Übungsaufgabe

Einführung in Akka: Actors, Message-Passing, Supervision, Lebenszyklus.

Universität:	Technische Universität Berlin
Kurs/Modul:	Programmieren II für Wirtschaftsinformatiker
Erstellungsdatum:	September 6, 2025

Aufgabe 1: Einführung in Akka – Actors, Message-Passing, Supervision, Lebenszyklus

In dieser Aufgabe setzen Sie sich grundlegend mit dem Actor-Modell in Akka auseinander. Beachten Sie die asynchrone Kommunikationsweise, die Trennung von Zustand und Verhalten sowie die Fehlertoleranz durch Supervisory-Strategien.

a) Erklären Sie, was ein Actor in Akka ist und welche drei Kernprinzipien dem Actor-Modell zugrunde liegen (Isolierung der Zustände, asynchrone Nachrichtenverarbeitung, fehlertolerante Struktur durch Supervision).

b) Welche Rolle spielen Messages (Nachrichten) im System? Nennen Sie Merkmale von Messages in Akka (z. B. Unveränderlichkeit) und erklären Sie, wie Actors auf Nachrichten reagieren.

c) Beschreiben Sie die wichtigsten Lebenszyklus-Methoden eines Actors (z.B. preStart, postStop, preRestart, postRestart) und erläutern Sie, wann sie aufgerufen werden und wofür sie genutzt werden.

d) Geben Sie einen kurzen Überblick über das Supervisory-Prinzip. Unterscheiden Sie zwischen OneForOneStrategy und AllForOneStrategy und nennen Sie typische Supervisory-Entscheidungen wie Restart, Resume, Stop und Escalate.

e) Skizzieren Sie einen einfachen Ablauf, wie ein Actor auf eine eingehende Nachricht reagiert (Receive-Verhaltenswechsel) und welche Rolle Become/ unbecome dabei spielen kann.

Aufgabe 2: Lebenszyklus, Fehlerbehandlung und Supervision in Akka

Diese Aufgabe vertieft das Verständnis für Lebenszyklus-Phasen, Fehlerbehandlung und Supervisory-Strategien in typischen Akka-Anwendungen.

a) Beschreiben Sie den Ablauf des typischen Actors-Lebenszyklus, inkl. Wann preStart, postStop, preRestart und postRestart aufgerufen werden und wofür sie genutzt werden.

b) Erläutern Sie den Mechanismus der Supervision. Unterscheiden Sie OneForOneStrategy und AllForOneStrategy und nennen Sie typische Supervisory-Entscheidungen (Restart, Resume, Stop, Escalate).

c) Skizzieren Sie den Ablauf, wie ein Supervisor auf das Scheitern eines Kind-Actors reagiert. Welche Auswirkungen hat ein Neustart eines Kind-Actors auf dessen Zustand, und wie wirkt sich dies auf die übrigen Actors im Supervisory-Baum aus?

d) Diskutieren Sie kurz, welche Rolle der Supervisor bei der Fehlertoleranz einer Geschäftslogik (z. B. Bestellverarbeitung) spielt und welche Vor- bzw. Nachteile Restart vs. Resume in einer produktiven Anwendung haben.

Aufgabe 3: Praktische Architektur – Beispiel-Architektur mit Akka

In dieser Aufgabe entwerfen Sie konzeptionell eine kleine Actor-Architektur und diskutieren das Fehlerverhalten.

a) Beschreiben Sie das grobe Design eines Daten-Dispatchers, der Rohdaten von mehreren Sensor-Actors an einen Data-Processor-Actor weiterleitet. Welche Rolle spielt der Supervisor in diesem Szenario, um Stabilität bei Ausfällen eines Sensors sicherzustellen?

b) Vergleichen Sie zwei typische Supervisory-Strategien (OneForOne vs AllForOne) hinsichtlich Komplexität, Fehlerisolation und Reaktionsverhalten im Falle eines Fehlers eines Child-Actors. Geben Sie zu jedem Fall eine kurze Begründung an, wann der jeweils bessere Einsatz ist.

c) Beschreiben Sie, wie Zustandsinformationen innerhalb eines Actors verwaltet werden sollten (z. B. unveränderliche Messages vs. veränderlicher Mutable State) und welche Best Practices in Bezug auf Become/State-Handling gelten, um Konsistenz trotz Fehlern zu wahren.

Lösungen

Aufgabe 1: Einführung in Akka – Actors, Message-Passing, Supervision, Lebenszyklus

a) Lösung. Ein Actor in Akka ist eine leichtgewichtige, isolierte Einheit der Ausführung, die auf Nachrichten reagiert und dabei Zustand kapselt. Die drei Kernprinzipien des Actor-Modells sind:

- Isolierung der Zustände: Jeder Actor besitzt seinen eigenen Zustand und externen Zugriff darauf gibt es nicht. Dadurch entfallen klassische Datenrennen.
- Asynchrone Nachrichtenverarbeitung: Actors kommunizieren ausschließlich über asynchrone Nachrichten (Messages) via Mailboxen; Nachrichten werden sequentiell verarbeitet, ohne blocking.
- Fehlertolerante Struktur durch Supervision: Actors werden von Supervisoren überwacht und bei Fehlern entsprechend einer definierten Strategie behandelt (Restart, Resume, Stop, Escalate).

b) Lösung. Messages in Akka besitzen mehrere Merkmale:

- Unveränderlichkeit (Immutable): Nachrichten sollten nach ihrer Erstellung nicht verändert werden, damit keine Nebenwirkungen durch gemeinsam genutzte Daten entstehen.
- Typische Repräsentation über Fallklassen (z. B. in Scala): messages dienen als klare Protokolle zwischen Actors.
- Eigenständige Semantik: Messages beschreiben konkrete Events oder Befehle (z. B. DataChunk, InitRequest, Ack).
- Verarbeitung durch pattern matching in der Receive-Funktion: Actors reagieren auf Nachrichten je nach Typ und ggf. Zustand.
- Delivery und Synchrone/Asynchrone Semantik: Nachrichten werden asynchron zur Mailbox des Empfängers geliefert; Verarbeitung erfolgt seriell innerhalb eines Actors.

Die Reaktion eines Actors erfolgt typischerweise durch Pattern Matching der empfangenen Messages, ggf. Durchführung von Seiteneffekten (z. B. Senden weiterer Messages, Erzeugen/Verwalten von Kind-Actors) und anschließendes Update des internen Zustands. c) Lösung. Wichtige

Lebenszyklus-Methoden eines Actors:

- preStart: Wird nach der Erstellung des Actor-Objekts aufgerufen, bevor der Actor erste Nachricht verarbeitet. Einsatz: Initialisierung von Ressourcen, Verbindungen, Registrierung bei Diensten.
- postStop: Wird aufgerufen, wenn der Actor gestoppt wird. Einsatz: Ressourcenfreigabe, Clean-up.
- preRestart: Wird auf dem alten Instance vor einem Restart aufgerufen (in der Regel bei einer Ausnahme). Standardverhalten: Kind-Actors beenden, Logik zur Aufräumung. Einsatz: Vorbereitendes Clean-up vor einem Neustart.
- postRestart: Wird auf dem neuen Instance nach dem Restart aufgerufen. Standardverhalten ruft oft wieder preStart auf. Einsatz: Wiederherstellung von Zustand bzw. Reinitialisierung.

d) Lösung. Überblick über das Supervisory-Prinzip:

- OneForOneStrategy: Die Supervisory-Entscheidung trifft nur den unmittelbar fehlgeschlagenen Child-Actor. Andere Children bleiben unverändert.
- AllForOneStrategy: Die Entscheidung gilt für alle Children des Supervisors; alle betroffenen Children können neu gestartet oder anderweitig behandelt werden.
- Typische Supervisory-Entscheidungen:
  - Restart: Der betroffene Actor wird neu erstellt (Zustand geht verloren, sofern er nicht explizit persistent ist).
  - Resume: Der Actor fährt fort, der Fehler wird ignoriert; Zustand bleibt erhalten.
  - Stop: Der fehlerhafte Actor wird beendet.
  - Escalate: Der Fehler wird an den nächsten Supervisory-Ebenen weitergereicht.

e) Lösung. Skizze eines einfachen Ablaufs beim Empfang einer Nachricht (Receive-Verhaltenswechsel) und Rolle von Become/unbecome:

- Anfangsverhalten (defaultReceive): Das Actor-Verhalten wird durch eine Receive-Funktion beschrieben, z. B. `def receive: Receive = case Init => ... ; case Data(d) => ...` .
- Verhaltenswechsel: Bei bestimmten Messages kann das Verhalten gewechselt werden, z. B. `context.become(readyForData)`. Dadurch wird das nächste Matching auf das neue Verhalten angewendet.
- Become/Unbecome: Mehrfaches Beutetreten von Verhaltenszuständen wird durch einen Stack realisiert; `context.become(newBehavior)` pusht auf den Stack, `context.unbecome()` poppt und kehrt zum vorherigen Verhalten zurück. Dies ermöglicht das modelling von Zustandsmaschinen innerhalb eines Actors.
- Vorteil: Saubere Trennung von Zuständen und Verhalten, geringere Seiteneffekte, bessere Wartbarkeit.

Aufgabe 2: Lebenszyklus, Fehlerbehandlung und Supervision in Akka

a) Lösung. Typischer Ablauf des Actor-Lebenszyklus:

- Erstellung des Actors (mit `actorOf`) – der neue Instance wird erzeugt.
- preStart wird aufgerufen, um Initialisierung vorzunehmen (Ressourcenaufbau, Registrierung, Verbindungen).
- Der Actor verarbeitet Nachrichten (Receive-Verhalten).
- Bei einem Fehler während der Verarbeitung kann der Supervisor eingreifen (z. B. durch Neustart).
- preRestart wird auf dem alten Instance vor dem Neustart aufgerufen (Ressourcen freigeben, ggf. Kinder stoppen).
- Neuer Instance wird erstellt; postRestart wird aufgerufen (standardmäßig ruft dies auch preStart auf – Initialisierung des neuen Zustands).
- Der Lebenszyklus endet mit Stop, wobei postStop aufgerufen wird (Ressourcen freigeben).

b) Lösung. Supervision und Strategien:

- Mechanismus der Supervision: Ein Supervisor (Eltern-Actor) verwaltet seine Child-Actors. Bei einem Fehler eines Children wird eine decider-Strategie angewendet, die eine Directive (Restart, Resume, Stop, Escalate) zurückgibt.
- OneForOneStrategy vs AllForOneStrategy:
  - OneForOneStrategy: Nur der fehlerhafte Child wird betroffen; Verhalten ist minimal invasiv.
  - AllForOneStrategy: Alle Children des Supervisors werden betroffen; nützlich, wenn Children gemeinsam Ressourcen nutzen oder inkrementelle Konsistenz wahren müssen.
- Typische Supervisory-Entscheidungen bleiben Restart, Resume, Stop, Escalate.

c) Lösung. Ablauf, wie ein Supervisor auf das Scheitern eines Kind-Actors reagiert:

- Ein Kind-Actor scheitert; der Supervisor erhält eine Failure-Nachricht.
- Die decider-Logik wählt eine Aktion (z. B. Restart).
- Bei Restart wird der betroffene Child-Actor neu erstellt; der Zustand des Child-Actors geht in der Regel verloren, sofern er nicht persistiert ist.
- In OneForOne gilt Restart nur für das fehlerhafte Kind; in AllForOne wird ggf. auch der Zustand der anderen Kinder beeinflusst (je nach gewählter Strategie).
- Auswirkungen auf übrige Actors: Bei AllForOne werden alle Kinder neu gestartet, wodurch deren Zustand ebenfalls auf Null gesetzt wird; bei OneForOne bleiben andere unverändert.

d) Lösung. Rolle des Supervisors in der Fehlertoleranz einer Geschäftslogik (z. B. Bestellverarbeitung) und Vor-/Nachteile von Restart vs. Resume:

- Rolle: Der Supervisor isoliert Fehler, verhindert globale Systemabstürze, ermöglicht automatische Wiederherstellung (Self-Healing), sorgt für robuste Verarbeitung von Geschäftsprozessen (z. B. Bestellpfad) durch kontrollierte Fehlerbehandlung.
- Restart-Vorteile: Saubere Initialisierung, Zustand-Reset, Wiederherstellung der Stabilität nach schweren Fehlern; Nachteil: Verlust von lokalem Zustand, potenziell Duplizierung oder Verlust von In-Memory-Progress.
- Resume-Vorteile: Behält bestehenden Zustand, keine Neuberechnung oder Neuanwendung von Logik; Nachteil: Fehlerzustand bleibt bestehen, ggf. inkonsistenter Zustand.
- Empfohlene Praxis: Bei operationskritischen Zuständen (Bestellverarbeitung mit Schrittfolge) oft Resume vermeiden, wenn der Fehler nicht manipulationsfrei repariert werden kann; Restart ist sinnvoll, wenn der Zustand wiederhergestellt oder neu initialisiert werden kann. Behandeln Sie Zustand konsequent (z. B. idempotente Operations, verbliebenes sicheres Verhalten) und nutzen ggf. Persistenz, Stashing oder Backpressure, um Konsistenz sicherzustellen.

Aufgabe 3: Praktische Architektur – Beispiel-Architektur mit Akka

a) Lösung. Grobes Design eines Daten-Dispatchers:

- Sensor-Actors: Mehrere Sensor-Actors erzeugen Rohdaten und senden sie an den Data-Dispatcher.
- Data-Dispatcher: Empfängt Rohdaten von Sensoren, führt ggf. Vorverarbeitung (Filterung, Normalisierung) durch und leitet Daten an den Data-Processor weiter. Er kann auch Routenlogik basierend auf Datenart, Sensor-ID oder Last implementieren.
- Data-Processor-Actor: Empfängt verarbeitete Daten, führt Aggregationen, Transformationsschritte oder BI-relevante Operationen durch.
- Supervisor-Ebene: Ein Supervisor überwacht Sensor-Actors und den Dispatcher; bei Ausfällen von Sensor-Actors sorgt der Supervisor durch Neustart für Stabilität, während der Dispatcher ggf. Backpressure-Mechanismen oder Pufferspeicher nutzt.
- Typische Strategie: OneForOne für isolierte Sensoren; AllForOne kann genutzt werden, wenn Sensoren gemeinsam Ressourcen (z. B. Datenpfad, Speicherräume) teilen.

b) Lösung. Zwei typische Supervisory-Strategien (OneForOne vs AllForOne) im Vergleich:

- OneForOne:
  - Vorteile: Geringe Beeinflussung anderer Kinder; bessere Skalierbarkeit; Fehler isoliert auf betroffenen Actor beschränkt.
  - Geeignet, wenn Sensoren unabhängig arbeiten und Fehler eines Sensors keinen globalen Zustand des Dispatchers beeinträchtigt.
- AllForOne:
  - Vorteile: Konsistenz sicherstellen, wenn Kinder gemeinsam Ressourcen nutzen oder ein Fehler das Gesamtsystem in einem gemeinsamen Zustand versetzt.
  - Geeignet, wenn das Versagen eines Kindes alle anderen betrifft oder deren Zustand stark verknüpft ist.

c) Lösung. Zustandsinformationen innerhalb eines Actors verwalten und Best Practices zu Become/State-Handling:

- Grundprinzip: Actors sind per Design mutable, aber konzeptionell isoliert; Zustand soll nicht durch geteilten Speicher mit anderen Actors geteilt werden.
- Behandeln von Zustand durch unveränderliche Messages: Verwende immutable Messages, um Zustandsänderungen zu beschreiben, und halte den internen Zustand innerhalb des Actors als nichtexterner, konsistenter Snapshot.
- Become/State-Handling:

  - Verwende Bewegt-Logik, um in eine Zustandmaschine zu wechseln (z. B. Initialisierung, Messdaten empfangen, Abschluss).
  - **Becoming**: `context.become(newBehavior)` pilzt einen neuen Satz von Reaktionen; dadurch wird der Zustand in Bezug auf das nächste Verhalten verwaltet.
  - **Unbecome**: Mit `context.unbecome()` wird zum vorherigen Verhalten zurückgefunden, sofern stack-basiert gearbeitet wird.

- Best Practices:
  - Nutze zustandsbezogene immutable State-Objekte; verwende Bewegunstate (become) statt über mutable Felder zu arbeiten, um Konsistenz zu verbessern.
  - Halte Transitions logg- oder auditierbar; nutze ggf. Stash, um Messages während Initialization zu puffern.
  - Splitting von Funktionen: Trenne Verarbeitung, Validierung und Zustandsänderungen in klare Schritte.