Lernzettel

Architektur-Patterns und -Praktiken: Saga, Circuit Breaker, Idempotency, Caching, Data Consistency Models, Eventual vs Strong consistency

Universität: Technische Universität Berlin
Kurs/Modul: Architektur von Anwendungssystemen

Erstellungsdatum: September 19, 2025

Lernzettel: Architektur-Patterns und -Praktiken: Saga, Circuit Breaker, Idempotency, Caching, Data Consistency Models, Eventual vs Strong consistency

(1) Saga-Pattern.
Sagas koordinieren langlaufende Transaktionen in verteilten Systemen durch eine Folge von Teiltransaktionen, wobei jede Transaktion eine Compensation-Transaktion auslöst, falls ein Fehler auftritt. Es gibt zwei Varianten: *Orchestrierung* (eine zentrale Steuerung koordiniert) und *Choreographie* (Services kommunizieren direkt miteinander, ohne zentrale Koordinatorin).
Wichtige Aspekte:

- Vorteile: keine verteilte Zwei-Phasen-Commit-Blockierung, bessere Verfügbarkeit, feingranulare Fehlertoleranz über long-running Prozesse.
- Kompensation statt Commit-Party: Bei Fehlern wird eine Gegenaktion ausgeführt, um den vorherigen Schritt rückgängig zu machen.
- Typische Einsatzfelder: Geschäftsprozesse über Microservices hinweg, ggf. mit asynchroner Kommunikation.

(2) Circuit Breaker.
Schützt Services vor cascading failures durch Sperren (Blocking) bei wiederholtem Scheitern. Zustände: *Closed* (alle Anfragen gehen durch), *Open* (keine Anfragen fließen), *Half-Open* (Kurze Probe).
Laufzeitparameter:

- Fehler-Schwellwert (failure rate) und Reset-Zeit.
- Fallback-Logik bei Open-Status.

(3) Idempotency.
Idempotente Operationen liefern bei wiederholter Ausführung immer denselben Effekt und dasselbe Resultat. Wichtig in verteilten Systemen bei Retries nach Fehlern.
Techniken:

- Idempotency Keys: Anfragen mit eindeutigen Schlüsseln werden nur einmal verarbeitet.
- Upsert-Strategien: bei gleichen Primärschlüsseln wird kein doppelte Änderung erstellt.
- Deduplication-Caches: speichern bereits verarbeitete Anfragen über kurzen Zeitraum.

(4) Caching.
Zweck: Reduktion von Latenz und Last auf Backend-Systemen. Typische Ebenen: Client-, Server- und Distributed-Caches (ggf. CDN).
Wichtige Konzepte:

- Cache-Strategien: Cache-Aside, Write-Through, Write-Behind, Read-Through.
- Invalidation/TTL: Zeitbasierte Gültigkeit oder ereignisbasierte Invalidation.
- Typische Probleme: Cache-Stale-Daten, Cache-Stampede, Poisoning.

(5) Data Consistency Models.
Unterscheidung der Konsistenzmodelle in verteilten Systemen:

- *Strong Consistency*: Alle Leser sehen die neuesten Writes sofort (linearizierbar).
- *Eventual Consistency*: Alle Replikate konvergieren eventual zu einem konsistenten Zustand.
- *Kausalität*, *Session* und andere Modelle als Kompromisse zwischen Latenz und Konsistenz.
- *CAP-Theorem*: In einem verteilten System kann man nur zwei der drei Eigenschaften verlässlich garantieren (Consistency, Availability, Partition tolerance) – oft werden trade-offs gewählt.

(6) Eventual vs Strong consistency.
Strong consistency sorgt für klare, sofortige Korrektheit, hat aber oft höhere Latenz und geringere Verfügbarkeit unter Partitionen.
Eventual consistency bietet bessere Verfügbarkeit und niedrigere Latenz, erlaubt vorübergehende Widersprüche, die sich mit der Zeit auflösen.
Beispiele:

- Einkaufswagen und Bestellprozesse profitieren häufig von eventual consistency.
- Inventar- oder Finanztransaktionen erfordern öfter starke Konsistenz oder partielle starke Konsistenz innerhalb bestimmter Transaktionsgrenzen.