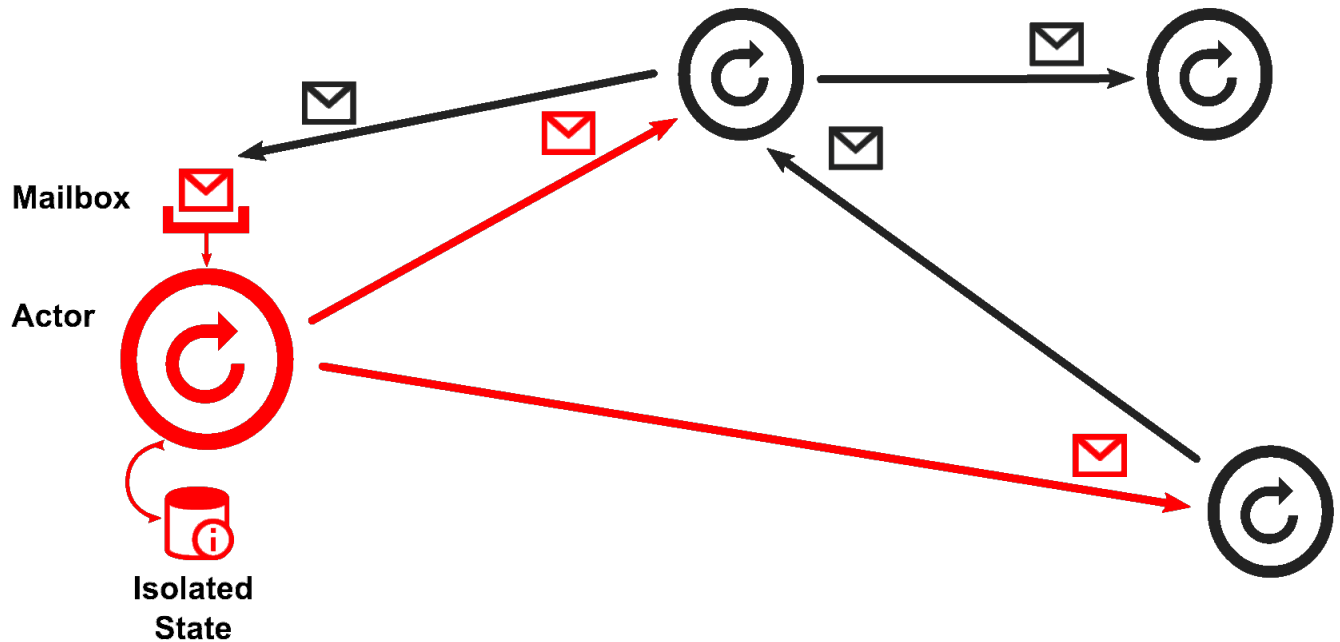
innFactory

Anwendung der funktionalen Programmierung mit Scala

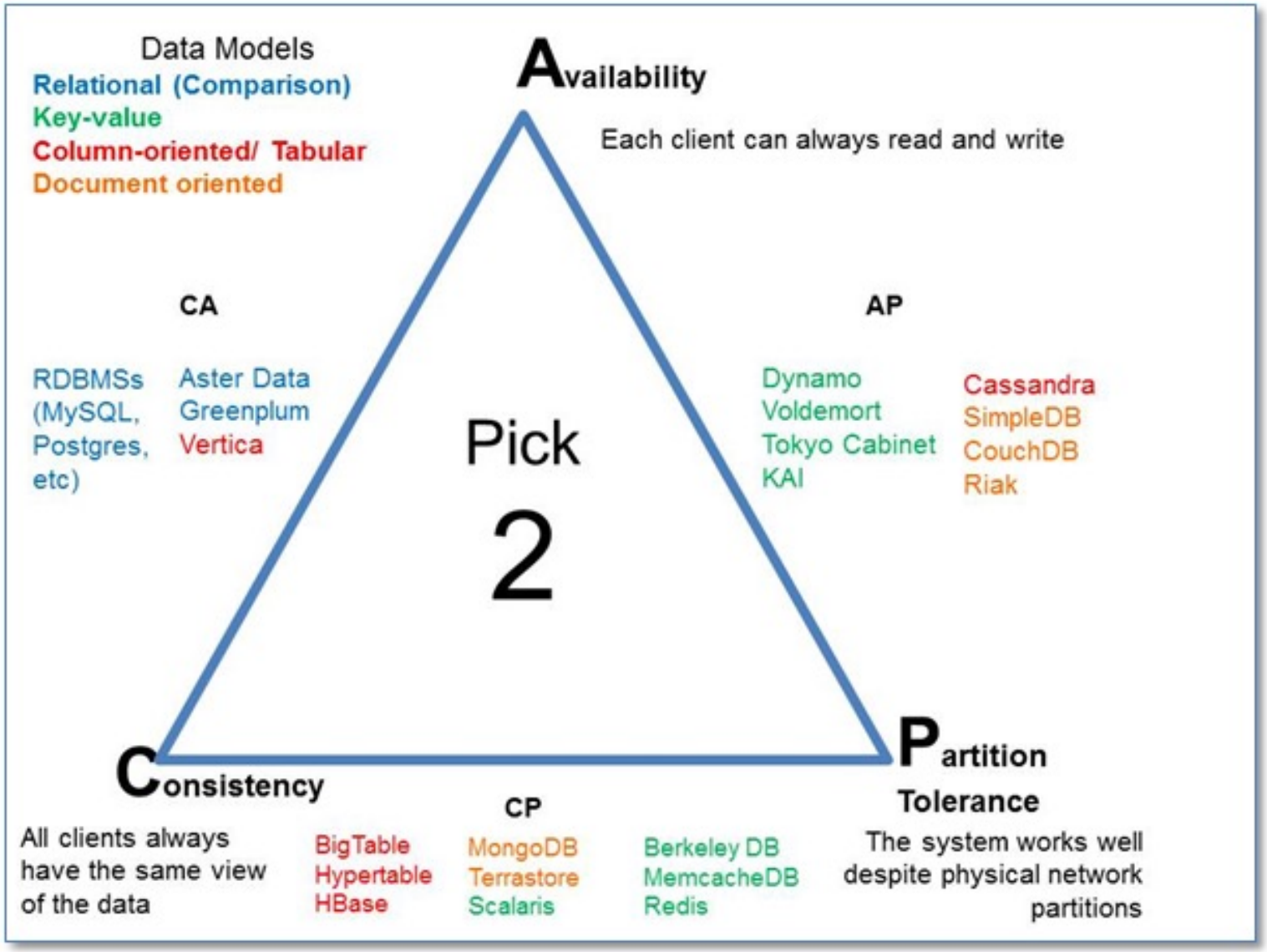
TH Rosenheim - SoSe 2025

innFactory

Pekko Persistence



Pekko Persistence
CAP-Theorem



Datenbanken 2
@ Master ;-)

Pekko Persistence
ACID

ACID ist ein Transaktionsmodell, das strengen Regeln folgt. Es gewährleistet Datenzuverlässigkeit und -konsistenz.

Das Akronym steht für:

- **Atomic:** Jede Transaktion wird vollständig abgeschlossen, bevor sie fortgesetzt wird. Kann eine Transaktion nicht fehlerfrei abgeschlossen werden, wird sie in den vorherigen Zustand zurückgesetzt, um die Datengültigkeit sicherzustellen.
- **Consistent:** Eine Transaktion behält die Struktur einer Datenbank bei.
- **Isolated:** Transaktionen werden unabhängig voneinander ausgeführt, um Konflikte zu vermeiden.
- **Durable:** Abgeschlossene Transaktionen bleiben auch nach Systemausfällen bestehen.

Pekko Persistence
BASE

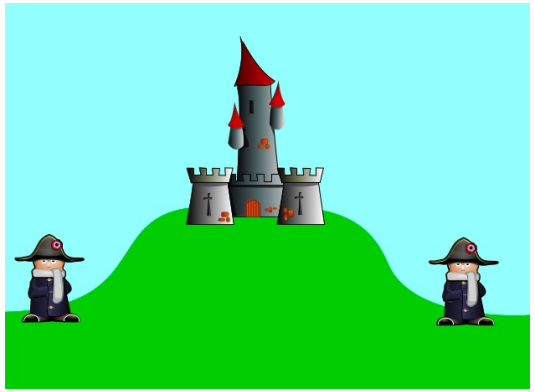
BASE ist ein flexibles Transaktionsmodell. Verfügbarkeit und Skalierbarkeit stehen bei BASE über Konsistenz.

Das Akronym steht für:

- **Basically Available** Hochverfügbar durch Datenreplikation über mehrere Knoten.
- **Soft State** Ermöglicht die Änderung von Datenwerten im Laufe der Zeit.
- **Eventually consistent:** Garantiert die Datenkonsistenz über einen längeren Zeitraum. Zwischenlesevorgänge können fehlerhafte Werte anzeigen.

Pekko Persistence
Aktordaten speichern, aber richtig.

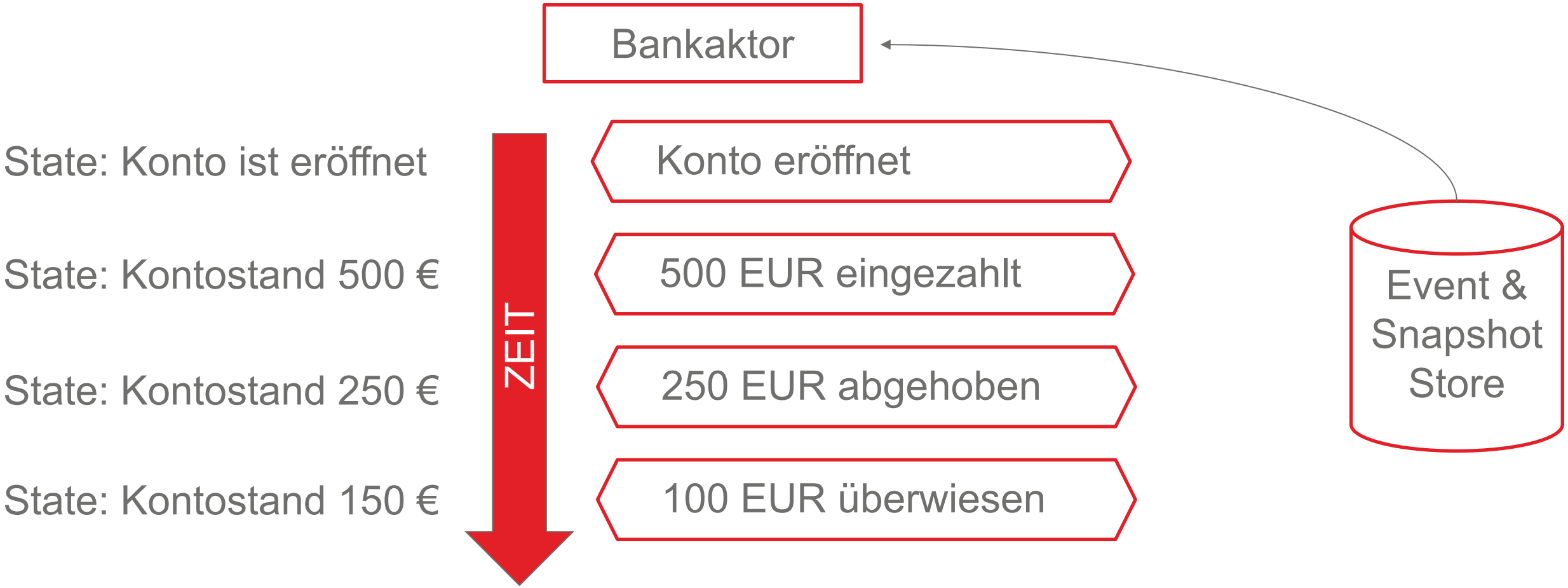
- Der Ablauf oder Zustand von Programmen wird nicht mehr als Zustand, sondern als Reihe von Events definiert
- CRUD (Create, Read, Update, Delete) reicht für Microservices bzw. hochparalle Systeme häufig nicht aus.
- ACID (atomicity, consistency, isolation, durability) ist für Transaktionen mit mehrerer Datenquellen nicht möglich. BASE (Basically Available, Soft state, Eventual consistency)
- Früher oder später wird man mit dem 2-Generals-Problem konfrontiert.
  - 2 Generale wollen die Stadt angreifen, können diese aber nur einnehmen, wenn sie zeitgleich angreifen. Die einzige Möglichkeit miteinander zu kommunizieren sind Nachrichten, die sie um die Stadt herum verschicken können. Diese können aber auch fehlschlagen. Dieses Problem führt zu einem unlösbaren Problem.



Pekko Persistence
Event Sourcing

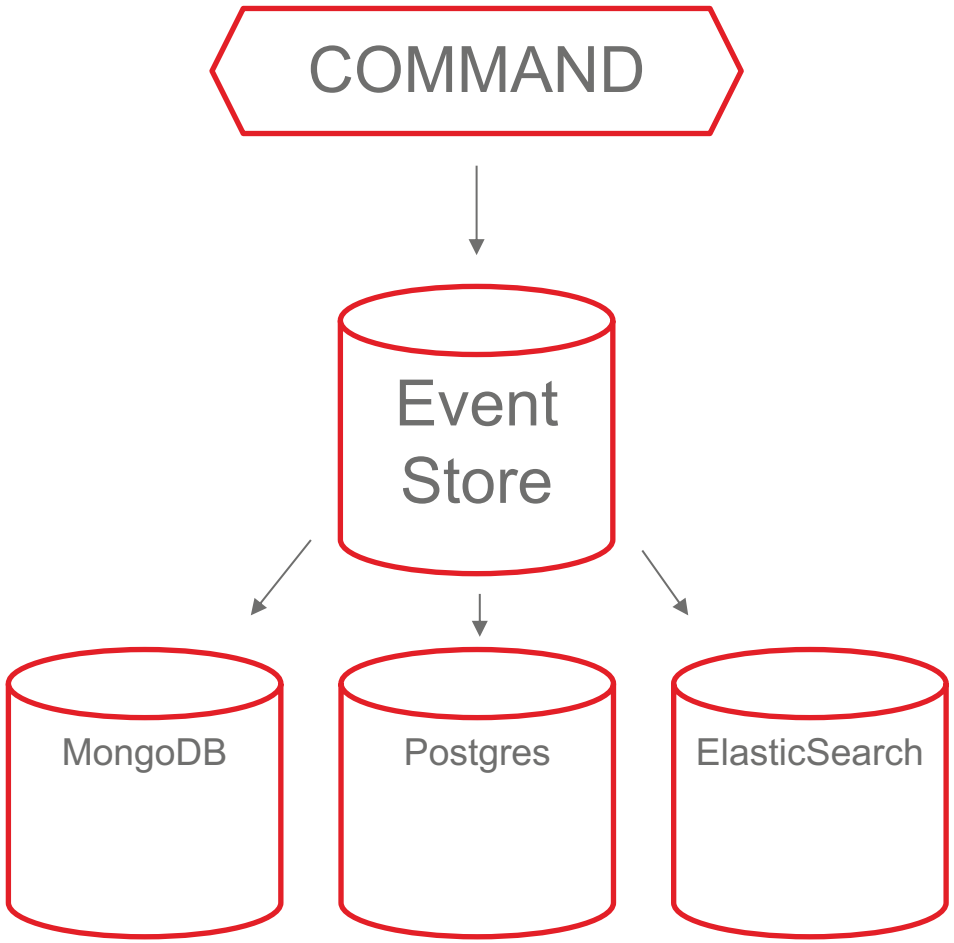
- Es wird nicht mehr der Zustand gespeichert, sondern die Events, die zu diesem Zustand führen.
- Der Zustand wird auf Basis der Events ausgerechnet.
- Im Kontext von Pekko bedeutet dies, das man seine Mailbox speichern muss, um aus allen Nachrichten den aktuellen Zustand wiederherstellen zu können.
- Die Erweiterung Pekko Persistence bietet diese Funktionalität und speichert die Mailboxdaten oder auch Snapshots (Zwischenspeicher)

Pekko Persistence
Event Sourcing

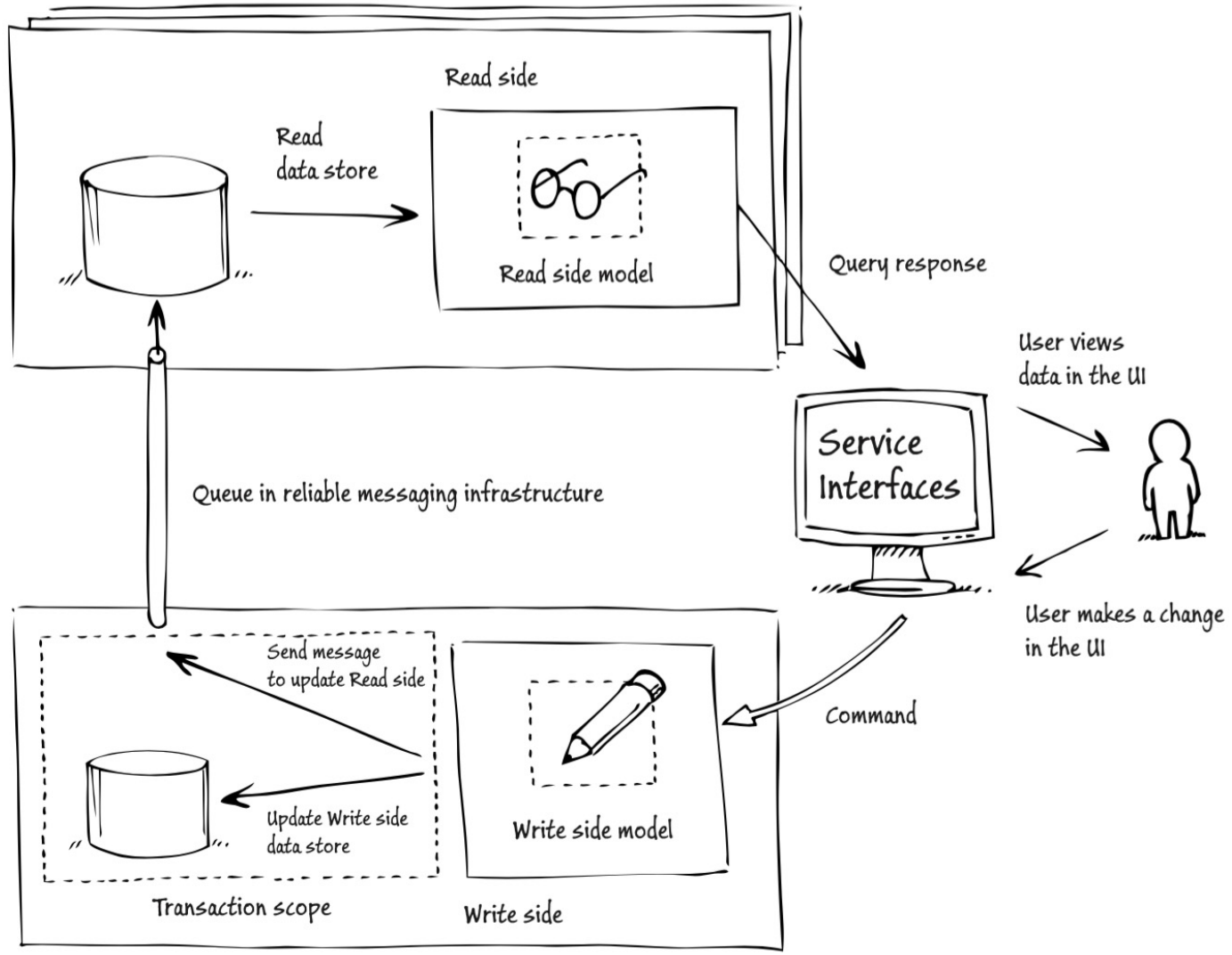


Pekko Persistence
CQRS

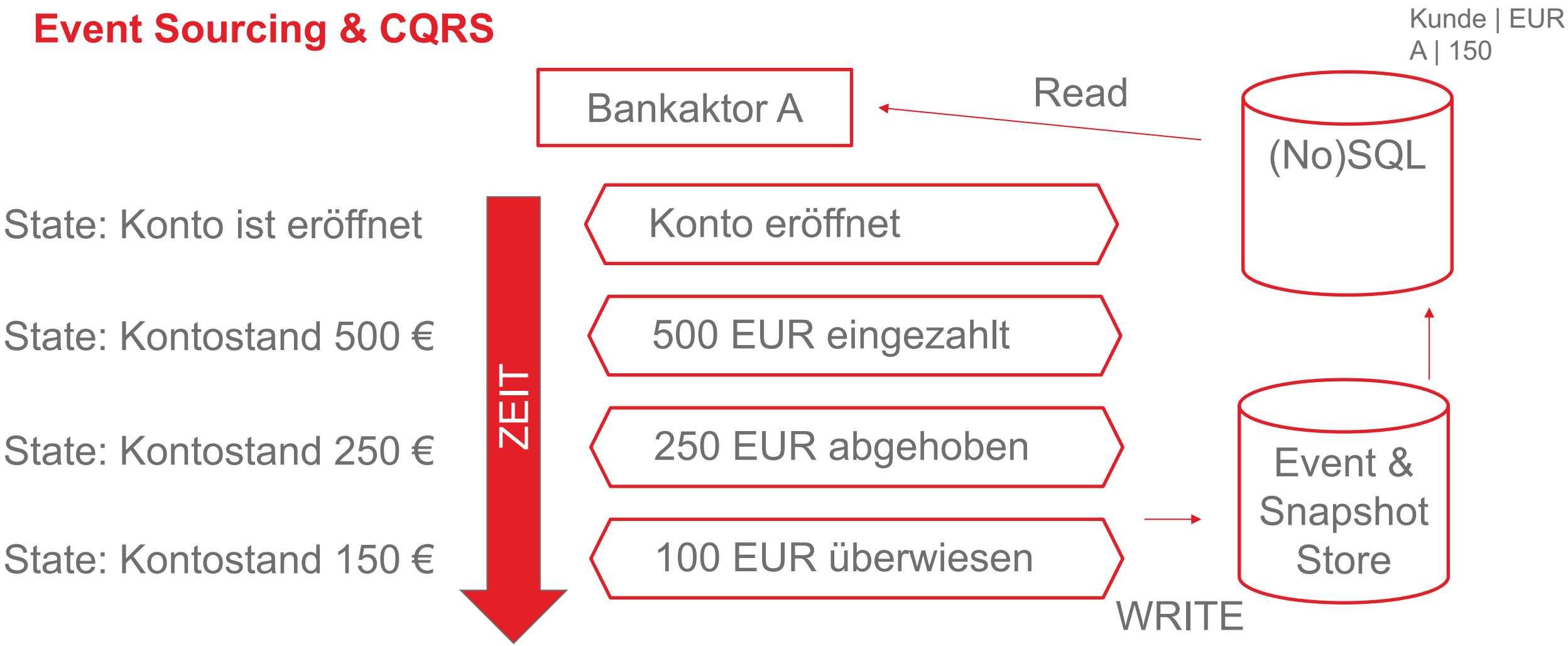
- Command Query Responsibility Segregation (CQRS) teilt eure Applikation in eine Lese- und eine Schreibseite.
- Die Commandseite verarbeitet create/update/delete requests
- Die Queryseite verarbeitet die Leseanfragen, indem sie Datenquellen anfragen, die auf Basis der Commandseite entstehen
- CQRS findet man häufig in Kombination mit Event Sourcing
- Wenn die Businesslogik über mehrere Services hinweg läuft gibt es ein Konzept der SAGA (out of scope)



Pekko Persistence
Event Sourcing & CQRS



Pekko Persistence
Event Sourcing & CQRS



Pekko Persistence
Überblick

- **Persistenz:** ermöglicht es Aktoren, ihren Zustand zwischen den Sitzungen zu speichern.
- **Journal:** speichert die Ereignisse, die den Zustand des Aktors beeinflussen.
- **Snapshot:** speichert einen Snapshot des aktuellen Zustands des Aktors, wodurch die Wiederherstellung beschleunigt wird.
- **Event Sourcing:** Nutzung von Ereignissen zur Rekonstruktion des Aktorzustands.
- **Rekonstruierung:** Zustandswiederherstellung durch Abspielen von gespeicherten Ereignissen oder Verwendung von Snapshots.
- Die Eventdaten können in verschiedensten (No)SQL-Datenbanken abgelegt werden: Cassandra, JDBC, MongoDB, DynamoDB, [..]

Pekko Persistence
Code - Event Sourcing

```scala
import akka.actor.typed.ActorRef
import akka.actor.typed.Behavior
import akka.persistence.typed.scaladsl.{EventSourcedBehavior, Effect, PersistenceId}

object Counter {
  def apply(): Behavior[CounterCommand] = EventSourcedBehavior[CounterCommand, CounterEvent, Int](
    PersistenceId.ofUniqueId("counter-1"),
    0, // initial state
    commandHandler,
    eventHandler
  )

  private def commandHandler(state: Int): (CounterCommand, ActorRef[CounterEvent]) => Effect[CounterEvent, Int] = {
    case Increment =>
      Effect.persist(CounterIncremented)
    case GetCount =>
      println(s"Current count: $state")
      Effect.none
  }

  private def eventHandler(state: Int, event: CounterEvent): Int = event match {
    case CounterIncremented => state + 1
  }
}
```

innFactory

Übung

Pekko Persistence
Übung

- Ziel der Übung ist es Temperatursensoren mit Akka Persistence abzubilden, die ihre Temperaturwerte in eine Datenbank als Event Stream schreiben. Dazu existiert bereits eine Main Applikation die zufällige Werte für New York, Rio, Rosenheim generiert.
- Diese Werte sollen darüber hinaus als Sensordaten getaggt werden und über einen Stream (Persistence Query) direkt nach dem schreiben wieder zentral gelesen werden.
- Es muss somit der persistente Aktor für den Sensor und die SensorDataView implementiert werden.
- Als Datenbank verwenden wir die In-Memory Datenbank h2. Zusatz: Umbau auf Persistence Cassandra + Cassandra via Docker
- https://pekko.apache.org/docs/pekko/1.1/persistence-query.html

Streams
Was sind Streams?

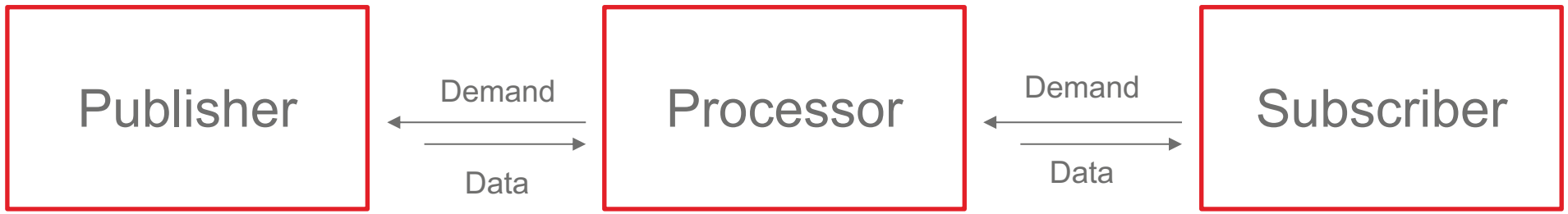
- Streams sind Datensequenzen von Daten
- Die Größe des Streams ist oft nicht bekannt oder kann auch unendlich groß sein.
- Oft sind Streams größer wie der Arbeitsspeicher / Speicher der Maschine selbst und so kann immer nur ein Teil der Daten verarbeitet werden. (z.B. ein „Stream“ von IoT Daten, Stream von Bildern)
- Use-Cases für Streams
  - Verarbeitung von live events (zum Beispiel IoT Daten, Tweets, Nachrichten, ..)
  - ETL-Systeme (Extract, Transform, Load)
  - Streaming media (audio, video, Text (LLMs))
- Reactive Streams sind „asynchrone Streams“

Streams
Reactive Streams

- „Reactive Streams is a standard for asynchronous stream processing with non-blocking backpressure, providing a way to handle streams of data in a reactive and efficient manner.“
- Wofür brauchen wir Reactive Streams? Für Fragestellung wie:
  - Wie kann ich Streams asychron verarbeiten?
  - Wie kann ich verhindern, dass ein Stream überlastet wird, wenn es einen langsamen Consumer gibt
  - Wie können wir garantieren, dass die Reihenfolge trotz des async Prozess erhalten bleibt?
- Komponenten eines Reactive Streams:
  - **Publisher:** Publisht Daten in den Stream und **Subscriber:** Verarbeitet Daten aus dem Stream
  - **Processor:** Arbeitet sowohl als Publisher, als auch als Subscriber
  - **Subscription:** Verbindet einen Subscriber mit einem Publisher

Streams
Backpressure

- Backpressure verwendet einen pull/push Mechanismus
- Subscripers schicken ihre Kapazität (Demand) upstream an den Publisher
- Publisher empfangen die Kapazität der Subscriber und schicken die Daten wenn verfügbar downstream
- Publisher dürfen nicht mehr Downstream schicken, wie die Subscriber angefordert haben



Streams
Pekko Streams

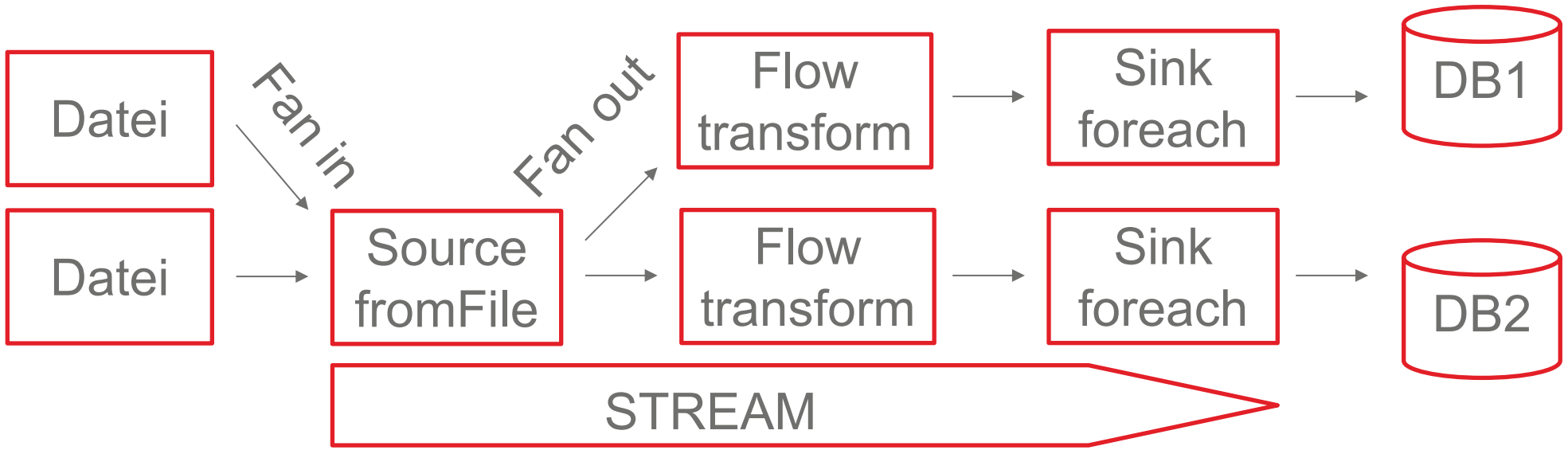
- Pekko Streams: Pekko Streams ist auf Basis der Reactive Stream interraces aufgebaut. Pekko Streams bieten eine Low-Level API an
- Pekko Actors abstrahiert mit Pekko Streams alle Konzepte weg, die wir bisher gelernt haben.
- Pekko Stream ist eine Kette von Prozessschritten (Processing Stage). Eine Stage besteht aus 0-n Input und 0-n Output. Eine Stage muss mindestens einen Input oder Output haben. Normalerweise läuft die Kette synchron von Anfang bis Ende ab. Eine asynchrone Verarbeitung ist aber möglich.
- Aufbau eines Streams:
  - Source: Datenquelle / Ursprung der Daten
  - Sinks: Ziel der Daten in einem Stream
  - Flow: Transformationsschritte der Daten in einem Stream
  - Runnable Graphs: Ein Stream in dem alle inputs und output verbunden sind.

Streams
Pekko Streams / Lineare Streams

- Source: Datenquelle / Ursprung der Daten
- Sinks: Ziel der Daten in einem Stream
- Flow: Transformationsschritte der Daten in einem Stream
- Runnable Graphs: Ein Stream in dem alle inputs und output verbunden sind.
- Jede Stage kann Sync + Async ausgeführt werden
- Normalerweise bleibt die Reihenfolge erhalten und Pekko kümmert sich um Backpressure

Streams
Pekko Streams / Graphen

- In den meisten Use-Cases reichen Lineare Streams aus, Graphen ermöglichen Kreuzungen durch Fan-In und Fan-Out
- Mithilfe von Graphen lassen sich komplexe Datenströme mit mehreren Inputs und Outputs abbilden



Streams
Pekko Streams / Code

- Definition und Ausführung von Streams und Graphen sind getrennt.
- Wenn der Stream läuft wird ein Materializer benötigt.

```
Source(1 to 100)
.via(Flow[Int].map(_ * 2))
.to(Sink.foreach(println))
.run
```

```
2
4
6
8
10
12
14
[..]
```

Pekko Connectors
Externe Datenquellen als (Reactive) Stream anbinden

- Erleichtert die Anbindung an externe Systeme wie Datenbanken, Messaging-Plattformen und APIs.
- Ermöglicht asynchrone Verarbeitung und Kommunikation zwischen verschiedenen Diensten und Komponenten.
- Unterstützt den Datenfluss zwischen Pekko-Aktoren und externen Systemen.
- Bietet typsichere Schnittstellen für die Kommunikation mit verschiedenen Systemen.
- Wird in Event-getriebenen Architekturen und Microservices verwendet.

Beispiele:

- Kafka, **Pekko Streams**, HTTP, MQTT, FTP, Webhooks, Datenbanken, NoSQL Datenbanken, Cloud-Dienste, [..]
- https://pekko.apache.org/docs/pekko-connectors/current/

Pekko
Weiterführende Themen / Ausblick

- In einer hochverfügbaren Applikation benötigt man ein Cluster, dass garantiert, dass ein Aktor nur einmalig existiert. Darüber hinaus darf kein Split Brain entstehen. (Split Brain Resolver)
- Verteilte Cluster Datentypen wie Conflict-free Repliacted Data Types (CRDTs)
- Überwachung und Monitoring wie Pekko Management, Failure Detection (Remote Death Watch)
- Multi-Datacenter Cluster
- Remote Protokolle für das Cluster, Serialisierung von Nachrichten (JSON, Avro, Protobuf)
- [..]