und zu verbessern. Ziel ist es, Unternehmen so anzupassen und neu auszurichten, dass sie den stetig wechselnden Geschäfts- und Marktanforderungen in der digitalen Ära gerecht werden. In einem wettbewerbsintensiven Umfeld ist die Fähigkeit eines Unternehmens, sich digital zu transformieren, entscheidend für seinen langfristigen Erfolg, seine Wettbewerbsfähigkeit und letztlich für seinen Bestand am Markt.

Damit das gelingen kann, ist ein (möglichst) einfacher und uneingeschränkter Zugang zu und Austausch von Daten (Rohfakten oder Zahlen) und Informationen (analysierte und interpretierte Daten) von zentraler Bedeutung. Doch die digitale Transformation bedeutet nicht nur die Einführung neuer Technologien, sondern es ist auch eine Veränderung der Unternehmenskultur und Arbeitswelt damit verbunden, denn ein entscheidender Faktor ist die Förderung einer digitalen Kultur, die Innovation, Flexibilität und kontinuierliches Lernen unterstützt. Das umfasst auch die Akzeptanz der Demokratisierung von Arbeit und die Einführung neuer Arbeitsweisen wie Remotearbeit, kollaborative Werkzeuge und Plattformen.

Das Internet der Dinge

Das Internet der Dinge, also generell das Konzept der Verbindung von Geräten und Systemen über das Internet, ermöglicht eine beispiellose Sammlung und Analyse von Daten, die Unternehmen nutzen können, um ihre Prozesse zu optimieren, Innovationen voranzutreiben und neue Geschäftsmodelle zu entwickeln.

Damit ist die Integration von IoT-Technologien nicht mehr nur eine Option für Unternehmen, die nach Innovation streben. Sie wird vielmehr zur Notwendigkeit für alle, die in der modernen, digital vernetzten Wirtschaft wettbewerbsfähig bleiben wollen. Die Fähigkeit, Daten zu erfassen, zu analysieren (Daten zu Informationen)

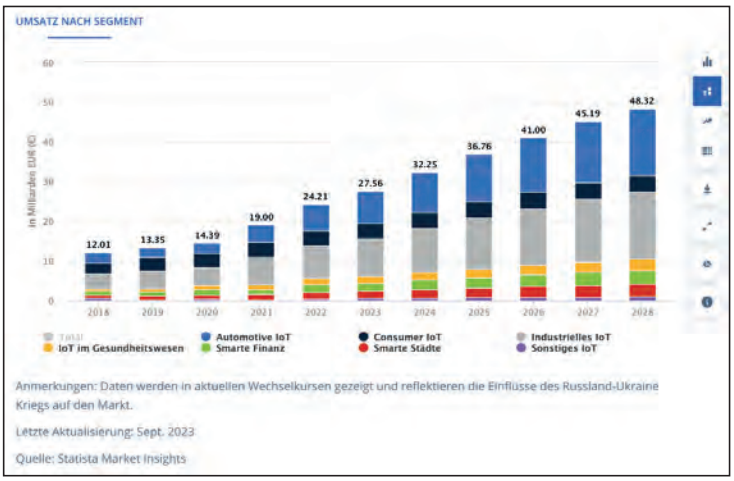

Abb. 1: Umsätze in Deutschland im IoT-Bereich von 2018 bis 2028 (in Milliarden) (Bildquelle [1])

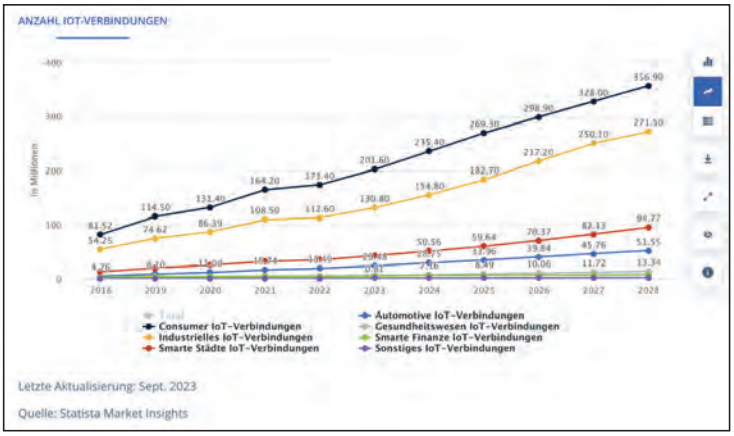

Abb. 2: Anzahl der IoT-Verbindungen in Deutschland von 2018 bis 2028 (in Millionen) (Bildquelle [2])

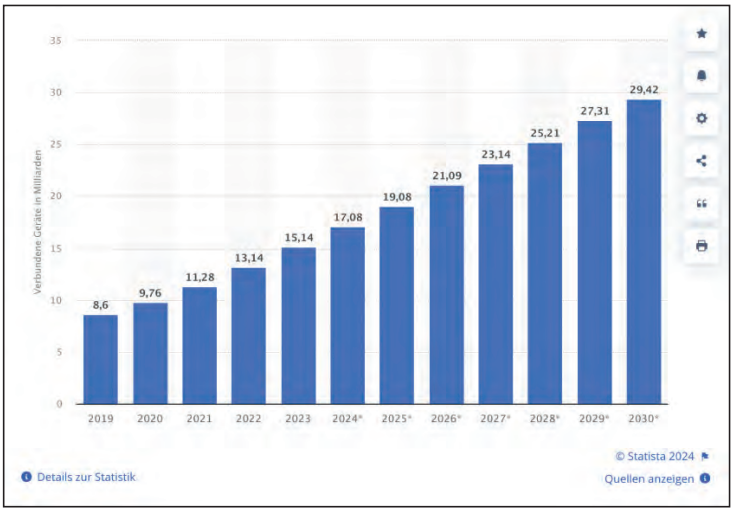

Abb. 3: Anzahl der mit dem IoT verbundenen Geräte weltweit von 2019 bis 2030 (in Milliarden) (Bildquelle [3])

und darauf basierend zu handeln, wird immer mehr zur Grundlage erfolgreicher Geschäftsstrategien und digitaler Transformation.

Im Folgenden soll der Begriff des Internets der Dinge etwas weiter definiert werden, denn IoT umfasst bei Weitem nicht nur die Datensammlung von Sensoren oder Anwendungen im Hobbybereich. IoT ist vielmehr ein rasant wachsendes komplexes Ökosystem, das Hardware, Netzwerktechnologien, Softwareplattformen, Anwendungen und Datenanalysen umfasst. Es beinhaltet die Integration von Technologien über mehrere Ebenen hinweg, von der physischen Geräteebene bis hin zu komplexen Datenanalyse- und Anwendungsebenen. Die Stärke des IoT liegt zunächst in seiner Fähigkeit, die physische und die digitale Welt zu verbinden, um intelligente, datengesteuerte Umgebungen und Erlebnisse zu schaffen.

In der Annahme, dass ein „Ding" jede Form von Ereignisquelle und -empfänger sein kann, umfasst IoT hier nicht nur die physische Welt, sondern auch „virtuelle Dinge" wie den Knopf einer digitalen Benutzeroberfläche, eine Spracheingabe oder ein ausgelöstes Ereignis eines Software-Service. So kann beispielsweise ein Bestellvorgang in E-Commerce- und IoT-Systemen durch eine Vielzahl von Mechanismen initiiert werden, die die Vielseitigkeit und Integration technologischer Lösungen in den Alltag widerspiegeln. Ein wesentliches Merkmal dieser vielfältigen Auslöser ist, dass die nachgelagerten Systeme, die die Bestellungen bearbeiten und ausführen, in der Regel nicht wissen, auf welche Weise ein Datenereignis generiert wurde. Die Hauptaufgabe dieser Systeme besteht darin, die eingehenden Bestellanforderungen effizient und zuverlässig zu verarbeiten, unabhängig von ihrer Herkunft:

- *Virtueller Check-out:* Er kann über einen Warenkorb in einem Onlineshop erfolgen, wobei der Kunde auf einen virtuellen Knopf klickt, unabhängig davon, ob das auf einem Desktopcomputer oder über eine Mobile-App geschieht.
- *Sprachbefehle:* Sprachassistenten wie Alexa, Google Assistant oder Siri ermöglichen es Benutzern, Bestellungen durch einfache Sprachbefehle auszulösen.
- *Physische Knöpfe:* In einigen kommerziellen oder Produktionsumgebungen können physische Knöpfe eingesetzt werden, um Nachbestell-Events für bestimmte Produkte oder Materialien auszulösen.
- *Sensoren:* In bestimmten Anwendungsfällen können Sensoren verwendet werden, um einen Bestellvorgang automatisch auszulösen, sobald die Stückzahl eines Produkts unter einen bestimmten Wert fällt. Zum Beispiel in automatisierten Produktionslinien oder dem oft zitierten smarten Kühlschrank, der eigenständig feststellt, welche Lebensmittel zur Neige gehen, und entsprechende Nachbestellungen veranlasst.

Die Vielfalt der Anwendungsfälle, die sich über unterschiedliche digitale und physische Domänen erstrecken können, unterstreicht die Bedeutung von Interoperabilität und Flexibilität in modernen I(o)T-Systemen, die eine reibungslose Integration und Kommunikation zwischen unterschiedlichen Technologien und Plattformen gewährleisten müssen.

Einige Beispiele von Vorteilen und IoT-Anwendungsfällen in verschiedenen Branchen fassen wir kurz zusammen:

- Industrie 4.0
- Durch genaue Erfassung der Auslastung und Anforderung an Maschinen durch bestehende Prozesse kann die Effizienz der Ausnutzung von einzelnen Maschinen oder ganzen Produktionslinien erhöht werden (Overall Equipment Effectiveness, OEE)
- Produktionsunternehmen können durch die Analyse gesammelter Sensordaten Wartungsbedarfe vorhersagen und Ausfallzeiten reduzieren
- E-Commerce, Banken, Versicherungen, Automotive
- Produktbasierte Dienstleistungen (z. B. Pay-per-use-Modelle)
- Produkte und Dienstleistungen personalisieren und verbessern, was zu einer erhöhten Kundenzufriedenheit und -bindung führt
- Einblicke in Markttrends und Kundenverhalten; Unternehmen können diese Informationen nutzen, um ihre Produkte und Dienstleistungen besser auf die Bedürfnisse ihrer Kunden abzustimmen und vorausschauend zu agieren
- Unternehmen in der Logistikbranche können die Lieferkettenintegrität durch lückenlose Überwachung sicherstellen
- Wettbewerbsvorteile durch die Fähigkeit, schnell auf Veränderungen zu reagieren, Prozesse zu optimieren und Innovationen zu entwickeln
- Nachhaltigkeit und Umweltschutz
- Nachhaltigere Wirtschaftsweise durch optimierte Ressourcennutzung und verringerte Umweltauswirkungen
- Reduktion des Energieverbrauch durch intelligente Gebäudesteuerungen

IoT-Technologien

Ganz generell geht es im IoT um Anwendungen, die physische Geräte über das Internet verbinden. Damit diese Anwendungen effizient funktionieren können, müssen die verwendeten Protokolle und Technologien spezielle Anforderungen erfüllen. Diese Anforderungen umfassen vor allem: Skalierbarkeit (von einigen wenigen Geräten bis zu Millionen oder sogar Milliarden von Geräten), Ausfallsicherheit, IT-Security (Verschlüsselung, Authentifizierung, Autorisierung und Integritätsschutz), Interoperabilität (nahtlose Kommunikation und Integration über Geräte- und Systemgrenzen hinweg), Energieeffizienz und geringe Latenzzeit.

Die Auswahl der geeigneten Protokolle und Technologien hängt von den spezifischen Anforderungen der jeweiligen IoT-Anwendung ab, einschließlich der Art der Daten, die übertragen werden müssen, der Umgebung, in der die Geräte betrieben werden, und der kritischen Bedeutung der Datenübertragung und -verarbeitung für die Anwendung.

Eines der am weitesten verbreiteten Protokolle ist das Ende der 1990er-Jahre entwickelte MQTT. MQTT ist für viele IoT-Anwendungen aufgrund seiner Leichtgewichtigkeit, Einfachheit, Effizienz und Flexibilität sehr gut geeignet, insbesondere in Szenarien, die eine zuverlässige Nachrichtenübermittlung über unzuverlässige Verbindungen erfordern.

Architekturmuster

Es lassen sich grundlegend zwei typische Architekturmuster unterscheiden.

Ereignisgesteuerte Architektur (Event-driven Architecture)

Einer erfolgreichen Implementierung von IoT-Anwendungen liegt ein wesentliches Architekturmuster zugrunde: Event-driven Architecture. Bei diesem Muster wird mittels Ereignissen (Events) eine Kommunikation zwischen verschiedenen Komponenten eines Systems implementiert. Dabei ist ein Ereignis eine signifikante Änderung im Zustand eines Systems oder einer Komponente, die übermittelt werden soll.

Im IoT geht es um Anwendungen, die physische Geräte über das Internet verbinden, und damit diese Anwendungen effizient funktionieren, müssen die verwendeten Protokolle und Technologien spezielle Anforderungen erfüllen.

Hier kommt ein Publish/Subscribe-Modell zur Anwendung, d. h. Produzenten (Publisher) veröffentlichen Ereignisse ohne Kenntnis der Empfänger (Subscriber). Subscriber abonnieren die für sie interessanten Ereignisse und reagieren darauf, ohne den Publisher kennen zu müssen. Der Fokus liegt auf dem Ereignis selbst und dessen Bedeutung. Daher ist dieses Muster so hervorragend geeignet für die Entkoppelung von Diensten und Komponenten in verteilten Systemen. Diese lose Kopplung ermöglicht es Entwicklern, Dienste unabhängig voneinander zu entwickeln, zu modifizieren und zu skalieren, und erlaubt eine sehr vereinfachte Integration heterogener Systeme und Technologien.

Zudem unterstützen ereignisgesteuerte Architekturen natürlicherweise asynchrone Verarbeitung: Ein Dienst kann ein Ereignis senden und sofort mit anderen Aufgaben fortfahren, ohne auf eine Antwort warten zu müssen. Insbesondere in verteilten Umgebungen, in denen die Latenzzeit ein Faktor ist, wird so die Effizienz und Leistung des Systems erheblich verbessert.

Nachrichtengesteuerte Architektur (Message-driven Architecture)

Eine „klassische“ Message-driven Architecture hingegen konzentriert sich auf die Verwendung von Nachrichten für die Anforderung von Aktionen oder die Übermittlung von Daten zwischen verschiedenen Teilen eines Systems. Eine Nachricht ist ein Datenpaket mit Informationen, die von einem Sender zu einem spezifischen Empfänger übertragen werden. Dabei kommt oft ein Point-to-Point- oder Request-Response-Modell zur Anwendung, bei dem der Sender eine Nachricht direkt an einen spezifischen Empfänger sendet. Durch den Request/Response-Ansatz verarbeitet der Empfänger die Nachricht und führt eine entsprechende Aktion aus und/oder sendet dann eine Antwort zurück.

Mit dem bereits erwähnten MQTT steht ein Protokoll zur Verfügung, das für beide Architekturmuster geeignet ist: Zunächst ist MQTT speziell im Hinblick auf ereignisbasierte Kommunikation (Publish/Subscribe) konzipiert, also für Szenarien, in denen Ereignisse oder Zustandsänderungen kommuniziert werden müssen, wie Sensorwerte, Statusupdates oder Alarme.

Aber es ist auch möglich, ein Request-Response-Muster mit MQTT zu implementieren. So könnten Clients Requests an ein spezifisches Thema (Topic) senden und auf eine Antwort via eines speziellen Antwortthemas warten. Ein anderer Client, der die Anfrage empfängt und bearbeitet, veröffentlicht die Antwort dann über dieses Antwort-Topic.

In MQTT v5.0 gibt es für die Korrelation von Anfragen und Antworten zudem die Möglichkeit, ein Response Topic in der MQTT-Message zu hinterlegen sowie eine eindeutige Zuordnung mittels frei wählbarer Correlation Data zu erlauben.

Real-World-Beispiele

Abschließend sollen einige Beispiele aus Case Studies von HiveMQ [4] einen Eindruck von konkreten Anwendungsfällen in verschiedenen Branchen geben, in denen MQTT eingesetzt wurde.

Öffentlicher Nahverkehr

Die Stadtwerke München setzen HiveMQ für Fahrgastinformationen in Echtzeit ein. [5]

- Herausforderungen
- Zuverlässige, hochverfügbare Bereitstellung von Echtzeitdaten
- Erforderliche Fernverwaltung der Anzeigen, z. B. Statusaktualisierungen, Neuladen und Neustart
- Lösung
- Anzeige- und Überwachungsgeräte abonnieren und veröffentlichen MQTT-Nachrichten, um Fahrgastinformationen in Echtzeit zu liefern
- Jedes Anzeigegerät veröffentlicht seinen Status alle 30 Sekunden in Form einer MQTT-Nachricht
- Ergebnisse
- Einsatz in über 500 Informationsmonitoren und über 2 000 Bus- und Straßenbahnhaltestellen
- Bereitstellung von Informationen in Echtzeit mit ständiger Überwachung
- Geringere Kosten für Implementierung und Wartung

Smart Cities

Die IAV GmbH nutzt MQTT und Fahrzeugdaten für das städtische Sturzflutmanagement. [6]

- Herausforderungen
- Minimierung der Auswirkungen von regionalen Sturzfluten durch Starkregenereignisse
- Die Vorhersage von außergewöhnlichen Regenereignissen ist schwierig

- Nutzung von Daten zur Warnung der Behörden vor lokal begrenzten Starkregenereignissen
- Lösung
- MQTT-Broker sendet zuverlässig ortsspezifische Echtzeitdaten von Autos in die Cloud
- MQTT-Daten werden zur Integration mit Wetterdiensten an Apache Kafka übertragen
- Ergebnisse
- Schaffung eines Frühwarnsystems, das Überschwemmungen in Städten aufgrund starker Regenfälle vorhersagen kann
- Einrichtung eines zentralen IT-Diensts für alle MQTT-Projekte

Consumer-Markt

Awair skaliert die Konnektivität für eine sichere Überwachung der Luftqualität. [7]

- Herausforderungen
- Herstellung von vernetzten Produkten für die Überwachung der Luftqualität in Privathaushalten und Unternehmen
- Bereitstellung einer skalierbaren und zuverlässigen MQTT-Plattform
- Lösung
- Kontrolle für jedes Gerät, um Over-the-Air-Updates der Gerätefirmware zu initiieren
- Einfache Integration mit MQTT in die Awair-Plattform
- Erstellung einer eigenen Geräteautorisierungslogik mittels MQTT, mit Awair-Geräte-Repository-Verbindung
- Ergebnisse
- HiveMQ verarbeitet über 100 Millionen Nachrichten pro Tag
- Nahtlose Integration für schnellere Einblicke

Jens Deters hatte in den vergangenen 25 Jahren diverse Rollen im Bereich IT und Telekommunikation: Softwareentwickler, IT-Trainer, Projektmanager, Produktmanager, Consultant und Niederlassungsleiter. Heute verantwortet er das Professional Services Team bei HiveMQ. Als langjähriger Experte im Bereich MQTT und IoT sowie Entwickler des beliebten GUI-Tools MQTT.fx unterstützen er und sein Team Kunden täglich bei der Implementierung der weltweit interessantesten IoT UseCases bei führenden Marken und Unternehmen.

Links & Literatur

[1] https://de.statista.com/outlook/tmo/internet-der-dinge/deutschland

[2] https://de.statista.com/outlook/tmo/internet-der-dinge/deutschland#volumen

[3] https://bit.ly/48ryWju

[4] https://www.hivemq.com/case-studies

[5] https://www.hivemq.com/case-studies/swm-munich-transit-system/

[6] https://www.hivemq.com/case-studies/iav/

[7] https://www.hivemq.com/case-studies/awair/

Einstieg in das Messaging-Protokoll MQTT

Postdienst für das Internet der Dinge

MQTT ist ein leichtgewichtiges, auf Publish/Subscribe basierendes Messaging-Protokoll, das für die Kommunikation zwischen Maschinen über das Internet (TCP/IP) konzipiert wurde. Dabei wurde besonders darauf geachtet, minimale Netzwerkbandbreite und Geräteressourcen zu verbrauchen, was es ideal für die Verwendung in einer Vielzahl von IoT-Szenarien macht, einschließlich solcher mit ferngesteuerten Standorten oder sehr eingeschränkter Netzwerkinfrastruktur. Wir schauen uns Konzepte und Features an.

von Jens Deters

MQTT ist durch sein Pub/Sub-Pattern vor allem dafür geeignet, ereignisgesteuerte Architekturen zu implementieren. Das Muster ermöglicht eine lose Kopplung zwischen Nachrichtenproduzenten (Publishern) und Konsumenten (Subscribers), ohne dass eine direkte Verbindung zwischen diesen erforderlich ist. Neben den grundsätzlichen Konzepten (Client/Broker, Publish/Subscribe) lernen wir im Folgenden auch MQTT-Features wie Quality of Service, Retained Messages, Persistent Sessions, Last Will and Testament und weitere kennen.

Kernkonzepte

Beginnen wir mit dem MQTT-Broker. Das ist der zentrale Server, der die Nachrichtenverwaltung übernimmt. Er empfängt alle Nachrichten von den Publishern und leitet sie an die Subscriber weiter, basierend auf den Themen, die sie abonniert haben. Dann gibt es die MQTT-Clients:

- *Publisher:* Ein Gerät oder eine Anwendung, die Nachrichten an den Broker sendet, typischerweise mit Informationen von Sensoren oder anderen Datenquellen
- *Subscriber:* Ein Gerät oder eine Anwendung, das bzw. die sich beim Broker für ein bestimmtes Thema registriert und Nachrichten zu diesem Thema empfängt
- *gleichzeitige Funktion:* MQTT-Clients können auch gleichzeitig als Publisher und Subscriber agieren, d. h. sowohl Informationen senden als auch auf Informationen reagieren (z. B. Steuerbefehle, Parameteränderungen)

Und schließlich gibt es Topics: In MQTT wird jede Nachricht einem Thema zugeordnet. Die Themen dienen als Filter, die bestimmen, welche Nachrichten an welche Subscriber weitergeleitet werden (**Abb. 1**).

MQTT vs. Message Queues

Der Name MQTT und die Frage, ob das Protokoll als Nachrichtenwarteschlange implementiert ist oder nicht, sorgt häufig für Verwirrung. MQTT bezieht sich auf das Produkt MQseries von IBM und nicht auf Message Queue. Daher ist es nicht richtig, MQTT als Message Queuing Telemetry Transport auszuschreiben, sondern vielmehr als MQ Telemetry Transport.

Seit 2013 ist es üblich, MQTT nicht mehr als Akronym, sondern als Eigenname für das Protokoll zu verwenden. Auch wenn die Arbeitsgruppe immer noch den Namen OASIS Message Queuing Telemetry Transport Technical Committee trägt, wurde seinerzeit beschlossen: „MQTT should not be standing for anything“ [1].

Auch unabhängig von der Namensgebung ist es wichtig, die Unterschiede zwischen MQTT und einer herkömmlichen Nachrichtenwarteschlange zu verstehen. So speichert Letztere Nachrichten, bis sie verbraucht werden. Bei der Verwendung einer Queue wird also jede eingehende Nachricht dort gespeichert, bis sie von einem Client (oft als Consumer bezeichnet) abgeholt wird. Holt kein Client die Nachricht ab, bleibt sie in der Warteschlange. In einer Message Queue ist es nicht vorgesehen, dass eine Nachricht von keinem Client verarbeitet wird, wie es in MQTT der Fall ist, wenn ein Topic von niemandem abonniert ist.

Ein weiterer großer Unterschied besteht darin, dass in einer herkömmlichen Warteschlange eine Nachricht in der Regel nur von einem Verbraucher verarbeitet werden kann. Die Last wird auf alle Verbraucher einer

Warteschlange verteilt. In MQTT verhält es sich genau umgekehrt: Jeder, der das Thema abonniert, erhält die Nachricht. Ähnlich dem Abonnieren eines E-Mail-Newsletters, der von vielen Subscribern gleichzeitig oder zeitversetzt empfangen wird.

Eine Warteschlange ist darüber hinaus viel unflexibler als ein MQTT Topic. Bevor sie verwendet werden kann, muss sie explizit mit einem separaten Befehl erstellt werden. Erst nachdem die Warteschlange benannt und erstellt wurde, ist es möglich, Nachrichten zu veröffentlichen oder zu empfangen. Im Gegensatz dazu sind MQTT Topics extrem flexibel und können im Handumdrehen erstellt werden.

Abhängig von den vom Broker bereitgestellten zusätzlichen Features kann mit dynamischen MQTT Topics auch eine entsprechend dynamische Role-based Access Control (RBAC) realisiert werden, um die Sicherheit und Flexibilität im Umgang mit der Nachrichtenübermittlung zu verbessern. RBAC basiert auf der Zuweisung feingranularer Zugriffsrechte für Benutzer und Geräte auf Basis von Berechtigungen für Rollen statt direkt für individuelle Nutzer oder Clients.

Durch die Anwendung von RBAC auf Topic-Ebene wird diese Strategie weiter verfeinert, indem die Zugriffsrechte dynamisch an die Struktur der Topics angepasst werden, über die die Kommunikation erfolgt. Sie sind jedoch spezifisch für die Implementierung des Brokers und nicht im MQTT-Protokoll selbst definiert [2]. Nun schauen wir uns die Eigenschaften von MQTT genauer an.

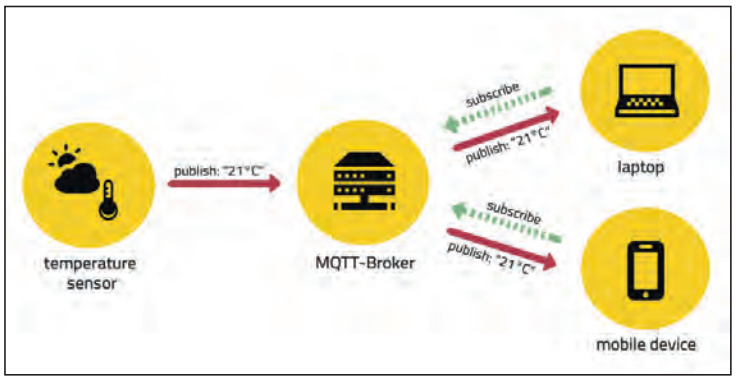

Abb. 1: MQTT-Architektur

Abb. 2: Die Topic-Ebenen werden mittels Schrägstrich getrennt

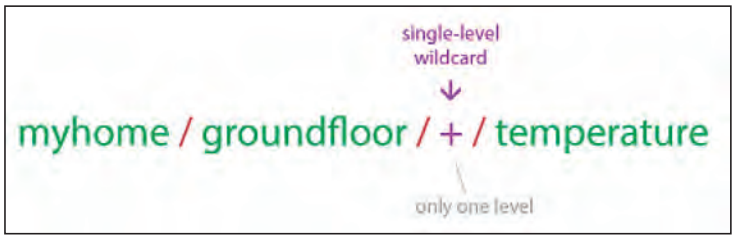

Abb. 3: Der Single-Level-Platzhalter

Leichtgewichtigkeit

MQTT ist leichtgewichtig, das heißt, es ist einfach, effizient und nicht ressourcenintensiv. Das Protokoll wurde mit dem Ziel entwickelt, kleine Datenmengen über unzuverlässige Netzwerke mit begrenzter Bandbreite und Konnektivität zu senden. Im Vergleich zu anderen Protokollen hat MQTT einen sehr kleinen Codefußabdruck, einen geringen Overhead und damit einen niedrigen Stromverbrauch. Durch den minimalen Paket-Overhead ist MQTT ideal für den Einsatz in Geräten mit begrenzter Verarbeitungsleistung, Speicher und Batterielebensdauer wie z. B. Sensoren und andere IoT-Geräte.

Im Gegensatz zu anderen Protokollen, die textbasierte Formate verwenden wie z. B. HTTP oder SMTP, verwendet MQTT ein binäres Nachrichtenformat für die Kommunikation zwischen Clients und Brokern. Damit ist MQTT sehr effizient in Bezug auf Bandbreiten- und Speichernutzung, vor allem im IoT-Kontext in Netzwerken mit begrenzter Kapazität oder bei der Kommunikation über Mobilfunknetze.

Ein weiterer wichtiger Aspekt seiner Leichtgewichtigkeit ist, dass MQTT auf Clientseite extrem einfach zu implementieren ist. Die Benutzerfreundlichkeit war bei der Entwicklung von MQTT ein zentrales Anliegen und macht es zu einer perfekten Lösung für Geräte mit begrenzten Ressourcen.

Topics

In MQTT bezieht sich der Begriff Topic auf eine UTF-8-Zeichenkette, die Nachrichten für einen verbundenen Client filtert. Ein Topic besteht aus einer oder mehreren Ebenen, die durch einen Schrägstrich getrennt sind (Topic-Level-Trennzeichen), wie es **Abbildung 2** zeigt.

Im Vergleich zu einer klassischen Message Queue sind MQTT Topics sehr leichtgewichtig. Der Client muss das gewünschte Topic nicht erstellen, bevor er es veröffentlicht oder abonniert. Der Broker akzeptiert jedes gültige Topic ohne vorherige Initialisierung.

Topic Wildcards

Es gibt verschiedene Platzhalter, die in den MQTT Topics verwendet werden können.

Single-Level: +

Der Single-Level-Platzhalter wird durch das Pluszeichen dargestellt und ermöglicht die Ersetzung einer einzelnen Topic-Ebene. Wenn Sie ein Thema mit einem einstufigen Platzhalter abonnieren, wird jedes Thema, das eine beliebige Zeichenfolge anstelle des Platzhalters enthält, gefunden (**Abb. 3**).

Ein Abonnement von *myhome/groundfloor/+/temperature* kann zum Beispiel die Ergebnisse liefern, die **Abbildung 4** zeigt.

Abb. 4: Ergebnisse von myhome/groundfloor/+/temperature

Abb. 5: Der Multi-level-Platzhalter

Multi-Level:

Der Multi-Level-Platzhalter deckt mehrere Topic-Ebenen ab. Er wird durch das Rautensymbol (#) dargestellt und muss als letztes Zeichen im Topic platziert werden, dem ein Schrägstrich vorangestellt ist (**Abb. 5**).

Wenn ein Client ein Topic mit einem Multi-Level-Platzhalter abonniert, empfängt er alle Nachrichten eines Topics, das mit dem Muster vor dem Platzhalterzeichen beginnt, unabhängig von der Länge oder Tiefe des Topics (**Abb. 6**). Wenn das Topic nur mit der Raute # angegeben wird, empfängt der Client alle Nachrichten, die an den MQTT-Broker gesendet werden.

Es ist jedoch wichtig zu bedenken, dass das Abonnieren mit einem mehrstufigen Platzhalter ein Antipattern sein kann, wenn ein hoher Durchsatz erwartet wird. Dann kann das Abonnement dazu führen, dass eine große Menge an Nachrichten an den Client übermittelt wird, was sich auf die Systemleistung und die Bandbreitennutzung auswirken kann.

Best Practice

Hier folgt eine Auswahl von Best Practices für das Erarbeiten und Verwenden von MQTT Topics:

- *Eine logische und hierarchische Struktur* erleichtert es generell, Nachrichten nach Funktion, Standort oder Gerätetyp zu klassifizieren. Sie erleichtert auch eine spätere Skalierung.

Trivia: Auf 3 folgt 5

Warum folgte auf MQTT 3.1.1 direkt MQTT 5.0 und nicht 4.0? Die Antwort ist erstaunlich einfach: Das MQTT-Protokoll definiert einen festen Header im *CONNECT*-Paket. Dieser Header enthält einen Ein-Byte-Wert für die Protokollversion. MQTT 3.1 hat den Wert *3* als Protokollversion und MQTT 3.1.1 hat den Wert *4*. Um den Wert der Protokollversion auf der Leitung mit dem offiziellen Namen der Protokollversion zu synchronisieren, verwendet die neue MQTT-Version „5“ sowohl für den Protokollnamen als auch für den Wert [9].

Abb. 6: Beispiel für die Ergebnisse beim Einsatz des Multi-Level-Platzhalters

- *Definition eines Namensraums:* Alle beteiligten Geräte und Anwendungen kennen, verstehen und verwenden die gleichen Begriffe und Beziehungen in diesem Bereich/Kontext. Zum Beispiel: *myhome/…*
- Über eine *Versionierung der Topic-Struktur* (vergleichbar mit REST APIs) wird Zukunftsfähigkeit sichergestellt; zudem kann damit das erwartete Payload-Schema bekannt gegeben werden. Die Topic-Struktur sollte zusammen mit dem Payload-Schema(s) sehr gut gemäß ihrer Version dokumentiert werden. Zum Beispiel: *myhome/v1/…* oder *myhome_v1/…*
- *Client-Identifier im Topic vorsehen:* Das ermöglicht zum einen eine gezielte Adressierung von Benutzern oder Geräten und zum anderen spielt es, eine entsprechende Unterstützung des Brokers vorausgesetzt, eine wichtige Rolle bei der Definition von Zugriffsrechten. Zum Beispiel:

```
myhome/v1/{insert_mqtt_client-id_here}/temperature
myhome/v1/{insert_mqtt_client-id_here}/humidity
```

- *Leerzeichen in Topics vermeiden:* In Skripten, Befehlszeilenschnittstellen oder Konfigurationsdateien können Leerzeichen das Parsing und die Handhabung erschweren, sodass zusätzliche Anführungszeichen oder Escaping erforderlich sind. Es sollten besser Unterstriche (_) oder Bindestriche (-) als Trennzeichen verwendet werden.
- *Führenden Schrägstrich vermeiden* (/), denn dieser erzeugt ein unnötiges Topic der ersten Ebene, das im Grunde eine leere Zeichenfolge ist, weil MQTT Topics immer mit einen Topic-Level beginnen, auf das dann der Topic-Trenner (/) folgt. Zum Beispiel impliziert */home/livingroom/light* ein leeres Topic der ersten Ebene vor dem /, gefolgt von *home* als zweiter Ebene. Das kann zu Ineffizienz (zusätzlichem Ressourcenverbrauch) und zu Inkonsistenzen bei der Strukturierung und Interpretation von Topics führen.
- *Großbuchstaben vermeiden:* MQTT Topics unterscheiden zwischen Groß- und Kleinschreibung. Das bedeutet, *Home/LivingRoom/Light* und *Home/Livingroom/Light* werden als völlig unterschiedliche Topics betrachtet. Die Verwendung von Großbuchstaben erhöht das Risiko der Fehleranfälligkeit, insbesondere in Umgebungen, in denen mehrere Entwickler oder Teams unabhängig voneinander (unternehmensweite) Topics definieren.
- *Das Abonnieren des Wildcard Topics # für sich allein sollte vermieden werden,* denn damit abonniert der Client alle aktuellen und zukünftigen Topics des Brokers. Somit erhält der Client jede einzelne Nachricht, die über den Broker läuft, unabhängig davon, ob sie für

den Client relevant ist oder nicht. Das kann zu erheblichen Leistungsproblemen führen, insbesondere wenn der Datenverkehr hoch ist oder die Anzahl der Nachrichten groß wird. Mehr zu Topics und zu benutzerdefinierten MQTT-Spezifikationen finden Sie unter [3].

Nutzlast (Payload)

Payload meint in MQTT den eigentlichen Inhalt oder die Daten einer Nachricht, die zwischen einem Client und einem MQTT-Broker übermittelt werden. In MQTT ist die Payload agnostisch. Das bedeutet, es wird kein spezifisches Format für die Daten festgelegt, die in der Payload einer Nachricht übertragen werden. Die Payload kann also Plain Text, binäre Daten oder strukturierten Text wie JSON, XML oder jedes andere Datenformat enthalten, das der Sender und der Empfänger verstehen und verarbeiten können. Diese Flexibilität ermöglicht es, MQTT in einer Vielzahl von Anwendungen und Systemen einzusetzen. Das geht allerdings zulasten der Sicherheit, denn während MQTT Mechanismen für eine sichere Kommunikation auf Transportebene (z. B. TLS/SSL) bietet, obliegt die Sicherheit der in der Payload übertragenen Daten den Anwendungsentwicklern und ist nicht Teil der MQTT-Spezifikation. Da die Payload sensible oder kritische Informationen enthalten kann, ist es wichtig, Sicherheitsmaßnahmen wie Verschlüsselung zu berücksichtigen.

Die maximale Größe der Payload in einer MQTT-Nachricht liegt theoretisch durch die MQTT-Spezifikation bei 256 MB. Allerdings hängt die tatsächliche maximale Größe der Payload in der Praxis von den Einschränkungen des verwendeten Brokers, der Netzwerkbandbreite und den Ressourcen der beteiligten Geräte ab. Für ressourcenbeschränkte Geräte oder Netzwerke mit geringer Bandbreite ist es ratsam, kleinere Payloads zu verwenden, um Effizienz und Leistung zu optimieren.

MQTT ist stateful!

MQTT ist ein zustandsbehaftetes (stateful) Protokoll. Der Zustand einer Kommunikation zwischen einem Client und einem MQTT Broker wird über die Dauer einer Netzwerksitzung hinweg aufrechterhalten. Diese wesentliche Eigenschaft von MQTT ermöglicht es dem Protokoll, eine außerordentlich zuverlässige Nachrichtenübermittlung zu gewährleisten.

Diese Zuverlässigkeit wird durch mehrere Mechanismen und Konzepte erreicht, die das Protokoll definiert. Zu den wichtigsten gehören Quality of Service (QoS) Levels, Persistent Sessions, Client Keep-alive, Last Will and Testament (LWT) (oder auch nur Will) und Retained Messages.

QoS

MQTT unterstützt drei Stufen von Quality of Service (QoS): QoS 0, QoS 1 und QoS 2. Mit den einzelnen Stufen hat es Folgendes auf sich:

- *QoS 0* bietet eine „At most once"-Zustellungsgarantie, bei der Nachrichten ohne Bestätigung gesendet werden und verloren gehen können. Es ist die niedrigste QoS-Stufe und wird in der Regel in Situationen verwendet, in denen ein Nachrichtenverlust akzeptabel oder die Nachricht nicht kritisch ist. QoS 0 eignet sich beispielsweise für den Versand von Sensordaten, bei denen ein gelegentlicher Datenverlust die Gesamtergebnisse nicht wesentlich beeinträchtigen würde.
- *QoS 1* bietet eine „At least once"-Zustellung, bei der der Nachrichtenempfang bestätigt und bei Bedarf die Nachricht erneut gesendet wird. Bei QoS 1 sendet der Publisher die Nachricht an den Broker und wartet auf die Bestätigung, bevor er fortfährt. Wenn der Broker nicht innerhalb einer bestimmten Zeit antwortet, sendet der Herausgeber die Nachricht erneut. Diese QoS-Stufe wird in der Regel in Situationen verwendet, in denen der Verlust von Nachrichten inakzeptabel, die Duplizierung von Nachrichten jedoch tolerierbar ist. QoS 1 eignet sich z. B. für das Senden von Befehlsnachrichten an Geräte, bei denen ein verpasster Befehl schwerwiegende Folgen haben könnte, eine Verdoppelung der Befehle jedoch nicht.
- *QoS 2* bietet eine „Exactly once"-Zustellung, bei der Nachrichten bestätigt und erneut gesendet werden, bis sie genau einmal vom Teilnehmer empfangen werden. QoS 2 ist die höchste QoS-Stufe und wird in der Regel in Situationen verwendet, in denen der Verlust oder die Duplizierung von Nachrichten völlig inakzeptabel ist. Bei QoS 2 führen der Publisher und der Broker einen zweistufigen Bestätigungsprozess durch, bei dem der Broker die Nachricht so lange speichert, bis sie vom Subscriber empfangen und bestätigt wurde. Diese QoS-Stufe wird in der Regel für kritische Nachrichten wie Finanztransaktionen oder Notfallwarnungen verwendet.

Es ist wichtig, zu beachten, dass höhere QoS-Stufen in der Regel mehr Ressourcen erfordern und zu höheren Latenzzeiten und mehr Netzwerkverkehr führen können. Daher ist es entscheidend, die geeignete QoS-Stufe auf der Grundlage der spezifischen Anforderungen Ihrer Anwendung zu wählen. Mehr Details zum Quality of Service in MQTT finden sie unter [4].

Persistent Sessions

MQTT ermöglicht es Clients, eine „Persistent Session" zu wählen. In diesem Fall behält der Broker Informationen über die Session des Clients, auch wenn die Verbindung getrennt wird. Dazu gehören Abonnements und Nachrichten, die mit QoS 1 oder QoS 2 gesendet, aber noch nicht an den Client ausgeliefert wurden. Wenn der Client sich erneut verbindet, identifiziert der Broker die Client-Session anhand der obligatorischen ID des Clients und setzt die Kommunikation nahtlos fort [5].

Client Keep-alive

Der Keep-alive-Mechanismus hilft dabei, festzustellen, ob eine Verbindung zwischen Client und Broker

noch aktiv ist. Clients senden in regelmäßigen Abständen *PINGREQ*-Nachrichten, auf die der Broker mit *PINGRESP* antwortet. Wenn innerhalb einer festgelegten Zeit keine Antwort erfolgt, geht der Client davon aus, dass die Verbindung unterbrochen ist, und kann entsprechende Maßnahmen ergreifen, wie z. B. einen erneuten Verbindungsversuch [6].

Retained Messages

MQTT-Broker können letzte Nachrichten für ein Topic speichern, auch wenn diese bereits an alle aktuellen Abonnenten ausgeliefert wurden. Neue Abonnenten des Topics erhalten sofort die zuletzt gespeicherte Nachricht, was sicherstellt, dass wichtige Informationen sofort verfügbar sind, selbst wenn sie erst nach der ursprünglichen Übermittlung abonniert werden [7].

Last Will and Testament (LWT)

Die Funktion „Last Will and Testament“ (LWT) ermöglicht es Clients, andere über deren unerwartete Verbindungsabbrüche zu informieren. Wenn ein Client eine Verbindung zu einem Broker herstellt, kann er direkt in der *CONNECT*-Nachricht eine Last-Will-Message angeben. Diese Nachricht folgt der Struktur einer regulären MQTT Message, einschließlich eines Topics, der Retained Flag, des QoS und einer Payload.

Der Broker speichert diese Nachricht, bis er eine ungewollte Trennung der Verbindung durch den Client feststellt. Wenn der Broker die Unterbrechung feststellt, sendet er den „letzten Willen“ an alle abonnierten Clients des entsprechenden Topics.

Der Broker verwirft die gespeicherte LWT-Nachricht, wenn der Client die Verbindung mit der *DISCONNECT*-Nachricht ordnungsgemäß trennt [8].

Getting started with MQTT

HiveMQ ist es ein großes Anliegen, die MQTT-Community zu fördern. Dafür gibt es zahlreiche frei nutzbare Angebote:

- Einen MQTT-Guide für Anfänger und Fortgeschrittene [10].
- Zum Ausprobieren gibt es HiveMQ Cloud. Das ist eine von HiveMQ bereitgestellte Cloud-native IoT-Messaging-Plattform, die eine verlässliche und skalierbare IoT-Gerätekonnektivität einfach macht. Die kostenlose Serverless-Variante stellt für den einfachen Einstieg einen MQTT-Broker bereit [11].
- Für den Einstieg in MQTT gibt es auch grafische Tools, wie die kostenlose MQTT.fx HiveMQ Cloud Edition für MacOS und Windows [12]. Eine ausführliche Anleitung ist unter [13] zu finden.
- Über die HiveMQ-Community erhält man Zugang zu exklusiven Veranstaltungen, Programmen und Ressourcen [14].
- HiveMQ hat außerdem mit der HiveMQ University ein freies Bildungsangebot rund um MQTT gestartet [15]. Neben kostenlosem Lehrmaterial wird als besonderes Highlight auch die Zertifizierung zum HiveMQ MQTT Associate sowie HiveMQ MQTT 3.1.1 Professional inklusive LinkedIn-Badge angeboten.

MQTT 5.0

Die bisher vorgestellten MQTT-Features basieren auf der Spezifikation MQTT 3.1.1, die im Juni 2014 von OASIS als Standard veröffentlicht wurde. Im März 2019 kam MQTT 5.0 heraus (Kasten: „Trivia“), mit dem eine ganze Reihe neuer Funktionen und Verbesserungen gegenüber MQTT 3.1.1 vorgestellt wurden, die darauf abzielen, die Skalierbarkeit, Zuverlässigkeit und Benutzerfreundlichkeit des Protokolls zu verbessern.

Shared Subscriptions

Shared Subscriptions sind eine sehr wichtige neue Funktion in MQTT 5.0, die speziell für die Skalierung und Effizienzsteigerung der Nachrichtenübermittlung in großen, verteilten Systemen konzipiert wurde. Diese Funktion kann als eine Form des clientseitigen Load Balancing betrachtet werden, das es ermöglicht, Nachrichten, die an ein bestimmtes Thema gesendet werden, nicht an jeden abonnierenden Client einzeln zuzustellen, sondern stattdessen unter den Mitgliedern einer Gruppe, die das Thema gemeinsam abonniert haben, zu verteilen.

Aber Achtung: Obwohl Shared Subscriptions eine Art Load Balancing auf Clientseite darstellen, unterscheiden sie sich von traditionellen Load-Balancing-Strategien, die typischerweise auf der Netzwerk- oder Serverebene implementiert werden. Bei MQTT Shared Subscriptions liegt der Fokus auf der effizienten Verteilung der Nachrichtenlast auf der Anwendungsebene, basierend auf der Abonnementlogik des MQTT-Protokolls. Dieses Konzept ist besonders nützlich in Szenarien, in denen die Lastverteilung oder die Reduzierung von Nachrichtenduplikaten für ein effizienteres Nachrichtenmanagement erforderlich sind.

User Properties

User Properties in MQTT 5.0 bieten eine flexible Methode zur Übertragung benutzerdefinierter Metadaten zusammen mit MQTT-Nachrichten und sind im Wesentlichen einfache UTF-8-String-Schlüssel-Wert-Paare, die an fast jede Kategorie von MQTT-Paketen angehängt werden können.

Die Stärke der User Properties liegt in ihrem unbegrenzten Potenzial – solange die maximale Nachrichtengröße nicht überschritten wird, kann eine unendliche Anzahl von Key-Value-Paaren verwendet werden. Das eröffnet enorme Möglichkeiten zur Anreicherung von MQTT-Nachrichten mit zusätzlichen Metadaten und erleichtert die flüssige Übertragung von Informationen zwischen Publishern, Broker und Subscribern, ohne die Informationen in die eigentliche Payload zu schreiben.

Payload Format Description

Die Payload Format Description ist ein optionaler Header in MQTT-Nachrichten, der angibt, ob die Payload der Nachricht binär oder textbasiert ist. Diese Information hilft dem Empfänger der Nachricht, die Payload korrekt zu interpretieren und zu verarbeiten, ohne auf externe Vereinbarungen oder Annahmen angewiesen zu sein:

- *Binär (0)* gibt an, dass die Payload binäre Daten enthält. Das ist nützlich für die Übertragung von nicht textbasierten Daten, wie z. B. verschlüsselten Daten oder binären Dateiformaten.
- *UTF-8-kodierte Charakterdaten (1)* geben an, dass die Payload als UTF-8-kodierter Text vorliegt. Das erleichtert die Übertragung und Interpretation von menschenlesbaren Informationen oder JSON/XML-Datenstrukturen.

Content Type

Der Content Type ist eine weitere optionale Eigenschaft in MQTT-5.0-Nachrichten, die eine genauere Beschreibung des Payload-Formats bietet. Dieser Header ähnelt dem Content-Type-Header in HTTP und kann verwendet werden, um den MIME-Typ der Payload anzugeben. Dadurch kann der Empfänger nicht nur die Art der Daten (Text oder binär) erkennen, sondern auch das spezifische Format verstehen, z. B., ob es sich um JSON, XML oder ein anderes Datenformat handelt. So signalisiert zum Beispiel ein Content Type von *application/json*, dass die Payload im JSON-Format vorliegt, was eine entsprechende Verarbeitung oder Deserialisierung durch den Empfänger ermöglicht.

Request-Response-Pattern

Das Request-Response-Pattern in MQTT 5.0 ist eine wichtige Erweiterung, die eine explizite Unterstützung der anfrage- und antwortbasierten Kommunikation zwischen Clients ermöglicht. In MQTT 3.1.1 mussten Entwickler eigene Konventionen implementieren, um ein Request-Response-Muster zu realisieren, was oft zu erhöhter Komplexität und inkonsistenten Implementierungen führte. MQTT 5.0 adressiert das durch die Einführung eingebauter Mechanismen, die diese Art der Kommunikation vereinfachen und standardisieren.

Das Request-Response-Pattern in MQTT 5.0 nutzt zwei neue Eigenschaften im *PUBLISH*-Paket, um Anfragen und Antworten zu verknüpfen:

- *Response Topic:* Beim Senden einer Anfrage (Request) kann der Client ein Response Topic angeben, an das die Antwort gesendet werden soll. Das informiert den Empfänger darüber, wohin die Antwortnachricht zu publizieren ist.
- *Correlation Data:* Zusätzlich kann der anfragende Client Correlation Data übermitteln – eine Art Identifikator oder Token, das in der Antwortnachricht enthalten sein sollte. Das ermöglicht dem anfragenden Client, die Antwort eindeutig seiner ursprünglichen Anfrage zuzuordnen.

Flow Control

Flow Control in MQTT 5.0 ist ein Mechanismus, der es Clients und Servern ermöglicht, den Nachrichtenfluss zu steuern, um Überlastung und mögliche Verluste von Nachrichten zu verhindern. Das Request-Response-Pattern stellt in gewisser Weise auch eine Art Flusskontrolle dar, wenn durch Koordinierung der Kommunikation dieser Mechanismus hilft, den Nachrichtenfluss zu verwalten und sicherzustellen, dass die Systeme nicht mit Anfragen überlastet werden.

Darüber hinaus kann über die Einstellung „Receive Maximum“ die maximale Anzahl von gleichzeitig nicht bestätigten *PUBLISH*-Nachrichten (QoS 1 oder QoS 2) definiert werden, die ein Client oder Server empfangen darf. Durch Anpassen dieses Wertes können Clients und Server ihre Verarbeitungs- und Speicherfähigkeiten berücksichtigen, um Überlastung zu vermeiden.

Zudem definiert die Angabe der „Maximum Packet Size“ die maximale Größe eines MQTT-Pakets, das gesendet oder empfangen werden kann. Auch diese Angabe ermöglicht es Clients und Servern, ihre Kommunikation entsprechend der Netzwerkkapazität und den Systemressourcen zu steuern. Indem Nachrichten, die größer als die festgelegte maximale Paketgröße sind, abgelehnt werden, hilft diese Einstellung, den Ressourcenverbrauch zu verwalten und die Systemstabilität zu gewährleisten.

Jens Deters hatte in den vergangenen 25 Jahren diverse Rollen im Bereich IT und Telekommunikation: Softwareentwickler, IT-Trainer, Projektmanager, Produktmanager, Consultant und Niederlassungsleiter. Heute verantwortet er das Professional Services Team bei HiveMQ. Als langjähriger Experte im Bereich MQTT und IoT sowie Entwickler des beliebten GUI-Tools MQTT.fx unterstützen er und sein Team Kunden täglich bei der Implementierung der weltweit interessantesten IoT UseCases bei führenden Marken und Unternehmen.

Links & Literatur

[1] https://www.oasis-open.org/committees/download.php/49028/OASIS_MQTT_TC_minutes_25042013.pdf

[2] https://docs.hivemq.com/hivemq-enterprise-security-extension/latest/index.html

[3] https://www.hivemq.com/resources/defining-mqtt-specification-to-meet-iot-iiot-industry-implementation/

[4] https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/

[5] https://www.hivemq.com/blog/mqtt-essentials-part-7-persistent-session-queuing-messages/

[6] https://www.hivemq.com/blog/mqtt-essentials-part-10-alive-client-take-over

[7] https://www.hivemq.com/blog/mqtt-essentials-part-8-retained-messages/

[8] https://www.hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament/

[9] https://www.hivemq.com/blog/mqtt5-essentials-part1-introduction-to-mqtt-5/

[10] https://www.hivemq.com/mqtt/

[11] https://www.hivemq.com/products/mqtt-cloud-broker/

[12] https://softblade.de/en/download-hivemq-cloud-edition/

[13] https://www.hivemq.com/article/mqtt-testing-debugging-using-mqttfx-hivemq-cloud/

[14] https://www.hivemq.com/community/

[15] https://university.hivemq.com

Echtzeit und Events im Frontend

Echt jetzt!

Echtzeitinteraktionen im Web werden immer wichtiger. Es ist also an der Zeit, sich einen Überblick darüber zu verschaffen. Zudem sehen wir uns einen realen Anwendungsfall an, der moderne Frontend-Architekturen mit Angular erfordert.

von Karsten Sitterberg

Echtzeitsysteme in der Informatik sind dadurch gekennzeichnet, dass sie eingehende Informationen innerhalb eines definierten Zeitraums verarbeiten. Dabei kann der Zeitraum überraschend lang sein, es geht also nicht um die Verarbeitungsgeschwindigkeit an sich. Der Browser ist eine recht dynamische Umgebung, vor allem, wenn man es mit JavaScript-Anwendungen zu tun hat. Hier arbeitet der Just-in-Time-Compiler, um den Code zu optimieren und damit zu beschleunigen. Das ist nicht möglich, während derselbe Anwendungscode gerade arbeitet. Folglich gibt es Unterbrechungen. Dazu kommt eine dynamische Speicherverwaltung mittels Garbage Collector, der ebenfalls Pausen verursacht. Selbst die Browser, die die Ablaufumgebung bereitstellen, sind nicht alle gleich: Je nach Art der Verarbeitung hat die eine oder andere Browserimplementierung die Nase vorn. Die zugrunde liegende Hardware könnte nicht unterschiedlicher sein: Das Spektrum reicht von eingeschränkten oder günstigen Mobilgeräten bis hin zu extrem schnellen Multi-Core-Maschinen.

Hart oder weich – Echtzeitsysteme und der Browser

Damit wird es schwierig bis unmöglich, harten Echtzeitanforderungen zu genügen, wenn man nicht gleichzeitig den Verarbeitungszeitraum entsprechend erweitert. Und doch setzen wir heute Browser für Videokonferenzen ein und spielen darauf Multiplayer-Spiele. Bei beiden Anwendungsfällen sind durchaus Auswirkungen zu spüren, wenn harte Echtzeitanforderungen nicht eingehalten werden. Je nach Nutzer und Szenario kann ein Ruckeln im Ego-Shooter sogar dazu führen, dass anschließend auch das Umfeld von den Auswirkungen der entstandenen Frustration betroffen ist. Um solchen Ansprüchen prinzipiell zu genügen, reicht jedoch ein weiches Echtzeitsystem. Und das kann ein Browser. Dieser Beitrag beleuchtet, welche Optionen es für eine passende Kommunikation mit dem Browser gibt und welche Architekturaspekte im Frontend zur Umsetzung existieren.

Treiber oder Quelle der Wahrheit?

Bevor wir uns verschiedenen Wegen der Kommunikation widmen, stellt sich noch eine grundsätzliche Frage: Was wird kommuniziert? Erhält der Browser einen kontinuierlichen Fluss von Daten oder eher Pakete mit dazwischenliegenden Pausen? Und wie sind die Daten beschaffen: Handelt es sich eher um kontinuierliche Rohdaten von Ereignissen, die die Webanwendung zu einem passenden Modell der Umwelt aggregiert und verarbeitet, oder sind es eher vollständige Aktualisierungen von (Teil-)Zuständen der Anwendung? Vereinfacht dargestellt, handelt es sich bei Ersterem um Event Sourcing. Dabei werden alle Informationen vollständig und derart übertragen, dass jede Anwendung aus dem Datenfluss wieder ihren jeweiligen Zustand ermitteln kann. Damit sind Anforderungen wie „Undo“ oder eine „Zeitreise“ zu beliebigen Punkten der Vergangenheit möglich. Dabei ist allerdings viel Kommunikation notwendig und die erforderliche Bandbreite bei der Übertragung sowie die notwendigen Ressourcen zur Verarbeitung können prohibitiv sein. Gerade bei mobilen Endgeräten, die sich teilweise nur auf geringe Bandbreiten stützen können, die dann auch noch nach Verbrauch zu bezahlen sind, kann ein optimierter Umgang mit den Daten gewünscht oder erforderlich sein. Auch hier ist es für weiche Echtzeitsysteme vorteilhaft, ein Kommunikationsverfahren zu wählen, bei dem die Anwendung aktiv über Änderungen informiert wird, um diese dann zu verarbeiten. Dabei spricht man von ereignisgesteuerten (Event-driven) Anwendungen.

Ein ganz anderes Verfahren wäre, dass sich die Anwendung hin und wieder Aktualisierungen abholt. Dabei würde sich die Oberfläche bei neuen Daten zwar

Wenn auf unterschiedliche Arten von Events gewartet werden soll, dann werden unter Umständen mehrere parallele Requests notwendig, was den Ressourcenverbrauch unnötig erhöht.

aktualisieren, aber eventuell viel später, als es angemessen wäre. Zudem muss die Anwendung bei umfangreichen Zustandsänderungen relativ viele Ressourcen in einem kurzen Zeitraum aufwenden, um diese zu verarbeiten. Und auch wenn es gar keine Änderungen gibt, entsteht Systemlast, da die Anwendung ja regelmäßig wegen potenzieller Aktualisierungen anfragen muss.

Qual der Wahl

Es gibt verschiedene Möglichkeiten, Events aus dem Backend im Frontend bereitzustellen. Generell können sogar einfache HTTP Requests genutzt werden, um via Long Polling auf Events aus dem Backend zu warten. Beim Long Polling ist der Begriff Echtzeit-Events allerdings mit Vorsicht zu genießen, da neue Events je nach Implementierung möglicherweise erst nach Ablauf des nächsten Polling-Intervalls zur Verfügung stehen. Damit wird die Aktualisierungsperiode unvorhersagbar. Wenn auf unterschiedliche Arten von Events gewartet werden soll, dann werden unter Umständen auch mehrere parallele Requests notwendig, was den Ressourcenverbrauch unnötig erhöht. Je nach Browser und verwendetem Protokoll kann es außerdem sein, dass nicht beliebig viele parallel offene Requests erlaubt sind (beispielsweise haben Browser für HTTP 1.1 ein Limit von sechs parallelen Requests je Host definiert).

Für beide Probleme stehen bessere Alternativen zur Verfügung, die im Folgenden jeweils kurz vorgestellt werden:

Server-sent Events (SSE) werden über eine persistente HTTP-Verbindung geschickt, und zwar immer nur vom Server in Richtung Client. Als Besonderheit hat SSE ein Verfahren eingebaut, um bei einer Verbindungsunterbrechung ohne zusätzliche Implementierung durch die Webanwendung eine Wiederaufnahme nach dem letzten verarbeiteten Element durchzuführen. Das wird direkt durch den Webbrowser implementiert, allerdings muss dazu im Backend auch ein entsprechender Puffer vorhanden sein. Sonst gehen bei Verbindungsunterbrechungen potenziell Events verloren. Positiv an SSE ist, dass die Events über eine normale HTTP-Verbindung laufen (Listing 1). So ist eine hohe Kompatibilität auch mit Unternehmensfirewalls gegeben.

Eine weitere Möglichkeit zur Echtzeitkommunikation im Web ist **WebRTC** (Web Real-Time Communication). WebRTC ist vor allem dafür gedacht, Multimediainhalte über Peer-to-Peer-Verbindungen direkt zwischen Clients auszutauschen. Damit werden z. B. Videokonferenzsysteme im Browser umgesetzt. Es ist zwar auch möglich, mit WebRTC Daten abseits von Audio und Video auszutauschen, allerdings ist für eine Kommunikation per WebRTC ein relativ komplexes Set-up erforderlich und viele Unternehmensfirewalls sperren die notwendigen Ports.

Von Google stammt **gRPC** (Remote Procedure Calls), ein bidirektionales Protokoll, das auf HTTP/2 aufsetzt. Damit ist es möglich, Daten zwischen Server und Client – und umgekehrt – zu streamen. Dank Codegenerierung des dabei verwendeten ProtoBuf-Datenformats, hoher Effizienz und breiter Sprachinteroperabilität hat sich gRPC vor allem in der Integration von Microservices und auch nativer mobiler Anwendungen etabliert. Auch wenn man wegen des zugrunde liegenden HTTP/2 vermuten könnte, dass gRPC im Browser und damit in Webanwendungen problemlos funktioniert, ist das leider nicht der Fall. Damit Webanwendungen mit einem gRPC-Backend kommunizieren können, wird gRPC-web benötigt, das z. B. durch einen Proxyserver in gRPC umgewandelt werden kann. Im Enterprise-Umfeld kann es auch zu Problemen mit Firmenfirewalls bzw. Proxyservern kommen, sodass gRPC aktuell besser für reine Backend-Kommunikation genutzt wird.

WebSockets bieten einen bidirektionalen Kommunikationskanal, um Events direkt zwischen Server und Client auszutauschen. WebSockets basieren zudem nicht direkt auf HTTP, sondern stellen ein eigenes Protokoll dar, bei dem alle Daten wie über eine reine TCP-Verbindung gesendet werden. Lediglich für den Verbindungsaufbau wird HTTP verwendet. Von Firmenfirewalls werden WebSockets häufig blockiert, das sollte vor Entwicklungsstart abgeklärt werden. Vor allem kollaborative Tools, aber auch Online-(Multiplayer-)Games oder Chats werden häufig mit WebSockets umgesetzt.

Für das hier betrachtete Anwendungsbeispiel werden keine Multimedia- sondern lediglich JSON-Daten be-

Listing 1: HTTP-Body eines Server-sent-Event-Datenstroms

```
id: 12343\n
data: {"msg": "First message"}\n\n
event: userlogon\n
id: 12344\n
data: {"username": "John123"}\n\n
id: 12345\n
data: GOOG\n
data: 556\n\n
```

nötigt, auch eine Peer-to-Peer-Verbindung wird nicht gebraucht. Es sollten jedoch zuverlässige Datenübertragungen auch in schwierigen Netzwerkumgebungen (z. B. auf Konferenzen bzw. Massenveranstaltungen oder hinter Firewalls/Proxies) möglich sein. Damit reduziert sich die Auswahl auf SSE und WebSockets. Beiden Alternativen ist gemein, dass sie eine stehende Verbindung vom Client zum Backend aufbauen und es dadurch ermöglichen, ohne Verzögerung Events vom Server zu empfangen. Außerdem können in beiden Fällen verschiedene Event-Typen, die in unterschiedlichen Teilen der Anwendung verarbeitet werden, einfach über dieselbe Verbindung übertragen werden (Multiplexing).

Bei Servers-sent Events muss das jeweilige Backend dafür speziell Unterstützung anbieten, zudem passt die Verarbeitung der Daten am besten zu einzelnen Datenpaketen, was eine Einschränkung bezüglich der Erweiterbarkeit dieser Lösung für bereits jetzt angedachte Anwendungsfälle bedeuten würde. Damit bleiben lediglich WebSockets in der engeren Auswahl. Diese sind jedoch ein reines Transportmittel, es fehlt dabei ein Protokoll für die Übertragung und Korrelation der Daten. Auch hier gibt es wieder verschiedene mehr oder weniger standardisierte Lösungen, sodass von der Entwicklung eines eigenen Kommunikationsprotokoll über WebSockets abgesehen werden kann. Auch dazu wollen wir uns einen kurzen Überblick verschaffen.

RSocket ist ein Protokoll, das ursprünglich von Netflix entwickelt wurde, um reaktive Datenströme zwischen Anwendungen bereitzustellen. Netflix war auch Vorreiter bei der Entwicklung von RxJava, den Reactive Extensions für Java. Nachdem innerhalb von Microservices mit dem reaktiven Programmiermodell gearbeitet wurde, lag es nahe, das auch auf die Kommunikation zwischen verschiedenen Anwendungen auszudehnen. Zudem stellte sich HTTP als ineffizient für bestimmte Verarbeitungsformen und Anforderungen heraus. RSocket ist ein binäres Streamingprotokoll mit reaktiver Semantik und kann über andere Transportprotokolle wie TCP und WebSockets gelegt werden. Es unterstützt auch die Wiederaufnahme einer Verbindung, nachdem diese unterbrochen wurde. Folgende Kommunikationsmodelle werden durch asynchrone Nachrichten unterstützt:

- Request-Response (RPC), Strom mit Einzelelement
- Request Stream, Strom mit potenziell unendlichen Elementen
- Fire-and-Forget, keine Antwort
- Channel, bidirektionales Streaming

RSocket wird ohne zusätzliche Infrastruktur durch die jeweiligen Anwendungen bereitgestellt.

Aus dem Umfeld von Message Brokern stammt **STOMP** (Simple Text Oriented Messaging Protocol). Es ist nachrichtenorientiert und kann beliebige Binärdaten als Inhalt transportieren, auch wenn die Kontrollstrukturen selbst im Textformat gehalten sind. Der Vorteil von STOMP liegt darin, dass es oft durch die Message-Broker-Infrastruktur angeboten wird. Ist ein Message-

Listing 2: Konfiguration von MQTT im MqttService

```
import mqtt, { IClientOptions } from 'mqtt';

@Injectable({providedIn: 'root'})
export class MqttService {
  private client?: MqttClient;

  constructor(private auth: AuthService, @Inject(SOCKET_HOST) private
                                                    socketHost: string) {}

  initMQTTSocket(options?: IClientOptions): void {
    options = {
      protocol: 'ws',
      hostname: this.socketHost,
      port: 80,
      path: '/mqtt',
      username: this.auth.getUsername(),
      password: this.auth.getPW(),
      ...options
    };
    this.client = mqtt.connect(options);
  }
}
```

Listing 3: Konfiguration von MQTT im MqttService

```
import mqtt, { IClientOptions } from 'mqtt';

export interface MqttEvent {
  topic: string;
  payload: string;
}

@Injectable({providedIn: 'root'})
export class MqttService {
  private messagesSub = new Subject<MqttEvent>();
  private client?: MqttClient;

  initMQTTSocket(options?: IClientOptions): void {
    // ... Parse Options (see Listing 1)
    this.client = mqtt.connect(options);

    this.client.on('message', (topic, payload) => {
      this.messagesSub.next({topic, payload: payload.toString()});
    });
  }
}
```

Broker im Einsatz, können auch Pub/Sub-Verfahren in Kombination mit STOMP zur Entkopplung eingesetzt werden. Gleichzeitig können die zugrunde liegenden Strukturen der Messaging-Infrastruktur eine Unterstützung beim Design der Kommunikationsmuster und der Aufteilung der Daten liefern.

MQTT (MQ Telemetry Transport) ist ein Protokoll, das ursprünglich zur Übertragung von Telemetriedaten zwischen Maschinen mit begrenzten Ressourcen entwickelt wurde. Die Daten werden bei MQTT als Nachrichten auf sogenannten Topics per Publish/Subscribe übertragen. Dadurch lassen sich beispielsweise Events entsprechend dem Typ oder der Verarbeitung direkt in verschiedene Topics separieren. MQTT wird typischerweise von entsprechender Messaging-Infrastruktur bereitgestellt. Diese Infrastruktur kann auch dazu genutzt werden, verschiedene Microservices lose gekoppelt miteinander zu integrieren, und stellt damit eine leichtgewichtige Alternative zu anderen Systemen wie RabbitMQ oder Kafka dar. In der Beispielanwendung könnte das genutzt werden, da die Events so mit Hilfe der Topics entsprechend der Stelle der Verarbeitung in der Webanwendung zugeordnet werden können.

Im Fall von Webanwendungen nutzt MQTT meist WebSockets für die Datenübertragung. Auf das Web bezogen könnte man also sagen: WebSockets sorgen für die Verbindung, MQTT für die Abwicklung der Datenübertragung und darin enthalten ist die konkrete (Event-)Datenstruktur. Um das MQTT-Protokoll nicht von Hand im Web nachbauen und anbinden zu müssen, wird im weiteren Verlauf die JavaScript-Library MQTT.js verwendet, die das erledigt.

Anbindung MQTT.js

Zur Verwendung von MQTT im Client bietet es sich an, das npm-Paket MQTT.js per *i -S mqtt* zu installieren. Um eine Verbindung zum MQTT-Broker aufzubauen, benötigt MQTT.js eine gewisse Grundkonfiguration, die in Listing 2 dargestellt ist. Da MQTT im Browser einen WebSocket als Verbindung benötigt, ist das verwendete Protokoll immer *ws* bzw. *wss* (WebSocket/WebSocket Secure, analog zu *http* und *https*). Weiterhin müssen Hostname, Port, Pfad und Autorisierungsdaten zum Aufbau der MQTT-WebSocket-Verbindung angegeben werden. Alternativ zu den Einzel-Properties kann auch ein URL angegeben werden.

Beispielhaft ist in Listing 2 auch gezeigt, dass der Hostname für den WebSocket (Variable *socketHost*) und Username/Passwort (durch den *AuthService*) z. B. per Angular Dependency Injection konfiguriert werden können. Um die MQTT-Verbindung aufzubauen, muss *mqtt.connect()* aufgerufen werden.

Messages und Events: MQTT und RxJS

Um MQTT-Nachrichten entgegenzunehmen, muss im Fall von MQTT.js ein Listener für das *message*-Event des *MqttClient* registriert werden. Der Event-Listener wird dann für jedes ankommende MQTT-Message-Event neu aufgerufen. Um diesen Event-Strom im Frontend besser verarbeiten zu können, bietet es sich an, eine Library zu nutzen, die sich darauf spezialisiert hat. Mit RxJS bringt Angular eine solche Library standardmäßig mit. Bei RxJS handelt es sich um die Reactive Extensions für JavaScript, also ein reaktives Programmiermodell. RxJS erlaubt es, Events in Form sogenannter Observable Streams zu verarbeiten. Mit Observables können die Event-Ströme anhand einer deklarativen Pipeline-Syntax gefiltert, modifiziert und aggregiert werden.

In Listing 3 ist zu sehen, dass nach Erzeugung der MQTT-Verbindung (*mqtt.connect()*) ein On-Message Listener am Client (*this.client.on()*) registriert wird. An dieser Stelle scheint bei MQTT.js allerdings das darunterliegende WebSocket API durch: Es kann lediglich ein allgemeiner Event Listener auf *message* Events registriert werden, nicht auf spezifische Topics. Die Events müssen also im Nachgang „sortiert“ bzw. pro Topic

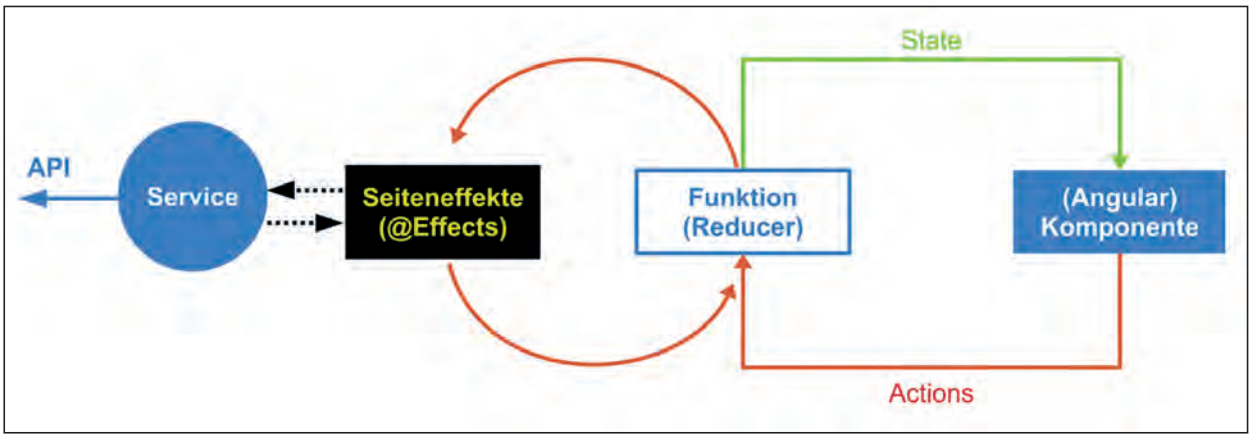

Abb. 1: Fluss von Actions und State in NgRx

gefiltert werden. Innerhalb des Listeners wird hier ein sogenanntes Subject genutzt, um MQTT-Nachrichten in RxJS-Events zu transformieren. Jedes Mal, wenn eine Message am Client ankommt, wird sie durch Aufruf von *this.messagesSub.next()* also in ein RxJS Event verwandelt.

Um nun an den Event-Strom der RxJS Events zu gelangen, kann einfach *this.messagesSub.asObservable()* aufgerufen werden. Da allerdings alle Events aller Topics über das *messagesSub* Subject gemulticastet sind, sollte zusätzlich eine Filterung pro Topic stattfinden. Listing 4 zeigt eine solche Filterung beispielhaft. Dabei wird in der Methode *getObservableForTopic()* zunächst am MQTT-Client auf ein spezielles Topic subscribed. Der Aufruf ist nötig, damit der MQTT-Broker die zu diesem Topic gehörenden Nachrichten überhaupt an unseren Client verschickt. Danach werden die Events aus dem *messagesSub* per Observable Pipeline (*.pipe()*) nach dem jeweiligen Topic gefiltert und auf die jeweilige Event Payload gemappt. Die Event Payload wird hier mit dem generischen Typparameter *T* typisiert. Die Methode *mqttService.getObservableForTopic()* kann also genutzt werden, um ein Topic des MQTT-Brokers zu subscriben und dessen Events als Observable Stream weiterzuverarbeiten.

Exemplarisch wird die Methode in Listing 5 genutzt, um das Topic für die Profil-Updates eines Nutzers zu subscriben. Da die Payload des Events bisher nur als String vorliegt, wird der String hier noch als JSON geparst, da das Profil natürlich ein Objekt ist.

Anwendungszustand mit MQTT und NgRx

Per MQTT werden Nachrichten zwischen Systemen verschickt. Im Frontend können diese Nachrichten entgegengenommen werden, indem sie mit Hilfe von RxJS in Observable Streams überführt werden. Diese Daten aus den Streams sollen nun im App-Zustand angebunden werden. Dazu gibt es verschiedene Möglichkeiten; Komponenten können beispielsweise den Zustand direkt halten, was kurzfristig praktisch sein kann. Für langfristige Projekte sollte man jedoch eine von der Architektur her robustere Variante nutzen, die auch in Bezug auf automatisierte Testbarkeit besser aufgestellt ist. Ein erster Schritt könnte hier sein, einen zustandsbehafteten Service zu wählen. Noch besser ist es, speziell auf komplexe Zustandsübergänge spezialisierte Pakete zu benutzen.

Hier bietet sich die auch sonst weit verbreitete Library NgRx an. Die Idee von NgRx ist sehr eng mit dem

Listing 4: Filterung des MQTT-Event-Observable im MqttService

```
@Injectable({providedIn: 'root'})
export class MqttService {
  private messagesSub = new Subject<MqttEvent>();

  getObservableForTopic(topic: string): Observable<string> {
    this.client?.subscribe(topic);
    return this.messagesSub.asObservable()
      .pipe(
        filter(ev => ev.topic === topic),
        map(ev => ev.payload)
      );
  }
}
```

Listing 5: Subscription auf Profile-Change-Events

```
@Injectable({providedIn: 'root'})
export class MqttService {

  getProfileChanges(userId: string): Observable<Profile> {
    const topic = 'user/${userId}/profile';
    return this.getObservableForTopic(topic)
      .pipe(
        map(payloadString => JSON.parse(payloadString))
      );
  }
}
```

Event-driven-Ansatz und teilweise mit der Idee von Event Sourcing verwandt. So werden Zustandsübergänge in NgRx durch bestimmte Events, die sogenannten Actions, ausgelöst. Diese Actions definieren die Absicht einer Zustandsänderung. In sogenannten Reducern werden die Actions verarbeitet. Basierend auf dem bisherigen Altzustand und der Action wird dann der neue Zustand ermittelt. Der neue Zustand wird dann von NgRx in einem lokalen Store gespeichert. Über sogenannte Selektoren wird der Zustand aus dem Store wiederum als Observable für die Komponenten zur Verfügung gestellt.

In Listing 6 ist exemplarisch ein Ausschnitt des *userReducer* gezeigt, der in diesem Beispiel die Informationen zum eingeloggten Nutzer verwaltet. Jeder Reducer benötigt einen Initialzustand, von dem alle Folgezustände abgeleitet werden. Der Initialzustand ist in Listing 5 in der Konstanten *initialState* festgehalten. Der Initialzustand wird dann der Reducer Factory (*createReducer()*) zusammen mit der eigentlichen Reducer-Logik übergeben. Letztere wird durch eine oder mehrere *on(action)*-Funktionen definiert. Der erste Parameter der *on()*-Funktion ist dabei der Action-Typ, der in diesem Teil des Reducers verarbeitet werden soll. In Listing 6 ist das die *setProfile*-Action.

Sobald eine solche Aktion ankommt, wird das Profil aus der Action vom Reducer in den Store-Zustand geschrieben. Der Reducer wird in diesem Beispiel auch genutzt, um das *userFeature* zu erzeugen. Das Feature ermöglicht eine vereinfachte Registrierung des Reducers in der Anwendung. Außerdem stellt es Selektoren für das Feature zur Verfügung, die es vereinfachen, den Zustand in den Komponenten aus dem Store auszulesen.

Im folgenden Code ist die Verwendung des Features zum Aufruf des *selectProfile*-Selektors dargestellt. Mit Hilfe des Selektors wird der Profilzustand aus dem Store selektiert und als *Observable<Profile>* an die Variable *this.userProfile* gebunden.

```
constructor(private store: Store) {
  this.userProfile = store.select(userFeature.selectProfile);
}
```

Was nun noch fehlt: Die Daten aus dem MQTT Observable müssen in eine Action verpackt und zur Verarbeitung an den Store weitergeleitet werden. Im Prinzip könnte dafür im *MqttService* der Store injected und dann *store.dispatch(action)* aufgerufen werden. Das ist jedoch eher unsauber und führt potenziell zu schlecht verständlichem, schlecht wartbarem Code. Best Practice ist es an dieser Stelle, einen sogenannten NgRx Effect zu verwenden.

Effects stellen in NgRx zentrale Bausteine dar, die zur Anbindung asynchroner Prozesse dienen. Im Gegensatz zu den Reducers, die Actions verarbeiten, sind Effects eine Quelle von Actions. **Abbildung 1** zeigt den grundlegenden Fluss von Actions und Zustand durch die NgRx-Bestandteile.

Effects können dabei sowohl Actions als Arbeitsanstoß haben als auch jede andere Form von Events. In diesem Fall sollten Änderungen am Nutzerprofil verarbeitet werden, die per MQTT am Frontend ankommen. Dafür wird die Nutzer-ID benötigt. Diese kann einfach mit Hilfe des Selectors *userFeature.selectUserId* aus dem Store besorgt werden. Solange der Nutzer nicht angemeldet ist, also keine *UserId* besitzt, gibt der Selektor *undefined* oder einen leeren String zurück. Um diesen Rückgabewert zu ignorieren, kann der *filter()*-Operator genutzt werden, der zusammen mit der *Boolean*-Funktion genau solche Werte aus dem Observable Stream herausfiltert. Wenn eine *UserId* existiert, wird damit die Service-Methode aus Listing 5 aufgerufen. Das führt innerhalb des Effect nun dazu, dass jedes Mal, wenn ein MQTT-Event ankommt, eine *setProfile*-Action ausgelöst wird. Das jeweilige *Profile*-Objekt wird dann vom *userReducer* verarbeitet und in den Store-Zustand geschrieben. Von da aus wird es an die Komponenten wei-

terverteilt, die sich auf dieses spezielle Zustands-Snippet registriert haben (Listing 7).

Offlinefähigkeit (fast) geschenkt

Durch die Wahl einer Infrastruktur, die in der Lage ist, eigenständig Nachrichten zu puffern, ist es deutlich einfacher, eine offlinefähige Anwendung zu erstellen. Nachdem das Netzwerk wieder verfügbar ist, liefert MQTT einfach die ausstehenden Nachrichten nach. Im Kontext von Echtzeitanforderungen kann dieses Verhalten, wenn es nicht transparent gemacht wird, dazu führen, dass der Nutzer im Offlinezustand einen alten Stand wahrnimmt und davon ausgeht, dass es keine Änderungen gibt. Das kann zu falschen Schlüssen und darauf basierenden Aktionen führen. Daher bieten Webbrowser ein passendes API, um den Zustand der Verbindung zu kommunizieren [1]. Zum einen kann der Status direkt abgefragt werden, zum anderen werden Events bereitgestellt, wenn sich am Verbindungsstatus etwas ändert. Damit ist es nun möglich, dem Nutzer einen Hinweis anzuzeigen, dass die Anwendung sich im Offlinemodus befindet. So weiß der Nutzer Bescheid und kann sich darauf einstellen.

Damit das nicht nur technisch funktioniert, sondern auch fachlich sinnvoll ist, müssen die auszutauschenden Nachrichten entsprechend konzipiert werden. Auch der Rest der Anwendung sollte so angelegt sein, dass die Offlinefähigkeit unterstützt wird. Beispielsweise müssen auch ausgehende Nachrichten auf eine Weise kommuniziert werden, dass der Offlinefall nicht zum Problem wird, sei es durch verlorene Daten oder eine mit einem Spinner blockierte Oberfläche.

Kommt es zu einer Aktivität im Frontend, die zu einem Backend-Aufruf führt, so sollte von der Visualisierung und Nutzerführung her klar sein, dass kein unmittelbares Ergebnis zu erwarten ist. Das kennt jeder von seinem Bankkonto: Eine eingestellte Überweisung führt typischerweise nicht dazu, dass eine entsprechende Buchung sichtbar wird, sondern erst nachdem die Bank den nächsten Buchungslauf durchgeführt hat. Der Nutzer kann jedoch jederzeit bei den Überweisungen sehen, dass diese vorgemerkt ist.

Nicht immer ist dieses auch als Eventual Consistency bekannte Verhalten akzeptabel. Würde bei einer Websuche nicht nach kürzester Zeit eine Liste mit passenden Ergebnissen auftauchen, sondern diese erst am nächsten Tag per E-Mail eintrudeln, würde wohl kaum jemand das System nutzen. Dieser Fall ist typisch für Abfragen, gerade dabei wird auf eine Antwort gewartet, da der Nutzer damit typischerweise weitere Aktionen durchführen möchte. Bei erfassten Daten ist das insbesondere dann kein Problem, wenn auf die Erfassung keine Antwort erfolgt, die für das weitere Arbeiten benötigt wird. Ein Mittelweg kann daher sein, für kurze Zeit auf eine passend korrelierte Antwort zu warten und den Nutzer aktiv zu informieren, dass eine Echtzeitantwort aktuell nicht möglich ist, seine Daten aber verarbeitet werden bzw. er die Antwort erhält, wenn er nächstes Mal online ist.

Damit stellt sich die Frage, wie erfasste Daten in der Webanwendung gepuffert werden können, um sie für den späteren Versand aufzubewahren. Es spielt dabei zunächst keine Rolle, über welchen Kanal die Daten versendet werden. Der ausgehende Kommunikationsweg kann bei einem geschickt gewählten Design vom Puffer entkoppelt werden. Dazu bietet sich das Outbox-Pattern an: Die Idee dabei ist, dass sämtliche zu versendenden Daten zunächst lokal persistiert werden. Erst danach wird der Versand asynchron durchgeführt. Ein ähnliches Vorgehen wird bei Microservices typischerweise gewählt, wenn ein transaktionaler Versand von Messages zusammen mit Datenbankoperationen erfolgen soll. Eine Darstellung des Konzepts findet sich in **Abbildung 2**.

Durch die Offlinefähigkeit entstehen neue Situationen, auf die das Gesamtsystem korrekt reagieren muss. Hier kann es hilfreich sein, bei Events explizit zu modellieren, dass es einen Ereignis- und einen Verarbeitungszeitpunkt gibt. Dabei kann es zu einem mehrfachen

Listing 6: Ausschnitt des userReducer, kann setProfile-Actions verarbeiten

```
export const initialState: UserState = {
  userId: ''
  profile: null,
};

export const userReducer = createReducer(
  initialState,
  on(setProfile, (state, action): AuthState => ({
    ...state,
    profile: action.profile
  }))
);

export const userFeature = createFeature({name: 'user', reducer:
                                                    userReducer});
```

Listing 7: Subscription auf Profile-Change-Events 2

```
@Injectable()
export class MqttEffects {

  profileChangeEvents = createEffect(() => {
    return this.store.select(userFeature.selectUserId)
      .pipe(
        first(Boolean),
        concatMap((userId) => {
          return this.mqttService.getProfileChanges(userId);
        }),
        map(payload => setProfile(payload))
      );
  });

  constructor(private store: Store, private mqttService: MqttService) {}
}
```

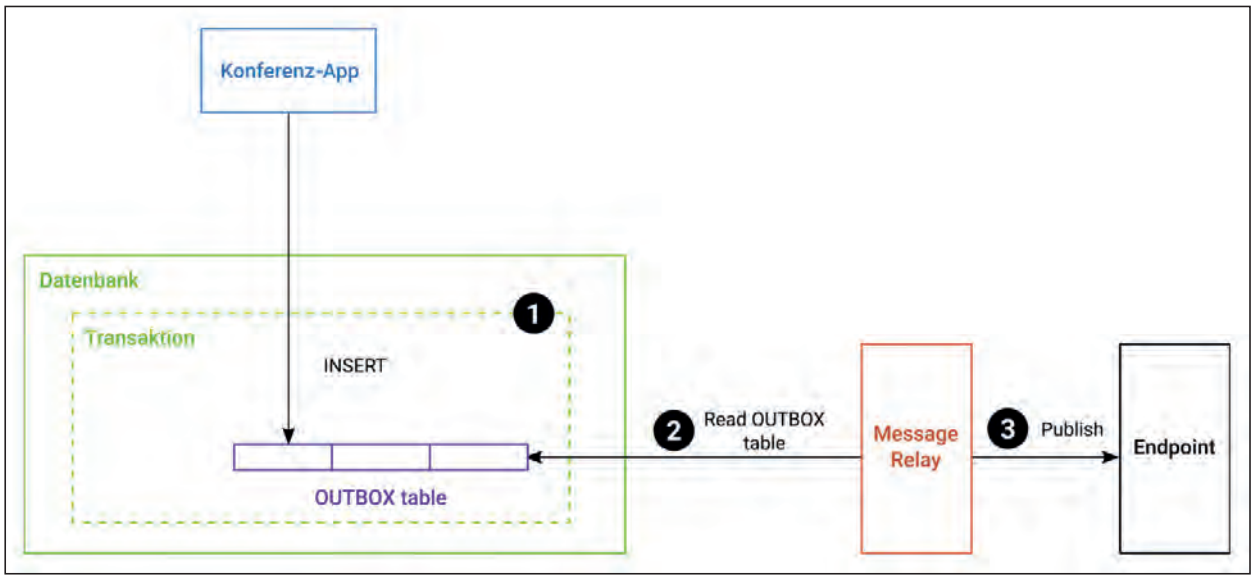

Abb. 2: Aufbau des Outbox-Patterns

Versand kommen, entsprechend sollte ein eindeutiges Merkmal zur Deduplizierung an Kommandos und ggf. Events vorgesehen werden.

Im Browser bieten sich für die lokale Persistenz im Wesentlichen zwei APIs an: *LocalStorage*, die sitzungsübergreifende Variante des *SessionStorage* und *IndexedDB*. Der *LocalStorage* bietet ein einfaches API mit den beiden Methoden *getItem(key)* und *setItem(key, value)*. Das API ist synchron (blockierend), bietet keine Transaktionen und ist nicht in Web-Workern oder Service-Workern verfügbar. Zudem ist der Speicherplatz typischerweise auf niedrige zweistellige MB-Bereiche begrenzt und es können lediglich Zeichenketten als Schlüssel und Werte genutzt werden.

Die *IndexedDB* bietet ein reichhaltigeres, aber auch komplexeres API. Abfragen sind asynchron, das API steht auch in Service-Workern und Web-Workern zur Verfügung. Schlüssel können Zahlen, Zeichenketten, Daten, Arrays und *ArrayBuffer*-Objekte sein. Werte können aus allem, was serialisierbar ist, bestehen, aber auch aus *Blob*-, *File*- und *ImageData*-Objekten. Durch die Möglichkeit, Indizes abzulegen, Cursors und die Möglichkeit, Abfragen zu nutzen, können Daten entsprechend einer gewünschten Ordnung verarbeitet werden. Zudem ist *IndexedDB* typischerweise für Datenmengen von mehreren Hundert Megabyte nutzbar. Damit eignet es sich deutlich besser für Anwendungsfälle, bei denen ein Cache oder Puffer, z. B. für Offlinefähigkeit, aufgebaut werden soll, als der *LocalStorage*. Allerdings ist das API nicht so intuitiv einsetzbar. Ein rudimentäres Beispiel zur Verwendung des API zur Implementierung des Outbox-Patterns ist in Listing 8 zu sehen.

Fazit und Ausblick

In diesem Beitrag haben wir fundamentale Bausteine zur Erstellung von echtzeitfähigen und offlinefähigen

Anwendungen kennengelernt. Durch die zunehmende Verbreitung von Webanwendungen steigen die Anforderungen an Robustheit und Benutzerfreundlichkeit. Sowohl der Umgang mit Offlinesituationen als auch der Wunsch nach aktuellen bzw. Echtzeitinformationen werden sicherlich zunehmend Anforderungen an typische Anwendungen werden.

Als Technologiestack für das Echtzeit-Frontend wurde hier das Angular-Framework mit NgRx State Management ausgewählt. Zur Anbindung an MQTT wurde MQTT.js genutzt. Der aktuelle Angular-Stack stellt neuerdings zwei Varianten bereit, um mit asynchronen Events umzugehen. Zum einen die hier gezeigten Observables, zum anderen die mit Angular 16 neu eingeführten Signals. Auf den Signals aufbauend wurde auch bei NgRx eine weitere Variante des Stores entwickelt, der Signal Store. Die Anbindung von MQTT könnte im Prinzip auch darüber geschehen. Dabei würde sich natürlich die Syntax im Vergleich zum Observable-Ansatz unterscheiden, der Aufbau wäre von den Konzepten her dem hier gezeigten Set-up aber sehr ähnlich.

Karsten Sitterberg ist als freiberuflicher Entwickler, Trainer und Berater für Webtechnologien tätig. Seine Schwerpunkte liegen im Bereich HTTP APIs, TypeScript und Angular. Karsten unterstützt seine Kunden sowohl bei der Einführung neuer Technologien als auch durch tatkräftige Mitarbeit in Projekten in Form von Entwicklung, Review und Coaching. Regelmäßig berichtet er in Vorträgen und Artikeln über aktuelle Trends und Hintergründe zu Themen, die für Entwickler und Architekten gleichermaßen relevant sind. Kontaktieren Sie ihn gerne für weitergehende Unterstützung zu den behandelten Themen.

karsten@sitterberg.com **sitterberg.com**

Links & Literatur

[1] https://caniuse.com/online-status

Listing 8: Beispiel einer simplen Outbox mit der IndexedDB

```
interface OutboxEntry {
  createdOn: Date;
  sentOn: Date | null;
  sent: boolean;
  endpoint: string;
  payload: object;
}

export class OutboxManager {
  private db: Promise<IDBDatabase>;
  constructor() {
    const request = indexedDB.open('OutboxDB', 3);
    this.db = new Promise((resolve, reject) => {
      request.onerror = ev => console.error('Fehler beim Öffnen der Datenbank');
      request.onupgradeneeded = (event) => {
        const db = (event.target as IDBRequest).result;
        db.createObjectStore('outbox', {autoIncrement: true});
      };
      request.onsuccess = (event) => {
        const db = (event.target as IDBRequest).result;
        resolve(db);
        this.processOutbox();
      };
    })
  }

  async addToOutbox(message: Object) {
    const transaction = (await this.db).transaction(['outbox'], 'readwrite');
    const objectStore = transaction.objectStore('outbox');
    const request = objectStore.add(message);

    request.onsuccess = (event) => {
      this.processOutbox();
    };
    request.onerror = ev => console.error('Fehler beim Hinzufügen zur Outbox');
  }

  async processOutbox() {
    if(!navigator.onLine) {
      return;
    }
    const transaction = (await this.db).transaction(['outbox'], 'readwrite');
    const objectStore = transaction.objectStore('outbox');

    objectStore.openCursor().onsuccess = (event) => {
      let cursor: IDBCursorWithValue = (event.target as IDBRequest).result;
      if (cursor) {
        // Code zum Abschicken des (MQTT/HTTP-)Request hier
        // Bei Erfolg: Nachricht aus Outbox löschen oder als versendet markieren
        let deleteRequest = cursor.delete();
        deleteRequest.onsuccess = () => console.log('Nachricht aus der Outbox
                                                                    gelöscht');
        cursor.continue();
      }
    };
  }
}

const outboxManager = new OutboxManager();
const message: OutboxEntry = {
  createdOn: new Date(),
  sentOn: null,
  sent: false,
  endpoint: '/api/demo/example',
  payload: {content: 'Hallo Welt'}
};
outboxManager.addToOutbox(message);

setInterval(() => outboxManager.processOutbox(), 5_000);
```

OAuth2 und der Spring Authorization Server

Leichtgewichtig und standardkonform

Der Spring Authorization Server ist ein mächtiges und nützliches Werkzeug, wenn es darum geht, spezielle Anforderungen umzusetzen oder standardkonform OIDC bzw. OAuth2 zu nutzen, ohne ein zusätzliches Umsystem als Implementierung zu erfordern. Wir stellen Grundlagen und Umsetzung vor.

von Thomas Kruse

Waren Anwendungen in der Vergangenheit oft monolithisch geprägt, so geht der Trend seit einigen Jahren hin zu verteilten Systemen. Damit gehen einige Herausforderungen einher, eine davon ist das Thema Authentifizierung. Bei monolithischen Anwendungen kann eine Anmeldung durch Username/Passwort lokal erfolgen. Gibt es jedoch mehrere per API miteinander interagierende Microservices oder Systeme, ist das nicht mehr so einfach möglich. Um nur zwei Gründe zu nennen, die auf der Hand liegen: Zur Sicherheit sollen natürlich weder die Credentials von Nutzern über Systemgrenzen hinweg verwendet werden, noch kann es eine Vertrauensbeziehung zwischen den Systemen geben, bei der sich eine erneute Authentifizierung des Nutzers erübrigt. Davon abgesehen gilt es aber auch, den Komfort für den Nutzer im Auge zu behalten: Sich ständig gegenüber anderen Systemen erneut authentifizieren zu müssen, ist bei der Vielzahl von Systemen indiskutabel.

Zu dieser Frage wurden im Lauf der Zeit bei Internetanwendungen eine Reihe von Standards herausgearbeitet, die vor allem durch Social-Media-Systeme wie Facebook, Xing, Twitter (neuerdings „X“), aber auch andere Integrationen wie zum Beispiel bei GitHub getrieben wurden. Dabei geht es im Prinzip um zwei unterschiedliche Dinge: einmal um die Delegation von Berechtigungen (Authorization), also die Frage, ob ein Social-Media-Managementsystem mit meinem Account eine Veröffentlichung in meinem Namen tätigen darf. Es handelt sich dabei um eine Delegation der Berechtigungen eines Nutzers an ein anderes System, das dann Aktivitäten über ein bereitgestelltes API durchführt. Der dazu etablierte Standard OAuth2 liefert ein Protokoll, mit dem das abgebildet werden kann.

Ergänzend, aber im Prinzip unabhängig davon, kommt der Aspekt der Authentifizierung ins Spiel: Zum einen soll die Komplexität aller mit einer Nutzerverwaltung einhergehenden Prozesse nicht an jedem System lokal und redundant implementiert werden, zum anderen geht es hier um ein gutes Nutzererlebnis und Vertrauen des Nutzers in den Anbieter. Wenn die Authentifizierung des Nutzers delegiert wird, gehen damit verschiedene Vorteile einher: Der Nutzer muss sich lediglich die Zugangsdaten zu wenigen Anbietern merken und erspart sich dabei auch die Anmeldung an vielen unterschiedlichen Systemen. Dem Nutzer ist dabei auch klar, dass ein großer Anbieter, wie zum Beispiel Google, sehr viel mehr Ressourcen in die Absicherung der eigenen Systeme investieren kann und wird, als das bei der WordPress-Installation des lokalen Kleingewerbes der Fall ist.

Das Prinzip ist aus dem Alltag vertraut: Der Staat stellt seinen Bürgern einen Ausweis und einen Reisepass zur Verfügung, mit dem sie sich identifizieren können. Das funktioniert sogar über Organisationsgrenzen (Landesgrenzen) hinweg und sorgt für reibungsarme und sichere Abläufe. Nachdem OAuth2 zunehmend nicht nur für die Autorisierung, sondern auch für die Authentifizierung genutzt wurde, entstand der Wunsch nach einer Normierung des Ablaufes, da Prozesse und APIs von jedem Anbieter individuell umgesetzt wurden. Das Ergebnis dieser Bestrebungen ist das OpenID-Connect-Protokoll: Es definiert bestimmte URL-Endpunkte und API-Formate, um Interoperabilität und Einheitlichkeit sicherzustellen. Auch wenn OAuth2 und OpenID Connect (kurz: OIDC) nahe verwandt sind, behandeln sie jeweils unterschiedliche Aspekte. Leider werden diese beiden Protokolle oft in einer Weise vermischt beschrieben und verwendet, dass es zu Verwechslungen und

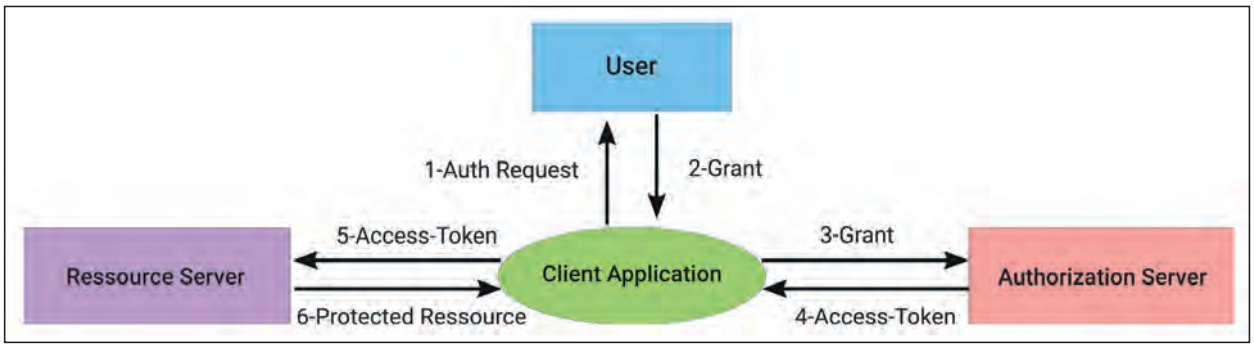

Abb. 1: Beteiligte Akteure und Systeme

Missverständnissen kommt. Aus diesem Grund werden wir sie hier strikt trennen und zunächst OAuth2, anschließend den Spring Authorization Server mit OIDC betrachtet.

OAuth2 mit Spring Boot

Für den optimalen Einstieg in OAuth2 empfiehlt es sich, einige Begriffe zu klären. Beginnen wir mit dem Begriff „Ressource". Dabei handelt es sich um etwas, das durch ein System verwaltet und geschützt wird. Wenn wir GitHub als Beispiel wählen, so wäre eine Ressource ein Git-Repository auf GitHub. Der Eigentümer dieser Ressource wird entsprechend „Resource Owner" genannt. Das ist typischerweise eine Person. Der Resource Owner zeichnet sich dadurch aus, dass er Zugriff auf die geschützten Ressourcen gewähren kann. Das System, das die Ressourcen bereitstellt, wird „Resource Server" genannt. In unserem Beispiel wäre das GitHub. Ein OAuth2-konformer Resource Server ist in der Lage, Anfragen via API entgegenzunehmen und auf Basis eines Access-Tokens den Zugriff zu gewähren. Bei HTTP-Anfragen wird das Token als HTTP-Header im Feld *Authorization* mit dem Präfix *Bearer* übertragen. Der Aufbau des Tokens wird durch OAuth2 nicht definiert, und man sollte sich vorstellen, dass es sich um eine zufällige, lange Zeichenkette handelt. Access-Tokens haben typischerweise einen kurzen Gültigkeitszeitraum, zum Beispiel fünf Minuten. Danach muss ein neues Token ermittelt werden. Eine HTTP-*GET*-Anfrage mit Access-Token im Authorization-Bearer-Header sieht beispielsweise so aus:

```
GET / HTTP/1.1
Authorization: Bearer secret-or-random-token-value
```

Eine weitere OAuth2-Rolle ist die des Clients. Beim Client handelt es sich um eine Anwendung, die anstelle des Resource Owners selbst Aufrufe gegen das API des Resource Servers durchführt. Dabei kann es sich durchaus um eine serverseitige Anwendung handeln, der Begriff Client soll hier keine spezielle Implementierung suggerieren. Die Aufrufe werden entsprechend mit einem Access-Token autorisiert. Diese Access-Tokens werden durch den Authorization Server für einen jeweiligen Client ausgestellt. Das setzt natürlich voraus, dass der Resource Owner sich gegenüber dem Authorization Server authentifiziert hat – z. B. durch ein Log-in mit Username und Passwort – und eine explizite oder implizite Einwilligung durch den Resource Owner für einen spezifischen Client gegeben ist.

Auf den ersten Blick können diese vielen Begriffe verwirren und der Ansatz mag sogar akademisch wirken, jedoch ist diese saubere Definition mit einer Trennung der Verantwortlichkeiten sehr nützlich, um angemessen über die damit einhergehenden Abläufe und Zuständigkeiten zu diskutieren. Zudem handelt es sich hier um sicherheitskritische Aspekte, sodass für Sorgfalt und Genauigkeit durchaus ein entsprechender Stellenwert gerechtfertigt ist. **Abbildung 1** zeigt den prinzipiellen Ablauf und die beteiligten Rollen in der Übersicht.

Eine Spring-Boot-Anwendung kann somit im Kontext von OAuth2 ebenfalls in verschiedenen Rollen fungieren:

- als OAuth2 Resource Server, der Requests empfängt und dabei verwendete Access-Tokens validiert, um Requests zu erlauben oder abzulehnen
- als OAuth2-Client, der Requests gegen das API eines Resource Servers im Namen eines Users (Resource Owner) durchführt und dazu Access-Tokens zur Autorisierung verwendet
- als eine OAuth2-Client-Sonderrolle, bei der lediglich eine Ressource des Resource Servers abgerufen wird, um Nutzerinformationen zu erhalten, darüber ein Log-in zu durchzuführen und die Authentifizierung zu implementieren

Hier ist es also sehr wichtig, sich stets die Rolle bzw. Perspektive klarzumachen, aus der die Anwendung gerade betrachtet wird. Es lohnt sich, auch über den Umgang mit Tokens nachzudenken: Wenn ein Access-Token zur Autorisierung dient, woher weiß der Resource Server, welche Operationen auf welchen Ressourcen erlaubt sind?

Tokens im Kontext von OAuth2 sind dazu da, Autorisierung stellvertretend für einen User zu gewähren. Nun sollen aber nicht immer alle Berechtigungen eines Users vollständig an ein externes System weitergegeben werden. Daher lässt sich die Nutzung eines Tokens einschränken. Dazu dient der sogenannte Scope. Schon bei der Anfrage an den Authorization Server kann der Client

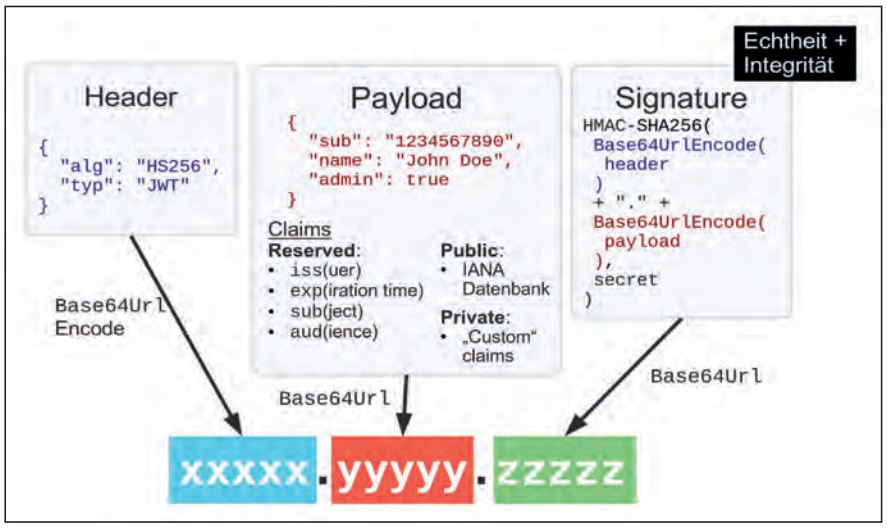

Abb. 2: Aufbau eines JWT

einen oder mehrere Scopes mitgeben und diese werden typischerweise dem Nutzer auch zur Einwilligung bzw. Bestätigung angezeigt, bevor dieser zustimmt. Bei Einwilligung erstellt der Authorization Server ein entsprechendes Access-Token, in dem die Scopes mit angegeben sind. Sie können durch den Resource Server bei der Nutzung validiert werden. Die Scopes sind spezifisch für ein jeweiliges System und müssen sowohl vom Resource Server als auch vom Authorization Server unterstützt werden. Dabei gilt es zu beachten, dass Scopes einen Typ von Ressource und eine Art von Operation beinhalten können, nicht jedoch die Instanz einer Ressource. Am Beispiel von GitHub wären Scopes: *read*, um alle Ressourcen lesend zu nutzen, auf die ein Nutzer Zugriff hat, oder *repo*, um alle Operationen, die ein Nutzer ausführen darf, auf allen Repositorys auszuführen, auf die der Nutzer Zugriff hat. Das kann auch kombiniert werden, z. B. *repo:read*, um lediglich lesend auf alle Repositories eines Nutzers zugreifen zu können.

Diese Informationen sind aus einem Token nicht direkt ersichtlich, ein Token ist schließlich zunächst in der Vorstellung eine Art temporäres Passwort oder eben Zufallsstring. Der Resource Server könnte entsprechende Ressourcen anbieten, die der Client – unter Verwendung des Tokens –

abfragt, um zu erfahren, auf welche Ressourcen es überhaupt Zugriff gibt und welche Operationen auf welchen Typen erlaubt sind. Es könnte aber auch sein, dass der Client weiß, welche Ressourcen anzusteuern sind, und das Token einfach dazu verwendet. Das ist mit der Situation in einem Hotel vergleichbar: Beim Check-in erhält man als Gast (Client) von der Rezeption (dem Authorization Server) eine Schlüsselkarte (Access-Token) und die Zimmernummer mitgeteilt. Nun kann man mit dieser Schlüsselkarte sein Zimmer (Ressource) nutzen, indem man sie am elektronischen Schloss (Ressource Server) vorlegt. Das Schloss weiß dabei nicht, welcher Gast gerade da ist, und diese Information ist auch nicht relevant. Genauso ist es für mich als Gast nicht wichtig zu wissen, welche Zimmer es alle im Hotel gibt.

Die nächste Frage ist jedoch: Woher weiß der Resource Server denn, ob ein Token gültig für eine bestimmte Operation ist? Dazu gibt es zwei Möglichkeiten. Bleiben wir zunächst dabei, dass ein Token lediglich einen Zufallswert beinhaltet. Mit diesem Wert kann der Resource Server den Authorization Server aufrufen und um Informationen zum Token bitten. Das wird „Token Introspection" genannt. Dieses von OAuth2 genormte API liefert verschiedene wichtige Informationen:

- Ist das Token (noch) gültig oder bereits abgelaufen (*active*)?
- Welche Scopes hat das Token (*scope*)?
- Für welchen Client ist das Token ausgestellt (*client_id*)?
- Wie ist der Nutzername des Users, für den das Token gilt (*username*)?

- Wie ist die technische Kennung (z. B. ID) des Users (auch *Subject* genannt), für den das Token gilt (*sub*)?
- Wann wird das Token ablaufen (*exp*)?

Abb. 3: Beispiel eines JWT

Listing 1: HTTP-Antwort des Token Introspection Endpoint

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{
  "active": true,
  "scope": "document:read folder:write email",
  "client_id": "4711",
  "username": "human3",
  "sub": "123-455-11"
  "exp": 1637355321
}
```

Listing 2: Spring-Boot-Konfiguration als OAuth2 Resource Server

```
spring:
  security:
    oauth2:
      resourceserver:
        opaque-token:
          introspection-uri: https://auth.example.com/introspect
          client-id: client
          client-secret: password
```

Listing 3: RestController mit Method Security

```
@PreAuthorize("hasAuthority('SCOPE_DEMO')")
@GetMapping("/user")
public String user(@AuthenticationPrincipal OAuth2AuthenticatedPrincipal principal)
{
  return principal.getName();
}
```

Diese Daten werden als JSON-Struktur vom Token Introspection Endpoint des Authorization Servers geliefert, wie Listing 1 zeigt. Um den Endpoint aufzurufen, muss der Resource Server sich selbst authentifizieren, er ist in diesem Moment quasi selbst ein Client des Authorization Servers. Das geschieht durch ein Client-Secret und eine Client-ID, die man sich als Username und Passwort eines technischen Users vorstellen kann.

Alternativ dazu besteht die Möglichkeit, dass das Token eben kein Zufallswert ohne eigene Aussagekraft (Opaque Token), sondern eine Struktur ist, die durch den Resource Server analysiert werden kann. Bei dieser Variante werden typischerweise JSON Web Tokens (JWT) eingesetzt. Dabei handelt es sich um eine dreiteilige Datenstruktur, die aus Metadaten, eigentlichen Inhaltsdaten und einer Signatur besteht. Damit kann das Token ohne zusätzliche Remoteaufrufe anhand der Signatur validiert (Echtheit, Integrität) und ausgewertet werden (**Abb. 2** und **3**).

Nach all den Grundlagen wollen wir uns nun die Umsetzung ansehen.

Im Kontext von Spring-Anwendungen muss man zwischen dem Core Spring Framework, Spring-Security- und Spring-Boot-Projekten unterscheiden. Spring Framework und Spring Security liefern Bausteine, um damit Anwendungen individuell zu entwickeln. Spring Boot liefert vordefinierte Konfigurationen und Vorgaben auf Basis dieser Bausteine. Für Spring Security gab es ein Erweiterungsprojekt, um OAuth2 umzusetzen: spring-security-oauth2. Die dadurch bereitgestellten Konfigurationen *@EnableAuthorizationServer* und *@EnableResourceServer* wurden zusammen mit dem Modul nach nunmehr über zehn Jahren abgekündigt. Das ist besonders wichtig, wenn weitergehende Dokumentationen, Tutorials oder Videos herangezogen werden. Wenn für diese Umstellung nicht das notwendige Bewusstsein vorhanden ist, kann schnell Verwirrung entstehen. Die neue, nun aktuelle Umsetzung ist in zwei separate Module, ebenfalls auf Spring Security aufsetzend, gesplittet: *spring-security-oauth2-resource-server* und *spring-security-oauth2-authorization-server*.

Um eine Spring-Boot-Anwendung als OAuth2 Resource Server zu konfigurieren (Listing 2), ist daher Folgendes nötig:

- Eine Spring-Boot-Anwendung, diese kann über *https://start.spring.io* erstellt werden
- Als Abhängigkeit ist *spring-security-oauth2-resource-server* erforderlich
- Die Konfiguration des zu verwendenden Authorization Servers bzw. Token Introspection Endpoint
- Credentials für den Resource Server, der als Client den Authorization Server auf dem Token Introspection Endpoint aufruft

Im Prinzip ist es das bereits. Angenommen, dass am Authorization Server der Resource Server als Client entsprechend registriert ist und ein anderes System in der

Rolle als Client des Resource Servers ein Access-Token – auf welchem Weg auch immer – erhalten hat, kann dieses Token nun durch den Resource Server entgegengenommen werden.

Im Java-Code stehen dann die üblichen Spring-Security-Mechanismen bereit. Hier gilt es wieder einmal, sich nicht verwirren zu lassen, wenn von Authentication statt Authorization die Rede ist: Für den Resource Server ist sehr wohl relevant, für welchen User eine Ressource aufgerufen wird. Daher stellt Spring Security bei Verwendung von OAuth2-Access-Tokens eine *BearerTokenAuthentication* zur Verfügung, in der ein *OAuth2AuthenticatedPrincipal*-Objekt liegt. Die erwähnten *scope*-Werte werden automatisch zu Authorities gemappt, wobei *SCOPE_* als Präfix verwendet wird. Das Verfahren ist somit analog zu den lange etablierten Rollen in Spring Security, die mit *ROLE_* als Präfix versehene Authorities sind. Analog können dann Berechtigungen durch SpEL oder in der HttpSecurity FilterChain abgefragt werden.

Listing 3 zeigt am Beispiel eines Spring *RestController*, wie zum einen die Controller-Methode durch Spring Security abgesichert und zum anderen auf die Authentifizierungsinformationen zugegriffen werden kann.

Es ist auch möglich, sich an einer Spring-Anwendung mittels OAuth2 und einer anbieterspezifischen Konfiguration oder OpenID Connect anzumelden. Damit kann ein externes System zur Nutzerverwaltung und zu allen damit verbundenen Aspekten genutzt werden. Der Log-in-Anwendungsfall ist auch eine geeignete Überleitung zum OAuth2-Client, denn für die Anwendung fungiert die Spring-Anwendung bereits als Client im Sinne von OAuth2. Entsprechend wird dazu die Abhängigkeit *spring-security-oauth2-client* in der Anwendung benötigt.

Damit der Client mit dem Authorization Server interagieren darf – er selbst muss sich mindestens identifizieren –, wird er bei diesem registriert. Handelt es sich dabei um einen „Confidential Client", also keine Anwendung, bei der die zugehörigen Credentials des Clients auf einfache Weise ausgelesen werden können, wie das typischerweise bei Webanwendungen der Fall ist, wird in diesem Zuge neben einer Client-ID auch ein Client-Secret zur Authentifizierung des Clients festgelegt.

Die folgenden Schritte verlaufen – leicht vereinfacht dargestellt – folgendermaßen: Der Client leitet einen nicht angemeldeten Nutzer an den Authorization Server weiter und gibt dabei zusätzlich seine Client-ID und einen URL zur Weiterleitung nach erfolgreicher Authentifizierung mit. Der Authorization Server führt die Authentifizierung durch. Er kann dazu ein beliebiges Verfahren nutzen, im einfachsten Fall ist das Username/Userpasswort. Anschließend wird der Nutzer an den mitgelieferten Redirect-URL weitergeleitet, zusätzlich wird ein Parameter-„Code" an den Client mitgegeben. Der Client kann diesen Code nun in einem neuen Aufruf an den Authorization Server nutzen, um darüber im Tausch ein Access Token zu erhalten. Mit dem Access Token können nun die Nutzerinformationen vom Authorization Server durch den Client abgerufen werden – der Nutzer ist damit angemeldet.

Dieser vereinfacht beschriebene Ablauf ist der sogenannte Authorization Code Grant. Es gibt verschiedene Grants, mit denen ein Client eine Autorisierung vom Authorization Server anfragen kann, für Webanwendungen wird in der Regel der Authorization Code Grant verwendet.

Für verbreitete Social-Log-in-Anbieter bringt Spring Security OAuth2 bereits vorbereitete Konfigurationen mit, sodass lediglich ein neuer Client bei dem jeweiligen Anbieter registriert werden muss und die Redirect-URLs bei dem Anbieter in die Liste erlaubter Ziele eingetragen werden müssen. Anschließend werden lediglich die Properties *spring.security.oauth2.client.registration.<provider>.client-id=<client-id>* und *spring.security.oauth2.client.registration.<provider>.client-secret=<client-secret>* benötigt. Provider können dabei zum Beispiel *google* oder *facebook* sein. Falls kein direkter Support für den zu verwendenden Provider existiert, müssen einige zusätzliche Konfigurationen vorgenommen werden. Eine beispielhafte Konfiguration für einen OAuth2 Authorization Server *auth.example.com* wird in Listing 4 gezeigt.

Soll die Spring-Anwendung gegenüber einem anderen System als Client fungieren, so ist das sowohl mit dem *RestTemplate* als auch mit dem reaktiven *WebClient* möglich. Die Konfiguration ähnelt dabei sehr der des Log-ins. Allerdings gibt es einen wesentlichen Unterschied: Wird für die Anmeldung nur einmalig ein Zugriff des Clients auf den Authorization Server benötigt, soll ein typischer Client in der Regel über einen längeren Zeitraum Requests durchführen – nicht nur an den Authorization Server, sondern auch an andere Resource Server. Dabei kann es durchaus sein, dass das im Hintergrund geschieht, ohne dass ein Nutzer gerade aktiv mit dem System arbeitet.

Listing 4: Konfiguration eines OAuth2-Providers für Log-in

```
spring:
  security:
    oauth2:
      client:
        registration:
          sample-client:
            client-id: sampleclient_1
            client-secret: client1_secret
            scope: profile
            client-name: OAuth2 Login
            authorization-grant-type: authorization_code
            provider: sample-provider
            redirect-uri: http://localhost:8080/login/oauth2/code/sample-provider
        provider:
          sample-provider:
            token-uri: https://auth.example.com/auth/oauth/token
            authorization-uri: https://auth.example.com/auth/authorize
            user-info-uri: https://auth.example.com/user/me
            user-name-attribute: name
```

Da die Access-Tokens aus Sicherheitsgründen nur über einen kurzen Gültigkeitszeitraum im Bereich weniger Minuten verfügen, wird ein Mechanismus benötigt, um „frische“ Access-Tokens zu erhalten. Dazu dient ein langlebiges Refresh-Token. Dieses Token wird nicht benutzt, um damit Resource Server aufzurufen, sondern lediglich in der Kommunikation zwischen Client und Resource Server. Je nach Architektur kann es sogar sein, dass der Client gesonderte Access-Tokens anfragen muss, mit denen die jeweiligen Resource Server aufgerufen werden.

Spring Security OAuth2 kann selbstständig Tokens aktualisieren, dazu dient der *OAuth2AuthorizedClientManager*. Damit kann beispielsweise ein Interceptor für das klassische *RestTemplate* erstellt werden (Listing 5) oder innerhalb von (*Rest*)*Controller*-Komponenten auch durch *@RegisteredOAuth2AuthorizedClient* ein passendes Argument bereitgestellt werden (Listing 6).

Spring Authorization Server und OIDC

OpenID Connect baut auf OAuth2 auf und erweitert diesen Standard zu einem Protokoll, das es dem Client erlaubt, die Identität eines Nutzers zu ermitteln und nicht nur eine Autorisierung für einen Resource Server zu delegieren. Die wesentlichen Unterschiede stellen sich zunächst folgendermaßen dar:

- Einführung eines neuen Tokens, des ID-Tokens; es hat zwingend das JWT-Format und enthält Claims, die das Token selbst sowie den Nutzer und seine Eigenschaften beschreiben
- Bereitstellung eines *Userinfo* HTTP Endpoints, über den mit einem passenden Access-Token die User Claims abgefragt werden können
- Eine definierte Datenstruktur, die der Authorization Server bereitstellt und die die Adressen der Endpoints auflistet

Listing 5: Interceptor für RestTemplate, der das Access-Token bereitstellt

```
@Component
public class OAuth2AuthorizedClientInterceptor implements
                                        ClientHttpRequestInterceptor {
  OAuth2AuthorizedClientManager manager;

  public OAuth2AuthorizedClientInterceptor(OAuth2AuthorizedClientManager
                                                          manager) {
    this.manager = manager;
  }

  public ClientHttpResponse intercept(
    HttpRequest request, byte[] body, ClientHttpRequestExecution execution) throws
                                                          IOException {

    // Interaktion durch den User
    Authentication principal = SecurityContextHolder.getContext().
                                                  getAuthentication();
    // Alternativ bei nicht-interaktiver Nutzung programmatische Erzeugung der
    Authentication - hier ohne konkreten User
      Authentication principal = new AnonymousAuthenticationToken ("key",
            "anonymous", AuthorityUtils.createAuthorityList("ROLE_ANONYMOUS"));

      OAuth2AuthorizeRequest authorizeRequest = OAuth2AuthorizeRequest
        .withClientRegistrationId("google")
        .principal(principal)
        .build();
      OAuth2AuthorizedClient authorizedClient = this.manager.
                                                  authorize(authorizeRequest);

      HttpHeaders headers = httpRequest.getHeaders();
      headers.setBearerAuth(
        authorizedClient.getAccessToken().getValue());
      return execution.execute(request, body);
    }
  }
```

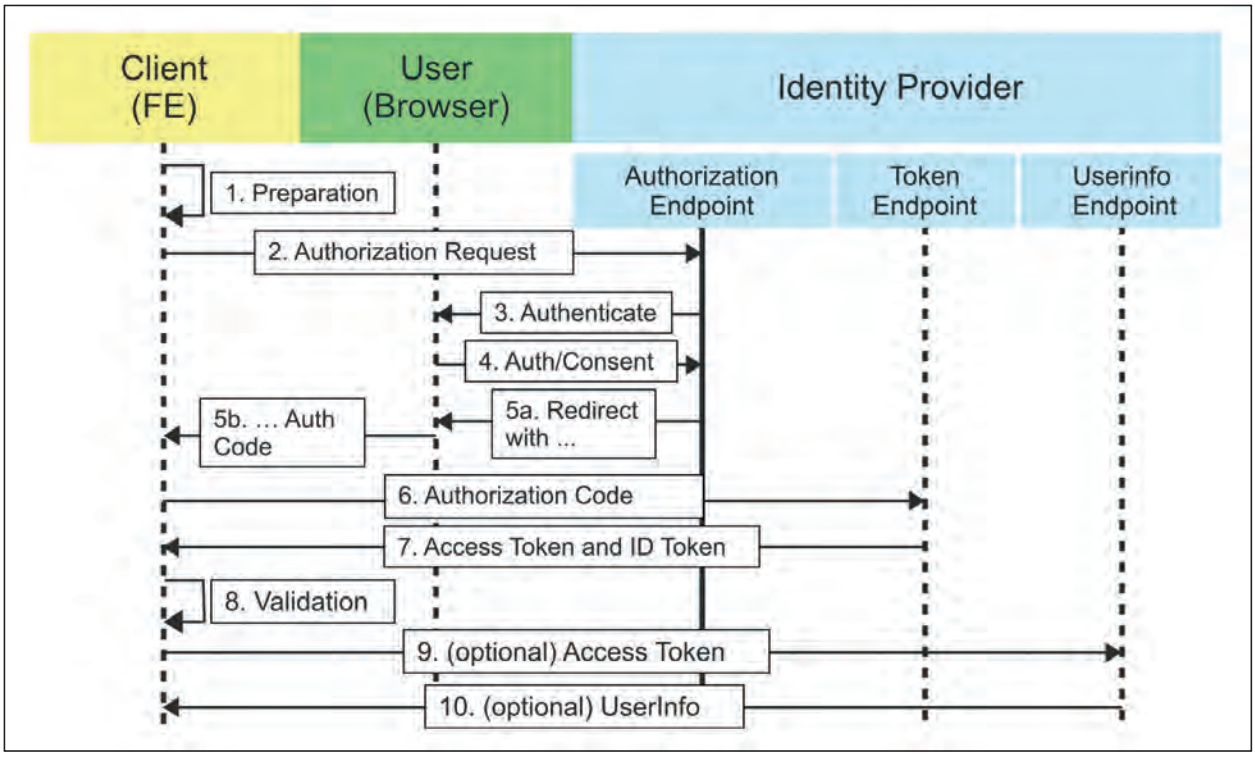

Abb. 4: Interaktion zwischen den beteiligten Komponenten bei einem Log-in mit OIDC am Spring Authorization Server

- Festlegung des URL, über den diese Datenstruktur abgerufen werden kann, der sogenannte Discovery URL: */.well-known/openid-configuration*

Dazu kommt nun ein Prozess, wie ein Client an das ID-Token und ggf. Access-Token gelangt. Diese Aspekte werden durch den Spring Authorization Server als OpenID-Connect-konforme Implementierung bereitgestellt. Damit ist der Schritt von einer OAuth2-Anwendung auch nicht mehr weit: Wir kennen die Rolle des Resource Servers und des Clients bereits. **Abbildung 4** zeigt den prinzipiellen Ablauf zwischen Client, Browser und Authorization Server bzw. OIDC Identity Provider.

Als Abhängigkeit wird *spring-security-oauth2-authorization-server* benötigt, mit Spring Boot kann *spring-boot-starter-oauth2-authorization-server* verwendet werden. Bei Spring Boot ist dann eine minimale Konfiguration bereits durch Propertys möglich. Die Konfiguration der Anwendung zeigt Listing 7.

Die Konfiguration durch die Propertys ist lediglich als Ausgangsbasis für einen Proof of Concept tauglich, für

Listing 6: Beispiel von RegisteredOAuth2-AuthorizedClient für Methodenparameter

```
@RestController
public class OAuth2ClientController {
  @GetMapping("/")
  public Object sample (@RegisteredOAuth2AuthorizedClient("google")
                        OAuth2AuthorizedClient authorizedClient) {
    var accessToken = authorizedClient.getAccessToken();
    ...
    return "index";
  }
}
```

Listing 7: Minimalkonfiguration des Spring Authorization Servers durch Properties

```
spring:
  security:
    oauth2:
      authorizationserver:
        client:
          oidc-client:
            registration:
              client-id: "oidc-client"
              client-secret: "{noop}secret"
              client-authentication-methods:
                - "client_secret_basic"
              authorization-grant-types:
                - "authorization_code"
                - "refresh_token"
              redirect-uris:
                - "http://example.com/login/oauth2"
              post-logout-redirect-uris:
                - "http://example.com/"
              scopes:
                - "openid"
                - "profile"
            require-authorization-consent: true
```

einen sinnvollen Einsatz ist eine weitergehende Konfiguration erforderlich.

Der Authorization Server benötigt jetzt nur eine Möglichkeit, um Clients zu verwalten, dazu dient das *RegisteredClientRepository*, und eine Möglichkeit, um Nutzer zu authentifizieren und zugehörige Daten wie z. B. Rollen zu erhalten. Dafür gibt es den *UserDetailsService*. Eine Beispielkonfiguration findet sich in Listing 8, dabei wird lediglich In-Memory-Persistenz verwendet. Beliebige andere Technologien lassen sich analog bzw. mit weiteren Optionen von Spring anbinden wie z. B. Data-JPA.

Da das ID-Token durch den Authorization Server signiert wird und eine JWT-Struktur aufweist, müssen entsprechende Konfigurationen auch dann bereitgestellt werden, wenn das Token lokal ausgewertet werden soll. Das sind im Wesentlichen *JwtSource* und *JwtDecoder*. Alternativ könnte das Token wie ein Opaque-Token behandelt und durch den Introspection Endpoint aufgelöst werden. Das ist mit zusätzlicher Konfiguration ebenfalls erreichbar.

Integration mit Webanwendungen am Beispiel Angular

Eine tokenbasierte Authentifizierung und Autorisierung eignet sich sowohl für serverseitig gerenderte als auch clientseitige Webanwendungen bzw. SPAs. Die Browseranwendung tritt dann in der Rolle eines OAuth2-Clients auf. Entsprechend kann die Anwendung dann das ID-Token nutzen, um Authentifizierungsinformationen des Nutzers zu erhalten. Eine Browseranwendung wird in Form der JavaScript-Dateien vollständig an den Nutzer übertragen und durch ihn ausgeführt. Damit ist es eine große Herausforderung, in der Anwendung geheim zu haltende Daten zu nutzen. Im OAuth2-Kontext scheidet damit ein Confidential-Client, also ein Client, der sich selbst mit Client-ID und Client-Secret authentifizieren kann, aus.

Der Spring Authorization Server bietet entsprechende Konfigurationen für Clients an. Aktuell ist es leider so, dass der Spring Authorization Server keine Refresh-Tokens für Public-Clients als Konfiguration anbietet. Der Hintergrund ist eine Interpretation der OAuth2-Spezifikation, die mit OAuth 2.1 präzisiert wurde und

Listing 8: Beispielkonfiguration für den Spring Authorization Server als Java Config Beans

```
@Bean
public RegisteredClientRepository registeredClientRepository()
{
 RegisteredClient frontend = RegisteredClient.withId(UUID.randomUUID().toString())
   .clientId("konferenz-frontend")
   //confidential client
   //.clientSecret("{noop}secret")
   //.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
   //public client
   .clientAuthenticationMethod(ClientAuthenticationMethod.NONE)
   //
   .clientSettings(ClientSettings.builder().requireAuthorizationConsent(false).build()
   )
   .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
   .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
   .redirectUri("http://127.0.0.1:4200")
   .redirectUri("http://127.0.0.1:4200/")
   .postLogoutRedirectUri("http://127.0.0.1:4200/")
   .scope(OidcScopes.OPENID)
   .scope(OidcScopes.PROFILE)
   .tokenSettings(TokenSettings.builder()
     .accessTokenTimeToLive(Duration.ofDays(2))
     .refreshTokenTimeToLive(Duration.ofDays(10))
     .build())
   .build();
 return new InMemoryRegisteredClientRepository(frontend);
}
@Bean
public JWKSource<SecurityContext> jwkSource(KeyPair keyPair)
{
  RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
  RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
  RSAKey rsaKey = new RSAKey.Builder(publicKey)
    .privateKey(privateKey)
    .keyID(UUID.randomUUID().toString())
    .build();
  JWKSet jwkSet = new JWKSet(rsaKey);
  return new ImmutableJWKSet<>(jwkSet);
}

@Bean
KeyPair loadOrGenerateRsaKey(Environment environment) throws JOSEException,
IOException
{
 var config = new String(getClass().getClassLoader().getResourceAsStream
                                                    ("dev-key.json").readAllBytes());
 final var key = convertConfigToRsaKey(config);
 return key.toKeyPair();
}
@Bean
public JwtDecoder jwtDecoder(JWKSource<SecurityContext> jwkSource)
{
  return OAuth2AuthorizationServerConfiguration.jwtDecoder(jwkSource);
}

@Bean
public JwtEncoder jwtEncoder(JWKSource<SecurityContext> jwkSource)
{
  return new NimbusJwtEncoder(jwkSource);
}

@Bean
public AuthorizationServerSettings authorizationServerSettings()
{
  return AuthorizationServerSettings.builder().build();
}
```

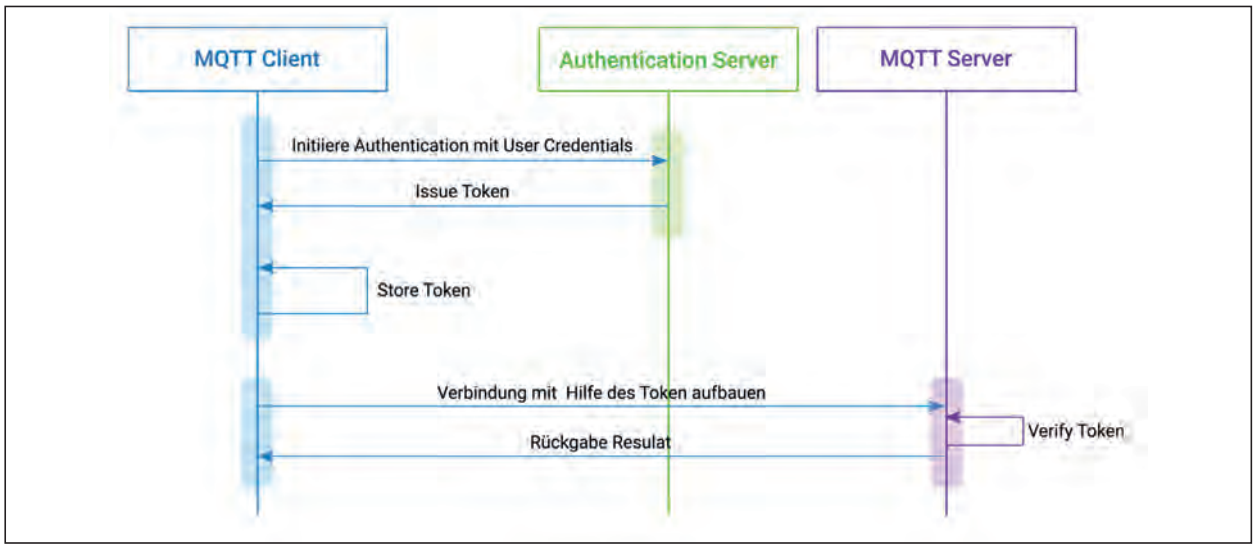

Abb. 5: Ablauf der Authentifizierung an Mosquitto mit mosquitto-go-auth

das Zusammenspiel von Refresh-Tokens bei Public-OAuth2-Clients definiert.

Das ist jedoch in der Praxis kein Problem, denn durch entsprechende Konfigurationspunkte kann das gewünschte Verhalten nachgerüstet werden, wie Listing 9 zeigt (mehr Informationen dazu finden sich in dem GitHub Issue unter [1]).

Auf der Seite einer Angular SPA ist die Integration recht einfach: Es gibt mindestens zwei populäre Bibliotheken zur Integration von OpenID-Connect. Verwendet man [2] sind nur wenige Schritte nötig:

- im Angular-Projekt ausführen: *$ ng add angular-auth-oidc-client*
- zu verwendenden Auth-Flow auswählen
- well-known URL des ID-Providers eintragen

Der Ablauf ist wie gewohnt: Zur Authentifizierung leitet die Angular-Anwendung auf den Spring Authorization Server weiter. Dort meldet sich der Nutzer an und wird dann mit dem Codeparameter zur Angular-Anwendung zurückgeleitet. Diese tauscht den Code nun gegen Access-Token, ID-Token und Refresh-Token aus. Analog ist natürlich die Integration in Vue, React und weitere Technologien möglich.

Anwendungsbeispiel: Anmeldung an den MQTT-Broker

Der OAuth2- bzw. OpenID-Connect-Standard ist mittlerweile extrem verbreitet und übertrifft häufig sogar die Integration des ehemaligen Platzhirsches SAML2. Selbst Systeme, die eine eigene Nutzerverwaltung bereitstellen, bieten oft direkt oder via Plug-in OIDC als alternatives Verfahren an.

Für unsere Beispielanwendung wird MQTT als Message-Bus eingesetzt. Als Produkt kommt Mosquitto zum Einsatz, eine leichtgewichtige Open-Source-Implementierung. Damit ein Single Sign-on (SSO) sowohl im Frontend (Angular, s. o.) als auch in den erforderlichen Umsystemen möglich ist, muss Mosquitto die Authentifizierung an den Spring Authorization Server delegieren. Die Rollen der einzelnen Komponenten gliedern sich dann wie folgt:

- Mosquitto-Broker: Resource Server
- Client: Angular-Browseranwendung
- Authorization Server: Spring Authorization Server

Mosquitto selbst bietet kein OAuth2 an, es gibt jedoch verschiedene Plug-ins, mit denen sich externe Authentifizierung nachrüsten lässt. Beispielsweise ist es mit mosquitto-go-auth [3] möglich, einen externen Web-

Listing 9: Prinzipielles Beispiel einer angepassten Konfiguration in SecurityFilterChain zur Verwendung von Refresh Tokens bei Public Clients

```
OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);
  http
    .getConfigurer(OAuth2AuthorizationServerConfigurer.class)
    // Enable OpenID Connect 1.0
    .oidc(Customizer.withDefaults())
    //refresh tokens for public clients
    .clientAuthentication(c ->
      c
        .authenticationConverter(new
AuthorizationServerConfigurationWithPublicClientAuthentication.
                                        PublicClientRefreshTokenAuthenticationConverter())
      .authenticationProvider(new
AuthorizationServerConfigurationWithPublicClientAuthentication.
                PublicClientRefreshTokenAuthenticationProvider(registeredClientRepository))
      ).tokenGenerator(tokenGenerator);
  http
    .exceptionHandling( exceptions -> exceptions
      .accessDeniedHandler(new AccessDeniedHandlerImpl())
    )
  return http.build();
```

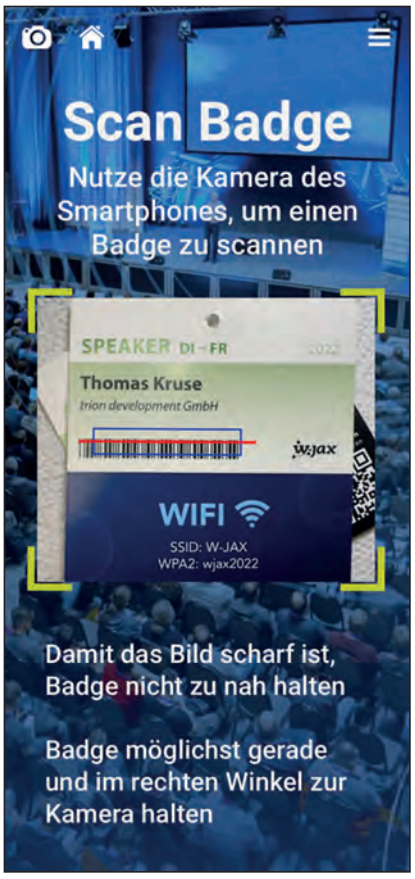

Abb. 6: Anmeldung an mobiler Anwendung mit Konferenz-Badge

dienst aufzurufen, der dann Authentifizierung und Autorisierung (ACL) bereitstellt. Das Plug-in bietet verschiedene Varianten an. Im JWT-Web-Modus sendet es den als Usernamen eines MQTT-Log-ins erhaltenen Wert als HTTP-Header-Authorization-Bearer, sodass dieser prinzipiell wie gewohnt genutzt werden kann (**Abb. 5**).

Der zugehörige Controller als Erweiterung des Spring Authorization Servers ist beispielhaft in Listing 10 gezeigt.

Anwendungsbeispiel Log-in mit Konferenz-Badge

Mit dem Spring Authorization Server erhält man viel Flexibilität. Sowohl bei der Anbindung verschiedener Persistenztechnologien und der Integration von und in Drittsysteme als auch bei der Option, eigene Authentifizierungsverfahren bereitzustellen. Da es sich um sicherheitskritische Bestandteile handelt, sollte das zugrunde liegende Design in jedem Fall wohlüberlegt sein. Hier bietet sich auch ein Review sowohl des Designs als auch der Implementierung durch eine externe Stelle an.

Nachdem wir die Sensibilität dafür geschaffen haben, soll es nun zu einem weiteren Aspekt aus unserem übergreifenden Beispiel gehen: Teilnehmer an der durch das System unterstützten Konferenzaktivität sollen sich mit Hilfe ihres individuellen Konferenz-Badges anmelden können. Dazu wird ein eigenes Authentifizierungsverfahren implementiert, das in diesem Fall von Frontend und Authorization Server unterstützt wird, sodass der Nutzerkomfort im Vordergrund steht und auf eine Umleitung und zusätzliches UI am Authorization Server verzichtet werden kann. Im Frontend findet eine lokale Erkennung des Barcodes des Konferenz-Badge statt. Damit soll verhindert werden, dass ein schlechtes Foto zuerst an das Backend übermittelt wird, nur um dann abgewiesen zu werden. Außerdem komprimiert die Frontendanwendung das Bild, um den Datenverkehr zu minimieren, und wandelt es zudem in ein Bild mit hohem Kontrast im Schwarz-Weiß-Format um.

Dann wird das Bild zum Backend übermittelt, wo der Barcode erkannt und als Authentifizierungsmerkmal genutzt wird. Ein Ausschnitt aus dem UI ist in **Abbildung 6** zu sehen

Fazit

Der Spring Authorization Server ist ein mächtiges und nützliches Werkzeug, wenn es darum geht, spezielle Anforderungen umzusetzen oder standardkonform OIDC bzw. OAuth2 zu nutzen, ohne ein zusätzliches Umsystem wie zum Beispiel Keycloak als Implementierung zu erfordern. Am Beispiel des Log-ins mit einem Konferenz-Badge wurde gezeigt, wie die Integration in bestehende Infrastrukturen bzw. besondere Anforderungen umgesetzt werden können. Nicht direkt offensichtlich, aber erwähnenswert ist der Anwendungsfall, dass eine Software als Gewerk zum Betrieb an und durch den Kunden übergeben wird und sich der Kunde eine möglichst geringe Komplexität wünscht. In diesem Fall ist die Auslieferung weniger Artefakte wünschenswert.

Liegen keine solchen Anforderungen vor und soll der OIDC-Provider möglicherweise sogar noch von anderen Anwendungen oder Systemen genutzt werden, ist es ratsam, die Verwendung einer dedizierten Anwendung mit entsprechend losgelöstem Lebenszyklus in Betracht zu ziehen. Dazu könnte beispielsweise Keycloak in Frage kommen.

Listing 10: Controller zur Log-in-Implementierung gemäß mosquitto-go-auth JWT Web

```
@RestController
@RequestMapping("/api/mosquitto/user")
public class UserProvider
{
  private final Logger logger = LoggerFactory.getLogger(getClass());
  @PostMapping
  public Map<String, Object> auth(Authentication authentication)
  {
    logger.info("USER check for {}", authentication != null ? authentication.getName() : "<>");
    return Map.of("ok", true, "error", "");
  }
}
```

Thomas Kruse unterstützt bei der Trion Unternehmen als Architekt, Trainer und Entwickler für Projekte, die Java-Technologien einsetzen. Sein Fokus liegt auf Java-basierten Webanwendungen und Cloud- und Containertechnologien, speziell mit Kubernetes. In seiner Freizeit engagiert er sich für Open-Source-Projekte und organisiert die Java User Group und die Frontend-Freunde in Münster. Kontaktieren Sie ihn gerne für weitergehende Unterstützung zu den behandelten Themen.

www.trion.de **tk@trion.de**

Links & Literatur

[1] https://github.com/spring-projects/spring-authorization-server/issues/297

[2] https://github.com/damienbod/angular-auth-oidc-client

[3] https://github.com/iegomez/mosquitto-go-auth

17 Toptipps aus der Praxis für bessere Anwendungen mit Kubernetes

Tuning für Kubernetes

Will man Anwendungen mit Kubernetes robuster und skalierbarer betreiben, lohnt es sich, einige wichtige Punkte zu beachten. Wir stellen Best Practices vor, um der Kritik, Kubernetes sei zu komplex, die Spitze zu nehmen.

von Thomas Kruse

Als Betriebsplattform hat sich Kubernetes in den vergangenen Jahren zunehmend etabliert. Dabei eignet es sich nicht nur als Abstraktion für verschiedene Cloud-Umgebungen, auch on Premises im eigenen Rechenzentrum oder auf gemieteten Servern vermag Kubernetes seine Stärken auszuspielen. Ein Kritikpunkt wird allerdings stets hervorgehoben: Es sei viel zu komplex im Vergleich zu individuellen Lösungen oder alternativen Ansätzen wie Docker. Es gelte viel zu lernen, um Kubernetes professionell nutzen oder gar betreiben zu können. Letzteres ist sicherlich nicht von der Hand zu weisen. Wenn es jedoch darum geht, Anwendungen in einer Kubernetes-Umgebung so zu betreiben, dass sie robust und skalierbar sind, kann man schon sehr weit kommen, wenn man einige Punkte berücksichtigt.

Genau diese Elemente samt ihren Hintergründen werden in diesem Beitrag behandelt. Die Punkte sind dabei entsprechend der empfohlenen Umsetzungspriorität sortiert. Einige beziehen sich speziell auf Spring-Boot- bzw. Java-basierte Anwendungen. Die Prinzipien lassen sich jedoch ohne Weiteres auf andere Programmiersprachen übertragen.

Probes: Health-Checks für die Anwendung

Falls eine Anwendung abstürzt, äußert sich das meistens zuerst dadurch, dass sich der Prozess beendet. Das bekommt Kubernetes mit und sorgt für einen Neustart des Containers, um die Verfügbarkeit wiederherzustellen. Doch nicht immer beendet sich die Anwendung bei schwerwiegenden Fehlern – gerade bei Java-Anwendungen ist das nicht unüblich. So kann zum Beipiel der Garbage Collector bei Speichermangel versuchen, freien Speicher zu beschaffen, bevor es zu einem *OutOfMemoryError* in der JVM kommt. Und selbst bei einem solchen Fehler kann es sein, dass die Anwendung als Prozess weiter vorhanden, jedoch so weit beeinträchtigt ist, dass die korrekte Funktion nicht mehr gewährleistet ist.

Hierzu bietet Kubernetes ein Verfahren, das sich Probes nennt, auf Deutsch „Prüfsonde“. Für den gerade beschriebenen Fall gibt es die *LivenessProbe*, also einen Check, ob die Anwendung noch lebendig ist. Schlägt die Prüfung fehl, wird der Prozess beendet. Nun greift wieder das normale Verhalten von Kubernetes: Crasht ein (Container-)Prozess, wird er neu erzeugt. Es ist also wichtig, eine solche Probe zu definieren, um „Zombie“-Situationen erkennen und darauf reagieren zu können.

Spring Boot stellt mit dem Actuator entsprechende Webendpunkte bereit, die durch Properties konfiguriert werden können. Ein Beispiel ist hier zu sehen:

```
management:
  endpoint:
    health:
      probes:
        enabled: true
```

Falls es sich um eine Anwendung handelt, die nicht per HTTP angesprochen wird, können andere Implementierungen der Probes verwendet werden, wie beispielsweise gRPC oder die Prüfung eines TCP-Sockets.

Nicht immer ist die Anwendung selbst für ein Problem verantwortlich zu machen: Es kann sein, dass ein wichtiges Umsystem nicht verfügbar ist oder dass die Anwendung z. B. aufgrund eines Mangels an begrenzten Ressourcen oder fehlender Initialisierungen nach dem Start nicht durch reguläre Nutzer aufgerufen werden kann. Ein Nutzer würde in diesem Fall lediglich einen Fehler erhalten, sodass es besser ist, wenn andere Instanzen der Anwendung genutzt werden. Das kann durch die *ReadinessProbe* gesteuert werden: Ist diese in einem Fehlerzustand, wird die Instanz aus der über Kubernetes Services umgesetzten internen Requestverteilung herausgenommen. Es ist auch möglich, eigene Elemente für den Spring Actuator bereitzustellen und damit individuelle Konfigurationen zu berücksichtigen. Dazu sei auf die sehr gute Dokumentation von Spring verwiesen.

Eine besondere Situation besteht dann, wenn eine Anwendung als Container neu startet und dabei von Initialisierungen eine gewisse Zeit in Anspruch genommen wird. In dieser Zeit ist der Prozess bereits gestartet, aber weder die *ReadinessProbe* noch die *LivenessProbe* werden korrekt arbeiten – gerade bei Webanwendungen. Den Prozess zu beenden, weil die *LivenessProbe* nicht arbeitet, würde lediglich zu endlosen Neustarts führen.

Um das zu vermeiden, gibt es die *StartupProbe*. Sie deaktiviert *LivenessProbe* und *ReadinessProbe* so lange, bis einmal ein Erfolgsfall eingetreten ist. Anschließend werden die anderen Checks aktiviert. Im Prinzip kann hier derselbe Endpunkt wie für die *LivenessProbe* verwendet werden.

Ein vollständiges Beispiel der Probes eines Containers in einem Pod ist in Listing 1 zu sehen. Dabei werden die Defaultpfade des Actuators verwendet. Zu weitergehenden Einstellungen sollte die Kubernetes-Dokumentation herangezogen werden.

Ressourcen: Optimale Clusternutzung

Ein Kubernetes-Cluster besteht typischerweise aus eher vielen und günstigen Maschinen (Nodes). Es gibt auch Set-ups, bei denen die Kubernetes Nodes durch virtuelle Maschinen auf wenigen leistungsfähigen physischen Maschinen abgebildet werden. Die Sinnhaftigkeit dieser Architektur soll an dieser Stelle einmal außen vor bleiben. Die Scheduler-Komponente von Kubernetes ist dafür zuständig, Workloads (Pods) so auf die verschiedenen Nodes zu verteilen, dass eine möglichst gute Nutzung der zugrunde liegenden Nodes erzielt wird und gleichzeitig der Ausfall einzelner Nodes nicht dazu führt, dass Anwendungen insgesamt nicht mehr verfügbar sind. Aber woher weiß der Scheduler, welchen Ressourcenbedarf ein Pod hat?

Das wird – natürlich – im Manifest deklariert. Dazu dient der Abschnitt *resources*, in dem sowohl für Speicher (*memory*) als auch für CPU-Nutzung (*cpu*) Unterabschnitte bereitstehen. Werden diese Daten nicht bekannt gegeben, kann der Scheduler lediglich als Heuristik auf die Anzahl Pods pro Node zurückgreifen. Dabei kann es schnell dazu kommen, dass einer Anwendung viel zu wenige Ressourcen zur Verfügung stehen und sie daher langsam läuft (zu wenig CPU) oder sogar mangels freiem Speicher beendet wird. Dabei können gleichzeitig andere Nodes eine viel zu geringe Auslastung aufweisen, da dort Pods mit sehr kleinem Ressourcenbedarf platziert wurden.

Es ist daher äußerst ratsam, die erforderlichen Ressourcen einer Anwendung unter typischer Last als Bedarf zu deklarieren. Dazu dient der Abschnitt *requests* unterhalb von *resources*. Der Scheduler wird dann unter Berücksichtigung der auf den jeweiligen Nodes vorhandenen Hardwareressourcen – abzüglich der durch andere *requests* als belegt markierten Ressourcen – eine möglichst gleichförmige und effiziente Nutzung sicherstellen. (Das ist alles andere als trivial, da es sich bereits um ein zweidimensionales Optimierungsproblem handelt, aber das überlassen wir einfach dem Scheduler.)

Man kann sich die Requests als eine Art Reservierung vorstellen. Kann der Cluster die angefragten Ressourcen nicht bereitstellen, bleibt der Pod im Status *pending* und wird keinem Node zugeordnet.

Was aber, wenn ein Pod sich, möglicherweise aus einer Fehlersituation heraus, anders benimmt, als das in einer typischen Lastsituation der Fall ist? Dann könnte ein Pod theoretisch alle Ressourcen eines Node verbrauchen und andere Pods könnten nicht zum Zuge kommen oder sogar beendet werden. Um das zu verhindern und zumindest nach dem Verursacherprinzip zu agieren, gibt es neben den *requests* auch *limits*. Die Limits werden nicht vom Scheduler beachtet, sondern zur Laufzeit der jeweiligen Container auf den Nodes als Begrenzung erzwungen. Falls die Limits überschritten werden, wird der Container in seiner CPU-Nutzung beschränkt, quasi gedrosselt, oder im Fall von zu hoher Speichernutzung beendet.

Nun können diese Werte unterschiedlich sein, zum Beispiel weil lediglich Requests angegeben wurden oder weil die Limits bewusst höher gewählt wurden als die Requests. Welchen Effekt hat das? Bei CPU ist

Listing 1: Konfiguration von Probes im Manifest von Kubernetes Pod

```
...
spec:
  containers:
  - ...
    startupProbe:
      httpGet:
        path: /actuator/health/liveness
        port: 8080
        initialDelaySeconds: 10
        periodSeconds: 5
    livenessProbe:
      httpGet:
        path: /actuator/health/liveness
        port: 8080
        initialDelaySeconds: 1
        periodSeconds: 3
    readinessProbe:
      httpGet:
        path: /actuator/health/readiness
        port: 8080
        initialDelaySeconds: 5
        periodSeconds: 2
```

Listing 2: Ressourcenkonfiguration am Pod

```
...
spec:
  containers:
  - ...
    resources:
      request:
        cpu: 2
        memory: 512Mi
      limits:
        cpu: 8
        memory: 512Mi
```

es im Prinzip kein Problem: Es werden so lange Pods auf Nodes zugeordnet, bis insgesamt keine CPU gemäß der Requests mehr übrig ist. Wollen ein oder mehrere Container mehr CPU-Zeit, als in Summe auf der Maschine vorhanden ist, passiert das, was sonst auch passiert: Es dauert eben etwas länger, die Prozesse erhalten abwechselnd Zeitscheiben der CPU zugewiesen. Verwendet ein Prozess jedoch mehr Speicher, als er per Request deklariert hat, kommt es darauf an, wie sich die anderen Prozesse verhalten. Ist noch Speicher übrig, der entsprechend verwendet werden kann, geschieht zunächst nichts. Ist jedoch kein Speicher mehr da, dann werden Prozesse ganz normal vom Kernel beendet (Out of Memory, kurz OOM). Dabei findet eine Gewichtung statt, die die Requests des Containers berücksichtigt. So wird eher ein Prozess beendet, der sehr stark über der angegebenen Nutzung liegt, als einer, der diese lediglich ein wenig überschreitet. Ein Grund mehr also, korrekte Requests zu deklarieren.

Ein weiterer Aspekt ist zu beachten: Ein zu starkes CPU-Limit kann eine geringe Startgeschwindigkeit der Anwendung bewirken. So kann es bei einer Anwendung, die im Regelbetrieb wenig CPU-Zeit benötigt, jedoch zum Start CPU-intensive Initialisierungsaufgaben durchführt, sinnvoll sein, das Limit entsprechend höher zu setzen. Andernfalls kann es sogar vorkommen, dass die Anwendung so langsam startet, dass *StartupProbe* oder *LivenessProbe* fehlschlagen und es zu einem Neustart kommt. Und das, obwohl möglicherweise auf dem Node noch ausreichend CPU frei wäre. Es ist daher ratsam, beim Speicherlimit nicht weit vom Request abzuweichen; beim CPU-Limit ist das dagegen durchaus möglich, bis dahin das Limit ganz entfallen zu lassen. Letzteres setzt jedoch eine gute Kenntnis des tatsächlichen CPU-Bedarfs und die entsprechende Deklaration voraus.

Aktuell ist es leider nicht ohne Weiteres möglich, verschiedene Limits in Abhängigkeit von dem Ergebnis der *StartupProbe* zu setzen, um zwischen Initialisierung und Regelbetrieb zu unterscheiden. Die technische Infrastruktur ist mit dem Vertical Pod Scaling prinzipiell jedoch bereits vorhanden, um darauf aufzusetzen. Listing 2 zeigt eine beispielhafte Ressourcendeklaration.

Die Java-Anwendung muss dazu passend ebenfalls konfiguriert werden, sodass sie nicht mehr Speicher verbraucht, als für sie reserviert ist. Leider ist das bei der JVM alles andere als ein trivialer Schalter *-Xmx512m* oder gar ein automatisches Tuning der JVM. Denn es gibt sehr viele unterschiedliche Elemente, die Speicher benötigen. Das bekannteste ist der Heap und evtl. noch der ehemalige PermGen, seit Java 8 Metaspace. Hier hilft dann nur, die Anwendung im Betrieb mit verschie-

denen Einstellungen zu beobachten und nachzusteuern. An dieser Stelle können sich auch Anhaltspunkte für gezieltes Profiling ergeben.

Umgebungsspezifische Konfiguration

Eine Anwendung wird in der Regel nicht ausschließlich produktiv betrieben, sondern auch als vorproduktive Variante, um durch Entwickler oder die Fachabteilung getestet zu werden. Dabei sind oft Modifikationen des Verhaltens nötig. So sollte ein Newsletter nicht an tatsächliche Empfänger versendet werden, wenn ein neues Layout getestet wird. Oder echte Überweisungen stattfinden, wenn die Umsetzung neuer regulatorischer Anforderungen validiert wird. Mit Kubernetes Namespaces ist es relativ leicht möglich, mehrere Umgebungen parallel bereitzustellen. Nun wäre es ziemlich unsinnig und aufwendig, für die Test- und Produktivumgebung unterschiedliche Container-Images zu bauen und dann in Betrieb zu nehmen. Zum einen ist es schwer, sicherzustellen, dass später in der produktiven Umgebung wirklich alles wie getestet funktioniert, wenn dort ein anderes Artefakt in Betrieb genommen wird, als vorher getestet wurde. Zum anderen verbrauchen derartig viele Container-Images auch unnötig viel Speicherplatz in der Registry und im Cluster. Ganz davon abgesehen, dass ein derartiges Vorgehen im regulierten Umfeld undenkbar ist. Damit dasselbe Artefakt gemäß den Anforderungen umkonfiguriert werden kann, sollten entsprechende Konfigurationsoptionen vorgesehen werden.

Die Konfiguration kann sich dabei auf zu verwendende Umsysteme wie Datenbank oder Messaging-Infrastruktur beziehen oder auf technische Einstellungen wie Poolgrößen und Timeouts, aber auch auf Fachlichkeit im Sinne von Feature-Toggles oder explizit vorgesehene Ausprägungen fachlicher oder technischer Konfigurationen, wie beispielsweise die Aktivierung und Parametrisierung von Regelwerken. Damit das funktioniert, müssen entsprechende Konfigurationsmechanismen von Anwendungen beim Design berücksichtigt werden.

Spring Boot bietet dabei mit Propertys eine flexible und gute Infrastruktur für technische Einstellungen. Diese können mittels Configuration-Propertys auch für den eigenen Bedarf der Anwendung verwendet werden. Kubernetes ermöglicht es, Umgebungsvariablen sehr einfach direkt im Manifest, durch *ConfigMap* oder Inhalte von *Secret*-Objekten zu definieren. Dabei bietet die Trennung der Verwendung am Container im Pod und der Definition der konkreten Ausprägungen in *Secret*- oder *ConfigMap*-Objekten Vorteile bezüglich der Verwaltbarkeit und im Hinblick auf die Trennung von Berechtigungen.

Eine weitere Möglichkeit besteht darin, *ConfigMap*- und *Secret*-Objekte im Dateisystem eines Containers als Volumes zu materialisieren, sodass sie wie reguläre Dateien ausgelesen werden können. Spring Boot bietet auch direkten Kubernetes-Support, um *ConfigMap*- und *Secret*-Objekte unmittelbar über das Kubernetes API zu nutzen. Ein Beispiel zur Einbindung von *ConfigMap* und *Secret* in einem Pod als Umgebungsvariablen zeigt Listing 3.

Listing 3: Einbindung von ConfigMap und Secret im Container als Environment-Variablen

```
...
spec:
  containers:
  - name: web-app
    env:
    - name: DB_PASS
      valueFrom:
        secretKeyRef:
          name: app-secrets
          key: DB_PASSWORD
    - name: DB_HOST
      valueFrom:
        configMapKeyRef:
          name: app-config
          key: DB_HOSTNAME
```

Robuster Umgang mit Umsystemen

Kubernetes stellt eine eher dynamische Umgebung dar. Aufgrund von Skalierungsereignissen, Ausfällen, nicht definierter Startreihenfolge oder simplen Updates kann es schon einmal sein, dass Umsysteme temporär nicht zur Verfügung stehen. Hier ist es wichtig, dass eine Anwendung damit umgehen kann und nicht in einen dauerhaften Fehlerzustand fällt oder gar nicht erst startet.

Der einfachste Mechanismus ist bereits recht effektiv: Nach einer gewissen Wartezeit erfolgt ein erneuter Versuch, ein entsprechendes Umsystem zu erreichen. Für den Hikari-Pool wäre die folgende Konfiguration ein Beispiel:

```
spring:
  datasource:
    hikari:
      initializationFailTimeout: -1
      connectionTimeout: 15000
```

Die Einstellungen sorgen dafür, dass die Anwendung auch dann startet, wenn die Datenbank zum Startzeitpunkt nicht verfügbar ist und bei Verbindungsproblemen zur Laufzeit automatisch neue Verbindungsversuche unternommen werden.

Dazu kann mit Spring Retry flexibel und mit wenig Implementierungsaufwand auch in fachlichem Code das entsprechende Resilience-Pattern eingesetzt werden. Ergänzend oder alternativ kann Resilience4j passende Implementierungen verschiedener Patterns bereitstellen. Hier sollte sich auch direkt eine Assoziation zur bereits erwähnten *ReadinessProbe* aufdrängen.

Alternativ könnte die Anwendung nach dem Prinzip „let it fail“ beim Start auch einfach fehlerhaft termi-

nieren. Das entspräche auch der Standardkonfiguration, und Kubernetes würde die Instanzen automatisch neu starten. Wenn dieser Zustand längere Zeit anhält, reduziert Kubernetes die Neustarts und der Status der Instanzen wird mit *CrashLoopBackoff* gekennzeichnet. Diese Neustarts erzeugen dabei unnötig Last im Cluster. Zudem ergibt sich, dass neu gestartete Instanzen sich anders verhalten als solche, die vor dem Problem gestartet wurden. Die sich daraus ableitenden Alerts können den Eindruck erwecken, dass die Anwendung selbst ein Problem hat, statt dass Umsysteme nicht erreichbar sind. Bei automatisierten Verfahren könnte es dann zu unerwünschten Rollbacks oder anderen manuellen Schritten kommen, die Aufwand generieren, der nicht unbedingt zielführend ist.

Zustandslose Instanzen

Kubernetes stellt innerhalb eines Clusters durch Service-Objekte definierte virtuelle IP-Adressen und DNS-Einträge zur Verfügung, über die sich eine Gruppe von Pods für einen Client einheitlich aufrufen lässt. Eine Verteilung von neuen TCP-Verbindungen und eingehenden UDP-Datagrammen auf die einzelnen Pod-Instanzen erfolgt automatisch durch Kubernetes. Die Verteilung ist dabei stochastisch mit dem Ziel einer gleichmäßigen Verteilung. Damit das gut funktioniert, ist es erforderlich, dass die einzelnen Anwendungsinstanzen keinen lokalen Zustand haben, von dem eine korrekte Verarbeitung der Requests abhängt. Das kann bereits bei Themen wie Authentifizierung anfangen: Hier empfiehlt sich ein tokenbasiertes Verfahren, das keinen lokalen Zustand benötigt. Dazu sollte das Token jeweils lokal validierbar sein – dazu später mehr. Auch die Darstellung sollte ohne serverseitigen Zustand auskommen. Das ist regelmäßig bei Angular, Vue und React bzw. allgemein SPA-Anwendungen der Fall. Wenn es sich um eine serverseitig gerenderte Webanwendung handelt, kann mit Spring Session der Zustand in eine separate Infrastruktur verlagert werden, sodass die einzelnen Anwendungsinstanzen austauschbar werden.

Ist das nicht der Fall, kann ein Service so konfiguriert werden, dass für einen Anwender, z. B. durch die Absender-IP bestimmt, stets dieselbe Instanz angesprochen wird. Nur wenn die Anwendung selbstständig Datenreplikation, Leader Election oder vergleichbare Verfahren einsetzt, sollte auf ein *StatefulSet* zurückgegriffen werden, um den einzelnen Instanzen eine dauerhafte Identität zu geben.

Replikate für Verfügbarkeit

Um eine hohe Verfügbarkeit sicherzustellen, geht Kubernetes den Weg, eher mehr Replikate einzusetzen, als die einzelnen Instanzen mit komplexen Verfahren beispielsweise durch Live-Migration abgesichert und dauerhaft bereitzustellen. Entsprechend sollte die Anwendung sich nicht nur aus Skalierungsgesichtspunkten, sondern vor allem auch für eine hohe Verfügbarkeit horizontal skalieren lassen. Die Umsetzung in Kubernetes ist dann typischerweise ein Deployment-Objekt oder *ReplicaSet*. Und das sollte auch mit einer Replikatanzahl von *>1* erfolgen!

Denn auch wenn ein *ReplicaSet* die entsprechende Anzahl von Pods sicherstellt und damit auf einen Node-Ausfall reagieren wird, dauert es eine gewisse Zeit. Wurde lediglich ein einzelnes Replikat als Ziel definiert, so ist in diesem Zeitraum die gesamte Anwendung nicht verfügbar. Das erschwert es auch, Nodes für Wartungsaufgaben außer Betrieb zu nehmen.

Als weiterführendes Thema zu diesem Aspekt sei noch auf *PodDisruptionBudgets* und die Möglichkeit, die Zuteilung von Pods auf Nodes bezüglich der Clustertopologie zu beeinflussen, verwiesen.

Aus Alt mach Neu: parallele Versionen

Anwendungscontainer müssen aus unterschiedlichen Gründen aktualisiert werden. So kann es Sicherheitsupdates geben, Fehler wurden behoben, neue Features ein- oder auch nur eine Konfigurationsänderung durchgeführt. Um während eines Rollouts einer neuen Version nicht die Verfügbarkeit einzubüßen, erlaubt das Kubernetes-Deployment ein sogenanntes Rolling-Update oder auch Blue-Green Rollout. Dabei stehen für eine gewisse Zeit die aktuelle und die vorherige Version parallel zur Verfügung und werden typischerweise von einem Service auch in zufälliger Ordnung angesprochen.

Das kann zu immensen Problemen führen, wenn die Anwendung nicht für dieses Szenario entwickelt wurde! Falls durch eine Datenbankmigration für oder durch die neue Version Veränderungen an den Tabellen- oder Feldstrukturen vorgenommen wurden, die für die alte Version nicht akzeptabel sind, kommt es nun zu Fehlern, wann immer die alte Version aufgerufen wird. Gleiches gilt für nach außen bereitgestellte APIs: Nicht jeder Client kommt damit klar, wenn zwischenzeitlich eine neuere API-Struktur geliefert oder erwartet wird – und dann ggf. wieder die vorherige.

Hier gilt es also sehr gut überlegt mit inkompatiblen Änderungen zu verfahren und diese optimalerweise über mehrere Anwendungsversionen in kleinen Schritten auszubringen. Auf diese Weise hält man sich auch den Weg eines Rollbacks bzw. Downgrades offen, sollte man etwa erst in der Konstellation der Produktivumgebung merken, dass mit der neuen Anwendungsversion etwas nicht korrekt funktioniert.

Verwendet man Werkzeuge zur Automatisierung von Datenbankmigrationen wie Liquibase oder Flyway – und das ist definitiv anzuraten –, muss man ggf. eine entsprechende Konfiguration vornehmen, damit diese auch mit einer neueren Schemaversion klarkommen, um ein Rollback besser zu unterstützen.

Herunterfahren ohne Panik

Es gibt auch im regulären Betrieb mehrere Gründe, aus denen eine Replik einer Anwendung beendet wird. Das kann bei einem Scale-down sein, bei einem Rolling-Update oder auch, weil die Pod-Instanz vom Node evaku-

iert wird. Dabei wird häufig erwartet, dass das zu keinen Problemen führt, schließlich bekommt Kubernetes durch den Statusübergang des Pods zum Terminating mit, dass diese Instanz heruntergefahren wird. Doch gibt es durchaus einige Konstellationen, die zu Problemen führen:

- Der Anwendungscontainer beendet sich als Reaktion auf *SIGTERM*, das Signal, das der Container zum Herunterfahren erhält, unmittelbar ohne Rücksicht auf den Anwendungszustand.
- Die Anwendung bearbeitet noch laufende Requests und beendet sich, bevor die Requests zu Ende bearbeitet wurden.
- Die Anwendung bearbeitet noch laufende Requests und benötigt dafür mehr Zeit, als Kubernetes einem Pod für den Shutdown einräumt.
- Der Anwendungscontainer erhält das *SIGTERM* und nimmt keine weiteren Requests entgegen, bevor die restliche Kubernetes-Infrastruktur, allem voran der Service, passend umkonfiguriert ist, um keine weiteren Requests an diese Instanz zu senden (es handelt sich hier im Prinzip um die umgekehrte Situation zu einer fehlenden *ReadinessProbe*).

Spring Boot verwendet üblicherweise einen Servlet-Container. Hier ist das Standardverhalten so, dass bei einem *SIGTERM* nicht nur keine weiteren Requests entgegengenommen, sondern aktuell laufende Requests abgebrochen werden. Hier hilft es, einen Graceful Shutdown zu konfigurieren. Dabei blockiert Spring Boot, dass der Servlet-Container neue Requests annimmt, ohne aber laufende abzubrechen.

Spring wartet per Voreinstellung nun bis zu 30 Sekunden auf die Abarbeitung laufender Requests, dann wird die Anwendung beendet. Dieser Wert ist mehr als großzügig bemessen, doch nicht jede Anwendung ist in der Lage, alle Requests in dieser Zeitspanne abzuarbeiten. Dazu kommt, dass auch Kubernetes 30 Sekunden als Schwellenwert vorsieht, bis eine Anwendung mit einem *SIGKILL* beendet wird, und der Countdown bei Kubernetes startet, bevor die Container im Pod den *SIGTERM* bekommen, der den Spring-Countdown startet. Hier sollte daher bewertet werden, welcher Zeitraum tatsächlich benötigt wird und entsprechende Einstellungen bei Spring (*timeout-per-shutdown-phase)* oder am Pod (*terminationGracePeriodSeconds*) vorgenommen werden.

Listing 4: Verzögerung des SIGTERM durch preStop Lifecycle Hook und Anpassung der maximalen Wartezeit zum Beenden in einem Pod-Manifest

```
spec:
  containers:
    image: ...
    lifecycle:
      preStop:
        exec:
          command: ["sh", "-c", "sleep 5"]
  terminationGracePeriodSeconds: 50
...
```

Nun gibt es noch die Situation, bei der Kubernetes einen Moment braucht, um die interne Verteilung anzupassen, während gleichzeitig der Anwendungscontainer schon keine Requests mehr annimmt. Um dieser Situation vorzubeugen, stellt Spring leider kein Mittel bereit. Jedoch bietet Kubernetes Lifecycle Hooks, darunter befindet sich *preStop*. Hier kann z. B. ein Kommando ausgeführt werden und erst wenn es terminiert, erhält der Container das tatsächliche *SIGTERM*-Signal. Da der Pod zum Zeitpunkt des Aufrufs von *preStop* bereits in der Phase *terminating* ist, wird der Pod bereits aus jedem zugehörigen Kubernetes Service herausgenommen. So reicht ein simples *sleep*, um Kubernetes Zeit für die Aktualisierung der Konfiguration zu geben, bevor die Anwendung das *SIGTERM* und somit auch weiterhin Requests akzeptiert. Ein Konfigurationsbeispiel für Graceful Shutdown und Anpassung der Zeitschwelle zum spätesten Herunterfahren bei Spring Boot ist das Folgende:

```
server:
  shutdown: graceful
spring:
  lifecycle:
    timeout-per-shutdown-phase: 40s
```

Ein weiteres Konfigurationsbeispiel zeigt Listing 4.

Definierter Prozess für Sicherheitsupdates

Viele Kunden treten mit dem Wunsch nach Schulung und Beratung im Kontext einer Kubernetes-Einführung an uns heran. Darunter sind erstaunlich viele, die durch Softwarelieferanten mehr oder minder sanft in Richtung Kubernetes gedrängt werden, da der Lieferant auf „Helm und Images“ umstellt. Nach den Prozessen gefragt, gibt es regelmäßig Verwunderung, dass derjenige, der die Container-Images erstellt, auch für das Patchmanagement und damit für einen wesentlichen Teil der Sicherheit verantwortlich ist. So kommt es nicht selten vor, dass es dann ein gemeinsames Meeting mit dem Zulieferer gibt, der sich bis zu dem Zeitpunkt in einer bequemen Lage angesichts der einheitlichen Zielumgebung und in der Führungsrolle durch den Wissensvorsprung wähnte. Und dann stellt sich heraus, dass die bisherigen Wartungsperioden, Lieferverträge und Übergabeprozesse auf keinen Fall Dinge wie wöchentliche Updates und Reaktionszeiten für kritische CVEs vorsehen.

Hier kann es auch zu unangenehmen Diskussionen mit weiteren Beteiligten kommen, wieso das Thema bisher nicht aufgekommen ist und wie man aus der Situation herauskommt. Hier sollten auf keinen Fall leichtfertig Zugeständnisse gemacht werden, selbst

wenn man in einem nicht regulierten Umfeld agiert. Dazu sagt zum Beispiel der Grundschutzbaustein SYS.1.6.A2 des Bundesministeriums für Sicherheit in der IT: „Die Verwaltung der Container DARF NUR nach einer geeigneten Planung erfolgen. Diese Planung MUSS den gesamten Lebenszyklus von Inbetrieb- bis Außerbetriebnahme inklusive Betrieb und Updates umfassen“, und SYS.1.6.A4: „Der Prozess zur Bereitstellung und Verteilung von Images MUSS geplant und angemessen dokumentiert werden“ [1].

Hier ist es essenziell, alle Zulieferer mit ins Boot zu holen und gemeinsame praxistaugliche Prozesse zu etablieren, die dann auch so gelebt werden. Empfehlenswert ist hier auch ein Investment in geeignete Automatisierungen für Übergabe, Test und Betriebseinführung. Viele Unternehmen setzen zunehmend auch auf Werkzeuge zur Schwachstellenerkennung. Diese können sicherlich eine Ergänzung, keinesfalls aber ein Ersatz für definierte Aktualisierungsprozesse der Images und den Umgang mit publizierten Schwachstellen sein.

Metriken bereitstellen – und auch benutzen

Keine Anwendung ist wie die andere, und auch die meisten Kubernetes-Cluster sind individuell. Sei es Art und Umfang der Datenverarbeitung bei Anwendungen, die JVM-Version mit den jeweiligen Optimierungen oder die konkrete Hardware, auf der etwas läuft. Es wird sich immer etwas unterscheiden. Damit wird das Sizing sowohl in vertikaler Hinsicht – welche Ressourcen sollen als Requests/Limits für einen Pod gewählt werden – als auch in horizontaler Hinsicht – wie viele Instanzen werden im Hinblick auf die Skalierung benötigt – eine Herausforderung. An dieser Stelle hat sich bewährt, dass von Infrastruktur und Anwendung Metriken bereitgestellt werden, um Aussagen zur Effizienz, aber auch zur Qualität des Systemzustands treffen zu können. Dabei lassen sich auf Basis der Metriken auch Prognosen für die Auslastung erstellen, sodass rechtzeitig zusätzliche Clusterkapazität geplant werden kann.

Metrikbasiertes Monitoring hat sich in dynamischen Umfeldern wie Kubernetes bewährt. Insbesondere, wenn dabei nicht nur technische, sondern auch fachliche Metriken bereitgestellt werden. So ist der Ausfall eines Nodes oder eines einzelnen Pods in der Regel keine Situation, die eine sofortige Handlung eines Administrators erfordert, sondern der Cluster nutzt die entsprechenden Konzepte zur Selbstheilung – ausreichend Pufferkapazität vorausgesetzt.

Doch wenn plötzlich Anomalien auftreten, können diese sehr nützliche Indikatoren für Probleme sein, die so behoben werden können, noch bevor eine Fehlermeldung einen Nutzer erreicht. So kann z. B. ein starker Anstieg von Passwortresets darauf hindeuten, dass etwas mit dem Identity-Service nicht korrekt funktioniert. Oder wenn in einem Onlineshop plötzlich die durchschnittliche Anzahl der Artikel im Warenkorb sinkt und das mit einem neuen Softwarestand korreliert werden kann, ist man möglicherweise einem Implementierungsproblem auf der Spur.

Bei Spring Boot werden dafür am einfachsten die Abhängigkeiten *spring-boot-actuator* und *micrometer-registry-prometheus* aufgenommen, wenn man mit Prometheus- oder OpenTelemetry-kompatiblen Metriksystemen arbeitet. Spring Boot bringt dann bereits automatische Integrationen in viele technische Infrastrukturmetriken wie z. B. Tomcat oder die JVM mit. Eigene Metriken lassen sich mit wenigen Zeilen Code oder gezielten Annotationen ebenfalls komfortabel bereitstellen.

Anwendungs-Images ohne Dämonen

Container-Images werden typischerweise mit Docker und einem Dockerfile gebaut. Der Standard hat den Vorteil, dass er weit verbreitet ist und durch das Dockerfile auch eine gute Reproduzierbarkeit gegeben ist. Allerdings wird damit auch stets ein Docker Daemon benötigt. Und der Daemon hat sehr weitreichende Rechte auf dem System, auf dem er läuft. Keinesfalls sollte hier die in manchen Tutorials beschriebene Option gewählt und der – sofern überhaupt dazu eingesetzte – Docker Daemon dem Kubernetes Node verfügbar gemacht werden. Damit

würden die gesamte Isolation und die Sicherheitsaspekte der Containerumgebung aufgeweicht. Auch die Option DinC/DinD (Docker-in-Container bzw. Docker-in-Docker) kommt mit einem Wermutstropfen: Damit Docker innerhalb eines Containers funktioniert, muss der Container *privileged* laufen. Also ergibt sich auch hier eine Aufweichung der Sicherheitsfunktionen.

Es gibt jedoch Alternativen. Eine davon ist Kaniko, ein Projekt von Google [2]. Kaniko ist in der Lage, ohne Docker Daemon auf Basis eines Dockerfile Images zu bauen und in eine Registry zu pushen. Kaniko kann dabei selbst problemlos in einem nichtprivilegierten Container ausgeführt werden und eignet sich damit auch sehr gut für CI-Pipelines. Ein weiteres Werkzeug – ebenfalls von Google – ist Jib, wobei der Fokus ursprünglich auf Java-Anwendungen lag [3]. Inzwischen gibt es neben Jib-Plug-ins für Maven und Gradle das Jib-CLI, mit dem auch beliebige andere Anwendungsarten in Container-Images verpackt werden können. Die Konfiguration übernimmt dabei eine Jib-Manifest-Datei. Im Gegensatz zu Docker oder Kaniko können keine Dockerfiles verwendet werden und Jib ist ausschließlich darauf ausgelegt, Images zusammenzustellen. Der Build der Anwendung muss damit an einer anderen Stelle stattfinden, was aber auch sinnvoll ist. Jib-CLI kann auch als Container-Image ausgeführt werden, eine detaillierte Anleitung findet sich unter [4].

Listing 5: Jib-Plug-in in einem Maven-Projekt

```
<project ...>
  <properties>
  </properties>
...
  <build>
    <plugins>
      <plugin>
        <groupId>com.google.cloud.tools</groupId>
        <artifactId>jib-maven-plugin</artifactId>
        <version>3.4.0</version>
        <configuration>
          <from>
            <image>my/baseimage</image>
          </from>
          <to>
            <image>repo.example.com/my/app:1</image>
          </to>
        </configuration>
      </plugin>
  ...
  </build>
</project>
```

Listing 6: Dockerfile, das eine Spring-Boot-Anwendung in separierten Layern als Image bereitstellt

```
FROM eclipse-temurin:21 as builder
WORKDIR application
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layertools -jar application.jar extract

FROM gcr.io/distroless/java21:nonroot
WORKDIR application
COPY --from=builder application/dependencies/ ./
COPY --from=builder application/snapshot-dependencies/ ./
COPY --from=builder application/resources/ ./
COPY --from=builder application/application/ ./
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

Listing 7: Jib-Plug-in mit eigenem Plug-in zur Optimierung von Spring Boot

```
<project ...>
  <properties>
  </properties>
...
  <build>
    <plugins>
      <plugin>
        <groupId>com.google.cloud.tools</groupId>
        <artifactId>jib-maven-plugin</artifactId>
        <version>3.4.0</version>
        <dependencies>
          <dependency>
            <groupId>com.google.cloud.tools</groupId>
            <artifactId>jib-spring-boot-extension-maven</artifactId>
            <version>0.1.0</version>
          </dependency>
        </dependencies>
        <configuration>
          <pluginExtensions>
            <pluginExtension>
              <implementation>com.google.cloud.tools.jib.maven.extension.
                springboot.JibSpringBootExtension</implementation>
            </pluginExtension>
          </pluginExtensions>
          <from>
            <image>my/baseimage</image>
          </from>
          <to>
            <image>repo.example.com/my/app:1</image>
          </to>
        </configuration>
      </plugin>
  ...
  </build>
</project>
```

In Listing 5 wird die Integration des Jib-Maven-Plugins in eine Spring-Boot-Anwendung gezeigt. Das Image wird automatisch als Teil des Maven-Lebenszyklus gebaut und kann direkt in eine Registry gepusht werden.

Container-Image-Layer optimieren

Ein Container-Image besteht aus verschiedenen Layern. Diese dienen dazu, Daten zu deduplizieren, da jeder Layer individuell gespeichert und verteilt werden kann. Das ist wichtig, weil für jede neue Anwendungsversion auch ein neues Image erzeugt wird. Die damit einhergehenden Datenmengen können beachtlich werden – besonders dann, wenn die Images ohne Sensibilität für das Thema erstellt werden.

Der häufigste Fall, in dem ein neues Container-Image erstellt wird, ist die Wartung der Anwendung. Bugfixes oder Erweiterungen führen dazu, dass eine neue Version getestet und hoffentlich auch in Betrieb genommen wird. Die verwendeten Bibliotheken und das gewählte Basisimage ändern sich dabei zunächst nicht. Daher wäre es wünschenswert, dass diese unveränderlichen Daten in eigenen Layern liegen, sodass sie auch als Teil des neuen Images einfach so wie sie sind wiederverwendet werden können. Zusätzlich ist dabei zu beachten, dass bei einer Änderung in einem Layer auch alle nachfolgenden Layer neu erzeugt werden müssen. Es ist auch nicht möglich, in einem Nachfolge-Layer Inhalte zu löschen: Ein Layer ist stets unveränderlich und kann auch von nachfolgenden Layern nicht direkt beeinflusst werden.

In der Praxis bedeutet das, dass die Layer entsprechend der Änderungsfrequenz der Inhalte sortiert werden und größere vor kleineren Layern liegen sollten.

Bei einer typischen Java-Anwendung machen den größten Teil der Inhalte zum einen das Basis-Image und zum anderen die verwendeten Libraries aus. Letztere ändern sich gewöhnlich nur bei einem Update des verwendeten Frameworks oder wenn einzelne Libraries manuell verwaltet werden. Es wäre also geschickt, diesen Teil in jedem Fall in einem anderen Layer zu platzieren als den selbst entwickelten Code, der sich eher häufig ändert. Dazu kommen noch Abhängigkeiten, die evtl. selbst in Entwicklung sind und typischerweise durch eine *-SNAPSHOT*-Versionsangabe gekennzeichnet sind.

Glücklicherweise bringt Spring Boot eigenständig Support dafür mit, diese Dateien aufzuteilen, sodass daraus passende Layer erstellt werden können. Es ist sogar möglich, eigene Konfigurationen anzugeben, wenn man zusätzliche Layer benötigt. Von Haus aus teilt Spring Boot so auf:

- *dependencies:* Abhängigkeiten mit Versionen, die kein *SNAPSHOT* beinhalten
- *spring-boot-loader:* der Spring Boot Loader für die JAR-Klassen
- *snapshot-dependencies:* Abhängigkeiten, die *SNAPSHOT* in der Version enthalten
- *application:* alle Klassen und Ressourcen der Anwendung

Die Anwendung wird als JAR paketiert, von dieser Vorbereitung zur Aufteilung auf Layer sieht man also zunächst nichts. Das Spring-Boot-Maven-Plug-in baut jedoch Unterstützung zum passenden Extrahieren in separate Ordner mit in das JAR ein. Das Werkzeug wird durch *java -Djarmode=layertools -jar eigene-app.jar* angesprochen. Ein beispielhaftes Dockerfile zur direkten Verwendung, nachdem die Anwendung mit Maven gebaut wurde, ist in Listing 6 zu sehen. Jib bringt automatischen Support für die Aufteilung in Layer durch ein eigenes Plug-in mit. Eine Beispielkonfiguration mit Maven ist in Listing 7 dargestellt.

Image-Größe optimieren

Wenn man darauf achtet, die Layer gut aufzuteilen, sollte man ebenfalls darauf achten, dass die Images lediglich die Größe haben, die erforderlich ist. Damit erhält man direkt mehrere Vorteile: Zum einen wird weniger Bandbreite bei der Verteilung benötigt. Zum anderen reduziert sich auch die Angriffsoberfläche, wenn im Image nichts enthalten ist, das nicht benötigt wird.

Es beginnt also mit der Wahl eines passenden Basis-Image. Viele beinhalten eine Linux-Distribution mit Shell und Hilfswerkzeugen samt zugehöriger Libraries. Ideal wäre hier eine Distribution, die auf geringe Größe optimiert ist. Das aus dem Routerumfeld stammende Alpine Linux ist so eine Distribution. Manche Anwendungen haben mit der Lib-C-Implementierung (musl statt glibc) jedoch Probleme und es gibt augenscheinlich auch eine geringere Performance bei bestimmten Nutzungsarten. Eine Evaluation lohnt sich, wenn die sich daraus potenziell ergebende Platzoptimierung dafürspricht. Alternativ gibt es abgespeckte Debian-Basis-Images, die dann eine glibc mitbringen und verhältnismäßig klein sind, wenn auch deutlich größer als auf Alpine basierende.

Hier stellt sich die Frage, welche Java-Laufzeitumgebung verwendet werden soll. Es gibt inzwischen einige Anbieter, die in der Regel mehr oder weniger nah am offiziellen OpenJDK-Standard sind. In der Regel werden hier auch Kombinationen aus Distribution und Java-Laufzeitumgebung angeboten. Eine ganz gute Balance zwischen Größe und Kompatibilität findet sich zum Beispiel bei den Distroless-Images von Google, die es für verschiedene Laufzeitumgebungen gibt [5]. Java 21 ist derzeit in Entwicklung und kann beispielsweise bereits als *gcr.io/distroless/java21:nonroot* verwendet werden. Vorsicht ist geboten, wenn eine Shell, zum Beispiel zum Debugging, gewünscht ist: Diese ist aus Platz- und Sicherheitsgründen lediglich in den Images der Debug-Variante enthalten.

Eine besondere Variante einer Java-Laufzeitumgebung wird durch die Kombination der Werkzeuge jdeps und jlink möglich, damit kann eine minimale Laufzeitumgebung individuell nach Bedarf der Anwendung gebaut werden. Durch jdeps werden alle Bestandteile der Anwendung und ihrer Abhängigkeiten auf Abhängigkeiten von der Java-Laufzeitumgebung untersucht. Das

Ergebnis wird dann verwendet, um eine minimale Java-Umgebung speziell für die Anwendung zu bauen. Auch auf dieser Ebene ergeben sich damit die Vorteile der Speicherplatzersparnis und reduzierten Angriffsoberfläche. Allerdings funktioniert damit die layerbasierte Deduplizierung nicht mehr anwendungsübergreifend.

Ein noch extremerer Weg ist die Verwendung von Native-Image aus dem GraalVM-Projekt. Damit wird eine Java-Anwendung in ein plattformunabhängig ausführbares Programm übersetzt, das die Substrate VM als Laufzeitumgebung mitbringt. Damit sind besonders kleine Images möglich; in bestimmten Konstellationen ist sogar ein Scratch-Basis-Image, also ein komplett leeres Image nutzbar. In der Praxis funktioniert das mit Spring Boot recht gut, wenn man einige Punkte beachtet (Details finden sich unter [6]).

Noch besser funktioniert es mit Frameworks, die speziell für diesen Einsatzzweck entwickelt wurden, beispielsweise Micronaut. Damit lassen sich dann Imagegrößen von deutlich unter 100 MB erreichen.

Einige Nachteile sind nicht direkt offensichtlich: Die Build-Zeit und der Ressourcenbedarf im Build sind wesentlich höher als für eine typische Java-Anwendung, bei der nur Class-Files erzeugt werden. Zudem ist das Ergebnis dann auch plattformspezifisch für die Kombination von Betriebssystem und CPU-Architektur.

Flexible Konfigurationen oder automatischer Operator

Wenn es erst einmal losgeht mit Kubernetes, dann ziehen dort in der Regel zügig viele Anwendungen und Dienste ein. Hier lohnt es sich, Verfahren zum Konfigurationsmanagement zu etablieren. Das kann Helm sein, muss es aber nicht: Gerade wenn es um Anwendungen geht, die lediglich innerhalb einer Organisation betrieben werden, gibt es regelmäßig wenig Freiheitsgrade für Konfigurationsoptionen. Ein schlankes Werkzeug wie Kustomize kann dann bereits ausreichen, um die Manifeste für eine Handvoll Umgebungen anzupassen. Helm spielt seine Stärken besonders dann aus, wenn Software für Nutzer bereitgestellt werden soll, bei denen stark unterschiedliche Betriebsumgebungen oder auch unbekannte Umgebungen und Anforderungen bestehen. Helm Charts können selbst wiederum versioniert und durch geeignete Artefakt-Repositorys bereitgestellt werden.

Sollen viele gleichartige Anwendungen, Anwendungen, deren Konfigurations-API als Kubernetes API bereitgestellt werden soll, oder Anwendungen mit einem anspruchsvollen Lebenszyklus zur Verfügung gestellt werden, so kann sich die Entwicklung eines passenden Operators lohnen. Ein Beispiel dafür ist die Bereitstellung einer Datenbank inkl. zugehöriger Wartungsprozesse, automatischer Datensicherung, Verfahren zum Ausbringen von Updates und Erstellung von Replikaten für Testumgebungen. Über Custom Resource Definitions (CRDs) könnte ein Operator diese Prozesse anbieten, aber auch klassische Verwaltungsfunktionen für Nutzer, Rechte und Datenbanken bzw. Schemata.

Lastautomatisierung

Wie verhält sich eine Anwendung unter typischer Last? Welche Ressourcen, Requests oder auch Limits sind sinnvoll? Funktioniert die Skalierung? Um diese Fragen belastbar zu beantworten, bedarf es geeigneter Tests. Optimalerweise werden diese so automatisiert, dass es möglich ist, eine vollständige Umgebung – hier spielt Kubernetes seine Stärken ganz klar aus – inkl. Umsystemen und Testdaten bereitzustellen. Anschließend können dagegen automatisierte Lasttests durchgeführt und anschließend die Umgebung vollständig wieder abgebaut werden.

Welches Werkzeug dazu konkret genutzt wird und ob ein verteilter Lasttest erforderlich ist, um aussagekräftige Ergebnisse zu erzielen, muss im Einzelfall entschieden werden. In jedem Fall sind derartige Tests eindeutig im Bereich der Entwicklungskompetenz anzusiedeln. Die Umsetzung der Automatisierung kann durch ein DevOps-Team erfolgen oder sie kann auch direkt durch die Entwickler als Teil des Anwendungslebenszyklus entstehen.

Stellen Anwendung und Infrastruktur Metriken bereit, so hat man in Kombination mit automatisierten Lasttests ein hervorragendes Fundament, um darauf aufbauend auch Performancetuning zu betreiben und sich gegen Regressionen bei technischen wie fachlichen Updates zu schützen.

Startzeit von Anwendungen optimieren

Im Enterprise-Umfeld kommt die Anforderung nach dynamischer Skalierung eher selten und lediglich für einzelne Anwendungen auf. Soll es jedoch einmal schnell gehen – und sei es, weil Ressourcen nach Verbrauch abgerechnet werden –, ist eine kurze Startzeit der Anwendung wünschenswert. Bevor eine Anwendung startet, muss natürlich zuerst das Image auf dem Ziel-Node vorhanden sein. Auch unter diesem Gesichtspunkt lohnt sich also eine Minimierung der Image- und Anwendungsgröße.

Nachdem das Image bezogen wurde, muss die Anwendung starten. Java-Anwendungen sind nun nicht gerade berühmt für schnelle Startzeiten. Bei Spring-basierten Anwendungen kommt zur JVM zusätzlich hoher Aufwand für das Scanning des Classpath und Reflection-basierter Zugriffe hinzu, die besonders aufwendig sind. Das inzwischen abgekündigte Projekt spring-content-indexer kann bei umfangreichen Codebasen helfen. Im Hinblick auf GraalVM Native-Image wird Spring AOT (Ahead of Time) entwickelt. Damit werden Initialisierungen von der Laufzeit auf den Build-Zeitpunkt verlagert. In Zukunft sollen davon auch JVM-Anwendungen profitieren, sodass die Startgeschwindigkeit verbessert wird. Mit GraalVM Native-Image reduziert sich die Startzeit einer typischen Anwendung dramatisch: Benötigt selbst eine kleine Spring-Boot-Anwendung oft mehrere Sekunden zum Starten, so liegen GraalVM-Native-Image-Anwendungen typischerweise im Bereich von wenigen hundert Millisekunden.

Doch nicht alle Anwendungen sind für Native-Image geeignet. Alternativ dazu gibt es das CRaC-Projekt: Coordinated Restore at Checkpoint erlaubt es einer JVM-Anwendung, direkt in einem initialisierten Zustand von JVM und Anwendung zu starten. Oder besser gesagt: fortzufahren. Denn die Anwendung wird im eigentlichen Sinne nicht neu gestartet, sondern aus einem vorherigen Start wird ein Snapshot generiert, der dann wieder zum Leben erweckt wird. Damit stehen dann alle Optimierungen der HotSpot JVM und ihrer Dynamik zur Verfügung, zudem startet ein darauf basierender Container extrem schnell.

Doch damit geht eine deutlich höhere Komplexität einher: Es muss ein passender Snapshot der Anwendung generiert werden. Dazu muss die Anwendung gestartet und optimalerweise geschickt unter Last gesetzt werden, damit die JVM bereits die gewünschten JIT-Optimierungen durchführt. Dazu muss ein passender Snapshot erzeugt werden, dessen Größe unter anderem davon abhängt, wie die Speicherbelegung des Prozesses ist. Der Snapshot muss auf der Zielplattform erzeugt werden. Da auch alle umgebungsspezifischen Einstellungen durch den Snapshot fixiert werden, muss potenziell ein Snapshot je Umgebung erzeugt werden. Das impliziert, dass er außerhalb des Container-Images, z. B. per Volume, für alle Instanzen der Umgebung zur Verfügung gestellt wird. Dabei ist zu beachten, dass auch alle Credentials und Secrets, die die Anwendung nutzt, in diesem Snapshot enthalten und daher besonders zu schützen sind. CRaC weist damit eine erhebliche Betriebskomplexität auf und dürfte sich eher für spezielle Anwendungsfälle lohnen. Für neu zu entwickelnde Anwendungen empfiehlt sich da eher ein Blick auf Micronaut oder Spring Native.

Speicherbedarf der Anwendung optimieren

Java braucht viel Speicher und belohnt dafür mit einer hohen Leistung, ohne dass Entwickler sich dazu besonders verrenken müssten. Doch wenn Anwendungen mit mehreren Replikaten laufen, um Verfügbarkeit oder Skalierungsanforderungen zu genügen, summiert sich das Ganze recht schnell. Um den Speicherbedarf zu reduzieren, bieten sich im Wesentlichen drei Optionen an:

- *Tuning der verschiedenen JVM-Settings:* So wird beispielsweise pro Thread ein Megabyte Speicher für den Stack alloziert, was nicht immer erforderlich ist. Zum anderen ist es oft möglich, die Anzahl der zu nutzenden Threads zu reduzieren, indem ein anderes Paradigma genutzt wird oder virtuelle Threads von Project Loom verwendet werden.
- *Optimierung des Anwendungsdesigns:* Zu cachende Daten beispielsweise nicht in der JVM halten, sondern in einer dedizierten Cacheinfrastruktur; auch die Verwendung geeigneter Datenstrukturen und ein wenig Umsicht bei der Implementierung können zu erheblichen Einsparungen beim Heap-Speicher führen.
- *Verwendung von GraalVM Native-Image:* Damit gebaute und optimierte Anwendungen brauchen je nach Art schon mal eine Größenordnung weniger Speicher als klassische HotSpot-Anwendungen. Das macht sich besonders dann bezahlt, wenn die Umgebung im Hinblick auf Ressourcen eingeschränkt ist, wie es bei Embedded-Systemen oft der Fall ist. Eine Micronaut-Anwendung unter niedriger Last mit Webserver und HTTP-Client braucht so auf einem ARM64-System mit Native-Image 31 MB Speicher insgesamt. Vor diesem Hintergrund können sich auch angesichts der anderen Vorteile wie Startgeschwindigkeit und Image-Größe entsprechende Evaluationen lohnen.

Davon unabhängig sollte natürlich stets auch nach Memory-Leaks Ausschau gehalten werden. Zur Identifikation bietet sich erneut metrikbasiertes Monitoring an. Wurden entsprechende Probleme identifiziert, gilt es mit den passenden Werkzeugen eine genaue Diagnose zu stellen und Abhilfe zu schaffen.

Fazit

Es gibt in der Tat einige elementare Dinge zu beachten, wenn Anwendungen in Kubernetes optimal betrieben werden sollen. Wer bei den obigen Punkten auf eine gute Umsetzung achtet, ist bereits sehr gut aufgestellt. Natürlich ist damit noch nicht alles perfekt, es gibt noch weitere Punkte im Hinblick auf Absicherung und Automatisierung, das würde jedoch den Rahmen dieses Beitrags sprengen.

Zudem kommen dabei auch zunehmend individuelle Aspekte ins Spiel, die von regulatorischen Vorgaben, der Art des Clusterbetriebs und auch von konkreten Anforderungen der jeweiligen Organisation abhängen, sodass pauschale Ratschläge nicht angemessen sind. Das Hinzuziehen von Beratern mit umfangreicher Erfahrung bei der Entwicklung von Anwendungen für Kubernetes und dem Betrieb großer Cluster macht sich dann oft schnell bezahlt.

Thomas Kruse unterstützt bei der Trion Unternehmen als Architekt, Trainer und Entwickler für Projekte, die Java-Technologien einsetzen. Sein Fokus liegt auf Java-basierten Webanwendungen und Cloud- und Containertechnologien, speziell mit Kubernetes. In seiner Freizeit engagiert er sich für Open-Source-Projekte und organisiert die Java User Group und die Frontend-Freunde in Münster. Kontaktieren Sie ihn gerne für weitergehende Unterstützung zu den behandelten Themen.

www.trion.de **tk@trion.de**

Links & Literatur

[1] Bundesamt für Sicherheit in der Informationstechnik: „SYS.1.6 Containerisierung (Edition 2022)“: https://www.bsi.bund.de, Stichwort Containerisierung

[2] https://github.com/GoogleContainerTools/kaniko

[3] https://github.com/GoogleContainerTools/jib

[4] https://www.trion.de/news/2021/04/02/jib-cli-docker-images.html

[5] https://github.com/GoogleContainerTools/distroless

[6] https://www.trion.de/news/2022/01/17/spring-native-produktiv.html

Event Sourcing für mobile Echtzeitanwendungen

Datengeschichte schreiben

Event Sourcing verspricht viele Vorteile wie Nachvollziehbarkeit, einfache Datenmigration oder leichtere Fehleranalyse. In diesem Artikel wollen wir uns anhand einer Beispielanwendung auf Basis von Spring Boot eine einfache Event-Sourcing-Architektur anschauen, mit der sich die Vorteile realisieren lassen.

von Stefan Reuter

Klassischerweise wird in einer Anwendung der aktuelle Zustand der Daten in einer relationalen oder NoSQL-Datenbank gespeichert. Dieser Ansatz ermöglicht einen direkten Zugriff auf die Daten, bietet aber wenige Möglichkeiten, zu ermitteln, wie der aktuelle Zustand erreicht wurde. Zwar gibt es zahlreiche Ansätze zur Speicherung der Historie von Aktionen, die zum aktuellen Zustand geführt haben, doch abgesehen von der zusätzlichen Komplexität können diese Modelle leicht voneinander divergieren und inkonsistent werden.

Event Sourcing ist eine Methode zur Speicherung des Zustands einer Anwendung durch eine Historie von Ereignissen, die in der Vergangenheit stattgefunden und zu diesem Zustand geführt haben. Der aktuelle Zustand wird auf Basis der vollständigen Ereignishistorie rekonstruiert, wobei jedes Ereignis eine Änderung oder einen Fakt in unserer Anwendung darstellt. Ereignisse bieten uns eine Single Source of Truth darüber, was in unserer Anwendung passiert ist. Der aktuelle Zustand wird dagegen nur aus der Liste der Ereignisse abgeleitet. Das ist besonders vorteilhaft in Anwendungen, die beispielsweise ein vollständiges Auditlog für externe Prüfer bereithalten müssen oder anderweitig hohe Anforderungen an die Nachvollziehbarkeit stellen. Der aktuelle Zustand wird dabei oft als materialisierter Zustand bezeichnet.

Welche Vorteile bietet Event Sourcing?

Event Sourcing hat gegenüber einem klassischen Design eine Reihe von Vorteilen, die wir zum Teil bereits angesprochen haben. Zunächst einmal wird die Transparenz des Systems durch Event Sourcing erheblich verbessert. Jedes Ereignis, das eine Veränderung bewirkt, wird als Event gespeichert, sodass der Zustand des Systems zu jedem Zeitpunkt in der Vergangenheit rekonstruiert werden kann. Diese Fähigkeit, die Historie Schritt für Schritt nachzuvollziehen, ist nicht nur für das Debugging und die Fehleranalyse von unschätzbarem Wert, sondern auch für das Verständnis der Evolution eines Systems über die Zeit.

Ein weiterer wesentlicher Vorteil liegt in der Unterstützung von Auditing und Compliance. In vielen Branchen, insbesondere in solchen, die stark reguliert sind, ist es erforderlich, nachzuweisen, wie ein bestimmter Zustand erreicht wurde und welche Ereignisse dazu geführt haben. Dadurch lässt sich offenlegen, dass das System in einer bestimmten Weise funktioniert und beispielsweise die Grundsätze ordnungsgemäßer Buchführung einhält. Event Sourcing ermöglicht eine lückenlose, unveränderliche Aufzeichnung aller Vorgänge, die für die Einhaltung gesetzlicher Vorschriften und für Auditzwecke entscheidend sein kann.

Die Migration und Evolution von Systemen wird durch Event Sourcing ebenfalls vereinfacht. Da der aktuelle Zustand eines Systems aus den gespeicherten Events rekonstruiert wird, können Änderungen an der Struktur oder der Logik des Systems durchgeführt werden, indem man die Events durch neue Verarbeitungslogiken führt. Dadurch wird das Risiko reduziert, das mit der Einführung neuer Funktionen oder mit der Migration von Daten verbunden ist, und eine flexible Weiterentwicklung und Erweiterung des Systems ermöglicht.

Auch die Fehleranalyse wird durch Event Sourcing deutlich erleichtert. Wenn ein Problem auftritt, können Entwickler die Sequenz von Events untersuchen, die zu dem unerwünschten Zustand geführt haben. Diese granulare Sicht auf Aktionen und deren Auswirkungen auf das System macht es einfacher, die Ursache von Problemen zu identifizieren und zu beheben, während in klassischen Systemen häufig der Umweg über ein mehr oder weniger vollständiges Logging genommen werden muss.

Ein weiterer Vorteil von Event Sourcing ist die verbesserte Skalierbarkeit. Durch die Trennung der Schreib- und Leseoperationen können Anwendungen so skaliert werden, dass sie mit einem hohen Volumen an Anfragen umgehen können. Die Schreiboperationen in die Ereignishistorie sind schnelle und einfache Operationen (append-only) und können zentralisiert bleiben, um Konsistenz und Integrität zu gewährleisten, während Lesemodelle auf mehrere Instanzen aufgeteilt werden können, um die Last zu verteilen und die Antwortzeiten zu verbessern. Insbesondere in transaktionsorientierten Systemen können so Lastspitzen beim Schreiben von Transaktionen gut unterstützt werden. Die Aktualisierung der Lesemodelle kann nachgelagert asynchron erfolgen.

Zusätzlich zu diesen Vorteilen bietet Event Sourcing eine hohe Flexibilität bei der Gestaltung von Abfragen und Ansichten, verbesserte Widerstandsfähigkeit gegen Datenverlust und das Potenzial für asynchrone Verarbeitung sowie Echtzeit-Updates.

Commands, Events und Aggregate

Event Sourcing lässt sich gut mit Domain-Driven Design (DDD) kombinieren. Insbesondere die Konzepte Command, Event und Aggregat aus dem taktischen Domain-Driven Design lassen sich im Kontext von Event Sourcing gut einsetzen. Im DDD-Kontext repräsentiert ein Command einen Auftrag oder eine Anforderung, eine bestimmte Aktion auszuführen. Commands sind in der Regel das Ergebnis von Benutzerinteraktionen oder anderen Systemen, die eine Änderung im Systemzustand bewirken möchten. Sie sind imperativ und beschreiben, was getan werden soll, ohne notwendigerweise zu spezifizieren, wie es getan werden soll. Ein Command kann entweder ausgeführt oder abgelehnt werden. So wäre beispielsweise ein Auftrag zur Überweisung eines Geldbetrags ein Command. Wird dieses Command ausgeführt, findet die Überweisung statt, der Betrag wird dem Quellkonto belastet und dem Zielkonto gutgeschrieben. Das Command kann aber auch mangels Deckung des Quellkontos abgelehnt werden. In diesem Fall ändert sich der Systemzustand nicht.

So gesehen wäre es auch möglich, statt Event Sourcing ein Command Sourcing zu implementieren. Hier würde statt einer Historie stattgefundener Ereignisse eine Historie von Commands, also von Aufträgen, gespeichert. Auch hier wäre eine gewisse Nachvollziehbarkeit gegeben, die Umsetzung würde sich aber ungleich schwieriger gestalten. Der Grund dafür ist, dass dem Command an sich nicht anzusehen ist, ob es erfolgreich ausgeführt oder abgelehnt wurde. Außerdem sind Commands im Gegensatz zu Events nicht idempotent. Ein zweimal ausgeführter Überweisungsauftrag führt gegebenenfalls zur Überweisung des doppelten Geldbetrags, während ein Ereignis dokumentiert, dass genau ein Überweisungsauftrag erfolgreich ausgeführt wurde.

Events repräsentieren im Domain-Driven Design Ereignisse, die bereits geschehen sind. Ein Event kann das Ergebnis der Verarbeitung eines Command sein und repräsentiert dann eine Tatsache, die nicht mehr rückgängig gemacht werden kann. Events sind damit von Natur aus unveränderlich. Das Verständnis eines Events im Kontext von DDD entspricht so ziemlich dem Konzept, das auch im Event Sourcing genutzt wird.

Das dritte Konzept aus dem taktischen Domain-Driven Design, das auch im Kontext von Event Sourcing hilfreich sein kann, ist das Aggregat. Es besteht aus einer Aggregatwurzel (Aggregate Root) und einer oder mehreren zugehörigen Entitäten (im Sinne von DDD, nicht zwingend JPA o. Ä.) und Value-Objekten. Die Aggregatwurzel ist eine Entität, über die der Zugang zum Aggregat von außen möglich ist. Sie verfügt über eine global eindeutige ID (Primärschlüssel), mit der das Aggregat angesprochen werden kann. Ein Aggregat hat eine klar definierte Grenze (Boundary). Diese legt fest, welche Objekte zum Aggregat gehören und wie sie interagieren können. Innerhalb des Aggregats müssen alle Änderungen in einer einzigen Transaktion durchgeführt werden, um die Konsistenz des Aggregatzustands zu gewährleisten.

Sowohl Commands als auch Events beziehen sich typischerweise jeweils auf genau ein Aggregat. Die erfolgreiche Verarbeitung eines Command führt zu einer Zustandsänderung des Aggregats und erzeugt ein Event, das diese Änderung repräsentiert. Konzeptionell ist es dabei ohne Bedeutung, ob zunächst die Zustandsänderung des Aggregats erfolgt oder zunächst das Event erzeugt wird. In Architekturen, die den Event-Sourcing-Ansatz verfolgen, wird der Zustand eines Aggregats allerdings stets durch die Anwendung eines Events geändert. Das bedeutet, dass das Event, das eine Zustandsänderung repräsentiert, vor der tatsächlichen Änderung des Zustands des Aggregats generiert und später genutzt wird, um den aktuellen Aggregatzustand zu ermitteln. Der Ablauf ist wie folgt:

- Ein Command, das eine spezifische Aktion oder Änderung anfordert, wird empfangen.
- Die Geschäftslogik des Aggregats prüft, ob die durch das Command angeforderte Aktion oder Änderung zulässig ist, basierend auf dem aktuellen Zustand des Aggregats und den Geschäftsregeln.
- Wenn die Aktion zulässig ist, erzeugt das Aggregat ein Event, das diese Zustandsänderung repräsentiert. Dieses Event beschreibt, was geschehen ist, doch der Zustand des Aggregats wurde zu diesem Zeitpunkt noch nicht geändert.
- Das generierte Event wird nun auf das Aggregat angewendet, um den Zustand entsprechend zu aktualisieren. Die Logik zur Handhabung des Events weiß, wie der Zustand des Aggregats basierend auf den Informationen im Event geändert wird.
- Das Event wird persistiert. So wird die dauerhafte Aufzeichnung der Zustandsänderung sichergestellt. Anschließend kann das Event an andere Teile des Systems oder externe Services weitergeleitet (publiziert)

werden, um weitere Aktionen auszulösen, die auf dieser Zustandsänderung basieren.

Der Schlüssel zum Verständnis dieses Prozesses ist, dass das Event sowohl die Absicht zur Änderung (basierend auf dem Command) als auch die Aufzeichnung der tatsächlich erfolgten Änderung darstellt. Die Reihenfolge – erst das Event, dann die Zustandsänderung – ermöglicht es, den Zustand des Aggregats ausschließlich durch die Anwendung von Events zu rekonstruieren. Das ist das zentrale Prinzip von Event Sourcing.

Um die Konsistenz und Integrität eines Aggregats zu wahren, werden die Commands, die sich auf ein bestimmtes Aggregat beziehen, stets sequenziell und transaktional verarbeitet. Das bedeutet, dass zu jedem Zeitpunkt für jedes Aggregat (also beispielsweise für ein bestimmtes Konto) höchstens ein Command bearbeitet wird. Dieses Vorgehen stellt sicher, dass die Verarbeitung des Command mit Validierung und Anwendung der Geschäftsregeln, das Erzeugen der Events und die zugehörige Zustandsänderung des Aggregats Hand in Hand gehen und innerhalb derselben Transaktion abgeschlossen werden. Es dient nicht nur der Vermeidung von Dateninkonsistenzen, indem es Race Conditions und gleichzeitige Zugriffe verhindert, sondern gewährleistet auch die vollständige Rückverfolgbarkeit und Nachvollziehbarkeit von Änderungen. Durch die Einbettung aller drei Schritte in eine einzige Transaktion wird sichergestellt, dass entweder alle Operationen erfolgreich durchgeführt oder im Falle eines Fehlers komplett zurückgerollt werden, sodass sich das System stets in einem konsistenten Zustand befindet.

Wie werden Events identifiziert?

Wir haben bisher gelernt, dass Events Ereignisse repräsentieren, die in der Vergangenheit stattgefunden haben. Wir haben gesehen, welche Bedeutung sie in einer Event-Sourcing-Architektur für die Persistenz des Systemzustands und die Nachvollziehbarkeit haben und wie sie im Kontext von Domain-Driven Design in Beziehung zu Commands und Aggregaten stehen. Doch wie kommen wir nun zu diesen Events, wie lassen sie sich identifizieren? Am Anfang kann eine gründliche Analyse der Geschäftsprozesse und -regeln stehen. Hier geht es darum, zu verstehen, welche Aktionen oder Bedingungen signifikante Änderungen innerhalb der Geschäftsdomäne auslösen. Events spiegeln oft diese Änderungen wider, wie z. B. „Bestellung aufgegeben“, „Zahlung erhalten“, „Produkt versendet“.

Im Umfeld von DDD ist das Workshopformat „Event Storming“ entstanden, das sich ebenfalls als Einstieg in die Identifikation von Events eignet. Es bringt Menschen aus verschiedenen Bereichen (wie Softwareentwicklung, Geschäftsanalyse, Produktmanagement und gegebenenfalls weitere Stakeholder) zusammen und dient dazu, in kurzer Zeit ein tiefes Verständnis der Geschäftsprozesse und Anforderungen zu erlangen. Eine komplexe Geschäftsdomäne wird dabei durch die Identifizierung von Schlüsselereignissen (also von Events) exploriert, visualisiert und modelliert.

Weitere Möglichkeiten, Events zu ermitteln, bestehen darin, Zustandsänderungen in den Entitäten oder Aggregaten der Domäne zu identifizieren. Jede dieser Zustandsänderungen kann potenziell ein Event darstellen. Daneben kann man sich von Benutzerinteraktionen oder Interaktionen mit anderen Systemen nähern. Jede Benutzeraktion, die eine signifikante Änderung im System verursacht, kann ein Event generieren. Ebenso können Informationen, die von anderen Systemen empfangen werden, ein Event darstellen. Schließlich können Events auch durch das Verstreichen von Zeit oder durch geplante Vorgänge ausgelöst werden, wie z. B. „Abonnement erneuert“ oder „Erinnerung versendet“.

Um das etwas konkreter zu machen, betrachten wir ein Beispiel. In unserer Beispieldomäne geht es darum, dass Konferenzbesucher an einem Spiel teilnehmen können, um andere Besucher kennenzulernen und sich mit ihnen auszutauschen. Während des Spiels erhalten die Teilnehmer Punkte für stattgefundene Interaktionen. Um an dem Spiel teilzunehmen, scannt ein Teilnehmer zunächst seinen eigenen Badge und wählt einen Nickname. Im Anschluss daran kann er die Badges anderer Konferenzteilnehmer scannen, mit denen er sich ausgetauscht hat. Ein Regelwerk entscheidet auf Basis der gescannten Badges darüber, wie viele Punkte der Nutzer bekommt. Für die Zukunft sollte das Spiel erweiterbar sein, sodass z. B. Interessen hinterlegt werden können und diese in die Punkteberechnung einfließen oder ein Feedback abgegeben werden kann. Weiterhin ist es denkbar, verschiedene Arten von Gamification in die Anwendung zu integrieren, um das Erlebnis spannender zu machen und die Nutzer zu mehr Interaktionen zu animieren. Es ist davon auszugehen, dass es Lastspitzen gibt, in denen besonders viele Interaktionen stattfinden, die verarbeitet werden müssen, aber zum Beispiel während der Vorträge deutlich weniger Interaktionen erfolgen. Darüber hinaus sollte die Anwendung damit umgehen können, dass das Frontend (Browseranwendung) nicht dauerhaft mit dem Internet verbunden ist, in dieser Zeit aber Interaktionen stattfinden, die dann zeitverzögert an den Server gemeldet werden.

Für diese Anwendung eignet sich eine Event-gesteuerte Architektur sehr gut. Ereignisse wie „Badge gescannt“ können zunächst lokal im Frontend festgehalten und bei Netzwerkverfügbarkeit an den Server übertragen werden. Im Server findet eine regelbasierte Auswertung dieser Ereignisse statt, die zu neuen Ereignissen wie „Punkte erhalten“ führt. Diese werden asynchron an das jeweilige Frontend übertragen und aktualisieren dort den Zustand oder lösen ihrerseits weitere Ereignisse aus. Wie kommen wir für unsere Anwendung nun zu einer Liste passender Events? Oft ist es hilfreich, zwischen Events zu unterscheiden, die direkt von einem Benutzer ausgelöst werden („Benutzer ruft im Call-Center an“) oder eine andere, externe Ursache haben („Pegelstand ist 8,30 m“) und davon abgeleiteten Events, die in unse-

rer Anwendung entstehen („Pegelstand erreicht Hochwassermarke 2“, „Rhein für Schifffahrt gesperrt“).

Für unsere Beispielanwendung ergeben sich aus der fachlichen Analyse folgende Events, die durch eine Benutzeraktion im Frontend ausgelöst werden:

- „Eigenen Badge gescannt“ (Log-in) mit der Nummer des gescanntes Badges
- „Nickname gewählt“ mit der Nummer des eigenen Badges und dem gewählten Nickname
- „Badge von anderem Konferenzteilnehmer gescannt“ mit den Nummern des eigenen und des gescannten Badges des anderen Teilnehmers

Zu diesen Basis-Events gibt es jeweils passende Commands, über die unser System von den gewünschten Aktionen erfährt („Badge scannen“, „Nickname ändern“). Neben diesen unmittelbaren Events leitet das System beim Verarbeiten der Commands über Regeln weitere Events ab:

- „Neuer Spieler nimmt teil“, wenn festgestellt wurde, dass ein Teilnehmer zum ersten Mal seinen Badge zur Anmeldung gescannt hat
- „Punkte erhalten“, wenn die Engine feststellt, dass sich ein Teilnehmer durch das Scannen eines anderen Badges Punkte verdient hat
- „Top-Ten-Liste aktualisiert“, wenn sich durch die Vergabe von Punkten die Zusammensetzung der Top-Ten-Liste geändert oder einer der Top-Ten-Teilnehmer einen neuen Punktestand erreicht hat

Neben den spezifischen Attributen, die ein Event auf Basis seiner Fachlichkeit hat – wie die Nummer des gescannten Badges beim Event „Badge von anderem

Listing 1: Beispiel für allgemeine Attribute eines Events

```
@Value.Immutable
public abstract class EventMessage<E extends DomainEvent<T>, T>
{
  public abstract String getEventId();

  public abstract String getEventType();

  public abstract String getAggregateId();

  public abstract Class<T> getAggregateType();

  public abstract Instant getTimestamp();

  public abstract Long getVersion();

  public abstract E getPayload();
}
```

Konferenzteilnehmer gescannt“ –, haben Events typischerweise eine Menge allgemeiner Attribute. Dazu zählen:

- ein eindeutiger Identifier
- der Zeitpunkt, zu dem das Event aufgetreten bzw. verarbeitet worden ist
- der Auslöser des Events – dies kann ein Benutzer oder ein externes System sein oder auch ein anderes Event; häufig wird auch eine Referenz auf das Command gespeichert, dessen Verarbeitung das Event erzeugt hat

Falls sich das Event auf ein Aggregat bezieht bzw. durch Änderung des Aggregats ausgelöst wurde, sind folgende zusätzliche Attribute hilfreich:

- Typ des Aggregats (z. B. dessen Klassenname)
- ID des Aggregats
- eine Sequenznummer oder Version, die innerhalb des Aggregats eindeutig ist und die Reihenfolge der Events festlegt

Die allgemeinen Attribute lassen sich gut in einer eigenen Java-Klasse abbilden, die entweder über Komposition oder Vererbung mit den konkreten Attributen verbunden wird. Ein Beispiel dazu findet sich in Listing 1.

Das jeweils konkrete Event wird dann in der Payload abgelegt und implementiert das *DomainEvent*-Interface. Ein *NicknameChanged* Event enthält dann nur noch die spezifischen Attribute für das Event und eine *apply*-Methode, die die Zustandsänderung auf das Aggregat (*Participant*) anwendet (Listing 2).

Umsetzung mit Spring Boot

Wollen wir die Konzepte mit Spring Boot umsetzen, ist das mit einfachen Bordmitteln möglich. Neben der bereits gezeigten Implementierung eines Events lassen sich auch Commands und ihre zugehörigen Handler als Java-Klassen umsetzen. Im Gegensatz zum Event, das sich seiteneffektfrei auf ein Aggregat anwenden lassen muss, kann es bei der Ausführung von Commands nötig sein, auf weitere Services zuzugreifen. Deshalb implementieren wir den *CommandHandler* als eigene Spring Bean und können so Dependency Injection nutzen.

Ein Command ist ein einfaches, unveränderbares Value-Objekt. Der *CommandHandler* ist dafür verantwortlich, die nötigen Geschäftsregeln zu prüfen und ggf. eine Liste von Events zu erzeugen, die die durch das Command ausgelöste Zustandsänderung des betroffenen Aggregats beschreiben. Entsprechend erhält die *decide*-Methode des *CommandHandlers* den Command

Listing 2: Interface DomainEvent mit konkreter Ausprägung für das Event NicknameChanged

```
public interface DomainEvent<T>
{
  void apply(T aggregateRoot);
}

@Value.Immutable
public abstract class NicknameChangedEvent implements DomainEvent<Participant>
{
  abstract String getNickname();

  @Override
  public void apply(Participant participant)
  {
    participant.setNickname(getNickname());
  }
}
```

Listing 3: SetNicknameCommand und SetNicknameCommandHandler

```
public interface Command<T>
{
}

@Value.Immutable
public abstract class SetNicknameCommand implements
Command<Participant>
{
  public abstract String getNickname();
}

public interface CommandHandler<C extends Command<T>, T>
{
  List<? extends DomainEvent<T>> decide(C command, T aggregateRoot);
}

@Service
public class SetNicknameCommandHandler implements CommandHandler<
                                    SetNicknameCommand, Participant>
{
  @Override
  public List<? extends DomainEvent<Participant>>
decide(SetNicknameCommand command, Participant participant)
  {
    if (command.getNickname() != null && !command.getNickname().
                                        equals(participant.getNickname()))
    {
      return List.of(ImmutableNicknameChangedEvent.builder()
        .nickname(command.getNickname())
        .build());
    }
    else
    {
      return List.of();
    }
  }
}
```

und den aktuellen Zustand des Aggregats als Parameter und liefert eine Liste von Events als Ergebnis zurück. Listing 3 zeigt das am einfachen Beispiel des Setzens eines Nicknames durch einen Teilnehmer.

Listing 4: Laden des aktuellen Zustands eines Aggregats

```
@Service
public class EventManager
{
  private final EventStore eventStore;
  private final ApplicationEventPublisher eventPublisher;

  public EventManager(EventStore eventStore, ApplicationEventPublisher
                                                        eventPublisher)
  {
    this.eventStore = eventStore;
    this.eventPublisher = eventPublisher;
  }

  public <T> T loadAggregate(Class<T> aggregateType, String aggregateId)
{
    return loadAggregateWithVersion(aggregateType, aggregateId).
                                                          aggregate();
  }

  @SuppressWarnings("unchecked")
  public <T> AggregateWithVersion<T> loadAggregateWithVersion(Class<T>
                                      aggregateType, String aggregateId)
  {
    final var eventMessages = eventStore.loadEventMessages(aggregateType,
                                                          aggregateId);
    final T aggregate;
    try
    {
      aggregate = aggregateType.getDeclaredConstructor(String.class).
                                                  newInstance(aggregateId);
    }
    catch (ReflectiveOperationException e)
    {
      throw new IllegalStateException("Unable to create initial state", e);
    }

    eventMessages.stream()
      .map(eventMessage -> (DomainEvent<T>) eventMessage.getPayload())
      .forEach(domainEvent -> domainEvent.apply(aggregate));
    final var version = eventMessages.isEmpty() ? 0L : eventMessages.
                                                  last().getVersion();
    return new AggregateWithVersion<>(aggregate, version);
  }

  public void saveEventMessages(List<? extends EventMessage<? extends
                                      DomainEvent<?>, ?>> eventMessages)
  {
    eventMessages.forEach(eventStore::saveEventMessage);
    eventMessages.forEach(eventPublisher::publishEvent);
  }
}
```

Wie erhalten wir nun den aktuellen Zustand eines Aggregats, wie wir ihn zum Beispiel für den Aufruf eines *CommandHandler* benötigen? Ganz einfach durch sequenzielles Anwenden der Events auf ein Aggregat. Wir laden die Events zu einem Aggregat aus einem *EventStore* (z. B. einer Datenbank) und wenden sie nacheinander an. Um das erste Event anwenden zu können, benötigen wir einen Initialzustand. In unserem Beispiel erzeugen wir diesen einfach durch Aufrufen eines Konstruktors des Aggregats, dem wir die ID mitgeben. Listing 4 zeigt einen einfachen *EventManager*, der genau das erledigt.

Damit gerüstet, können wir einen einfachen *CommandManager* implementieren, der den *EventManager* nutzt, um den aktuellen Zustand eines Aggregats und dessen aktuelle Version zu ermitteln und dann den passenden *CommandHandler* ermittelt und aufruft. Schließlich schreibt er die bei der Ausführung des Commands erzeugten Events über den *EventManager* in den *EventStore*. Beim Erzeugen der Events wird die Version jeweils um eins erhöht. Dadurch wird nicht nur die Reihenfolge der Events festgeschrieben, sondern es lässt sich auch ein einfaches optimistisches Locking implementieren, wie wir im folgenden Abschnitt sehen werden. Listing 5 zeigt unseren einfachen *CommandManager*.

Persistenz

Ein zentrales Element der Persistenz in einem System, das auf Event Sourcing basiert, ist der Event Store, in dem die Events dauerhaft gespeichert werden. Wird für die Persistenz eine relationale Datenbank verwendet, besteht die Herausforderung darin, eine Struktur zu schaffen, die es ermöglicht, Events effizient zu speichern, abzurufen und zu verarbeiten. Wenn man bereits relationale Datenbanken nutzt, kann man sie auch für den Einstieg in Event Sourcing verwenden. Dabei kann man zunächst eine Tabelle erstellen, um die Events zu speichern. Sie sollte so gestaltet sein, dass sie alle notwendigen Informationen über jedes Event aufnehmen kann. Für die typischen Spalten in einer Event-Tabelle helfen uns die eben identifizierten allgemeinen Attribute von Events. Die Tabelle könnte folgende Spalten enthalten:

- *EventID:* Eindeutiger Identifier für jedes Event und Primärschlüssel der Tabelle
- *EventType:* Typ des Events (z. B. *NicknameGewählt*, *PunkteErhalten*)
- *AggregateID:* Identifier des Aggregats, auf das sich das Event bezieht; ermöglicht es, alle Events zu einem bestimmten Aggregat abzufragen
- *AggregateType:* Typ des Aggregats; alternativ kann auch je Aggregat eine eigene Tabelle verwendet werden, in diesem Fall wird der Typ des Aggregats in den Tabellennamen codiert
- *Timestamp:* Zeitpunkt, zu dem das Event aufgetreten ist
- *Version:* Versionsnummer bezogen auf das Aggregat, die mit jedem Event inkrementiert wird; hilft,

die Reihenfolge der Events sicherzustellen und kann in Verbindung mit einem Unique Constraint auf die Spalten *AggregateType*, *AggregateID* und *Version* auch für optimistisches Locking verwendet werden; bei einer parallelen Verarbeitung von Commands für dasselbe Aggregat kommt es zu einer Verletzung des Constraints und eine der beiden Transaktionen wird zurückgerollt

- *Payload:* Die eigentlichen Daten des Events; dabei handelt es sich oft um einen JSON- oder XML-String, der die Zustandsänderung beschreibt; in PostgreSQL würde man für dieses Feld den JSONB-Datentyp wähle, auch die meisten anderen relationalen Datenbanken unterstützen JSON-Daten ebenso wie der SQL-Standard

Wenn ein neues Event generiert wird, fügt man einen Eintrag in die Event-Tabelle ein. Der Payload-Teil des Events sollte alle Informationen enthalten, die notwendig sind, um den Zustand des Aggregats aus den Events

Listing 5: Ein einfacher CommandManager

```
@Service
public class CommandManager
{
  private final EventManager eventManager;
  private final ApplicationContext context;
  private final Clock clock;

  public CommandManager(EventManager eventManager, ApplicationContext
                                                        context, Clock clock)
  {
    this.eventManager = eventManager;
    this.context = context;
    this.clock = clock;
  }

  public <C extends Command<T>, T> void handle(CommandMessage<C,
                                                    T> commandMessage)
  {
    CommandHandler<C, T> commandHandler = getCommandHandler
                                                    (commandMessage);
    final var aggregateType = (Class<T>) commandMessage.
getAggregateType();
    final var aggregateId = commandMessage.getAggregateId();
    final var aggregateWithVersion = eventManager.loadAggregateWithVersion
                                        (aggregateType, aggregateId);
    final var events = commandHandler.decide(commandMessage.
getPayload(), aggregateWithVersion.aggregate());
    final var eventMessages = new ArrayList<ImmutableEventMessage
                                                <DomainEvent<T>, T>>();
    for (int i = 0; i < events.size(); i++)
    {
      final var event = events.get(i);
      eventMessages.add(ImmutableEventMessage.<DomainEvent<T>,
                                                        T>builder()
        .aggregateType(aggregateType)
        .aggregateId(aggregateId)
        .eventType(event.getClass().getSimpleName())
        .eventId(UUID.randomUUID().toString())
        .timestamp(clock.instant())
        .version(aggregateWithVersion.version() + i + 1)
        .payload(event)
        .build());
    }
    eventManager.saveEventMessages(eventMessages);
  }

  @SuppressWarnings("unchecked")
  public <C extends Command<T>, T> CommandHandler<C, T>
          getCommandHandler(CommandMessage<C, T> commandMessage)
  {
    return (CommandHandler<C, T>) getCommandHandler(
      ResolvableType.forClass(commandMessage.getPayload().getClass()),
      ResolvableType.forClass(commandMessage.getAggregateType()),
      null
    );
  }

  public Object getCommandHandler(ResolvableType commandClass,
      ResolvableType aggregateRootClass, BeansException originalException)
  {
    ResolvableType type = ResolvableType.forClassWithGenerics(
      CommandHandler.class,
      commandClass,
      aggregateRootClass
    );

    try
    {
      return context.getBeanProvider(type).getObject();
    }
    catch (BeansException e)
    {
      final var superType = commandClass.getSuperType();
      if (!superType.equalsType(NONE))
      {
        return getCommandHandler(superType, aggregateRootClass, e);
      }
      else
      {
        throw originalException == null ? e : originalException;
      }
    }
  }
}
```

zu rekonstruieren. Da die Payload flexibel ist, kann man komplexe Datenstrukturen speichern, ohne das Schema der Datenbank ändern zu müssen (Tabelle 1).

Um den aktuellen Zustand eines Aggregats zu rekonstruieren, selektiert man alle Events für das betreffende Aggregat (über *AggregateType* und *AggregateID*), sortiert nach *Timestamp* oder *Version* und wendet sie in der Reihenfolge ihres Auftretens an. Dieser Prozess stellt den Zustand des Aggregats, basierend auf seiner Event-Historie wieder her. Bei Aggregaten mit vielen Events kann die Rekonstruktion des Zustands zeitaufwendig werden. In solchen Fällen kann man in regelmäßigen Abständen Snapshots des Aggregatzustands speichern. Ein Snapshot speichert den vollständigen Zustand eines Aggregats zu einem bestimmten Zeitpunkt. Um den aktuellen Zustand wiederherzustellen, findet man den neuesten Snapshot und wendet nur die Events an, die nach diesem Snapshot aufgetreten sind. Statt Snapshots zu verwenden, kann man bereits bei der fachlichen Modellierung darauf achten, dass zu einem Aggregat nicht zu viele Events entstehen. So kann man beispielsweise eine zeitliche Komponente einfließen lassen, sodass ein Aggregat nur noch für einen bestimmten Zeitraum genutzt wird. Bei der Verwaltung von Konten und Buchungen entspricht das dem regelmäßigen Erzeugen eines Saldos, wobei der Saldo der Snapshot ist. Um den aktuell verfügbaren Betrag auf einem Konto zu ermitteln, wird der letzte Saldo abgerufen und es werden diejenigen Vorgänge dazugerechnet, die noch nicht gebucht worden sind, d. h. die noch nicht Teil des Saldos sind. Berücksichtigt man die Reduktion der Event-Menge pro Aggregat bereits bei der Modellierung, gelangt man zu einem Modell, indem Buchungen pro Monat oder pro Jahr gespeichert und zum Abschluss der Periode abgerechnet werden.

Die Art und Weise, wie Snapshots in einer relationalen Datenbank gespeichert werden, ist abhängig von den konkreten Queryanforderungen. Entweder werden die Snapshots analog zu den Events in eher generischen Tabellen als JSON-Payload gespeichert oder die Aggregatstruktur wird ausmodelliert und ggf. auf mehrere Tabellen verteilt. Das Speichern in generischen Tabellen bietet sich dann an, wenn auf die Aggregate vorwiegend über den Identifier zugegriffen wird und nur geringe Anforderungen an eine Suche bestehen. Die Ausmodellierung eines komplexen Modells bedeutet dagegen mehr Aufwand, ermöglicht aber auch komplexe Suchen über den aktuellen Systemzustand. Hier bietet es sich an, Event Sourcing mit anderen Methoden wie CQRS zu kombinieren.

Neben den weit verbreiteten relationalen Datenbanken eignen sich auch NoSQL-Datenbanken gut als Datenspeicher für Event Sourcing. In Document Stores wie MongoDB oder Couchbase könnte jedes Event als separates Dokument in der Datenbank gespeichert werden. Das Fehlen eines festen Schemas bei Document Stores erweist sich hier als gutes Match. In Wide Column Stores wie Apache Cassandra oder Google Bigtable können Events in Tabellen mit dynamischen Spalten gespeichert werden. Jede Zeile in einer solchen Tabelle entspricht dabei einem Aggregat und jede Spalte repräsentiert ein Event. Das ermöglicht eine effiziente Abfrage von Events pro Aggregat. Wide Column Stores bieten oft Unterstützung für Zeitstempel und können Events automatisch, basierend auf dem Zeitpunkt ihres Eintreffens, sortieren. Das erleichtert die effiziente Rekonstruktion des Aggregatzustands.

Weitergabe von Events

Nachdem ein Event persistiert wurde und so die dauerhafte Aufzeichnung der Zustandsänderung sichergestellt ist, kann das Event an andere Komponenten des Systems (z. B. an ein Frontend) oder an externe Systeme weitergeleitet (publiziert) werden, um dort gegebenenfalls weitere Aktionen auszulösen, die auf dieser Zustandsänderung basieren. Soll die Zustandsänderung innerhalb der Anwendung selbst weitergegeben werden, so bieten sich leichtgewichtige Mechanismen wie die Nutzung von *ApplicationEvents* im Spring-Framework an. Auf diese Weise erreicht man eine lose Kopplung der einzelnen Komponenten ohne zusätzliche Komplexität, wie sie Message Broker oder Event-Streaming-Plattformen mitbringen. Die Kommunikation erfolgt lokal durch direkten Methodenaufruf entweder

synchron im selben Thread oder asynchron. Das Event wird durch ein einfaches Java-Objekt repräsentiert, das den Methoden als Parameter übergeben wird. Sender und Empfänger kennen sich nicht, sondern jeweils nur die Java-Klasse des Events.

Sollen Events über Netzwerkgrenzen hinweg an externe Komponenten weitergeleitet werden, so werden klassisch Message Broker wie RabbitMQ und Event-Streaming-Plattformen wie Apache Kafka genutzt. Eine geschätzte Eigenschaft ist dabei die Unterstützung von Publish/Subscribe. Dabei werden die Events so veröffentlicht, dass sie von mehreren externen Systemen konsumiert werden können, ohne dass der Sender explizit die Empfänger kennen muss. Die Verantwortung wird dabei aufgeteilt, in den Sender, der die Events veröffentlicht, und die Empfänger, die den entsprechenden Eventstrom abonnieren und die Events empfangen und verarbeiten. Neben dieser Gemeinsamkeit gibt es jedoch zahlreiche Unterschiede zwischen Message Brokers und Event-Streaming-Plattformen sowie den jeweiligen Implementierungen, die bei der konkreten Auswahl berücksichtigt werden sollten. Eine Diskussion an dieser Stelle würde den Rahmen des Artikels allerdings sprengen.

Eine weitere Option ist MQTT (MQ Telemetry Transport), ein leichtgewichtiges, Publish/Subscribe-basiertes Messaging-Protokoll, das speziell für die Kommunikation in Netzwerken mit geringer Bandbreite und unzuverlässigen Verbindungen entwickelt wurde. Es eignet sich sehr gut für die Weiterleitung von Events, insbesondere in verteilten Systemen und Internet-of-Things-(IoT-)Anwendungen mit Echtzeit-Kommunikationsanforderungen. Als Besonderheit von MQTT unterstützen Topics eine hierarchische Struktur, die eine flexible Organisation und Filterung von Nachrichten ermöglicht. Viele moderne MQTT-Broker unterstützen MQTT über WebSockets, eine Technologie, die eine bidirektionale Kommunikation zwischen Webbrowsern und Servern ermöglicht. Das erlaubt Webanwendungen, die im Browser laufen, eine Verbindung zum MQTT-Broker herzustellen und als MQTT-Client zu fungieren, wodurch Events in Echtzeit an das Frontend weitergeleitet werden können.

Unabhängig vom verwendeten Transport sollten Events nur dann publiziert werden, wenn sie auch im Event Store gespeichert werden konnten, und die Zustandsänderung so tatsächlich festgeschrieben wurde. Umgekehrt muss sichergestellt werden, dass auch zu jeder Zustandsänderung ein Event publiziert wird. Anders formuliert: Idealerweise erfolgen die Persistierung des Events und das Publizieren innerhalb derselben Transaktion. Technisch ist das durch die unterschiedlichen Technologien, die zum Speichern und Versenden eingesetzt werden, oft schwierig zu realisieren, und verteilte Transaktionen bringen ihre eigenen Herausforderungen mit. Als Lösung bietet sich hier häufig das Outbox-Pattern an, mit dem die Konsistenz zwischen einer Datenbank und einem Nachrichtensystem (wie einem Message Broker oder Event Stream) sichergestellt werden kann. Bei diesem Muster wird eine Datenbanktabelle verwendet, die als Outbox fungiert. Werden Events persistiert, so wird gleichzeitig ein Eintrag in die Outbox-Tabelle geschrieben, der das Event repräsentiert, das publiziert werden soll. Ein separater Thread, Prozess oder Dienst, oft als Outbox Relay oder Event Publisher bezeichnet, überwacht kontinuierlich die Outbox-Tabelle auf neue Einträge. Wenn ein neuer Eintrag erkannt wird, liest ihn der Prozess, veröffentlicht das entsprechende Event an das Nachrichtensystem und markiert den Eintrag in der Outbox-Tabelle als verarbeitet oder löscht ihn. Statt einer separaten Tabelle kann in einigen Fällen auch der Event Store selbst als Outbox verwendet werden. Einige Event Stores bieten die Weiterleitung von Events nativ an.

EventID	EventType	AggregateID	AggregateType	Timestamp	Version	Payload
fbec934b	NicknameChanged	1234	Participant	2024-04-03T10:15:30Z	1	{ "nickname" : "Alfred" }
f61ce23f	NicknameChanged	1234	Participant	2024-04-03T10:16:30Z	2	{ "nickname" : "Barnie" }
36b39859	BadgeScanned	1234	Participant	2024-04-03T10:16:40Z	3	{ "image" : "AA==", "clientScan" : "MYBADGE", "serverScan" : "MYBADGE" }
5fc8c088	PointsReceived	1234	Participant	2024-04-03T10:16:40Z	4	{ "pointsReceived": 10 }
e75b031c	BadgeScanFailed	1234	Participant	2024-04-03T10:18:30Z	5	{ "image" : "AB==", "clientScan" : "invalid badge", "serverScan" : "MYBADGE"}

Tabelle 1: Beispielhafter Zustand der Event-Tabelle nach dem Setzen des Nicknames und Scannen von Badges

Während beim Speichern von Events für das Event Sourcing nur die Informationen relevant sind, die die Zustandsänderung an sich beschreiben, kann es bei der Weiterleitung von Events für den Empfänger hilfreich sein, die Events mit zusätzlichen Informationen anzureichern. Allgemein bietet es sich an, zwischen der internen und externen Repräsentation von Events zu unterscheiden. Die externe Repräsentation von Events sollte als API verstanden werden, dem ähnliche Überlegungen zugrunde liegen wie dem Design von HTTP APIs. Auch hier helfen Werkzeuge aus dem Domain-Driven Design: Anticorruption Layer, hexagonale Architektur und die explizite Modellierung der Beziehungen zwischen Bounded Contexts im strategischen Domain-Driven Design.

Bezogen auf unsere Beispieldomäne reicht es bei einem Event „Punkte erhalten“ für die vollständige Beschreibung der Zustandsänderung aus, die Veränderung der Punktezahl zu speichern. Ein externer Empfänger ist dagegen vermutlich ebenfalls an dem neuen Punktestand des Teilnehmers interessiert. Hier bietet es sich an, die Repräsentation des Events anzupassen und entweder den vollständigen Zielzustand des betroffenen Aggregats oder einen relevanten Ausschnitt daraus mit dem Event zu publizieren. Auf diese Weise spart man sich einen zusätzlichen Leseaufruf der Empfänger beim Verarbeiten der Events. Je nach Anforderungen kann es sinnvoll sein, die externe Repräsentation von Events eher allgemein zu entwerfen, sodass sie von allen oder möglichst vielen Konsumenten nutzbar ist. Ähnlich wie beim Ansatz „Backend for Frontend“ für Web-APIs kann es aber auch nützlich sein, Events speziell für ein System aufzubereiten, sodass dieses sie möglichst effizient verarbeiten kann. Wichtig ist aber, in allen Fällen darauf zu achten, dass die externe Repräsentation von Events über die Zeit möglichst stabil bleibt und Änderungen im Vorfeld mit den jeweiligen Empfängern abgestimmt werden.

Ausblick

Der Einstieg in eine Event-gesteuerte Architektur markiert einen bedeutenden Wandel für Unternehmen, die von traditionellen, monolithischen und synchron arbeitenden Anwendungsstrukturen kommen. Während der vollständige Umstieg auf Event Sourcing eine umfangreiche Transformation darstellt, die viele Bereiche von der Entwicklung über den Betrieb bis hin zu Geschäftsprozessen beeinflusst, bietet ein schrittweises Vorgehen oft eine praktikable Alternative. Um mit Event-gesteuerten Mustern zu experimentieren und Erfahrungen zu sammeln, kann man damit beginnen, Event Notifications für spezifische Aktionen oder Zustandsänderungen in Datenbanken zu implementieren. Mit diesem Ansatz bleibt die Persistenz der Anwendung zunächst erhalten und wird nicht auf Event Sourcing umgestellt. CDC-Technologien wie die Open-Source-Lösung Debezium erleichtern den Übergang, indem sie Änderungen in Datenbanken in Echtzeit erfassen und als Events an Event-Streaming-Systeme oder Message Broker weiterleiten. Diese Technik ermöglicht es, die Einführung von Event-gesteuerten Mustern schrittweise vorzunehmen, ohne die bestehenden Anwendungen und Datenstrukturen grundlegend ändern zu müssen.

Die Nutzung von CDC zur Erleichterung des Übergangs bietet mehrere Vorteile. Sie ermöglicht eine Integration bestehender Systeme in die neue Architektur, reduziert das Risiko während der Umstellungsphase und fördert die Entkopplung von Services durch Event-basierte Kommunikation. Ausgehend davon kann CDC dazu beitragen, Event Sourcing selektiv in bestimmten Bereichen einer Anwendungslandschaft zu implementieren und so die nötige Erfahrung zu sammeln, um es später großflächiger auszurollen. Hat man auf der anderen Seite bereits erste Erfahrungen mit Event Sourcing und Event-gesteuerten Architekturen gesammelt und möchte diese weiter ausbauen, so kann es hilfreich sein, ein Framework auszuwählen, das die Implementierung erleichtert und für wiederkehrende Aufgaben einheitliche Lösungen bereitstellt. Ein Kandidat in dieser Kategorie ist Axon. Axon ist ein Framework für Java-basierte Anwendungen, die den Prinzipien von Domain-Driven Design, Command Query Responsibility Segregation (CQRS) und Event-gesteuerter Architektur folgen. Es unterstützt Event Sourcing als eine Option für die Persistenz und lässt sich gut mit Spring bzw. Spring Boot kombinieren.

Insgesamt bietet Event Sourcing einen interessanten Ansatz für die Persistenz von Anwendungen und erlaubt einen natürlichen Einstieg in Event-gesteuerte Architekturen. Es sollte vor allem dann in Erwägung gezogen werden, wenn hohe Anforderungen an Nachvollziehbarkeit und Auditierung gestellt werden, wie sie häufig in sicherheitskritischen Bereichen und im Finanzumfeld zu finden sind. Aber auch andere Bereiche wie die Verarbeitung von Sensordaten, IoT-Anwendungen, Air Traffic Control und Defense können von Event Sourcing profitieren. Wichtig ist dabei zu beachten, dass Event Sourcing neben den genannten Vorteilen eine nicht unerhebliche zusätzliche Komplexität mitbringt. Bei der Entscheidung lohnt es sich daher, sich von erfahrenen Experten beraten zu lassen.

Als Enterprise-Architekt bei der Postbank war **Stefan Reuter** verantwortlich für Multikanal-Integrationslösungen. Heute unterstützt er bei der Trion Unternehmen bei der Transformation ihrer IT-Landschaft, wenn es darum geht, zukunftssichere, kosteneffiziente und performante Lösungen zu schaffen. Als erfahrener Projektleiter und IT-Architekt verfügt er über umfassendes Wissen und praktische Erfahrung in der Gestaltung und Umsetzung anspruchsvoller IT-Projekte. Er ist seit 25 Jahren auf der Java-Plattform zu Hause und hat in dieser Zeit umfangreiche Erfahrungen gesammelt, die er als gefragter Experte gerne auf Konferenzen, in praxisorientierten Schulungen und im Rahmen von Beratungsaufträgen weitergibt. Kontaktieren Sie ihn gerne für weitergehende Unterstützung zu den behandelten Themen.

sr@trion.de **www.trion.de**