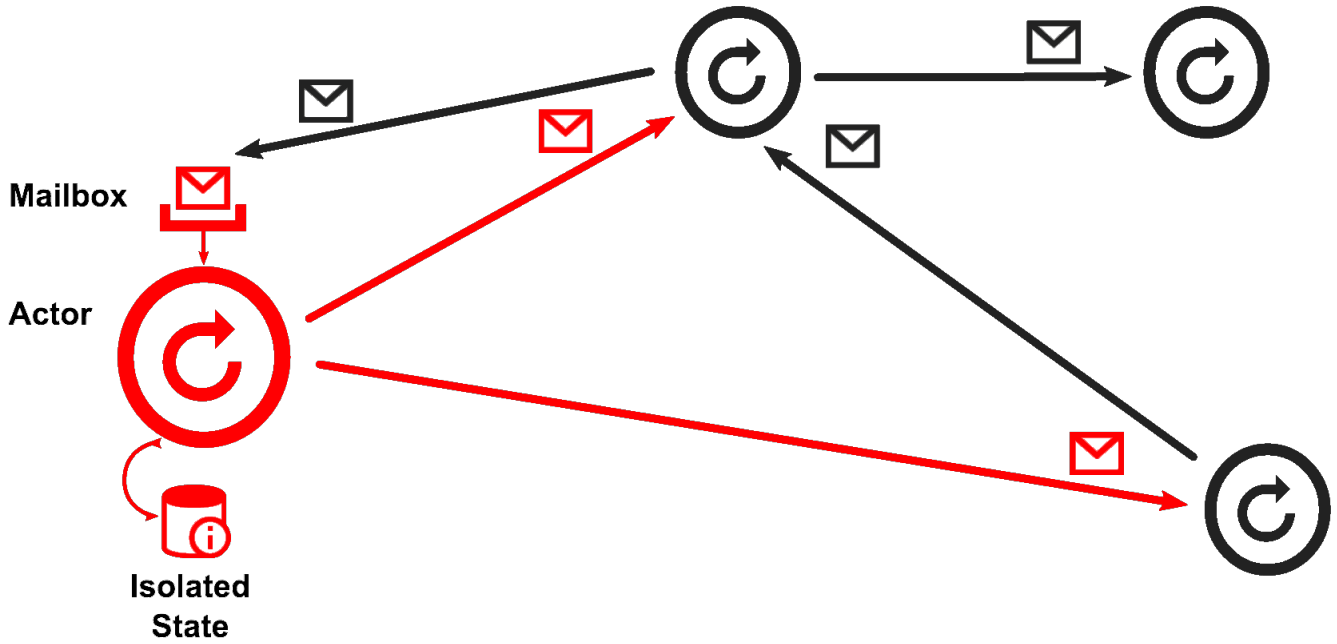
innFactory

Anwendung der funktionalen Programmierung mit Scala

TH Rosenheim - SoSe 2025

innFactory

Aktorenmodell



Das Aktorenmodell
Grundlagen

- „The actor is the fundamentel unit of computation emodying processing, storage and communication“
- Das Aktorenmodell wurde 1973 von Carl Hewitt erfunden. Inspiert von der Physik, Simula-67 & Smalltalk-72
- Es gibt verschiedenste Implementierungen des Aktorenmodells wie z.B. Erlang, Pekko oder Swift Actors und es hat auch viele andere Programmiersprachen beeinflusst.

Ein Aktor

- ist die Recheneinheit innerhalb eines Aktorenmodells
- hat eine Adresse mit der andere Aktoren kommunizieren können
- erhält Nachrichten und verarbeitet diese synchron
- kommuniziert mit anderen Aktoren über Nachrichten

Aktorenmodell
Grundlegende Konzepte

Wenn ein Aktor eine Nachricht erhält, kann dieser

- seinen eigenen Zustand verändern
- sein zukünftiges Verhalten für die nächsten Nachrichten verändern
- neue Aktoren erzeugen
- Nachrichten an andere Aktoren oder sich selbst schicken
- sich zerstören

Wenn ein Aktor keine Nachricht verarbeitet ist er Leerlauf

➔ 1 Mio+ Aktoren 🥳 , 1 Mio Threads 💣 (Max Threads = 2 * #vCPU)

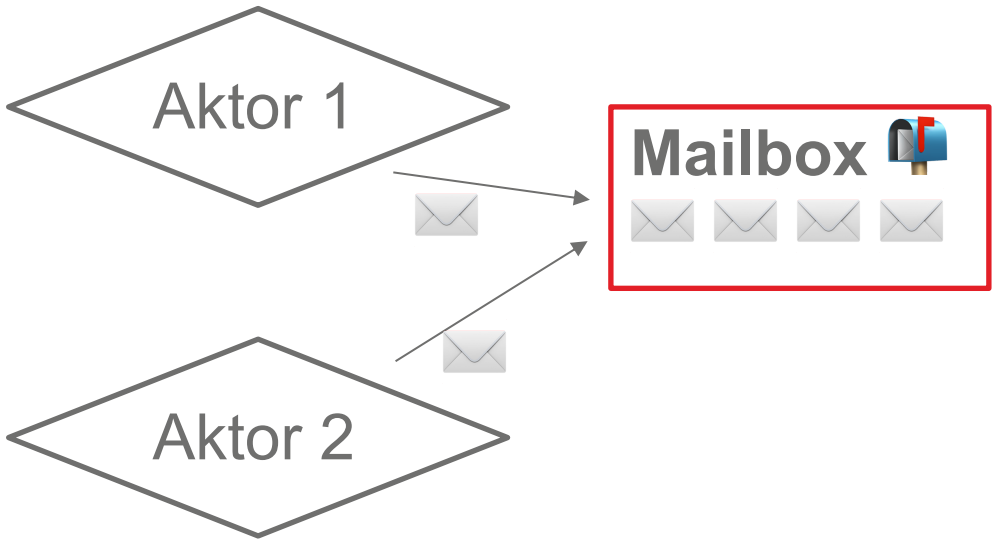
Aktorenmodell
Vorteile

- Komplexe Probleme können in einfache Teilstücke zerlegt werden
- Garantiert eine hohe Effizienz und eine sichere parallele Verarbeitung (vgl. Probleme mit Threads-Safety, Semaphore)
- Erlaubt durch die Leichtgewichtigkeit sehr hohe Parallelität
- Wenn man dem Modell strikt folgt können verteilte und fehlertolerante Systeme deutlich leichter entwickelt werden. (Vgl. Reactive Manifesto)

Aktor
Komponenten eines Aktors

Ein Aktor hat 4 Kernkomponenten

- Nachrichten
- Mailbox
- Zustand
- Verhalten



Aktor
- **Zustand**
- **Verhalten**

Ein Aktor arbeitet immer nur dann, wenn er eine Nachricht verarbeitet werden muss.

Synchron)

Aktoren
Nachrichten (Message) & Mailbox

- Nachrichten sind die Ein- und Ausgaben von Aktoren
- Nachrichten MÜSSEN unveränderbar sein (immutable)
- Aktoren definieren ähnlich wie bei REST APIs die Schnittstellendefinition des Aktors
- Sind typischerweise Strukturen, Objekte oder Werte

- Jeder Aktor hat eine eigene in-memory Mailbox die verschiedene Eigenschaften wie die maximale Nummer von Nachrichten, Prioritätseinstellungen, Zustellreihenfolge, uvm. definiert. Ein Aktor kann seine Nachricht immer nur zu der Mailbox eines Aktors schicken.

Aktoren
Zustand (State)

- Ein Aktor kann einen internen Zustand haben, der aber nicht von außen, sondern nur durch den Aktor selbst thread-sicher bearbeitet werden kann
- Der Aktor kann nur während der Verarbeitung von Nachrichten auf den Zustand zugreifen
- Andere Aktoren können den Zustand eines Aktors nur über eine Nachricht erfragen
- Der Zustand eines Aktors ist nicht persistent (außer man verwendet Module wie Pekko Persistence)

Aktor
Verhalten (Behaviors)

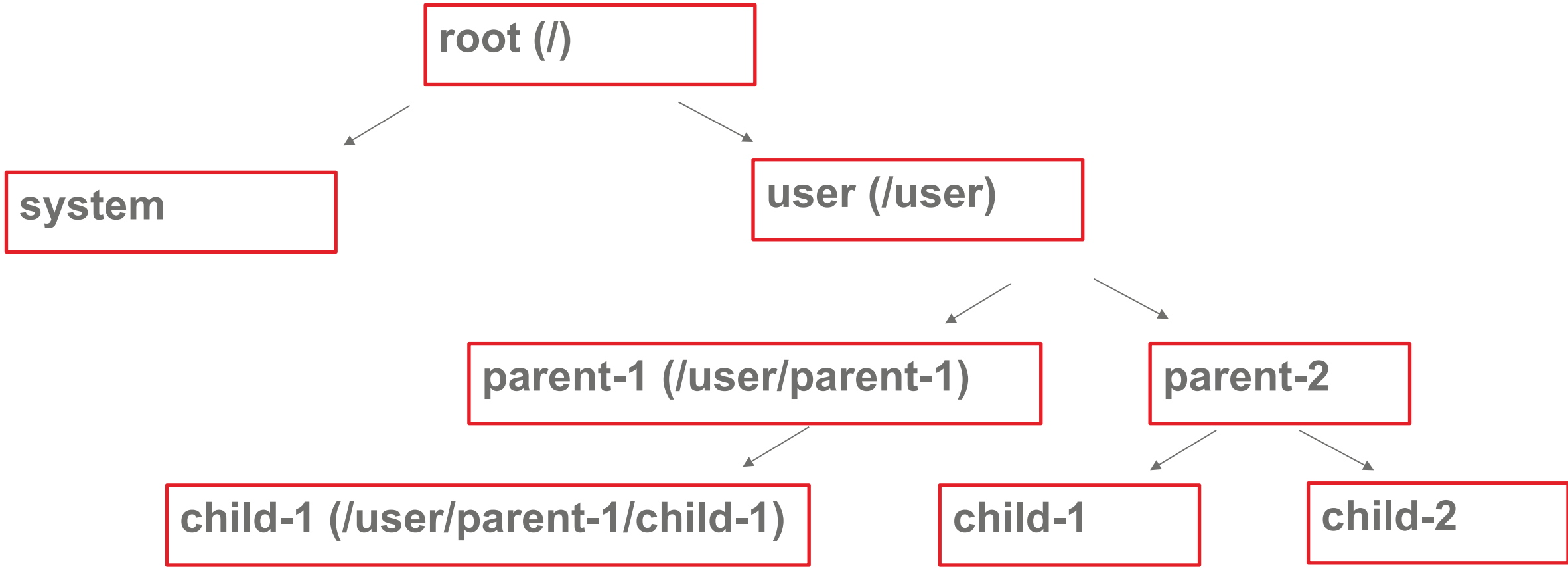
- Das Verhalten eines Aktors definiert, wie neue Nachrichten verarbeitet werden
- Ein Aktor kann mehrere Verhalten haben, allerdings wird immer nur ein Verhalten pro Nachricht ausgeführt.
- Während der Verarbeitung einer Nachricht kann das Verhalten für die nächste Nachricht definiert werden
- Das Verhalten sollte im Idealfall immer typsicher sein.

Das Verhalten eines Aktors kann:

- den Zustand des Aktors verändern
- Neue (Kind-)Aktoren erzeugen
- Nachrichten an andere Aktoren verschicken, wenn die entsprechende Referenz im Zustand oder der Nachricht bekannt ist (z.B. Antwort zurück an den Absender)
- Das Verhalten des Aktors verändern

Aktorenmodell

Hierarchie im Aktorenmodell (Aktor System)



innFactory

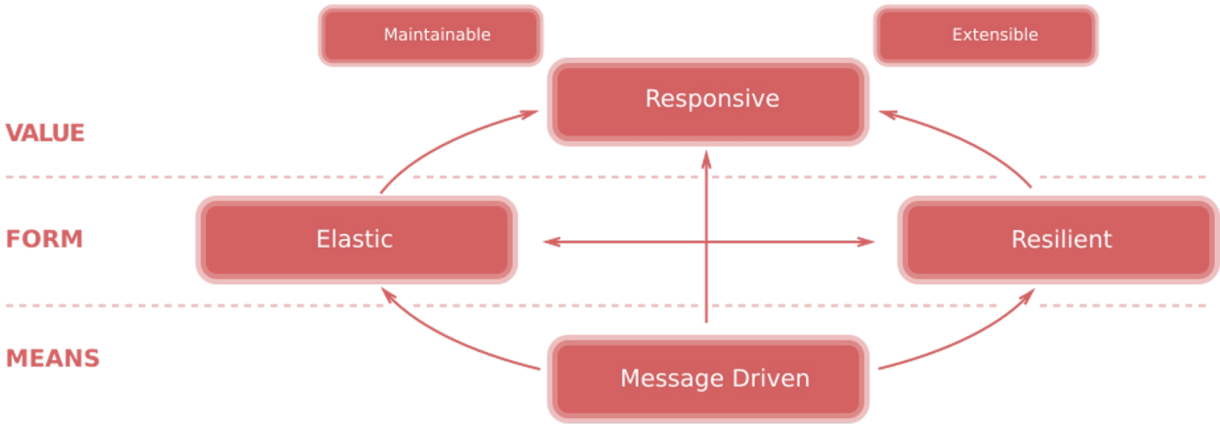
Pekko

Aktorsysteme in Scala
Pekko (Apache2 Fork von Akka)

- Pekko ist ein Framework um hoch-parallelisierte, verteilte, widerstandsfähige, nachrichtenbasierte Applikationen auf der JVM zu programmieren.
- Pekko folgt den Prinzipien des ***Reactive Manifesto***
- Pekko erlaubt es uns innerhalb eines Aktors “single-threaded“ Code zu programmieren
- Bei Konsequenter Einhaltung des Prinzips können Deadlocks und Synchronisation ausgeschlossen werden, was zu einem deutlich einfacheren Systemdesign von parallelen Systemen führt.
- Pekko erlaubt es Code auf einer Single-CPU zu schreiben und diesen dann auf Millionen JVMs im Netzwerk zu verteilen, sodass diese dann über das Netzwerk über Nachrichten kommunizieren.
- Pekko bietet alle Werkzeuge die Notwendig sind, diese Techniken und alle Sonderformen sinnvoll abzudecken. (Zusatzmodule wie: Pekko Cluster, Pekko Remote, Pekko Stream, Pekko Persistence, …)

Aktorsysteme in Scala
Reactive Manifesto

- **Responsive:** Das System antwortet immer in einem definierten Zeitraum
- **Resilient:** Das System bleibt antwortbereit, auch wenn es auf Fehler stößt
- **Elastic:** Das System bleibt antwortbereit, egal mit welcher Arbeitslast es konfrontiert wird
- **Message Driven:** Die Nachrichten im System sind die Grundlage für ein elastisches, resilientes und Antwortbereites System



https://www.reactivemanifesto.org/

Pekko
Grundkonzepte

- Pekko repräsentiert das Aktorenmodell bzw. das Gesamtsystem innerhalb der JVM über ein ActorSystem
  - Das ActorSystem kümmert sich um alle wichtigen Aufgaben wie Threadmanagement, Aktor Dispatching, Scheduling, Konfiguration, Logging, Messages, Deadletters, …
- Ein ActorContext kümmert sich innerhalb des ActorSystem um die Repräsentation eines Aktors (Erzeugen, Zerstören)
- Über Supervision entscheidet der „Parent“ über die Aktionen die im Fehlerfall bei den „Childs“ auftreten können.
- Eine ActorRef ist die eindeutige Adresse innerhalb eines Aktorsystems zu einem konkreten Aktor und ermöglicht den Versand von Nachrichten an diesen.
- Typsichere Aktoren (Typed Actors) & Nicht-typsichere Aktoren (Classic Actors -> Nutzen wir nicht)

innFactory

Pekko
Coding

```
object HelloWorld {

  sealed trait MessageA
  case class SendGreeting(to: ActorRef[MessageB]) extends MessageA

  def receiverBehavior(): Behavior[MessageB] = ???
  def senderBehavior(): Behavior[MessageA] = ???

  def main(args: Array[String]): Unit = {
    val system = ActorSystem[Nothing](Behaviors.setup { context ⇒
      val actorB = context.spawn(receiverBehavior(), "actor-b")
      val actorA = context.spawn(senderBehavior(), "actor-a")
      actorA ! SendGreeting(actorB)
      Behaviors.empty
    }, "HelloWorldSystem")

    Thread.sleep(500)
    system.terminate()
  }
}
```

Pekko
Next

- Pekko bietet viele weitere Module, die wir uns **nächste Woche** anschauen werden:
  - **Pekko Persistence**
  - **Pekko Connectors**
  - Pekko gRPC
  - Pekko HTTP (+ Play Framework)
  - Pekko Cluster
  - Pekko Management

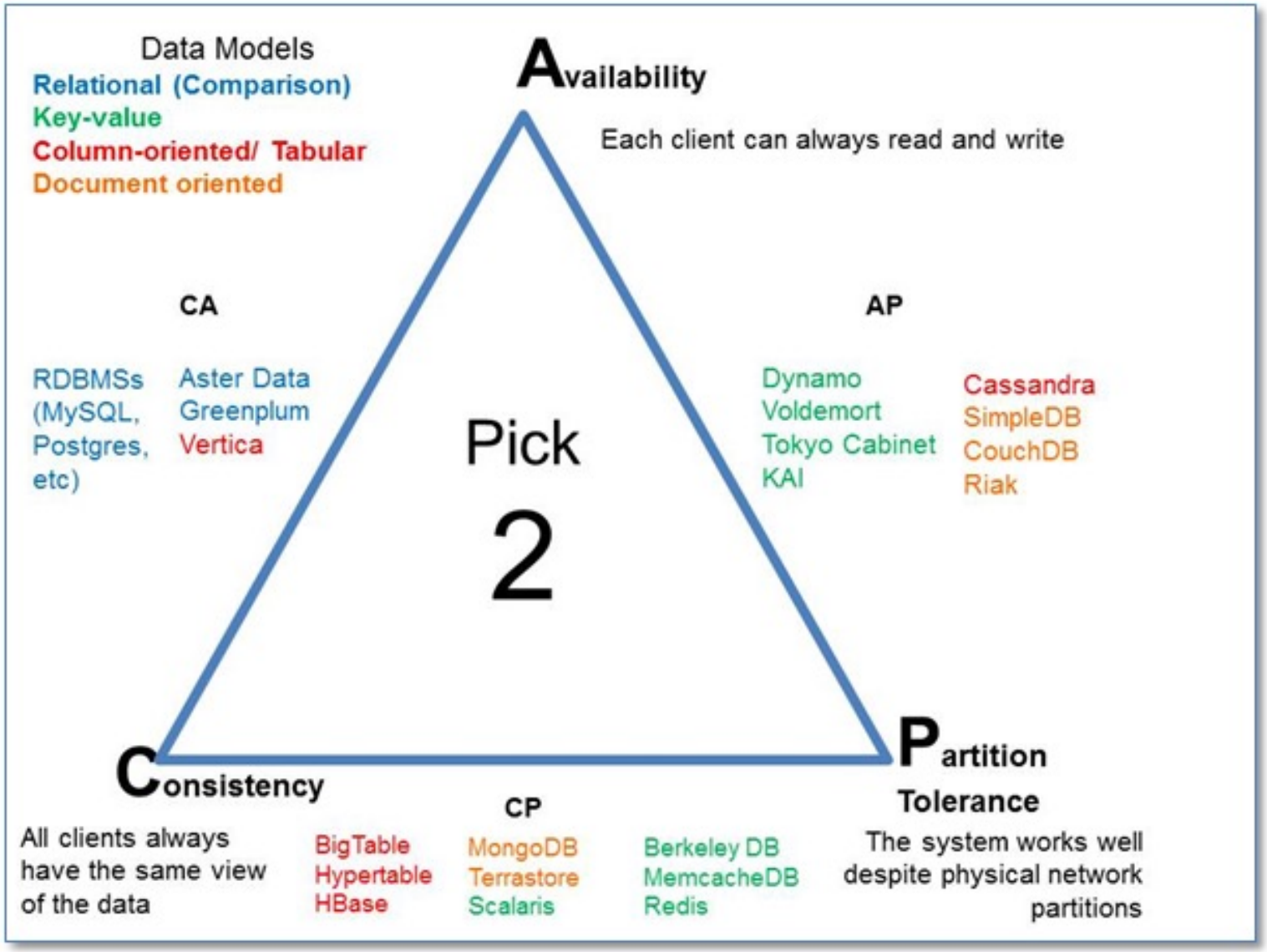
innFactory

Übung

Pekko Typed
Übung #1

- Ziel der Übung ist ein Typed Actor eines Bankkontos. Das Konto soll ähnlich wie im Livecoding sein Verhalten nach der Eröffnung verändern, sodass Beträge eingezahlt und ausgezahlt werden können.
- Jedes Konto soll mit einem Namen und einem EUR Wert initialisiert werden.
- Die Aktoren sollen Überweisungen tätigen können, sodass „Aktor 1“ zum Beispiel 500 EUR an “Aktor 2“ überweisen kann, sofern der Sender über ausreichend Guthaben verfügt.
- Zusatz 1: Beim Start soll ein möglicher Dispokredit hinterlegt werden, sodass ein Konto auch bis zu einem Beitrag ins Minus gehen kann.
- Zusatz 2: Der Empfänger soll den Eingang des Geldes bestätigen. Der Empfänger soll in Fall A) warten B) weiterarbeiten und später den Empfang quittieren)

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

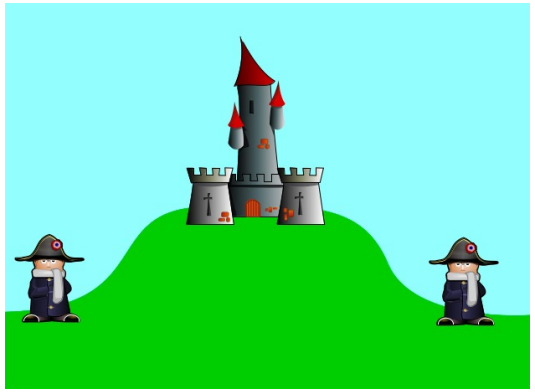
BASE ist ein flexibles Transaktionsmodell. Verfügbarkeit und Skalierbarkeit stehen bei BASE über Konsistenz.

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.

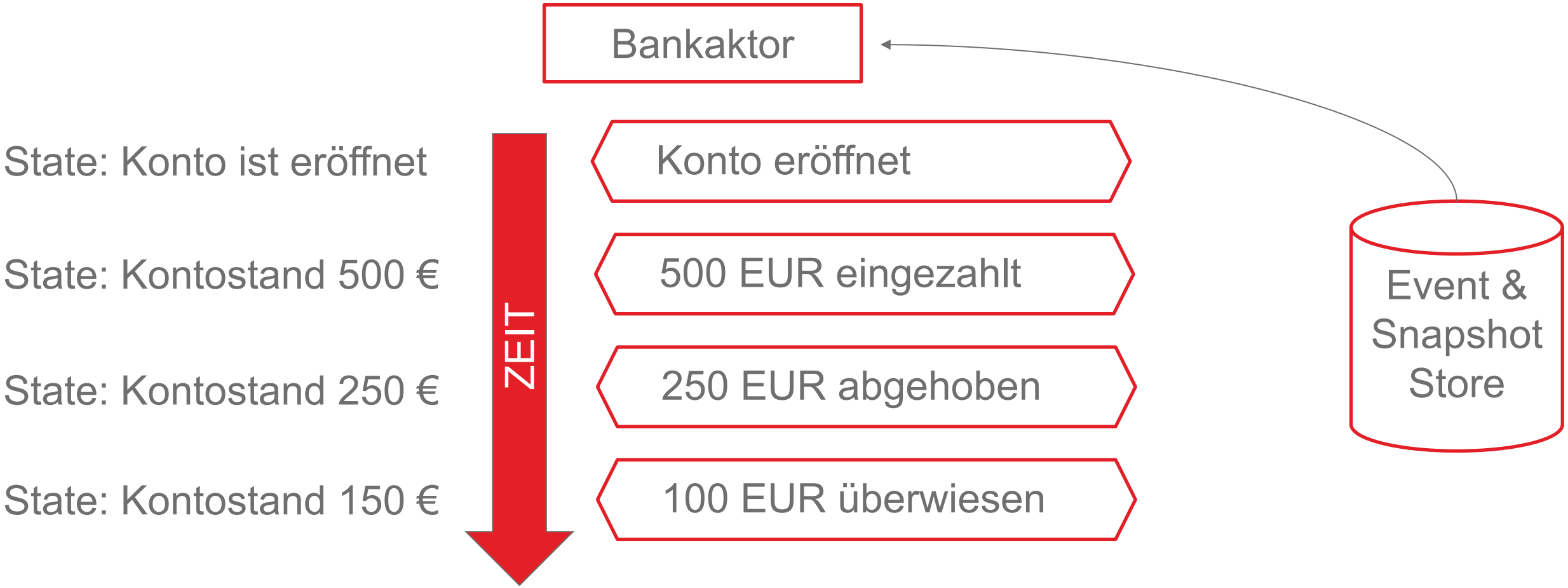
- Der Ablauf oder Zustand von Programmen wird nicht mehr als Zustand, sondern als Reihe von Events definiert
- CRUD (Create, Read, Update, Delete) reicht für Microservices bzw. hochparalle Systeme häufig nicht aus.
- ACID (atomicity, consistency, isolation, durability) ist für Transaktionen mit mehrerer 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

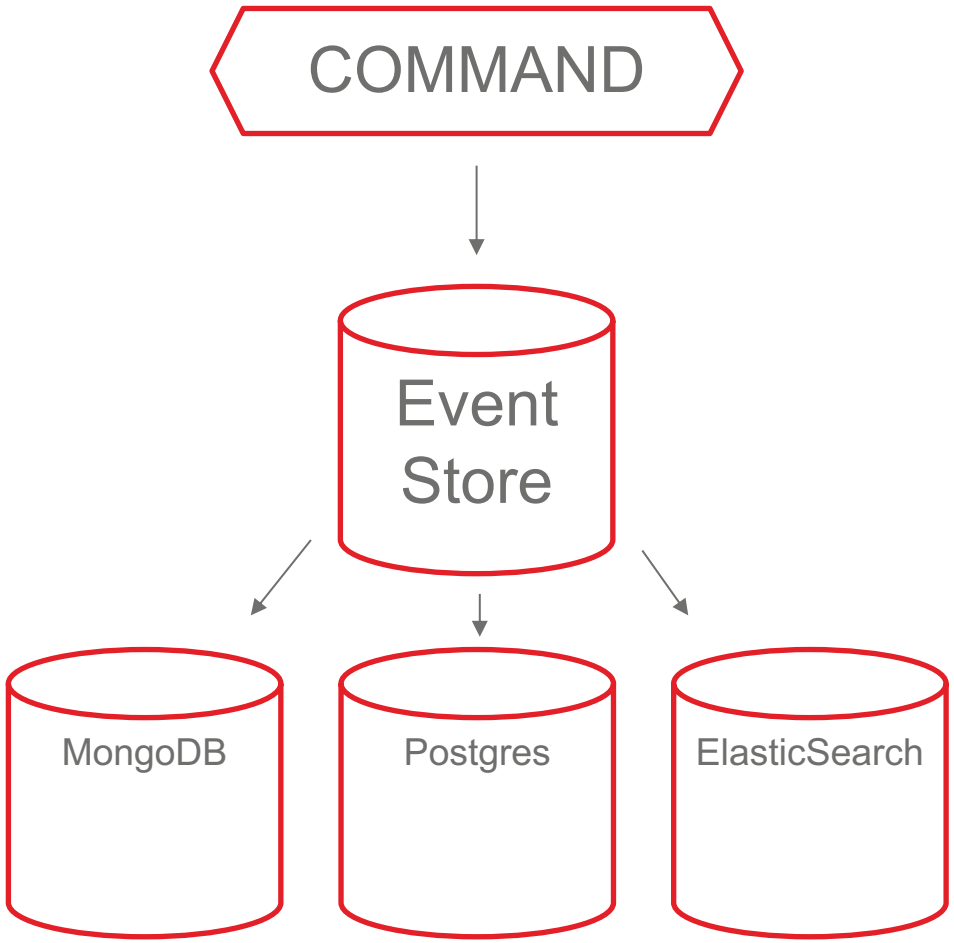
- Es wird nicht mehr der Zustand gespeichert, sondern die Events, die zu diesem Zustand führen.
- Der Zustand wird auf Basis der Events ausgerechnet.
- Im Kontext von Pekko bedeutet dies, das man seine Mailbox speichern muss, um aus allen Nachrichten den aktuellen Zustand wiederherstellen zu können.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

Pekko Persistence
Event Sourcing

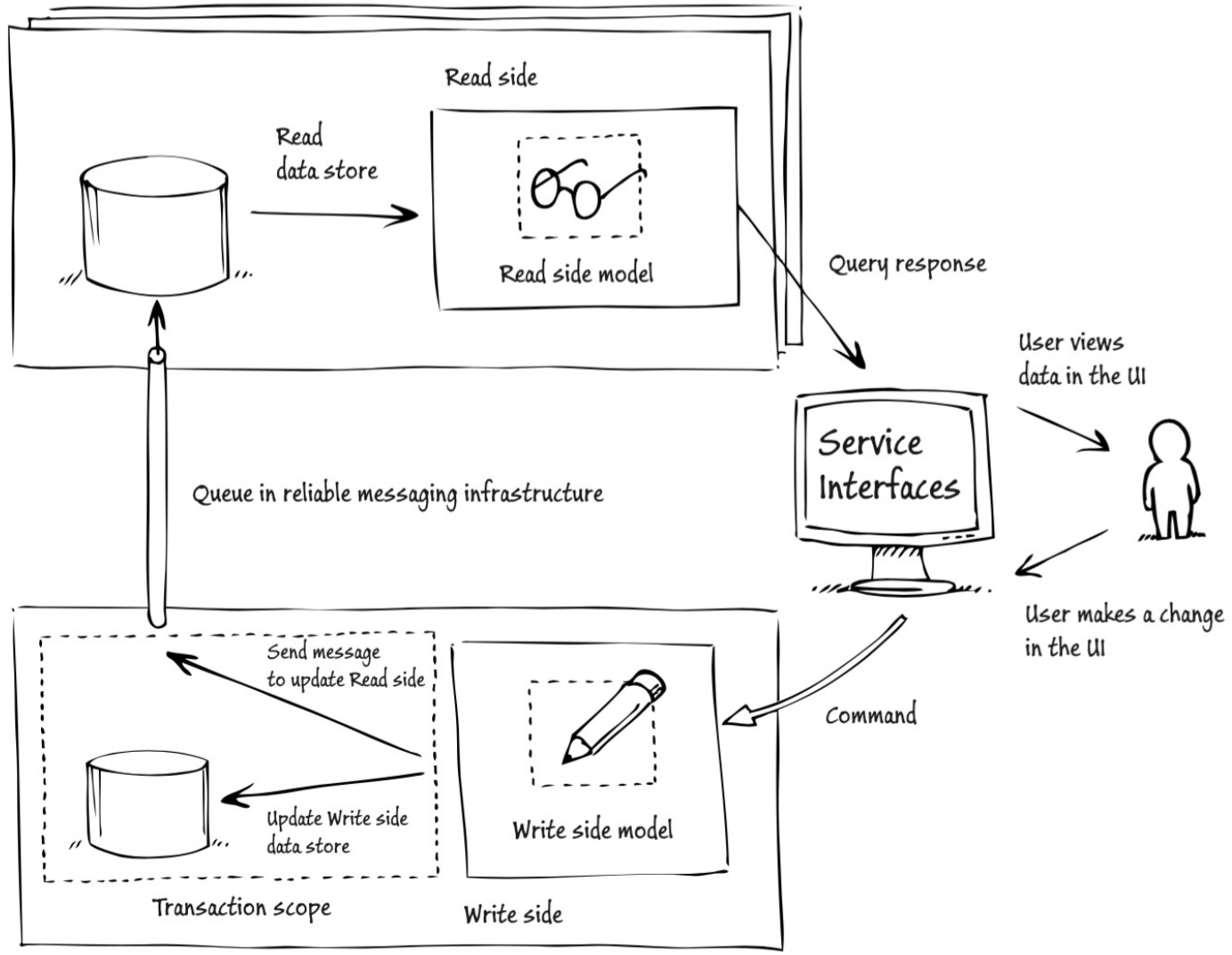


Pekko Persistence
CQRS

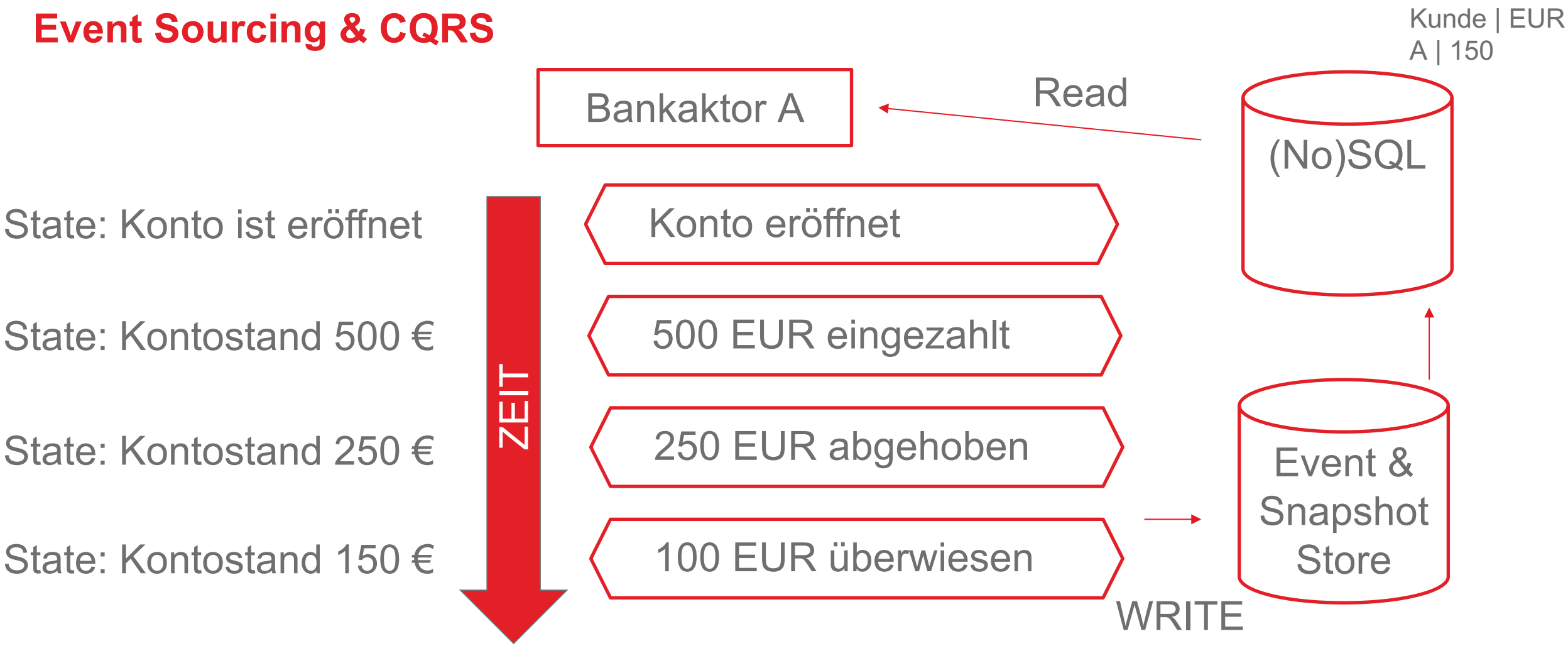
- Command Query Responsibility Segregation (CQRS) teilt eure Applikation in eine Lese- und eine Schreibseite.
- Die Commandseite verarbeitet create/update/delete requests
- Die Queryseite verarbeitet die Leseanfragen, indem sie Datenquellen anfragen, die auf Basis der Commandseite entstehen
- CQRS findet 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
  }
}
```

Pekko Connectors
Externe Datenquellen als 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/

innFactory

Übung

Pekko Typed
Übung #2

- Ziel der Übung ist es die Übung aus der letzten Woche um Pekko Persistence zu erweitern, sodass die Daten in einem lokalen Journal (Docker Postgres) gespeichert werden und nach jedem 10. Command ein Snapshot erstellt wird.
- Zusatz: Eine Leseseite mit Pekko Persistence Querys aufbauen
- https://pekko.apache.org/docs/pekko/1.1/persistence-query.html