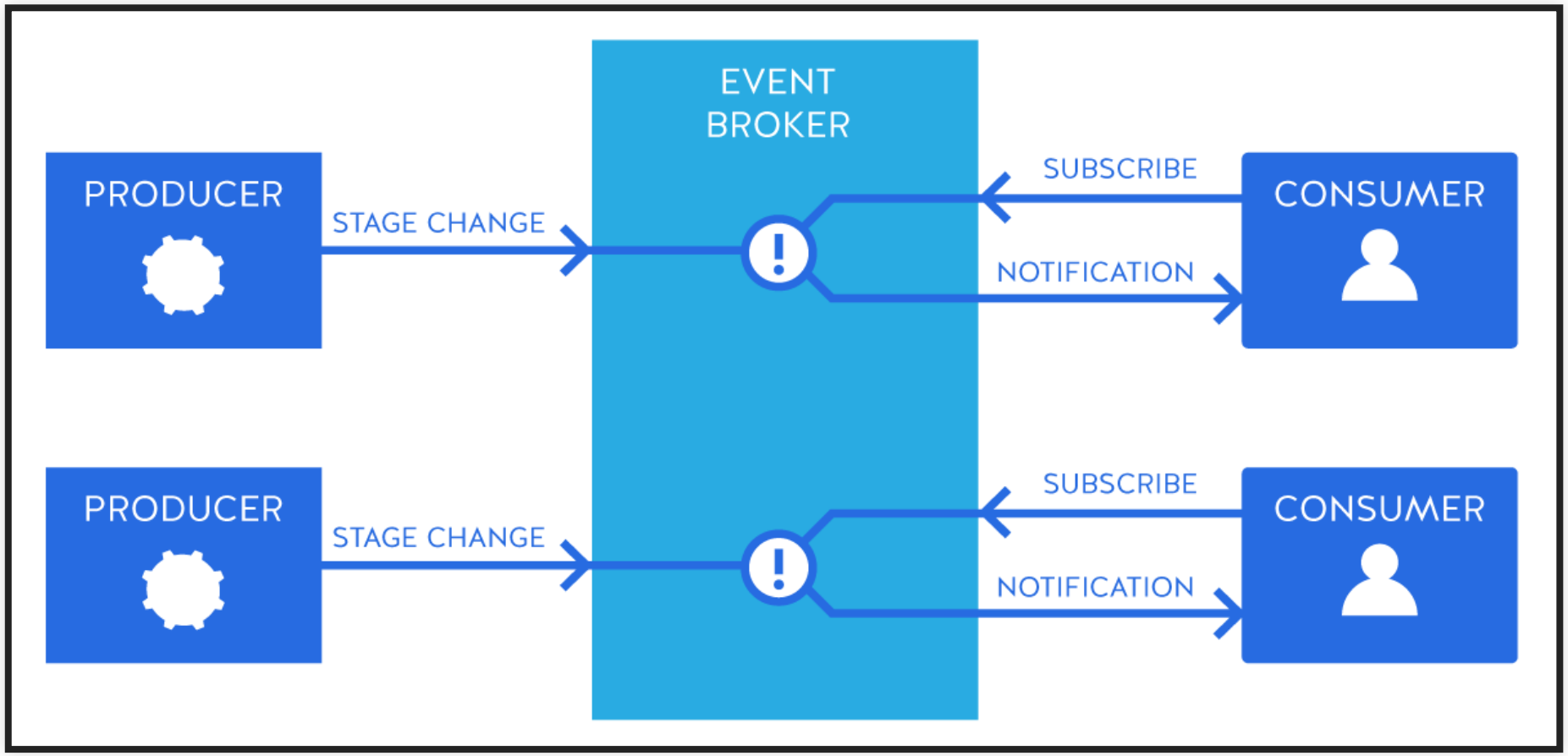
KAPITEL 2: VERTEILEN VON EREIGNISSEN UND NACHRICHTEN

LERNZIELE

- Point-to-Point und Publish / Subscribe Pattern erklären sowie mittels Diagrammen die Funktionsweise darstellen
- Verschiedene Formen der Nachrichtenzustellung unterscheiden und erklären
- Message- und Event-Broker differenzieren
- APIs und Protokolle wie JMS, MQTT und AMQP unterscheiden und (kurz) erklären
- Software-Lösungen wie Mosquitto, ActiveMQ und Kafka unterscheiden, für einen konkretes Szenario gezielt wählen und Client-Anwendungen realisieren

2.1 MOTIVATION

Architekturansätze wie MOM und EDA werden über eine zentrale Komponente realisiert, die Nachrichten bzw. Events verteilt.



Quelle: https://nordicapis.com/whats-the-difference-between-event-brokers-and-message-queues/

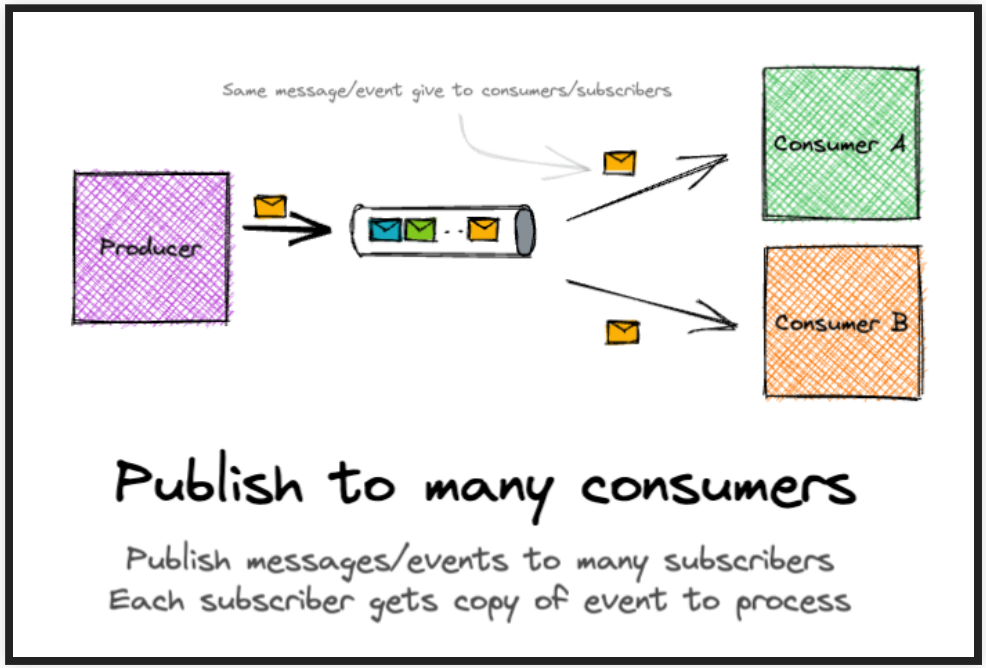
Inhalt der folgenden Kapitel

- **Messaging Models**: Wie werden Nachrichten ausgetauscht? Hierfür gibt es es verschiedene Ansätze, die bekanntesten sind Point-to-Point-Messaging und Publish / Subscribe
- **Message Delivery**: Wie werden Nachrichten versendet? Werden Nachrichten abgesendet, kann es zu Verlusten, Fehlern und Ausfällen kommen. Wie wichtig ist die Zustellung und Verarbeitung?
- **Message- vs. Event-Broker**: Häufig in einer Platform vereint, können sie grundlegend unterschieden werden
- **Protokolle**: API-basierte Ansätze wie JMS oder bekannte Kommunikationsprotokolle wie MQTT und AMQP stehen hier im Vordergrund
- **Mosquitto, ActiveMQ und Kafka**: Einstieg in typische Software-Lösungen sowie einige Beispiele

2.2 GRUNDLAGEN

PUBLISH / SUBSCRIBE

Grundlegendes Konzept ist häufig das Pub/Sub Pattern



Mit Pub/Sub-Kanälen können Sie Nachrichten an viele Verbraucher veröffentlichen, die an dieser Nachricht interessiert sind. Der Produzent veröffentlicht und die Konsumenten abonnieren.

Quelle: https://serverlessland.com/event-driven-architecture/visuals/publish-subscribe

- Mit diesem Muster kann der Produzent Nachrichten an viele nachgeschaltete Konsumenten veröffentlichen. Die Konsumenten erhalten ihre eigene Kopie der Nachricht zur Verarbeitung.
- Verbraucher kommen und gehen, und dem Produzenten ist es egal; dies Flexibilität, Verbraucher hinzuzufügen, wenn sich die Geschäftsanforderungen ändern
- Mit Pub/Sub werden Nachrichten/Ereignisse oft an nachgelagerte Verbraucher weitergegeben (anstatt dass Verbraucher Ereignisse aus einer Warteschlange/einem Kanal abhören und abrufen müssen).

Quelle: https://serverlessland.com/event-driven-architecture/visuals/publish-subscribe

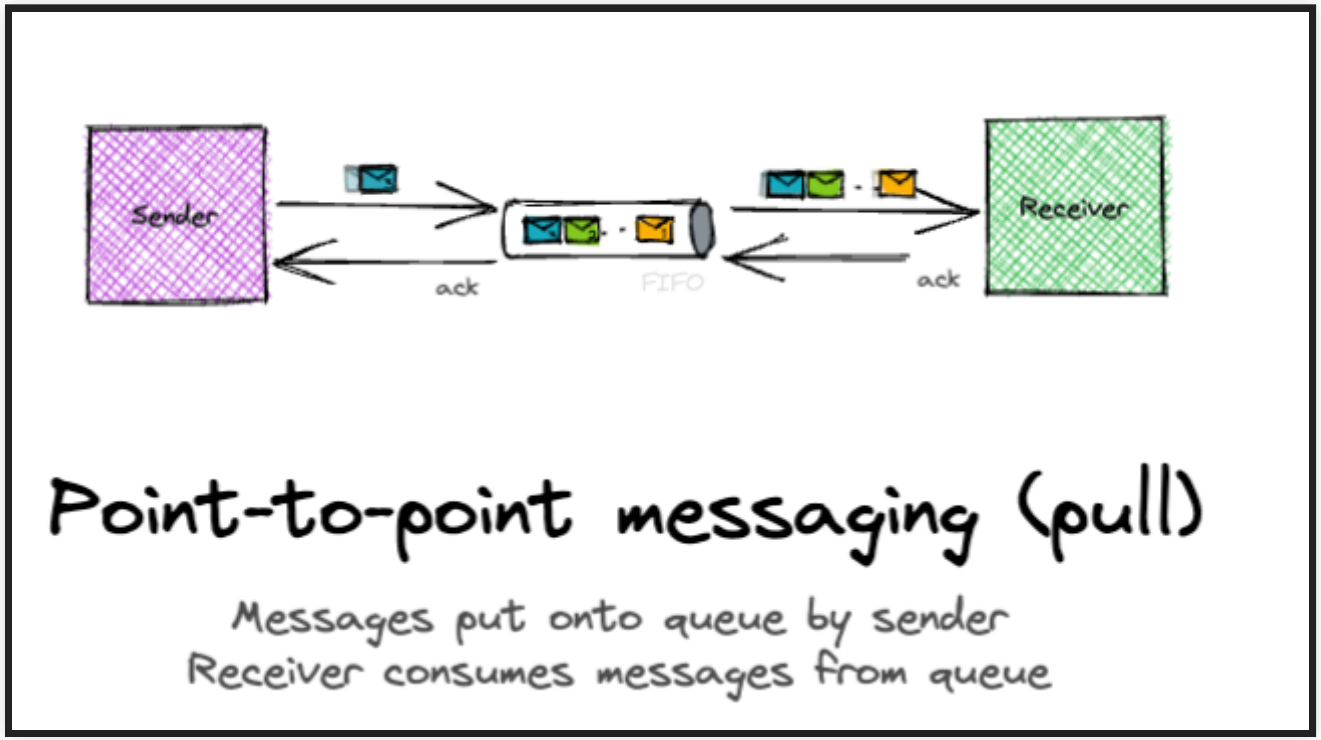
- **Geringe Kopplung** auf Seiten des Publishers: Anzahl, Identität oder Art der Nachrichten, an denen die Abonnenten interessiert sind, muss nicht bekannt sein. Sie geben einfach Daten als Reaktion auf die richtigen Ereignisse über ihre API aus. Dies ermöglicht Flexibilität und Skalierbarkeit, da neue Subscriber einfach zum System hinzugefügt werden können, ohne dass sich dies auf den Publisher auswirkt.
- **Geringere kognitive Belastung** für die Subscriber: Innenleben des Herausgebers ist nicht von Interesse, Zugriff auf Implementierungsdetails des Herausgebers ist nicht notwendig. Sie können nur über die öffentliche API des Publishers mit dem Herausgeber kommunizieren, was ihr Verständnis des Systems vereinfacht.

Quelle: https://www.enjoyalgorithms.com/blog/publisher-subscriber-pattern

- **Separation of Concerns**: Die Einfachheit des Pub/Sub-Patterns (Daten fließen in eine Richtung vom Herausgeber zum Abonnenten) ermöglicht es den Entwicklern, eine Trennung von Belangen vorzunehmen. Das bedeutet, dass verschiedene Nachrichtentypen in verschiedene Kategorien aufgeteilt werden können, die jeweils einen einzigen, eindeutigen Zweck erfüllen. So können beispielsweise Daten mit dem Thema `/cats` Informationen über Katzen enthalten, während Daten mit dem Thema `/dogs` Informationen über Hunde enthalten können.
- **Verbesserte Testbarkeit**: Durch die fein abgestufte Kontrolle über die Themen lässt sich leicht überprüfen, ob die verschiedenen Ereignisbusse die erforderlichen Nachrichten übermitteln.
- **Verbesserte Sicherheit**: Die Pub/Sub-Architektur eignet sich gut für das Sicherheitsprinzip der Zuweisung von minimalen Privilegien oder Informationen. Entwickler können problemlos Module erstellen, die nur die Mindestanzahl von Nachrichtentypen abonniert haben, die zum Funktionieren erforderlich sind.

Quelle: https://www.enjoyalgorithms.com/blog/publisher-subscriber-pattern

POINT-TO-POINT-MESSAGING



Quelle: https://serverlessland.com/event-driven-architecture/visuals/inside-event-driven-architectures

Senden von Nachrichten an einen Kanal zur Verarbeitung durch nachgeschaltete Verbraucher

- Ist ein Verteilungsmuster, das in Nachrichtenwarteschlangen mit einer Eins-zu-Eins-Beziehung zwischen dem Absender und dem Empfänger verwendet
- Jede Nachricht in der Warteschlange wird nur an einen Empfänger gesendet und nur einmal konsumiert bzw. verarbeitet
- Mehrere Verbraucher können Nachrichten aus der Warteschlange zur gleichzeitigen Verarbeitung abrufen
- Beispiele für geeignete Anwendungsfälle für diese Art der Nachrichtenübermittlung sind die Verarbeitung von Gehaltsabrechnungen und Finanztransaktionen, bei denen die Verarbeitung nur einmal erfolgen darf

Quelle: https://www.ibm.com/de-de/topics/message-brokers

MESSAGE DELIVERY

Verschiedene Systeme bieten unterschiedliche Lösungen für Nachrichtenzustellung an.

MESSAGE DELIVERY

Verschiedene Systeme bieten unterschiedliche Lösungen für Nachrichtenzustellung an.

At most once delivery

MESSAGE DELIVERY

Verschiedene Systeme bieten unterschiedliche Lösungen für Nachrichtenzustellung an.

At most once delivery

At least once delivery

MESSAGE DELIVERY

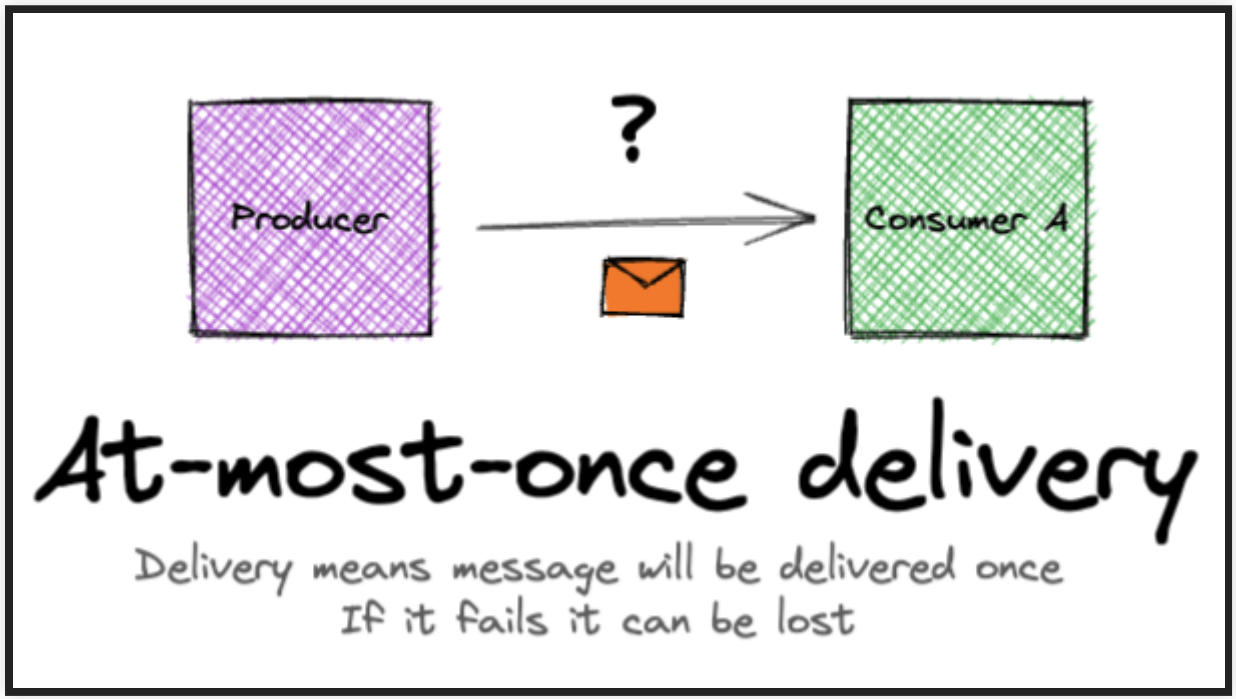
Verschiedene Systeme bieten unterschiedliche Lösungen für Nachrichtenzustellung an.

At most once delivery

At least once delivery

Exactly once delivery

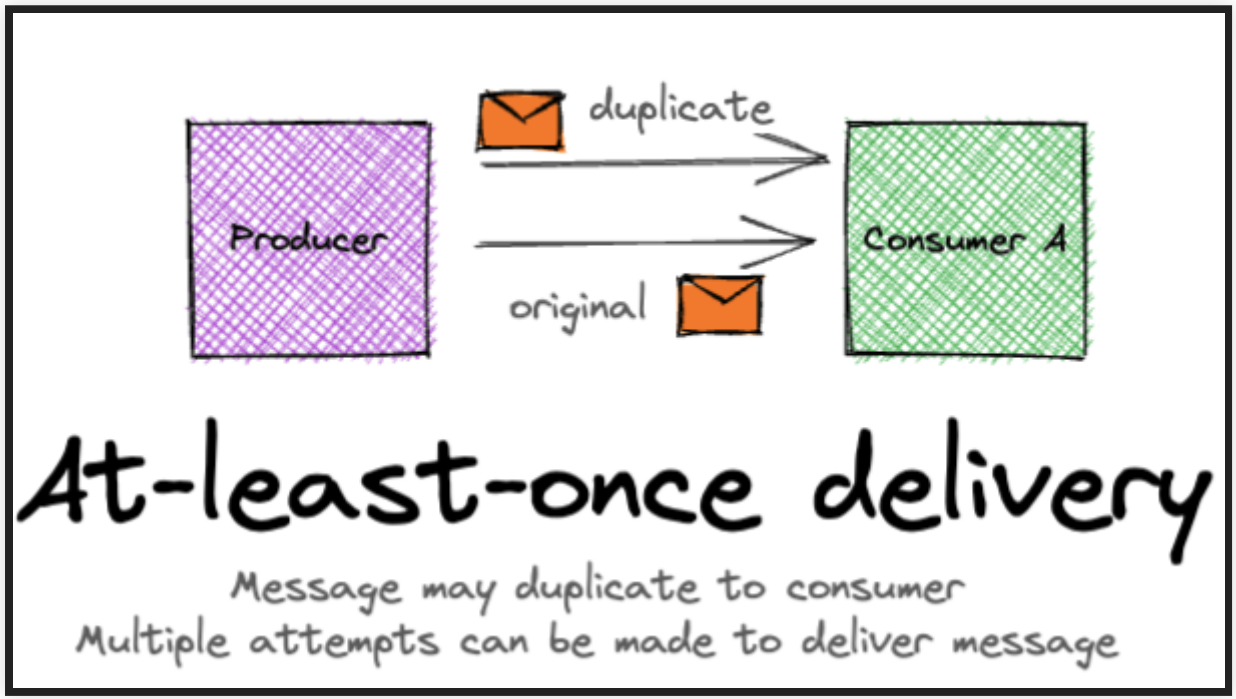
At most once delivery



Bedeutet, dass die Nachricht einmal zugestellt wird. Wenn sie fehlschlägt, kann sie verloren gehen

Quelle: https://serverlessland.com/event-driven-architecture/visuals/message-delivery

At least once delivery



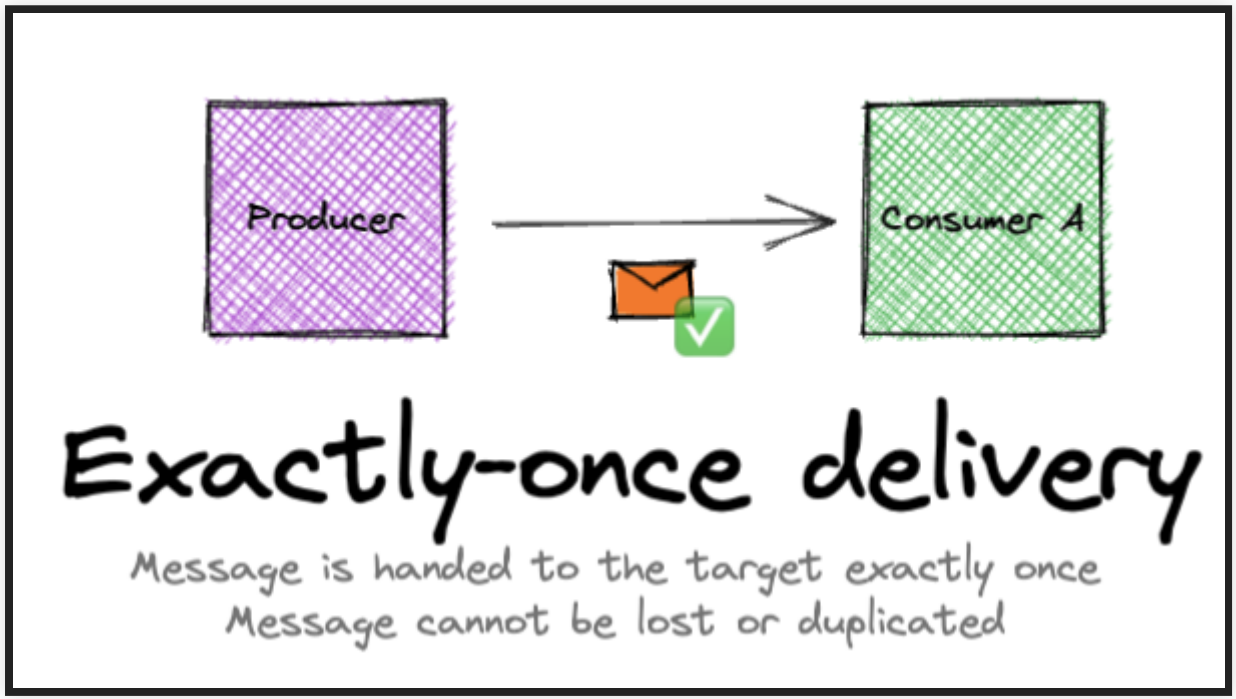
Bedeutet, dass die Nachricht mindestens einmal zugestellt wird. Fehlschläge beim versenden werden damit vermieden.

Quelle: https://serverlessland.com/event-driven-architecture/visuals/message-delivery

- Es können mehrere Versuche unternommen werden, die Nachricht zuzustellen.
- Die Nachricht kann dem Verbraucher doppelt zugestellt werden.
- Es ist wichtig, idempotente Konsumenten zu haben.
- Es können mehrere Versuche unternommen werden, die Nachricht an das Ziel zuzustellen, bis einer davon erfolgreich ist.
- Dies bedeutet, dass Nachrichten dupliziert werden können, aber nicht verloren gehen.

Quelle: https://serverlessland.com/event-driven-architecture/visuals/message-delivery

Exactly once delivery



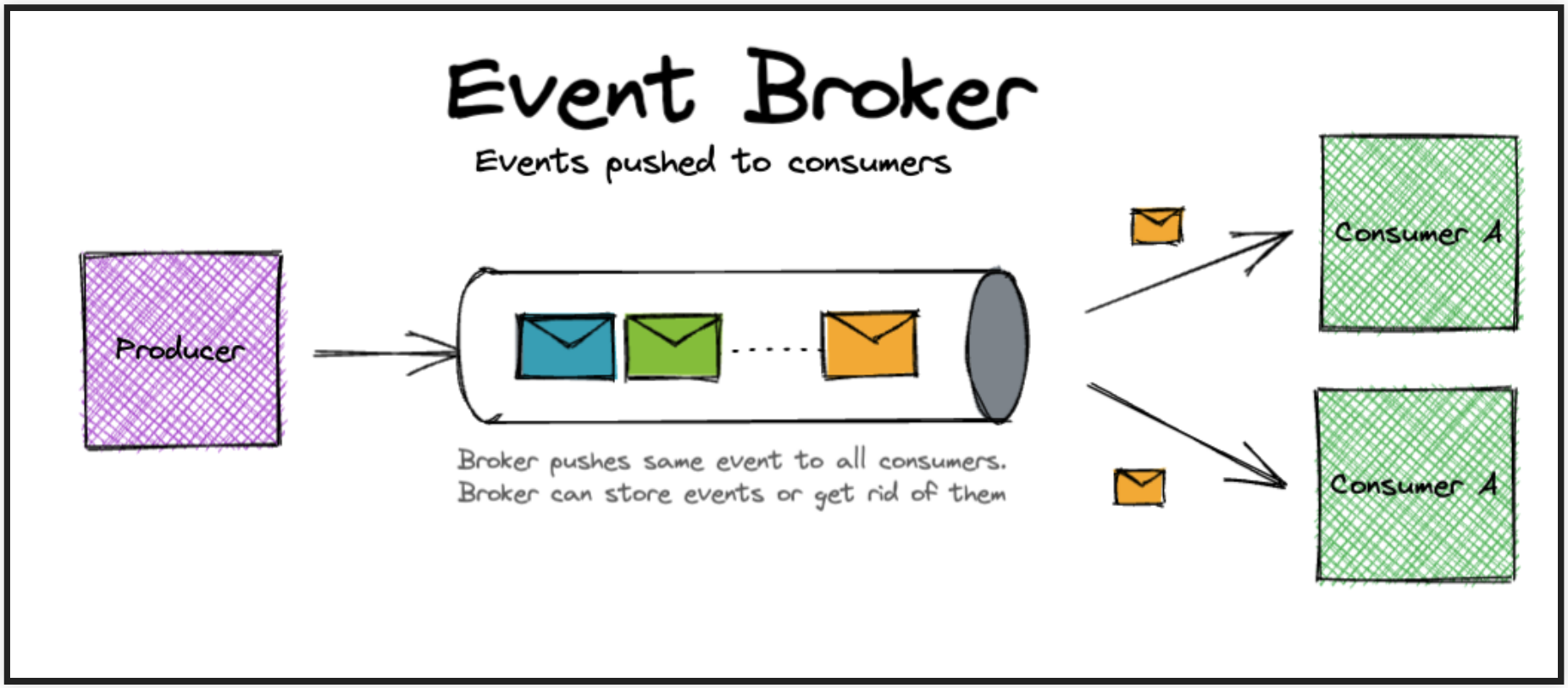
Die Nachricht wird genau einmal an das Ziel übermittelt. Nachricht kann nicht verloren gehen oder dupliziert werden.

Quelle: https://serverlessland.com/event-driven-architecture/visuals/message-delivery

MESSAGE QUEUE VS. EVENT BROKER

What's The Difference Between Event Brokers and Message Queues?

Hierfür ist die Unterscheidung zwischen Event und Message, sowie Message-oriented Middleware und Event-driven Architecture wichtig.



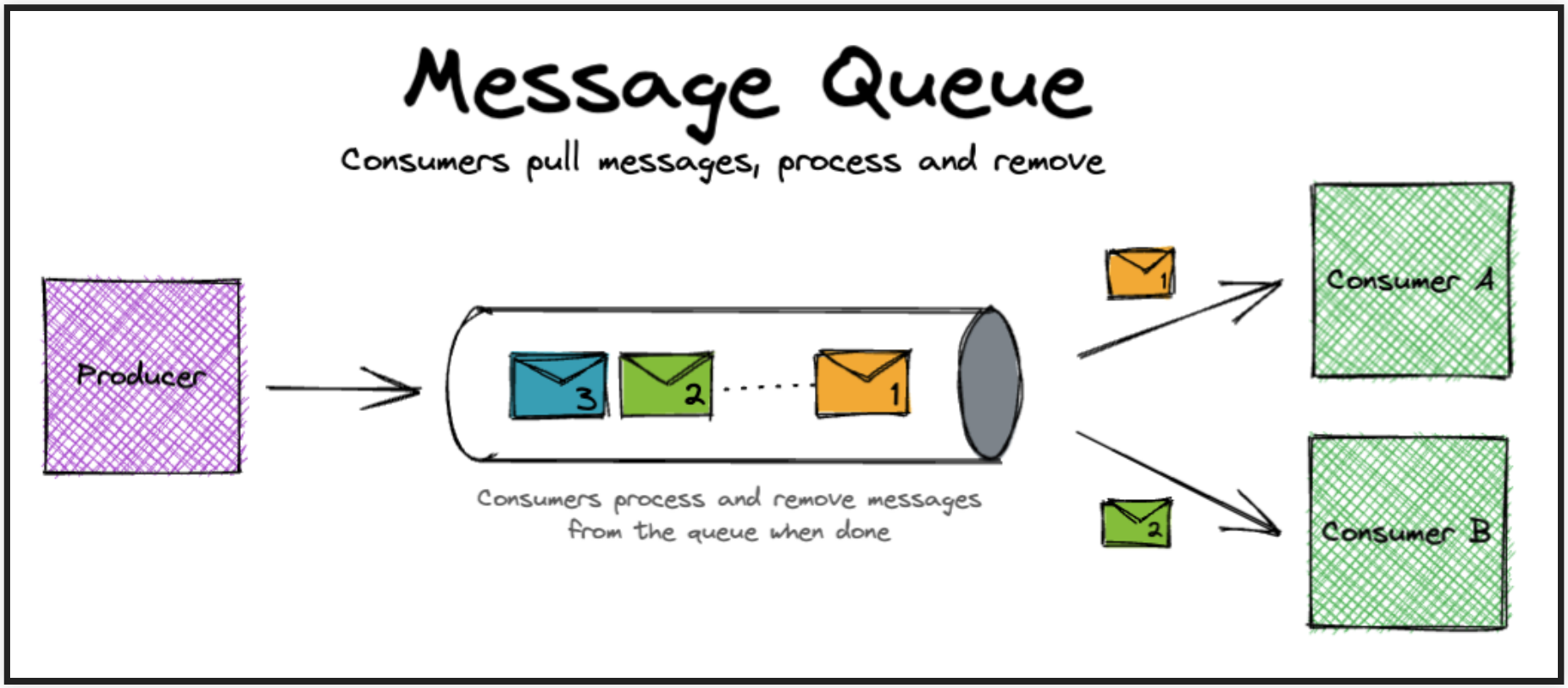
Quelle: https://serverlessland.com/event-driven-architecture/visuals/message-queue-vs-event-broker

- In dieser Architektur gibt es drei Akteure. Der Ereignisproduzent ist die Einheit, die für die Erzeugung des Ereignisses verantwortlich ist
- Als Nächstes gibt es den Mediator oder Broker; wenn dieses Ereignis an den Verbraucher gesendet wird, wird es entweder in geordneter Weise an bestimmte Verbraucher (Mediator) oder gleichmäßig an alle interessierten Parteien (Broker) verteilt.
- Der letzte Akteur, der Verbraucher, empfängt diese Ereignisse und verarbeitet sie für den von ihm gewünschten Zweck.

Quelle: https://nordicapis.com/whats-the-difference-between-event-brokers-and-message-queues/

- es gibt verschiedene Arten von Brokern
- Einige Broker können einfache Subscribe-Pushes anbieten, während andere Ereignisse auf der Grundlage zuvor definierter Attribute in ein Log schieben können
- Letztendlich bleibt das Paradigma dasselbe: Ein Ereignis tritt ein, wird an den Broker und dann an die interessierten Parteien weitergeleitet.

Quelle: https://nordicapis.com/whats-the-difference-between-event-brokers-and-message-queues/



Quelle: https://serverlessland.com/event-driven-architecture/visuals/message-queue-vs-event-broker

- Message-Queues lösen das gleiche Problem wie Event-Broker, aber auf eine andere Weise
- Die einfache Idee einer Message-Queues besteht darin, einzelne Dienste eine Queue haben können, die eine Vielzahl von Aufrufen enthält, die sequentiell sortiert und an alle interessierten Parteien gesendet werden.
- Anstatt einfach "Es ist etwas passiert" zu verteilen, nimmt eine Queue eine zu verarbeitende Nachricht auf und fügt sie einer Warteschlange hinzu
- Im weiteren Verlauf können Verbraucher oder zusätzliche Prozessoren die Warteschlange einsehen und die Nachricht aus der Warteschlange nehmen, um sie weiterzuverarbeiten, zu behalten oder zu ihren eigenen Prozessen hinzuzufügen

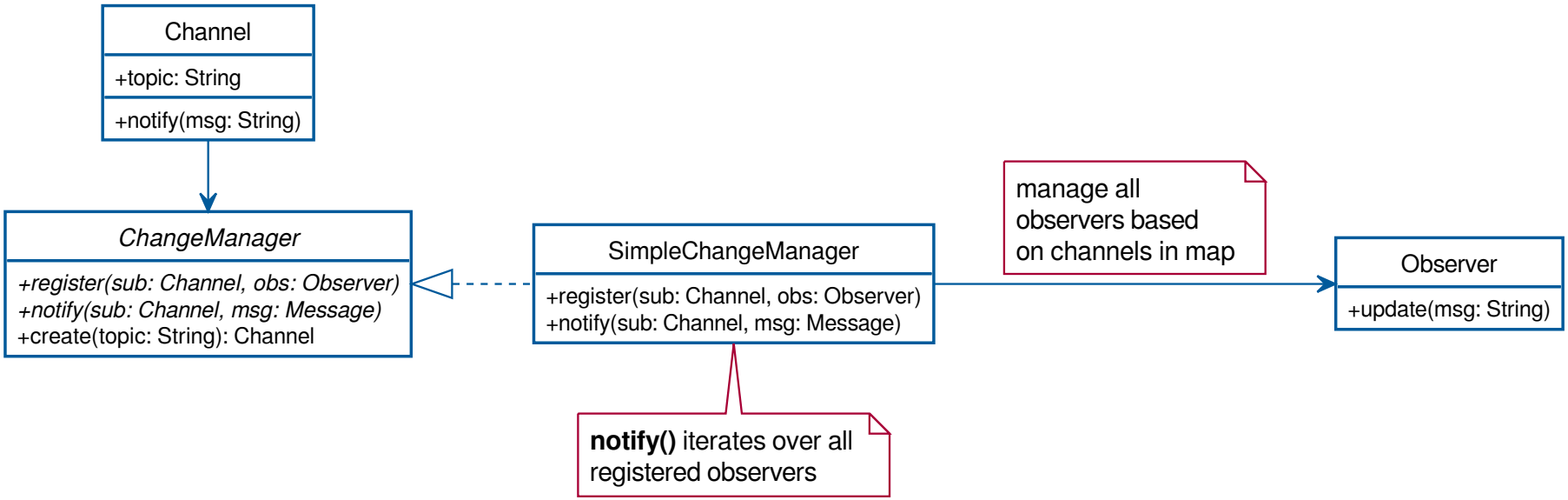
Quelle: https://serverlessland.com/event-driven-architecture/visuals/message-queue-vs-event-broker

- Nachrichten-Warteschlangen können Patterns verwenden, um ihre Nachrichten zu adressieren und so die Zustandsänderungen im System richtig zu steuern
- Themenbasiertes Routing kann dabei helfen, Zustandsänderungen an relevante Parteien zu adressieren, ohne dass diese Parteien die Daten tatsächlich anfordern
- Alternativ können die Beteiligten mit Hilfe von Pub-Sub-Beziehungen allgemeine Interessengebiete abonnieren, die dann als Kategorien für diese Nachrichten verwendet werden können.

Quelle: https://serverlessland.com/event-driven-architecture/visuals/message-queue-vs-event-broker

ÜBUNG: PUBLISH / SUBSCRIBE

Was ist notwendig um das Pub/Sub Konzept in einer (objektorientierten) Sprache zu realisieren? Skizzieren Sie eine mögliche Lösung unter Verwendung von Klassen-, Komponenten-, Sequenz- und/oder Kommunikationsdiagrammen. Alternativ kann auch Pseudocode verwendet werden.


Channel
+topic: String
+notify(msg: String)
ChangeManager
+register(sub: Channel, obs: Observer)
+notify(sub: Channel, msg: Message)
+create(topic: String): Channel
SimpleChangeManager
+register(sub: Channel, obs: Observer)
+notify(sub: Channel, msg: Message)
manage all observers based on channels in map
Observer
+update(msg: String)
notify() iterates over all registered observers

Implementierung des beschriebenen Ansatzes

Nutzen Sie das bereitgestellte Projekt-Archiv publish-subscribe.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Verwendung des Ansatzes in einer Main-Methode

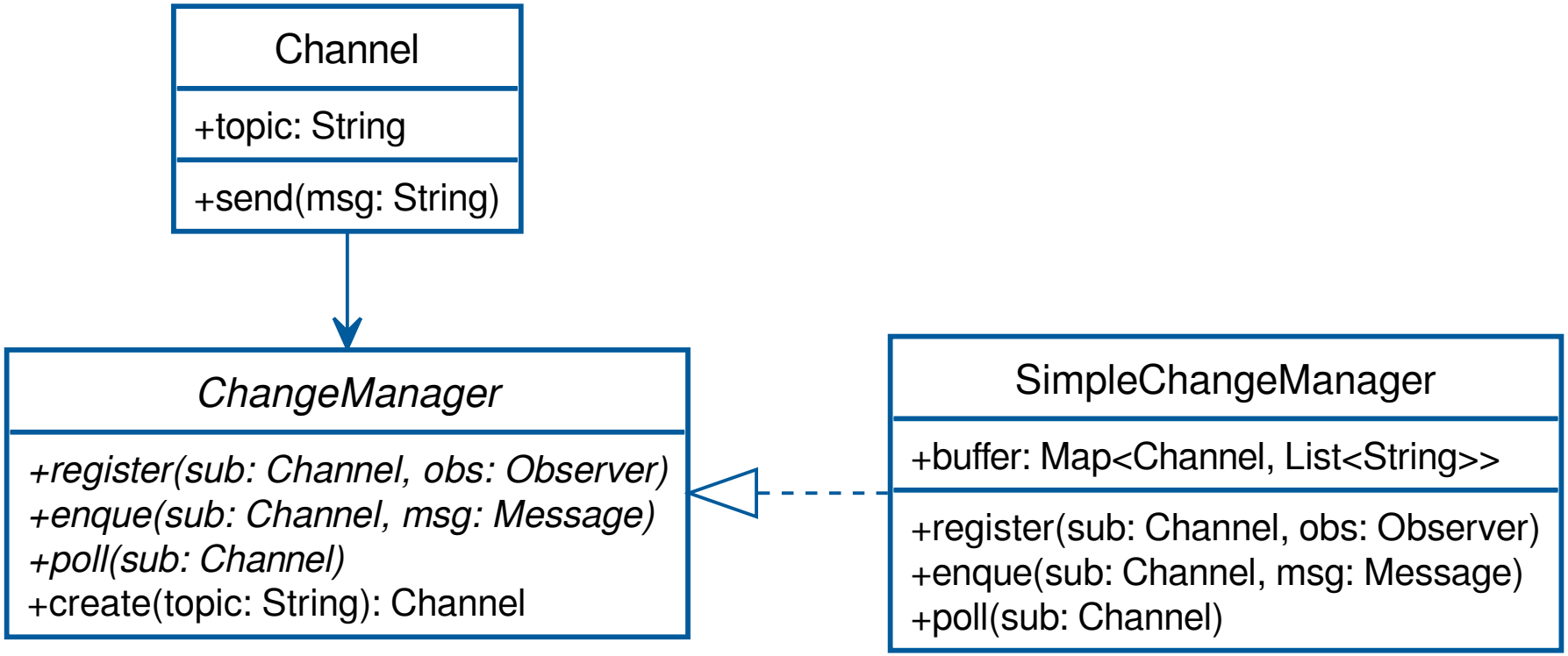
```
ChangeManager manager = new SimpleChangerManager();
Channel channel1 = manager.create("test1");
Channel channel2 = manager.create("test2");
manager.register(channel1, new Observer() {
  public void update(String message) {
    System.out.println(message);
  }
});
channel1.notify("Hello, World!");
channel2.notify("Hello, Other!");
```

- Erzeugt zwei Channels, den `channel1` und `channel2`
- Subscribed einen Observer in `channel1`
- Informiert alle Subscriber bzw. Observer in `channel1`
- Informiert alle Subscriber bzw. Observer in `channel2`, da es keine gibt passiert nichts

ÜBUNG: POINT-TO-POINT

Was ist notwendig um das Point-to-Point Konzept in einer (objektorientierten) Sprache zu realisieren? Skizzieren Sie eine mögliche Lösung unter Verwendung von Klassen-, Komponenten-, Sequenz- und/oder Kommunikationsdiagrammen. Alternativ kann auch Pseudocode verwendet werden.

Es kann ein vergleichbarer Ansatz verwendet werden



Es ist darauf zu achten, dass bei neuen Ereignissen ein geeigneter Empfänger die Nachricht erhält. Falls keiner da ist, muss die Nachricht gepuffert werden.

Eine Mögliche Nutzung würde sich dann wie folgend aussehen

```
ChangeManager manager = new SimpleChangerManager();
Channel channel1 = manager.create("test1");
Channel channel2 = manager.create("test2");
channel1.send("Hello, World!");
channel2.send("Hello, Other!");
String msg = manager.poll(channel1);
```

Ein Consumer könnte mittels Poll am Manager Nachrichten für einen Kanal abholen. Gibt es mehrere Aufrufe, sollte dies nicht die selbe Nachricht noch einmal liefern.

ÜBUNG: REALISIERUNG MITTELS HTTP

Erweitern Sie die Pub/Sub und Point-to-Point Ansätze in einer HTTP-Anwendung. Was ist notwendig, damit die Patterns umgesetzt werden können?

Beispiel-Implementierung des Pub/Sub mit Spring über einen HTTP-Endpunkt

Nutzen Sie das bereitgestellte Projekt-Archiv pub-sub-http-spring.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

ÜBUNG: REALISIERUNG MITTELS WEBSOCKETS

Erweitern Sie die Pub/Sub und Point-to-Point Ansätze in einer Anwendung die Websockets unterstützt. Was ist notwendig, damit die Patterns umgesetzt werden können?

Beispiel-Implementierung des Pub/Sub mit Jarkata WebSockets

Nutzen Sie das bereitgestellte Projekt-Archiv pub-sub-websocket-java.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Beispiel-Implementierung des Pub/Sub mit NodeJS WebSockets

Nutzen Sie das bereitgestellte Projekt-Archiv pub-sub-websocket-nodejs.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

2.3 DATENFORMATE, APIS UND PROTOKOLLE

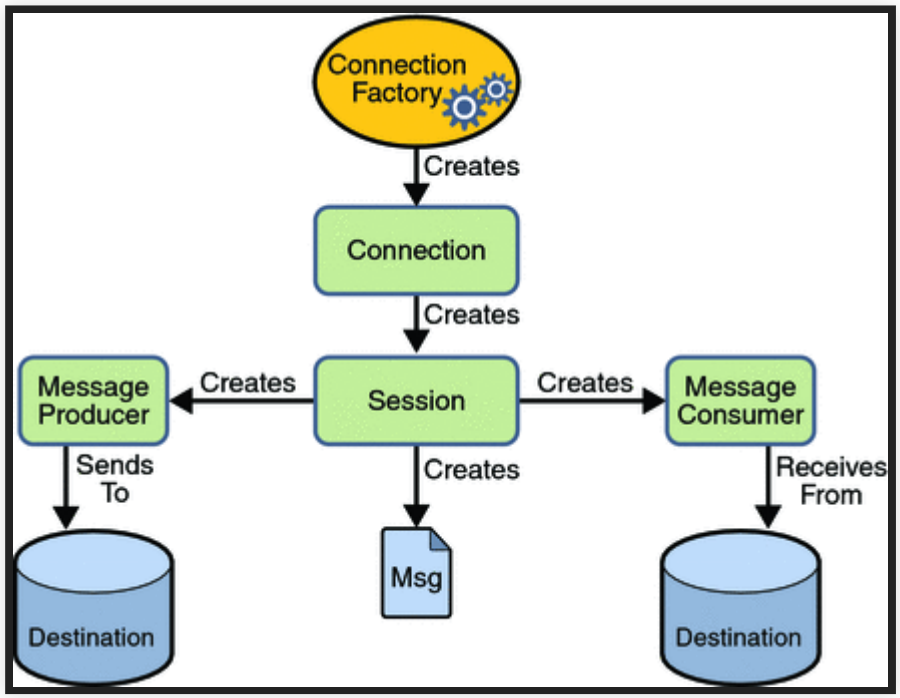
Zum Austauschen von Nachrichten gibt es verschiedene APIs und Protokolle

- JMS als standardisierte API in Java
- MQTT im Bereich IoT
- AMQP (Advanced Message Queuing Protocol)
- CoAP (Constrained Application Protocol)
- XMPP (Extensible Messaging and Presence Protocol)
- ... sowie WebSockets direkt oder Sub-Protokolle wie STOMP

Im folgenden soll auf die klassischen Ansätze wie JMS, MQTT und AMQP eingegangen werden.

JMS

JMS (Java Messaging Service) ist eine standardisierte API für die Programmier- sprache Java und Teil des Java-EE-Standards.



Quelle: https://docs.oracle.com/javaee/5/tutorial/doc/bnceh.html

- Bekannte Implementierungen sind Apache ActiveMQ (http://activemq.apache.o
  oder IBM MQ (http:// www-03.ibm.com/software/products/en/ibm-mq)
- Es gibt weitere JMS-Produkte
  (https://en.wikipedia.org/wiki/Java_Message_Service#Provider_implementatio
- Java Application Server, die nicht nur das Web Profile sondern das vollständige Java-EE-Profil unterstützen, müssen eine JMS enthalten

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

Beispiel mit Spring und JMS sowie lokaler Queue

Nutzen Sie das bereitgestellte Projekt-Archiv jms-example.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Quelle: https://github.com/JavaProgramTo/Spring-Boot-ActiveMQ

Beispiel für den Publisher

```
@RequestMapping("/api/publish")
@RestController
public class Publisher {
    Logger logger = LoggerFactory.getLogger(getClass());
    @Autowired private JmsTemplate jmsTemplate;
    @Autowired private Queue queue;
    @GetMapping("/{msg}")
    public String publishMessage(@PathVariable("msg") String content ){
        jmsTemplate.convertAndSend(queue, content);
        logger.info("Message published : "+content);
        return "Success";
    }
}
```

- Beispiel zeigt einen REST-Controller mit Spring welcher über http://localhost:8080/api/publish bereit steht
- Wenn ein GET-Request eingeht, wird die Nachricht in die JMS-Queue übergeben
- Per Dependency Injection wird innerhalb der Anwendung die JMS-Queue bereitgestellt

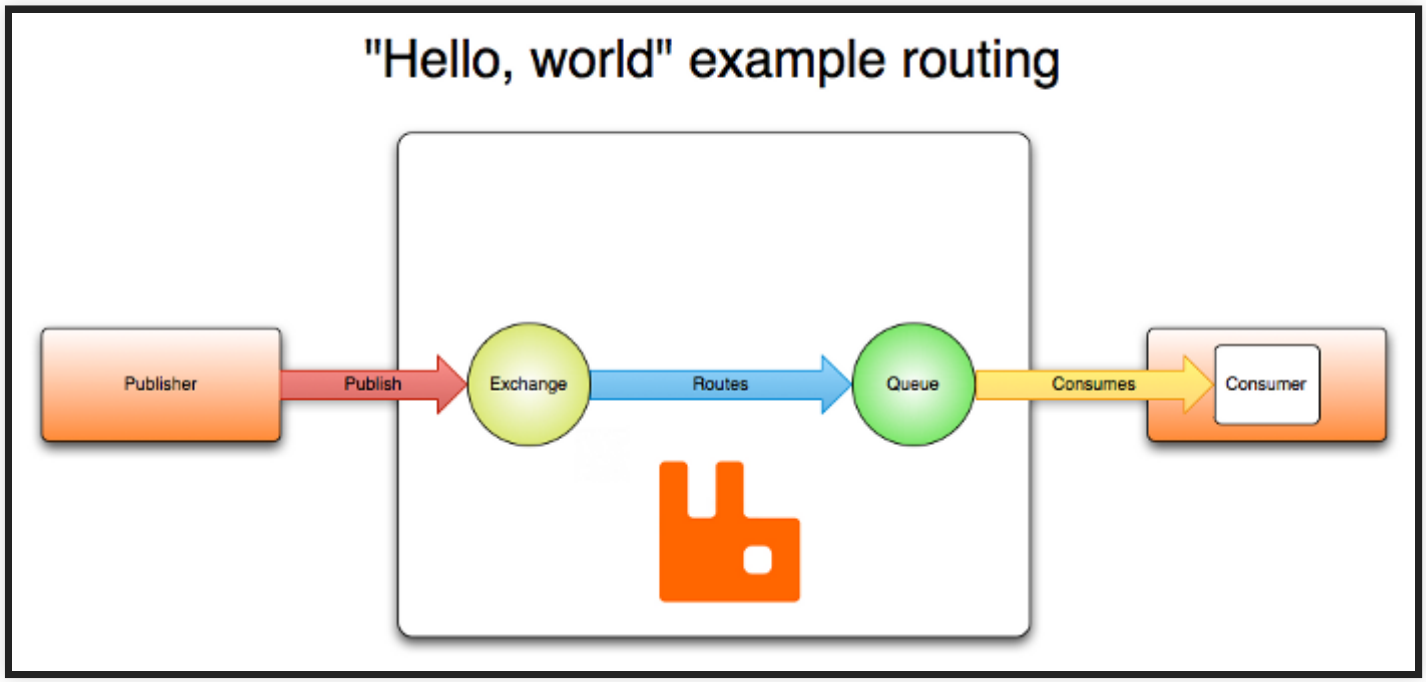
Beispiel für den Consumer

```java
@Component
public class Consumer {
    Logger logger = LoggerFactory.getLogger(getClass());
    @JmsListener(destination = "local.inmemory.queue")
    public void onMessage(String content){
        logger.info("Message received : "+content);
    }
}
```

- Eine unabhängige Komponente wird hier zur Verarbeitung verwendet
- Mithilfe einer Annotation angegeben, dass die folgende Methode aufgerufen werden soll, wenn in der Queue `local.inmemory.queue` eine Nachricht eingeht
- Die Methode nimmt die Nachricht als Parameter entgegen und kann diese verarbeiten

AMQP

AMQP (Advanced Message Queuing Protocol) standardisiert keine API, sondern ein Netzwerk-Protokoll auf TCP/IP-Ebene.



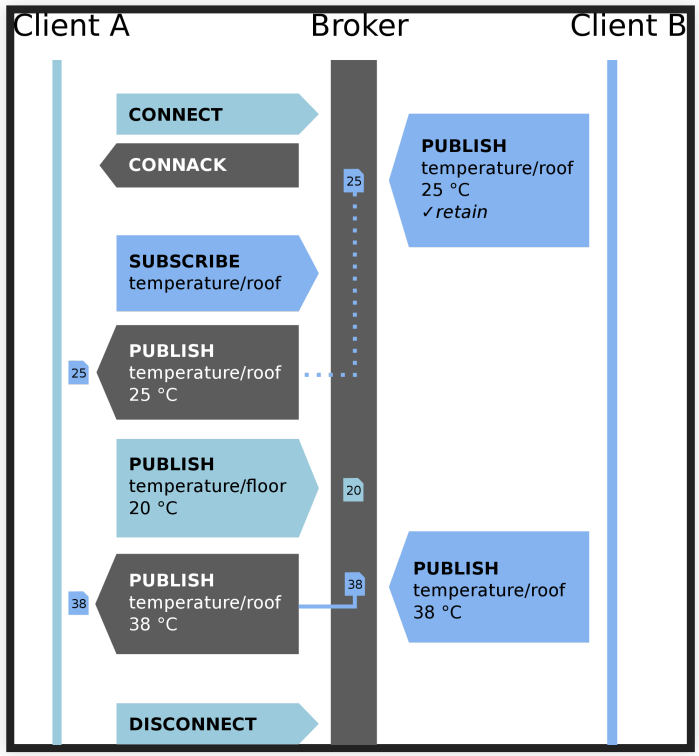
Quelle: https://www.rabbitmq.com/tutorials/amqp-concepts.html

- Open-Source-Standard für ein asynchrones Messaging
- AMQP ermöglicht verschlüsselte und kompatible Nachrichtenübertragung
- Ermöglicht einen einfacheren Austausch der Implementierung
- Bekannte Umsetzungen:
  - RabbitMQ (https://www.rabbitmq.com/)
  - Apache ActiveMQ (http://activemq.apache.org/)
  - Apache Qpid (https://qpid.apache.org/)
- Auch bei AMQP gibt es weitere Alternativen (https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol#Implementations

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

MQTT

MQTT ist ein Kommunikationsprotokoll mit Funktionen, die speziell auf IoT-Lösungen ausgerichtet sind:



Quelle: https://de.wikipedia.org/wiki/MQTT#/media/Datei:MQTT_protocol_example_without_QoS.svg

- Ursprünglich **M**essage **Q**ueue *T*elemetry *T*ransport; seit Version 3.1.1 wurde definiert, dass MQTT für kein Akronym steht.
- 1999 von IBM entwickelt, seit 2013 standardisiert über OASIS als IoT-Protokoll
- Setzt auf TCP/IP auf (Port 1883 und 8883 reserviert durch Internet Assigned Numbers Authority (IANA))
- Verwendet TCP-Verbindungen für Zuverlässigkeit (gesicherte Zustellung und Paketfehlerprüfung), Fragmentierung und Ordnung.
- Ziel ist es, den Daten-Overhead eines jeden MQTT-Pakets zu minimieren.

Quelle: https://www.hivemq.com/blog/mqtt-vs-amqp-for-iot/ und https://chemnitzer.linux-tage.de/2018/media/programm/papers/157_MQTT_Das_Protokoll_der_Dinge_folien_mqtt_clt2018.pdf

- Der letzte bekannte Datenwert für ein Gerät kann gespeichert werden (Retained-Messages).
- Benachrichtigungen, wenn der Client unerwartet die Verbindung trennt (Will-Messages), damit der Client-Status überwacht werden kann.
- Bidirektionaler Nachrichtenfluss - Daten von und Befehle an Geräte können dieselbe TCP-Verbindung nutzen.
- Publish-Subscribe-Routing, das die einfache Hinzufügung weiterer Konsumenten und Produzenten von Daten ermöglicht.
- MQTT-SN → Abwandlung für non-TCP/IP-Netzwerke (z.B. ZigBee)

Quelle: https://www.hivemq.com/blog/mqtt-vs-amqp-for-iot/ und https://chemnitzer.linux-tage.de/2018/media/programm/papers/157_MQTT_Das_Protokoll_der_Dinge_folien_mqtt_clt2018.pdf

Topics sind hierarchisch organisiert

```
sensoren/sensor1/temperature
sensoren/sensor1/humidity
sensoren/sensor2/temperature
sensoren/sensor2/humidity
```

Hinweis: Topics sollten nicht mit einem `/` beginnen, damit wäre eine Hierarchie-Ebene mit leerem Namen verbunden.

Subscribe auf einem Topic erlaubt Wildcards

```
sensoren/
sensoren/sensor1/#
sensoren/+/humidity
```

- Mit dem Zeichen # kann ab einer Hierarchie-Ebene alles, und was darunter liegt, adressiert werden
- Ein + entspricht einer einzigen, beliebigen Hierarchie-Ebene

Retained-Messages

- Normalerweise bekommen Subscriber erst dann Nachrichten zu einem abonnierten Topic zugestellt, wenn ein Publisher aktuell eine Nachricht zum Broker sendet...
- ... mit dem Retained-Flag gekennzeichnete Nachrichten werden vom Broker zwischengespeichert
- ... und werden vom Broker sofort ausgeliefert, wenn sich ein Publisher entsprechend beim Broker anmeldet

Quelle: https://chemnitzer.linux-tage.de/2018/media/programm/papers/157_MQTT_Das_Protokoll_der_Dinge_folien_mqtt_clt2018.pdf

Last Will and Testament

- Was soll passieren, wenn die Verbindung zwischen einem Client (Publisher/Subscriber) und Broker abbricht?
- Es kann eine Nachricht (Topic/Payload) vom Client definiert werden, die, bei Verbindungsabbruch, vom Broker publiziert wird;

Bei Verbindungsaufbau:

```
LWT-Definieren: sensor/status/ "OFF"
Publish: sensor/status/ "ON"
```

Bei Verbindungsabbruch:

```
Publish sensor/status/ "OFF"
```

Quelle: https://chemnitzer.linux-tage.de/2018/media/programm/papers/157_MQTT_Das_Protokoll_der_Dinge_folien_mqtt_clt2018.pdf

Quality of Service (QoS)

- QoS definiert/gewährleistet Garantien bezüglich der Zustellung von Nachrichten
- MQTT-Protokoll definiert 3 Stufen:
  - 0 → höchstens einmal (...aber auch keinmal!)
  - 1 → mindestens einmal (...aber auch mehrmals!)
  - 2 → exakt einmal
- QoS gilt in beide Richtungen:
  - publish: Client → Broker
  - subscribe (push): Broker → Client
- Der Client (Subscriber/Publisher) gibt den QoS-Level vor

Quelle: https://chemnitzer.linux-tage.de/2018/media/programm/papers/157_MQTT_Das_Protokoll_der_Dinge_folien_mqtt_clt2018.pdf

Bekannte Implementierungen für MQTT-basierte Broker

- Moquette
- Mosquitto
- MQTTRoute
- Emqttd
- HiveMQ
- HBMQTT

Bekannte Client-Bibliotheken

- C/C++ (z.B. libmosquitto, libmosquittopp)
- Java (z.B. Eclipse Paho Java, moquette)
- Javascript, Node.js (z.B. mqtt.js)
- PHP (z.B. phpMQTT, Mosquitto-PHP)
- Python (z.B. Eclipce Paho Python)
- Nodemcu (...Lua-Firmware für ESP-Chips)
- Tcl (→ http://wiki.tcl.tk/48822)

Quelle: https://chemnitzer.linux-tage.de/2018/media/programm/papers/157_MQTT_Das_Protokoll_der_Dinge_folien_mqtt_clt2018.pdf

MQTT VS. AMQP

Quelle: https://www.hivemq.com/blog/mqtt-vs-amqp-for-iot/

	MQTT	AMQP
Command targets	Topics	Exchanges, Queues
Underlying Protocol	TCP/IP	TCP/IP
Secure connections	TLS + username/password (SASL support possible)	TLS + username/password (SASL support possible)
Client observability	Known connection status (will messages)	Unknown connection status
Messaging Mode	Asynchronous, event-based	Synchronous and asynchronous
Message queuing	The broker can queue messages for disconnected subscribers	Core capability, flexible configuration

	MQTT	AMQP
Message overhead	2 bytes minimum	8 bytes (general frame format)
Message Size	256MB maximum	2GB theoretical, 128MB max recommended
Content type	Any (binary)	Any (binary)
Topic matching	Level separator: / Wildcards: + #	Level separator: . Wildcards: * #
Reliability	Three qualities of service: 0 - fire and forget 1 - at least once 2 - once and only once	Two qualities of service: - without acks (=0) - with acks (=1)
Connection “multiplexing”	No	Yes - channels
Message attributes	MQTT 5.0 only	Yes
Object persistence	Yes	Yes

Die Vorteile von MQTT

- Es gibt zwei Funktionen von MQTT, die keine direkte Entsprechung in AMQP haben und im IoT-Bereich nützlich sind: Will- und Retained-Messages.
- MQTT hat einen geringeren Byte-Verbrauch beim Transfer als AMQP, da es sich mehr auf IoT-Lösungen fokussiert.
- MQTT ist einfacher zu konfigurieren und in Betrieb zu nehmen.
- Das Publish-Subscribe-Paradigma ermöglicht es vielen Clients, Informationen zu erzeugen und untereinander sowie mit Backend-Systemen auszutauschen.

Quelle: https://www.hivemq.com/blog/mqtt-vs-amqp-for-iot/

Vorteile von AMQP

- AMQP bietet mehr Möglichkeiten als MQTT für das Message Queuing
- Die Konfiguration von Routing-Topologien ist fehleranfällig
- Die Tatsache, dass AMQP 0.9.1 und 1.0 so unterschiedlich sind, führt zu einer gewissen Komplexität in diesem Lösungsraum - Version 1.0 ist die einzige, die bei OASIS standardisiert ist

Quelle: https://www.hivemq.com/blog/mqtt-vs-amqp-for-iot/

DATENÜBERTRAGUNGSFORMATE

- Protokolle wie MQTT und AMQP erlauben beliebige Content-Types.
- Hierbei sind proprietäre Formate denkbar, z.B. das direkte Kommunizieren von Messdaten in MQTT
- Alternativ kann auf übliche Formate im Bereich der Webtechnologien zurückgegriffen werden:
    - JSON
    - XML
    - AVRO
    - sowie JSON-B, Protobuf und FlatBuffers

Hinweis: Nachrichtenformate sollten gut gewählt werden und die Schemata der Formate bedarfen einem klaren Management. Ein effizientes properitäres Format kann später im Datastreaming zusätzlichen Aufwand bringen. Änderungen an Formaten betreffen die beteiligten Dienste bei der Erzeugung und Verarbeitung von Nachrichten bzw. Ereignissen.

2.4 JMS WITH APACHE ACTIVEMQ ARTEMIS

- Apache ActiveMQ Artemis ist ein Open-Source-Projekt für "multi-protocol, embeddable, high performance, clustered, asynchronous Messaging-Systems"
- Fungiert als Message Oriented Middleware (MoM)
- Ist lizenziert unter der Apache Software License v 2.0

Quelle: https://activemq.apache.org/components/artemis/documentation/1.5.0/book.pdf

- In Java geschrieben, läuft auf jeder Plattform mit einer Java 8+ Laufzeitumgebung
- Kann als Standalone-Lösung, integriert in JEE-Applikationsserver oder eingebettet in eigene Produkte eingesetzt werden
- HA-Lösung mit automatischem Client-Failover, so dass Sie bei einem Serverausfall keine Nachrichten verloren gehen oder duplizieren werden müssen

Quelle: https://activemq.apache.org/components/artemis/documentation/1.5.0/book.pdf

BEISPIEL MIT SPRING

Am Beispiel zu JMS findet sich folgend eine Realisierung mit Spring. Hier kommt erneut ein REST-Controller und eine unabhängige Componente in Spring zum Einsatz. Die verschiedenen Bestandteile könnten voneinander getrennt werden.

Nutzen Sie das bereitgestellte Projekt-Archiv apache-activemq-artemis.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

REST-Controller in Spring

```
@RestController
@RequestMapping("/api")
public class Endpoint {
  @Autowired
  Producer producer;

  @GetMapping(value = "/publish/{msg}")
  public String produce(@PathVariable("msg") String msg) {
    producer.send(msg);
    return "Done";
  }
}
```

Möglichkeiten zum Senden wird hier über eine Komponente ausgelagert und per Dependency Injection bereitgestellt.

Producer

```
@Component
public class Producer {
  @Autowired
  private JmsTemplate jmsTemplate;
  @Value("${jms.queue.destination}")
  private String destinationQueue;

  public void send(String msg) {
    jmsTemplate.convertAndSend(destinationQueue, msg);
  }
}
```

Der Producer verwendet erneut die Klasse JmsTemplate wovon ein Objekt ebenfalls per Dependency Injection bereitgestellt wird.

```java
@Component
public class Consumer {
  @JmsListener(destination = "${jms.queue.destination}")
  public void receive(String msg) {
    System.out.println("Got Message: " + msg);
  }
}
```

Die Komponente zum Verarbeiten muss anschließend nur erneut eine Methode zur Verarbeitung markieren.

2.5 MQTT WITH MOSQUITO

Run your first Mosquitto-Instance

Nutzen Sie das bereitgestellte Projekt-Archiv mosquitto.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Das Beispiel kann genutzt werden, um damit eine Mosquitto-Instanz lokal zu starten. Hierfür ist Docker und Docker Compose notwendig, welche üblicherweise in einer Installation von Docker Desktop enthalten sind.

```
~/mosquitto $ docker compose up -d
```

Zum Starten ist dann im entpackten Ordner das Kommando `docker compose up -d` notwendig, welches alle Container im Hintergrund startet.

Hilfreiche Docker-Compose Befehle

```
~/mosquitto $ docker compose stop
~/mosquitto $ docker compose start
~/mosquitto $ docker compose down
~/mosquitto $ docker compose down -v
```

- Stoppen der Container
- Starten der Container
- Entfernt alle Container lokal
- Entfernt alle Container lokal sowie dazugehörige gespeicherte Daten

Einfache Tests mit mosquitto_sub und mosquitto_pub

Subscribe für eine Topic

```
~/mosquitto $ docker compose exec mosquitto mosquitto_sub -h localhost -t te
```

Publish einer Nachricht in einem Topic

```
~/mosquitto $ docker compose exec mosquitto mosquitto_pub -h localhost -t te
```

BEISPIEL MIT NODEJS

Für den Zugriff mit NodeJS ist lediglich ein geeignetes Package für den Client notwendig, hier MQTT.js.

Nutzen Sie das bereitgestellte Projekt-Archiv mosquitto-nodejs.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Subscribe to Topic

```
const client = require('mqtt').connect(process.env.MQTT_HOST || 'mqtt://loca
client.on('connect', () =>
    client.subscribe(`test/rnd`,
        (err) => err && console.log("Error while subscribing to test")))
client.on('message', (topic, message) =>
    console.log(topic, message.toString()))
```

- Aufbau der Verbindung zum MQTT-Broker
- Callback, wenn die Verbindung aufgebaut wurde
- Falls erfolgreiche Verbindung vorliegt, wird ein Subscribe auf dem Topic `test/rnd` durchgeführt
- Jede eingehende Nachricht wird mit einem eigenen Callback behandelt, hier mit einer Ausgabe auf die Konsole

Publish to Topic

```
const client = require('mqtt').connect(process.env.MQTT_HOST || 'mqtt://loca
client.on('connect', () =>
    setInterval(() => {
        const v = Math.random();
        client.publish(`test/rnd`, `${Math.random()}`)
        console.log(`Send ${v} to test/rnd`);
    }, parseInt(process.env.FREQUENCY) || 1000))
```

- Aufbau der Verbindung und der Callback ist identisch zum Subscribe
- Im Beispiel wird jede Sekunde ein Ereignis veröffentlicht
- Zum Versenden in die Topic `test/rnd` ist der Aufruf `publish` notwendig

BEISPIEL MIT JAVA

Für den Zugriff mit Java wäre in Anlehnung an NodeJS ein geeignete Bibliothek für den Client notwendig, hier der Eclipse Paho Java Client.

Nutzen Sie das bereitgestellte Projekt-Archiv mosquitto-java.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Subscribe to Topic

```
MqttClient sampleClient = new MqttClient("tcp://localhost:1883",
    "ExampleServer", new MemoryPersistence());
MqttConnectOptions connOpts = new MqttConnectOptions();
connOpts.setCleanSession(true);
sampleClient.connect(connOpts);
sampleClient.subscribe("example", (t, msg) -> {
    System.out.println(msg);
});
```

Publish to Topic

```
MqttClient sampleClient = new MqttClient("tcp://localhost:1883",
  "ExampleServer", new MemoryPersistence());
MqttConnectOptions connOpts = new MqttConnectOptions();
connOpts.setCleanSession(true);
sampleClient.connect(connOpts);
MqttMessage message = new MqttMessage("Hello, World!".getBytes());
message.setQos(qos);
sampleClient.publish("example", message);
```

BEISPIEL ZUR ERFASSUNG VON SYSTEM-INFORMATIONEN

Das folgende Beispiel erweitert die Verwendung von Mosquitto und MQTT in NodeJS zur Erfassung von System-Informationen.

```
servers/278eb741a19a/cpu-usage              0.01908396946564883
servers/278eb741a19a/cpu-free               0.9808917197452229
servers/278eb741a19a/free-mem             695.3125
servers/278eb741a19a/total-mem           1984.89453125
servers/278eb741a19a/freemem-percentage     0.3503019878653817
```

Das Beispiel erfasst CPU, RAM und Speichernutzung der aktuellen Umgebung und publiziert diese in verschiedenen Topics.

Nutzen Sie das bereitgestellte Projekt-Archiv mosquitto-system-info.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

```
~/mosquitto $ docker compose up -d
```

Zum Starten ist dann im entpackten Ordner das Kommando `docker compose up -d` notwendig, welches alle Container im Hintergrund startet.

BEISPIEL ZUR INTEGRATION VON MQTT IN SPRING

Das Beispiel erzeugt einen Spring REST-Controller, welcher über `http://localhost:8080/example` aufgerufen werden kann.

Nutzen Sie das bereitgestellte Projekt-Archiv mosquitto-spring.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

ZUGANGSKONTROLLE

MQTT erlaubt die Zugriffskontrolle und somit über Credentials eine Absicherung der Topics.

Nutzen Sie das bereitgestellte Projekt-Archiv mosquitto-acl.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

```
~/mosquitto $ docker compose up -d
```

Zum Starten ist dann im entpackten Ordner das Kommando `docker compose up -d` notwendig, welches alle Container im Hintergrund startet.

Im Beispiel kann mittels dem folgenden Befehl ein Nutzer `someone` erstellt werden.

```
~/mosquitto $ docker compose exec mosquitto mosquitto_passwd -c /mosquitto/c
Password:
Reenter password:
```

Für den authentifizierten Zugriff sind zusätzlich zwei Konfigurationen notwendig:

```
allow_anonymous false
password_file /mosquitto/config/passwd
```

Darüber hinaus lassen sich Zugriffsrechte bis auf Topic-Ebene steuern. Hierfür wird eine Access Controll List bereitgestellt:

```
user someone
topic readwrite test1
topic read test2
topic write test3

user admin
topic readwrite #
```

- Erlaubt das Lesen und Beschreiben des Topics `test1` für den Nutzer `someone`
- Erlaubt das Lesen des Topics `test2` für den Nutzer `someone`
- Erlaubt das Beschreiben des Topics `test3` für den Nutzer `someone`
- Sowie ein administrativer Zugriff für einen Nutzer `admin`

Ein Test kann dann anschließend mit den folgenden Befehlen ausprobiert werden:

```
~/mosquitto $ docker compose exec mosquitto mosquitto_pub -h localhost \
  -t test1 -m "Hello, World" -u someone -P 12345678

~/mosquitto $ docker compose exec mosquitto mosquitto_sub -h localhost \
  -t test1 -v -u someone -P 12345678
```

WILL-MESSAGES

Nachrichten die beim Abbruch der Verbindung hinterlassen werden sollen, können beim Verbindungsaufbau als Konfiguration angegeben werden.

```
MqttClient sampleClient = new MqttClient(broker, clientId, persistence);
MqttConnectOptions connOpts = new MqttConnectOptions();
connOpts.setWill(topic, "Good bye!".getBytes(), 2, false);
connOpts.setCleanSession(true);
```

Hierbei wird in Java z.B. der Payload als Byte-Array hinterlegt.

RETAINED-MESSAGES

Retained Messages werden über einen Flag realisiert

```
// ...
MqttMessage message = new MqttMessage("Hello, World!".getBytes());
message.setQos(qos);
message.setRetained(true);
sampleClient.publish("example", message);
```

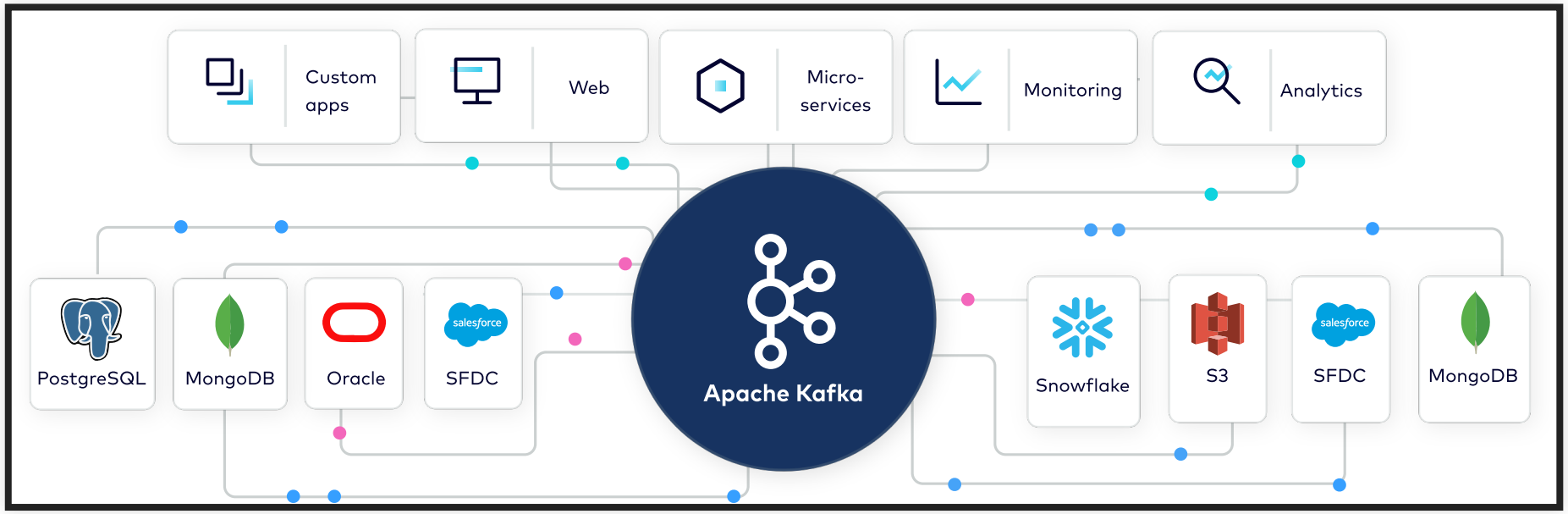
Nachrichten dieser Art bleiben als letzte bekannte Nachricht erhalten. Verbindet sich ein neuer Subscriber (neue Client-ID) wird diese Nachricht zugestellt.

2.6 AMQP MIT RABBITMQ

TODO

2.7 APACHE KAFKA

Apache Kafka kann als Message- bzw. Event-Broker verwendet werden und erlaubt performantes verteilen von Nachrichten / Ereignissen in einer skalierbaren Umgebung. Ereignisse in Kafka können persistiert und in einem Hochverfügbarkeitscluster betrieben werden.



Quelle: https://www.confluent.io/de-de/what-is-apache-kafka/

- Kafka ist in Java geschrieben.
- Kafka steht unter der Apache-2.0-Lizenz
- Die APIs werden mit einem sprachneutralen Protokoll angesprochen, diese unterscheiden sich in:
    - Producer API
    - Consumer API
    - Streams API
    - ... sowie Admin API und Connect API

Es gibt verschiedene Client-Implementierungen, um mit einem Kafka Broker zu arbeiten: https://cwiki.apache.org/confluence/display/ KAFKA/Clients

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

Run your first Kafka-Instance

Nutzen Sie das bereitgestellte Projekt-Archiv apache-kafka.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Das Beispiel kann genutzt werden, um damit eine Kafka-Instanz lokal zu starten. Hierfür ist Docker und Docker Compose notwendig, welche üblicherweise in einer Installation von Docker Desktop enthalten sind.

```
~/apache-kafka $ docker compose up -d
```

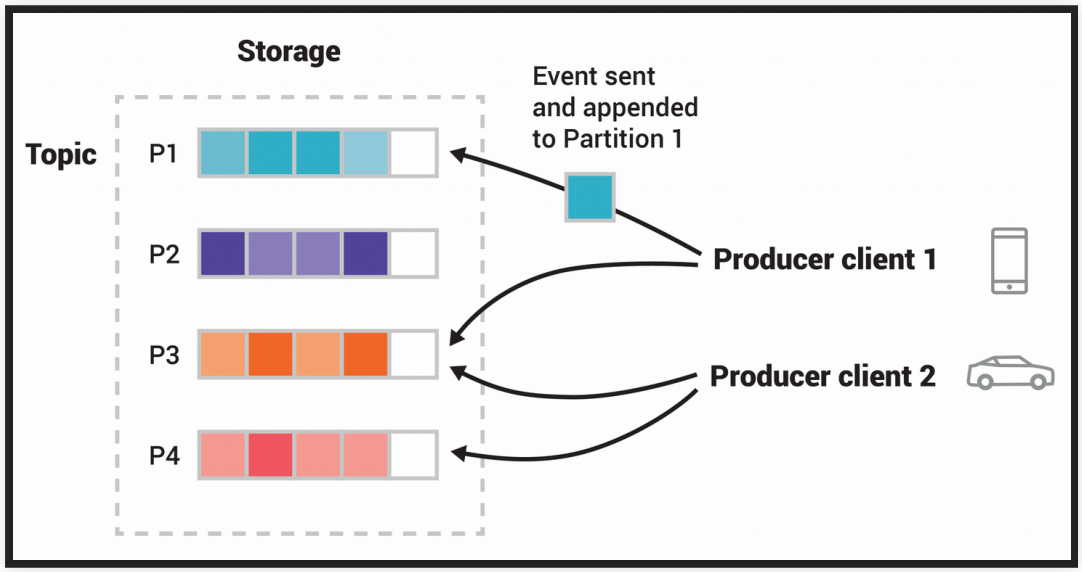
Zum Starten ist dann im entpackten Ordner das Kommando `docker compose up -d` notwendig, welches alle Container im Hintergrund startet.

Hilfreiche Docker-Compose Befehle

```
~/apache-kafka $ docker compose stop
~/apache-kafka $ docker compose start
~/apache-kafka $ docker compose down
~/apache-kafka $ docker compose down -v
```

- Stoppen der Container
- Starten der Container
- Entfernt alle Container lokal
- Entfernt alle Container lokal sowie dazugehörige gespeicherte Daten

Wichtige Begriffe im Kontext von Kafka



- Producer / Consumer: Erzeuger und Verbraucher von Ereignisse / Nachrichten
- Records: Ereignisse und Nachrichten in Kafka
- Topics: Zusammengefasste Topics
- Partitionen: Unterteilung von Topics

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

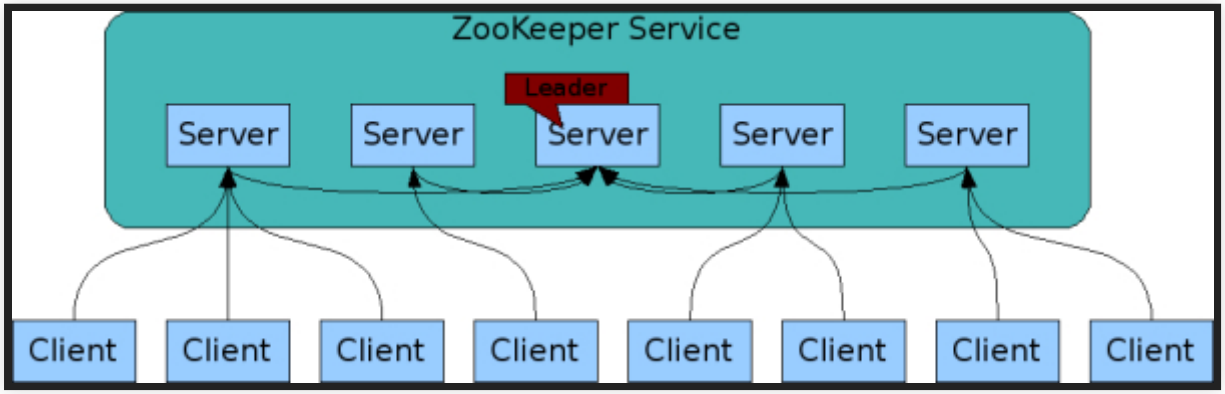
Kafka CLI

```
docker compose exec kafka kafka-topics.sh --create --topic examples --bootst
docker compose exec kafka kafka-topics.sh --describe --topic examples --boot
docker compose exec kafka kafka-topics.sh --list --bootstrap-server localhos
docker compose exec kafka kafka-console-producer.sh --topic examples --boots
docker compose exec kafka kafka-console-consumer.sh --topic examples --from-
```

- Erzeugt die Topic `examples`
- Gibt Details zur Topic `examples` aus
- Listet alle Topics auf
- Erzeugt einen Nachrichten-Produzenten in Form einer Konsole
- Liest alle Nachrichten aus

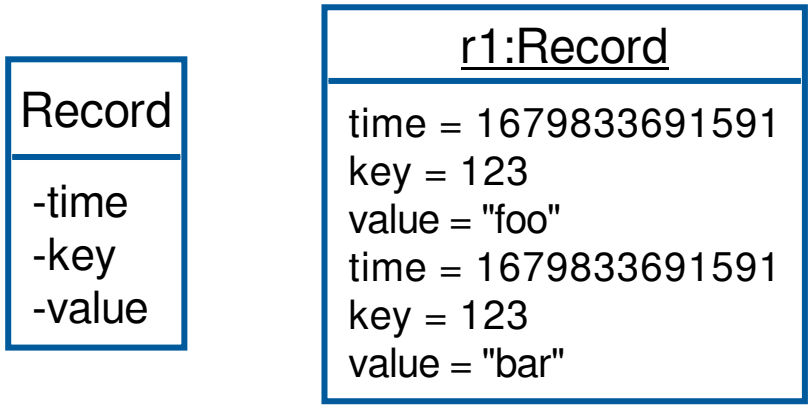
Zookeeper

- ZooKeeper ist ein zentraler Dienst zur Verwaltung von Konfigurationsinformationen, zur Namensgebung, zur verteilten Synchronisierung und zur Bereitstellung von Grouping Services
- Configuration, Naming, Synchronisierung und Grouping Services sind in verteilten Anwendungen häufig notwendig
- Zookeeper bietet eine fertige Lösung, die diese Funktionalitäten anbietet



Quelle: https://zookeeper.apache.org

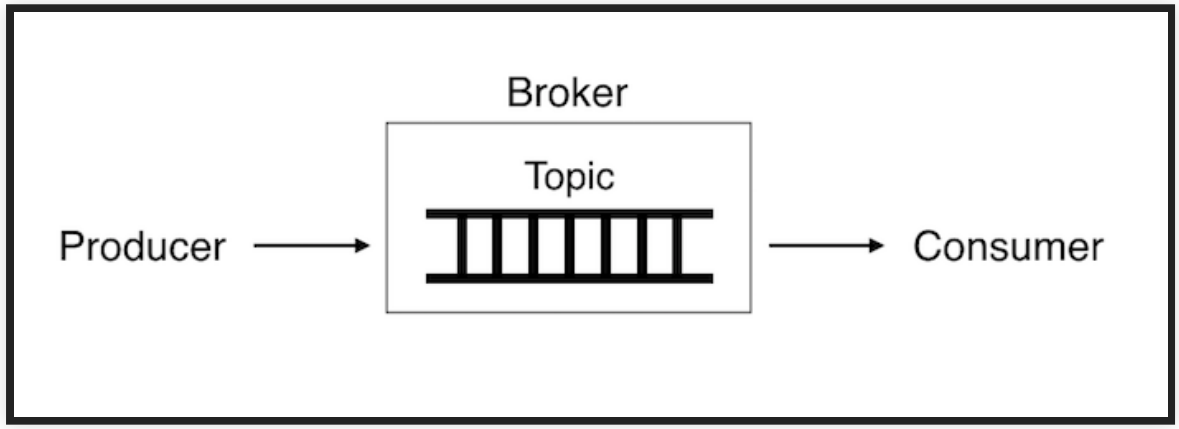
RECORDS



- Kafka organisiert Daten in Records
- Ein Record transportiert Werte als Value
- Kafka behandelt den Value als eine Black Box und interpretiert die Daten nicht
- Außerdem haben Records einen Schlüssel (Key) und einen Zeitstempel (Timestamp)
- Kafka speichert darüber hinaus keine Metadaten zu den Records

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

TOPICS



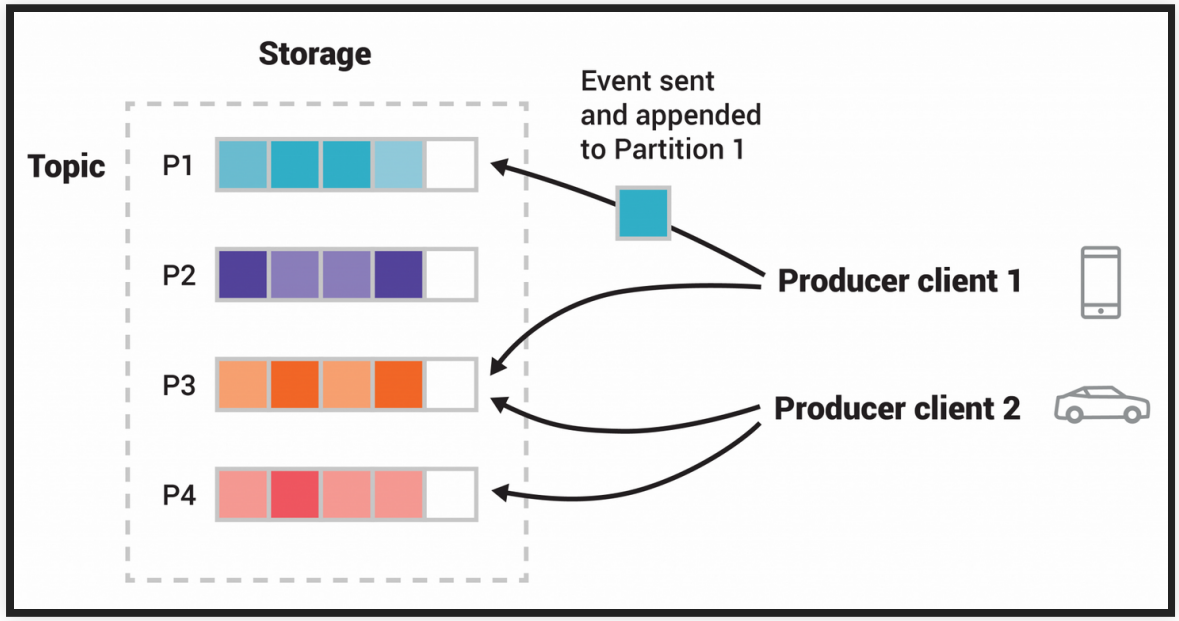
- Topics fassen Records zusammen.
- Producer senden Records an das Ende einer Topic, es entsteht eine Art Log
- Über Topics kann eine logische Separierung von Records erreicht werden, ähnliche wie bei einer Datenbanktabelle
- Ein Topic könnte z.B. alle Records zu Bestellungen enthalten

Quelle: https://hevodata.com/learn/kafka-topic/

- Werden Nachrichten in Topics platziert, die nicht existieren wird Kafka diese erstellen
- Topics können bei Bedarf nachträglich konfiguriert werden
- Zur Steuerung der Log-Größe eines Topics kann global die Retention Time od. Bytes konfiguriert werden sowie je Topic
    - Retention Time steuert nach wieviel Zeit Nachrichten entfernt werden, globale Konfiguration ist hier `log.retention.hours`, `log.retention.minutes` und `log.retention.ms`
    - Retention Byte steuert nach wieviel Bytes in einem Topic alte Nachrichten entfernt werden, globale Konfiguration ist hier `log.retention.bytes`

```
kafka:
  # ...
  environment:
    - KAFKA_CFG_LOG_RETENTION_HOURS=24
```

PARTITIONEN



- Topics sind in Partitionen unterteilt
- Wenn ein Producer einen neuen Record erstellt, wird er an eine Partition angehängt

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

- Kafka speichert für jeden Consumer den Offset für jede Partition
- Dieser Offset zeigt an, welchen Record in der Partition der Consumer zuletzt gelesen hat.
- Partitionen erlauben Producern das Anhängen neuer Records
- Producer profitieren davon, dass das Anhängen von Daten zu den effizientesten Operationen auf einem Massenspeicher gehört
- Außerdem sind solche Operationen sehr zuverlässig und einfach zu implementieren
- Über Partitionen hinweg ist die Ordnung nicht garantiert
- Das Lesen in einer Partition ist linear, über Partitionen hinweg ist es parallel

Mehr Partitionen haben verschiedene Auswirkungen.
Typisch sind Hunderte von Partitionen.

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

KEYS

Jeder Record kann einen Key haben

- Kafka verarbeitet Records in der richtigen Reihenfolge, wenn Sie einen Key haben
- Wichtig: Topics in Kafka sind in Partitionen unterteilt, für die Zuweisung eines Records auf eine Partition dient der Hash eines Keys, z.B. im Java-Client mittels *murmur2* ermittelt über die Formel `murmur2(key) % number_of_partitions`
- Beispiele für Keys: Key als ID für einen Auftrag in einem Topic, damit Zustandsänderungen in der richtigen Reihenfolge verarbeitet werden

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

- Bei Null-Schlüsseln wird der Standard-Partitionierer die Nachrichten automatisch auf alle verfügbaren Partitionen verteilen.
- UUIDs sind in der Praxis eine gute Variante für Keys
- **Wichtig:** Die Wahl des Schlüssels ist ein relevantes Designkriterium, je nach Producer-Konfiguration und Eigenschaften des Schlüssel entscheidet sich die Verteilung über Partitionen.

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

COMMIT

- Wenn Consumer einen Record verarbeitet haben, committen sie einen neuen Offset.
- So weiß Kafka jederzeit, welche Records welcher Consumer bearbeitet hat und welche noch bearbeitet werden müssen.
- Consumer Records können committen, bevor sie tatsächlich bearbeitet sind
- Ein Consumer kann ein Batch von Records committen, wodurch eine bessere Performance möglich wird, weil weniger Commits notwendig sind, bei Absturz können dadurch duplizierte Verarbeitungen entstehen
- Kafka unterstützt Exactly once, also eine garantiert einmalige Zustellung.

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

BEISPIEL MIT JAVA

Consumer zum Empfangen von Nachrichten

```
Properties config = new Properties();
config.put("client.id", InetAddress.getLocalHost().getHostName());
config.put("bootstrap.servers", "localhost:9092");
config.put("group.id", "foo");
config.put("key.deserializer", StringDeserializer.class.getName());
config.put("value.deserializer", StringDeserializer.class.getName());
```

- Grundlegend ist eine Sammlung von Konfigurationen notwendig, welche notwendig sind, damit Nachrichten konsumiert werden können.
- Hier findet sich eine Client-ID, für den die Offsets verwaltet werden
- Host und Port für die Server, hier können mehrere angegeben werden
- Eine Group-ID, dazu später mehr
- Deserializer zum Lesen der Keys und Values von Records

```
try(Consumer<String, String> consumer = new KafkaConsumer<>(config)) {
  consumer.subscribe(List.of(topic));
  while (true) {
    ConsumerRecords<String, String> records =
      consumer.poll(Duration.ofDays(Long.MAX_VALUE));
    records.forEach(el -> logger.info(el.key() + ": " + el.topic()));
    consumer.commitSync();
  }
}
```

- Zum Lesen wird als nächstes der Consumer erzeugt, welcher die Konfiguration benötigt
- Anschließend wird der Consumer an verschiedenen Topics registriert
- Je Verarbeitungsschritt wird nach Nachrichten gefragt
- Die Nachrichten verarbeitet, hier ausgegeben
- ... sowie der Commit realisiert um den Offset zu verändern

Producer zum Senden von Nachrichten

```
Properties config = new Properties();
config.put("client.id", InetAddress.getLocalHost().getHostName());
config.put("bootstrap.servers", "localhost:9092");
config.put("acks", "all");
config.put("key.serializer", StringSerializer.class.getName());
config.put("value.serializer", StringSerializer.class.getName());
```

- Hier sind ebenso Konfigurationen notwendig, welche ähnlich sind
- Anzahl der Acknowledgments die der Leader benötigt, bevor er eine Nachricht-Anfrage als vollständig betrachtet (`0`, `1`, `all`)
- Für Keys und Values wird ein Serialisier benötigt

```
try(Producer<String, String> producer = new KafkaProducer<>(config)) {
  ProducerRecord<String, String> record =
    new ProducerRecord<>("test-topic", "test", "abc");
  Future<RecordMetadata> future = producer.send(record);
  RecordMetadata metadata = future.get();
  logger.info(metadata.toString());
}
```

- Wie beim Consumer wird anschließend ein Producer erzeugt
- Nachrichten werden anschließend in Records verpackt, unter Angabe einer Topic (vgl. `test-topic`), eines Keys (vgl. `test`) und eines Values (vgl. `abc`)
- Anschließend wird der Record versendet
- Im Fall einer synchronen Verarbeitung, kann mittels `get` gewartet werden

```
try(Producer<String, String> producer = new KafkaProducer<>(config)) {
  ProducerRecord<String, String> record =
    new ProducerRecord<>(topic, "test", "abc");
  producer.send(record,
      (metadata, e) -> logger.info(metadata.toString()));
}
```

Alternativ ist eine asynchrone Verarbeitung möglich, indem hier ein Callback-Handler hinterlegt wird.

Administration von Topics

Erzeugen eines Topics

```
Properties properties = new Properties();
properties.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092")
try(Admin admin = Admin.create(properties)) {
  NewTopic topic = new NewTopic("test-topic", 1, (short) 1);
  CreateTopicsResult result = admin.createTopics(Collections.singleton(topic
  KafkaFuture<Void> future = result.values().get("test-topic");
  future.get();
}
```

Erzeugt ein Topic `test-topic` mit einer Partition und einer Replikation.

Konfiguration von Retention Time

```
// ...
NewTopic topic = new NewTopic("test-topic", 1, (short) 1);
Map<String, String> config = new HashMap<>();
config.put(TopicConfig.RETENTION_MS_CONFIG, String.valueOf(60*60*1000));
topic.configs(config);
CreateTopicsResult result = admin.createTopics(Collections.singleton(topic))
KafkaFuture<Void> future = result.values().get("test-topic");
future.get();
```

Über Konfigurationen lassen sich Topics spezifisch anpassen, hier wird die Retention Time auf eine Stunde reduziert.

Ändern eines Topics

```
Properties properties = new Properties();
properties.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092")
try(Admin admin = Admin.create(properties)) {
  Map<String,NewPartitions> map = new HashMap<>();
  map.put("test-topic", NewPartitions.increaseTo(2));
  CreatePartitionsResult result = admin.createPartitions(map);
  KafkaFuture<Void> future = result.values().get("test-topic");
  future.get();
}
```

Ändert das Topic `test-topic` auf zwei Partitionen.

Nutzen Sie das bereitgestellte Projekt-Archiv apache-kafka-java.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

BEISPIEL MIT NODEJS

Wie in Java lässt sich die selbe Variante auch in anderen Programmiersprachen realisieren. Folgend ein kleines Beispiel mit NodeJS und KafkaJS.

```
const { Kafka } = require('kafkajs')
const kafka = new Kafka({
  clientId: 'example-consumer',
  brokers: ['localhost:9092']
})
```

Grundlage für das Nachrichten Lesen und Empfangen ist eine Instanz der Klasse `Kafka`.

Consumer

```
const consumer = kafka.consumer({ groupId: 'test-group' })
const run = async () => {
  await consumer.connect()
  await consumer.subscribe({ topic: 'test-topic', fromBeginning: true })
  await consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
      console.log({ partition, offset: message.offset,
        value: message.value.toString()})
    }
  })
}
run().catch(console.error)
```

- Dabei finden sich wie in Java Konzepte zum Erzeugen des Consumers
- ... Verbinden und Subscriben
- ... sowie eine Logik zum wiederholten Empfangen von Nachrichten
- Dabei erzeugt `eachMessage` einen Commit nach jeder Nachricht
- Alternativ wäre `eachBatch` möglich, bei denen mehr Steuerungsmöglichkeiten gegeben wären

Producer

```
const producer = kafka.producer({ createPartitioner:
  Partitioners.DefaultPartitioner })
const run = async () => {
  await producer.connect()
  await producer.send({ topic: 'test-topic',
    messages: [
      { value: JSON.stringify({time: new Date().getTime(),
        rnd: Math.random()}) } ]
  })
}
run().catch(console.error)
```

Ebenso vergleichbar ist der Producer, welcher hier genau eine Nachricht mit einem zufälligen Wert in Form eines JSON-Strings als Value versendet.

POLLING

Wie kommen Daten zum Consumer?

- Consumer pollen Daten
- Polling der Records zu den Consumern schützt davor, unter zu viel Last zu geraten, wenn gerade eine große Zahl von Records verarbeitet werden muss
- Die Consumer können selbst entscheiden, wann sie die Records verarbeiten
- Bibliotheken wie das im Beispiel verwendete Spring Kafka pollen im Hintergrund neue Records

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

REPLIKATION

Hochverfügbarkeit und Ausfallsicherheit mit Kafka

- Partitionen speichern Daten und können über Server verteilt werden.
- Jeder Server bearbeitet dann einige Partitionen, und erlaubt somit Lastverteilung
- Die Partitionen können repliziert werden, die Daten liegen dann auf mehreren Servern für die Steigerung der Ausfallsicherheit
- Die Anzahl N der Replicas kann konfiguriert werden.
- Beim Schreiben lässt sich festlegen, wie viele In-Sync-Replicas Änderungen committen müssen
- Bei $N=3$ Replicas und zwei In-Sync-Replicas bleibt der Cluster einsatzfähig, wenn eines der drei Replicas ausfällt; es kann dann auf zwei Replicas geschrieben werden
- Beim Ausfall einer Replica gehen keine Daten verloren, da jedes Schreiben mindestens auf zwei Replicas erfolgreich gewesen sein muss

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

Leader und Follower

- Replikation ist so umgesetzt, dass ein Leader schreibt und die restlichen Replicas als Follower schreiben
- Der Producer schreibt direkt an den Leader
- Mehrere Schreiboperationen können in einem Batch zusammengefasst werden, es dauert dann länger, bis ein Batch vollständig ist und die Änderungen tatsächlich abgespeichert werden; ist jedoch durch den Durchsatz performanter

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

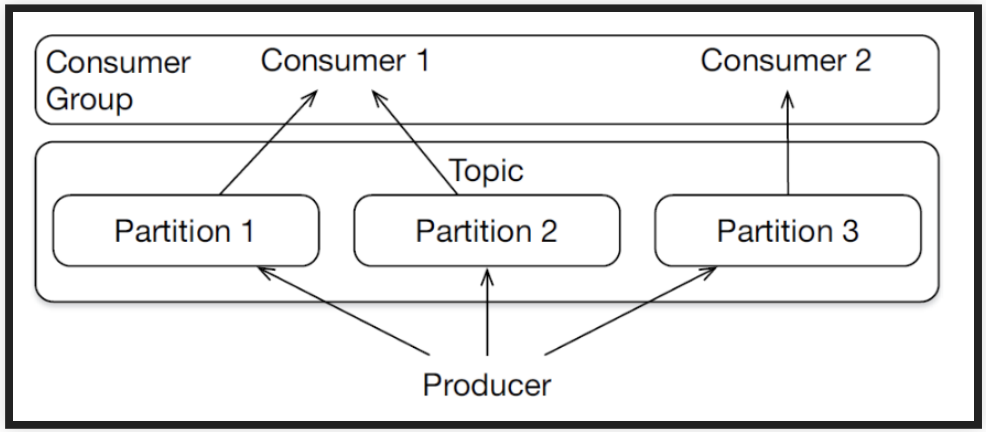
Schreibenwiederholen

- Wenn eine Schreiboperation nicht erfolgreich war, kann der Producer über die API festlegen, dass die Übertragung erneut versucht wird
- Die Standardeinstellung ist, dass das Verschicken eines Records nicht wiederholt wird, dadurch können Records verloren gehen
- Wenn die Übertragung mehrfach erfolgt, kann es vorkommen, dass der Record trotz des Fehlers erfolgreich übertragen wurde
- In diesem Fall würde es ein Duplikat geben, womit der Consumer umgehen können muss
- Eine Möglichkeit ist es, den Consumer so zu entwickeln, dass er idempotente Verarbeitung anbietet
- Damit ist gemeint, dass der Consumer in demselben Zustand ist, egal wie oft der Consumer einen Record verarbeitet
- Beispielsweise kann der Consumer bei einem Duplikat feststellen, dass er den Record bereits bearbeitet hat und ihn ignorieren

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

CONSUMER GROUPS

Consumer sind in Consumer Groups organisiert.



- Jede Partition hat genau einen Consumer in der Consumer Group. Ein Consumer kann für mehrere Partitionen zuständig sein.
- Ein Consumer bekommt Nachrichten von einem oder mehreren Partitionen

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

- Die Abbildung zeigt ein Beispiel:
  - Der Consumer 1 bekommt die Nachrichten aus den Partitionen 1 und 2.
  - Der Consumer 2 erhält die Nachrichten aus Partition 3.
  - Wenn ein Consumer eine Nachricht mit einem Key bekommt, erhält er auch alle Nachrichten mit demselben Key, weil alle zur selben Partition gehören.
  - Ebenso ist die Reihenfolge der Nachrichten pro Partition festgelegt.
  - Somit können Records parallel bearbeitet werden und gleichzeitig ist die Reihenfolge bei bestimmten Records garantiert.
  - Das gilt natürlich nur, wenn die Zuordnung stabil bleibt. Wenn zum Beispiel zur Skalierung neue Consumer zu der Consumer Group dazukommen, dann kann sich die Zuordnung ändern.

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

- Die maximale Anzahl der Consumer in einer Consumer Group ist gleich der Anzahl der Partitionen, weil jeder Consumer mindestens für eine Partition verantwortlich sein muss.
- Idealerweise gibt es mehr Partitionen als Consumer, um bei einer Skalierung noch weitere Consumer hinzufügen zu können.
- Wenn jeder Consumer alle Records aus allen Partitionen bekommen soll, dann muss es für jeden Consumer eine eigene Consumer Group mit nur einem Mitglied geben.

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

PERSISTENZ

Kafka ist eine Mischung aus einem Messaging-System und einem Datenspeicher

- Die Records in den Partitionen können von Consumern gelesen und von Producern geschrieben werden.
- Sie werden dauerhaft gespeichert.
- Die Consumer speichern lediglich ihren Offset.
- Ein neuer Consumer kann daher alle Records verarbeiten, die jemals von einem Producer geschrieben worden sind, um so seinen eigenen Zustand auf den aktuellen Stand zu bringen.

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

LOG COMPACTION

Kafka muss mit der Zeit immer mehr Daten speichern

- Einige Records werden jedoch irgendwann irrelevant.
- Wenn ein Kunde mehrfach umgezogen ist, will man vielleicht nur noch den letzten Umzug als Record in Kafka halten
- Log Compaction kann dazu genutzt werden, dass alle Records mit demselben Key bis auf den letzten entfernt werden
- Deswegen ist die Wahl des Keys sehr wichtig und muss aus fachlicher Sicht betrachtet werden, um auch mit Log Compaction noch alle relevanten Records verfügbar zu haben

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

AVRO

- Avro ist ein Datenformat, das zusammen mit Kafka und Big-Data-Lösungen aus dem Hadoop-Bereich genutzt wird
- Avro ist ein binäres Protokoll, bietet ebenso eine JSON-basierte Repräsentation an
- Avro-Bibliotheken gibt es u.a. für Python, Java, C#, C++ und C
- Avro verwendet Schematas zur Beschreibung der Datenformate, welche im Projekt oder in einem Repository bereitgestellt werden können
- Aus den Schema wird in Quellcode überführt und erlaubt anschließend das Arbeit z.B. in Java mit den zugehörigen Klassen
- Beim Lesen können Daten in andere Formate konventiert werden um Schema-Evolution zu unterstützen.

Quelle: Eberhard Wolff. Das Microservices-Praxisbuch. dpunkt.verlag, Heidelberg, 2018, 1. Auflage.

Beispiel eines Schemas: user.avsc

```
{
    "namespace": "de.thi.informatik.edi.dto",
    "type": "record",
    "name": "User",
    "fields": [
        { "name": "name", "type": "string" },
        { "name": "favorite_number", "type": [ "int", "null" ] },
        { "name": "favorite_color", "type": [ "string", "null" ] }
    ]
}
```

- Erzeugt einen neuen Record im angegeben Namespace mit dem Namen `User`
- Der Record hat die drei Attribute `name`, `favorite_number` und `favorite_color`
- Optionale Werte werden über `null`-Typen realisiert

Ergebnis in Java wäre eine Klasse der folgenden Form

(einige Methoden wurden ausgelassen)

User
-name: CharSequence -favorite_number: Integer -favorite_color: CharSequence
+User(name: CharSequence, favorite_number: Integer, favorite_color: CharSequence) +getName(): CharSequence +setName(value: CharSequence) +getFavoriteNumber(): Integer +setFavoriteNumber(value: Integer) +getFavoriteColor(): CharSequence +setFavoriteColor(value: CharSequence) +newBuilder(): Builder +getSchema(): Schema

Generierte Klassen in Java verwenden

```
User firstUser = new User();
firstUser.setName("Alyssa");
firstUser.setFavoriteNumber(256);
User secondUser = User.newBuilder()
  .setName("Charlie")
  .setFavoriteColor("blue")
  .setFavoriteNumber(null)
  .build();
```

- Nutzer lassen sich auf den üblichen Weg erstellen
- ... oder einen Builder
- Builder validieren die gesetzten Daten und initialieren alle Felder; sind damit weniger performant

Serialisierung

```
DatumWriter<User> userDatumWriter =
  new SpecificDatumWriter<User>(User.class);
try(DataFileWriter<User> dataFileWriter =
  new DataFileWriter<User>(userDatumWriter)) {
  dataFileWriter.create(firstUser.getSchema(), new File("user.avro"));
  dataFileWriter.append(firstUser);
  dataFileWriter.append(secondUser);
} catch (IOException e) {
  e.printStackTrace();
}
```

- Erzeugt einen DataFileWriter
- ... welcher anschließend mehrere Objekte im AVRO-Format in die Datei `user.avro` serialisiert

Deserialisierung

```
DatumReader<User> userDatumReader =
  new SpecificDatumReader<User>(User.class);
try(DataFileReader<User> dataFileReader =
  new DataFileReader<User>(new File("user.avro"), userDatumReader)) {
  User user = null;
  while (dataFileReader.hasNext()) {
    user = dataFileReader.next(user);
    System.out.println(user);
  }
} catch (IOException e) {
  e.printStackTrace();
}
```

- Erzeugt einen DataFileReader
- ... welcher anschließend mehrere Objekte im AVRO-Format aus der Datei `user.avro` deserialisiert

Beispiele finden sich in folgenden Paket

Nutzen Sie das bereitgestellte Projekt-Archiv apache-kafka-avro.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

ABSICHERUNG / ZUGANGSKONTROLLE

- Absicherung über Verschlüsselung mit SSL/TLS
- Authentifizierung über SSL oder SASL
  - SASL/PLAINTEXT speichert Nutzername / Passwort im Kafka-Cluster, SSL wäre zwingend empfehlenswert
  - SASL/SCRAM speichert "salted hashs" im Zookeeper, SSL wäre ebenfalls zwingend empfehlenswert
  - SASL/GSSAPI basiert auf Kerberos Ticket mechanismen und stellt eine sichere Möglichkeit zur Authentifizierung dar
- HTTP Basic Authentication ist ebenfalls möglich
- Autorisation über ACL

Configuration eines Servers und Clients mit SASL / PLAIN

EXAMPLE_WEB: apache-kafka-security

SCALING

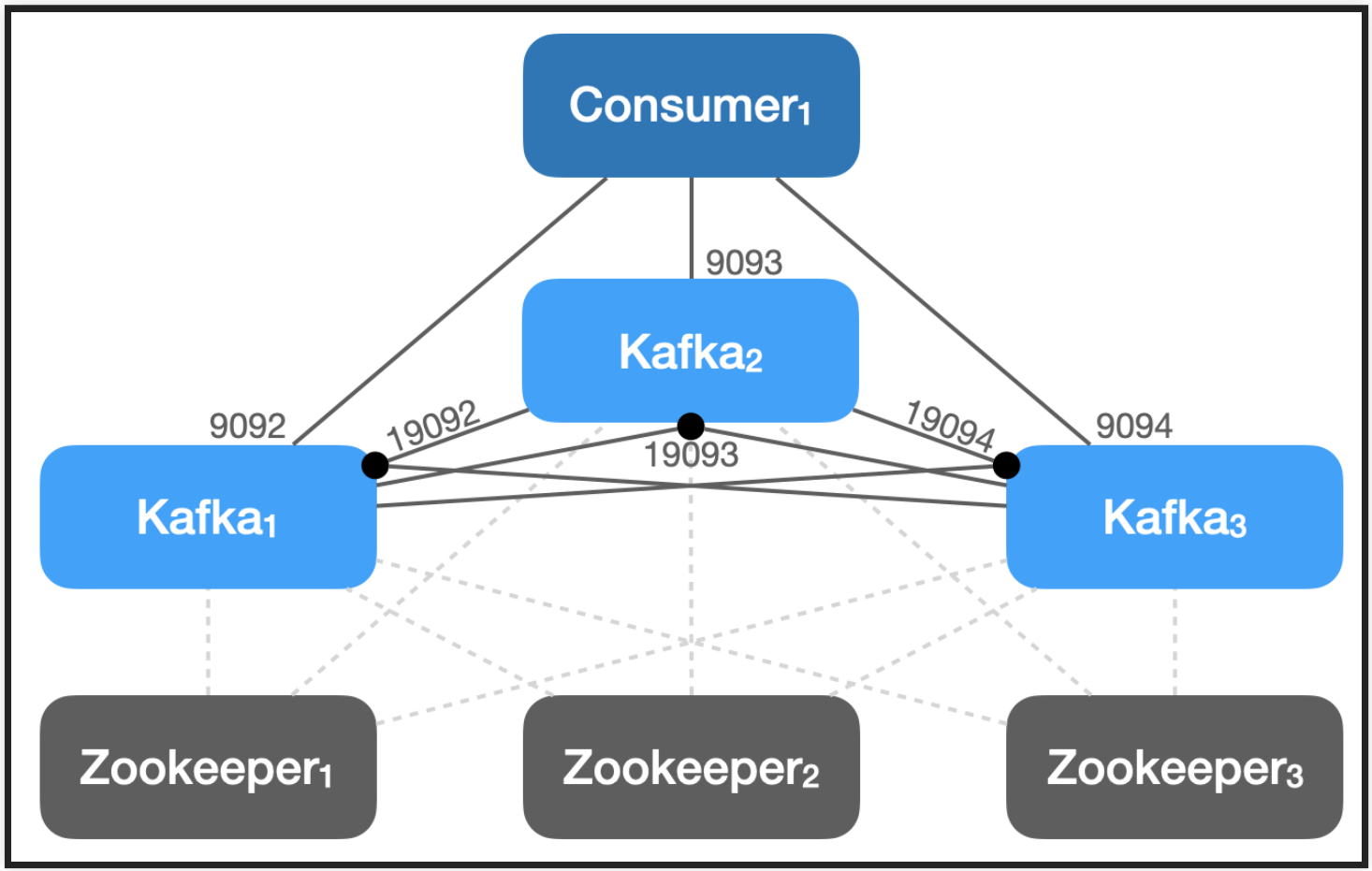
Wie unterstützt Kafka die Skalierung?

Grundlage hier sind Partitionen und Replikationen. Topics werden in verschiedene Partitionen aufgeteilt, Partitionen werden Repliziert im Cluster und Consumer werden auf Partitionen zugewiesen.

Beispiel eines Kafka-Clusters zum Experimentieren

Nutzen Sie das bereitgestellte Projekt-Archiv apache-kafka-scale.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Beispiel Cluster



- Jeder Kafka-Knoten erhält eine eigene Broker-ID
- ... definiert zwei Kommunikationsentpunkte
    - externer Zugriff, z.B. durch Clients (hier über Port 9092, 9093 und 9094)
    - interner Zugriff, insbesondere für andere Kafka-Knoten (hier über Port 19092, 19093 und 19094)
- Je Kommunikationskanal kann das Sicherheitslevel definiert werden und ob z.B. SSL/TLS genutzt werden soll, im Beispiel ist dies ungesichert