CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Microservices

(5) Integrationsstile

Messaging – Event-driven Architectures

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 1.0 Lizenz** bereitgestellt.

Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).

Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CCo Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

KAPITEL 12

Microservice Architekturen

12.1 Eigenschaften von Microservices

12.2 Integrationsmuster für Microservices

- Datenbank-basierte Integration
- gRPC, REST
- API Versioning
- Event-Driven Architectures

12.3 Architekturelle Sicherheit

- Circuit-Breaker, Bulkhead
- Idempotente API-Operationen

12.4 Skalierung von Microservices

- Load Balancing
- Messaging
- Scaling for Reads/Writes
- Command Query Responsibility Segregation (CQRS)
- Caching

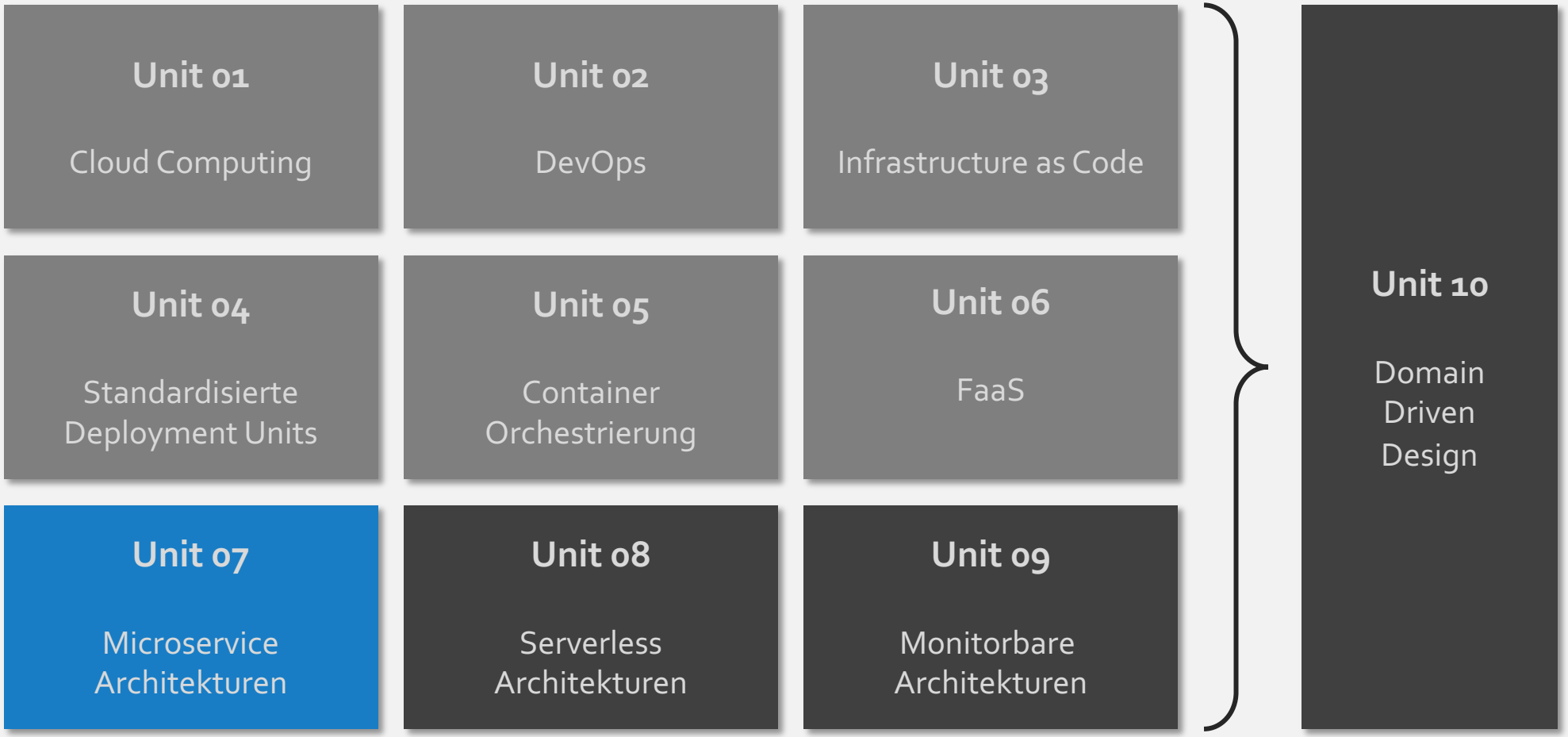
12.5 Prinzipien zur Entwicklung von Microservices

12.6 Serverless-Architekturen

- Architekturelle Konsequenzen von Serverless-Limitierungen
- Das API-Gateway Pattern
- Abgrenzung zu Microservices

INHALTSVERZEICHNIS

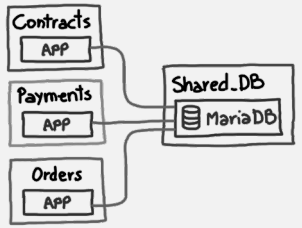
Überblick über Units und Themen dieses Moduls



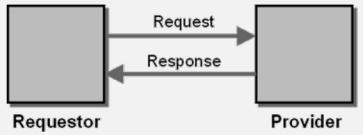
AUSBLICK

Integrationsstile

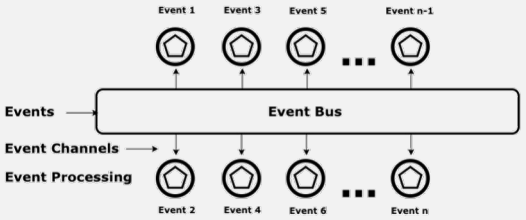
- Daten-basierte Integration
  - The shared database



- Request/Response-basierte Integration
  - Remote Procedure Calls
  - REST
  - GraphQL



- **Ereignis-basierte Integration**
  - Event Driven Architectures
  - Reactive Programming



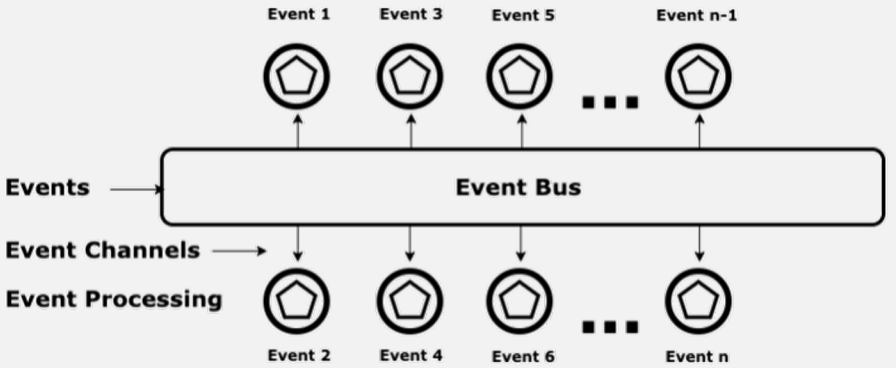
EREIGNISGESTEUERTE ARCHITEKTUREN (EDA)

Dezentrale Architekturen

CLOUD NATIVE COMPUTING

- Eine **ereignisgesteuerte Architektur (EDA)** ist ein Architekturmuster, bei dem die Kommunikation und die Verarbeitung zwischen verschiedenen Anwendungen und Diensten durch Ereignisse oder Nachrichten ausgelöst wird.
- **Ereignisse** können von Benutzern, Geräten, anderen Systemen oder Diensten generiert werden und ausgelöst werden, um eine bestimmte Aktion auszulösen oder Informationen auszutauschen.
- EDA ist eine **dezentrale Architektur**, bei der jede Anwendung oder jeder Dienst unabhängig von anderen agieren und auf Ereignisse (in Echtzeit) reagieren kann.
- Dienste müssen sich hierzu untereinander nicht kennen **(sehr lose Kopplung)**.

Im Gegensatz dazu ist ein Request-Response-basierter Ansatz eine zentrale Architektur, bei der Anwendungen oder Dienste Anforderungen an eine zentrale Stelle senden und auf eine Antwort warten. In diesem Modell wird die Kommunikation zwischen Anwendungen oder Diensten durch eine zentrale Schnittstelle koordiniert.



NATS

APACHE kafka

redis

EREIGNISGESTEUERTE ARCHITEKTUREN (EDA)

Vor- und Nachteile

CLOUD NATIVE COMPUTING

Vorteile

- **Einfache horizontale Skalierbarkeit**
  da jeder Dienst unabhängig auf Ereignisse in Echtzeit reagieren kann.
- **Flexibilität**
  EDA ermöglicht eine einfache Integration von Diensten. Neue Dienste können problemlos in das System integriert werden, indem sie einfach Ereignisse senden oder empfangen.
- **Echtzeitverarbeitung**
  von Ereignissen, da Anwendungen oder Dienste sofort auf Ereignisse reagieren können, anstatt auf eine zentrale Schnittstelle warten zu müssen.

Nachteile

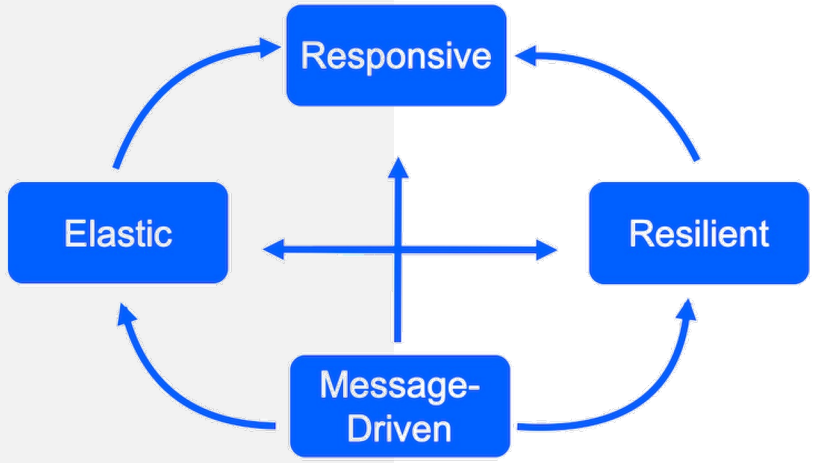
- **Komplexität**
  EDA ist oft komplexer, da jeder Dienst unabhängig von anderen agiert und auf Ereignisse reagiert. Das erfordert eine verteilte Koordination von Ereignissen zwischen Diensten.
- **Mangelnde zentrale Kontrolle**
  EDA erfordert daher eine genaue Überwachung der Ereignisse. Dies kann schwieriger sein als bei Request-Response-basierten Ansätzen, bei denen eine zentrale Schnittstelle koordiniert.
- **Schwierige Fehlerbehebung**
  EDA kann aufgrund der dezentralen Natur der Architektur schwieriger zu debuggen und zu diagnostizieren sein, da Ereignisse von verschiedenen Diensten ausgelöst werden können.

EREIGNISGESTEUERTE ARCHITEKTUREN (EDA)

CLOUD NATIVE COMPUTING

Das Reaktive Manifest

*„Anforderungen, die heute an Computersysteme gestellt werden, sind nur zu erfüllen durch die gleichzeitige Ausrichtung an vier Qualitätseigenschaften, deren Wert bislang nur einzeln betrachtet wurde: Systeme müssen stets **antwortbereit (responsive)**, **widerstandsfähig (resilient)**, **elastisch (elastic)** und **nachrichtenorientiert (message driven)** sein. Dann nennen wir sie reaktive Systeme.“*



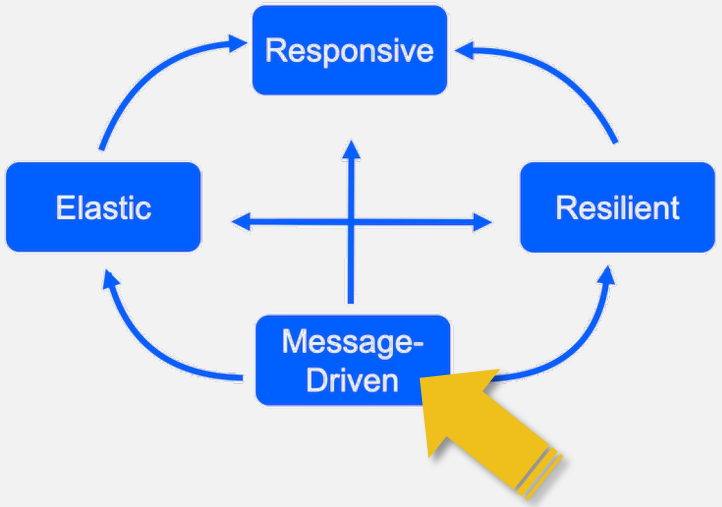
- Microservice Architekturen bestehen stets aus mehreren Komponenten und sind daher abhängig von deren Reaktivität.
- Deshalb müssen die genannten vier reaktiven Qualitäten in der Architektur einer jeden Ebene des Gesamtsystems berücksichtigt werden.
- Hierdurch sind reaktive Subsysteme miteinander komponierbar.

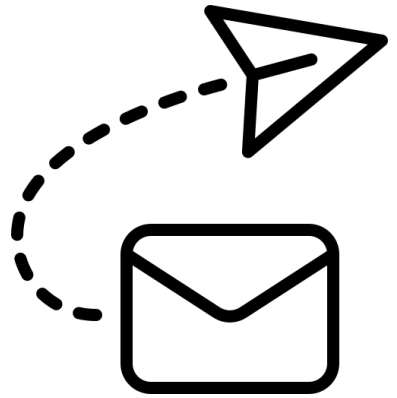
Quelle: https://www.reactivemanifesto.org/de

REAKTIVE SYSTEME

Sind nachrichten- oder ereignisbasiert (engl.: message driven)

- Das System verwendet **asynchrone Nachrichtenübermittlung** zwischen seinen Komponenten zur Sicherstellung von deren Entkopplung und Isolation.
- Die explizite Verwendung von Nachrichten führt zu **ortsunabhängigen** Komponenten und erlaubt die transparente Skalierung.
- **Ortsunabhängigkeit** bedeutet, dass Code und Semantik des Programms nicht davon abhängen, ob dessen Teile auf demselben Computer oder verteilt über ein Netzwerk ausgeführt werden
- Die **Überwachung von Nachrichtenpuffern** ermöglicht kontinuierlichen Einblick in das Laufzeitverhalten des Systems.
- Nicht-blockierende nachrichtenorientierte Systeme erlauben eine **effiziente Verwendung von Ressourcen**, da Komponenten beim Ausbleiben von Nachrichten vollständig inaktiv bleiben können.



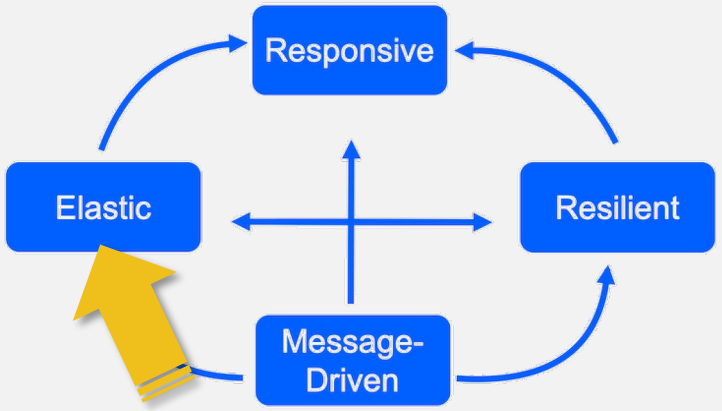


Quelle: https://www.reactivemanifesto.org/de

REAKTIVE SYSTEME

Sind elastisch (engl.: elastic)

- Das System bleibt auch unter sich **ändernden Lastbedingungen** antwortbereit.
- Bei Laständerungen werden automatisch die genutzten Ressourcen angepasst **(Up- oder Down-Scaling)**.
- Das System wird in **unabhängige Komponenten** gegliedert, die auf beliebig viele Ressourcen verteilt werden können (horizontale Skalierbarkeit)
- Reaktive Systeme unterstützen die Erfassung ihrer **Auslastung zur Laufzeit**, um automatisch regelnd eingreifen zu können.
- Dank ihrer **horizontalen Elastizität** können sie auf Speziallösungen verzichten und mit handelsüblichen Komponenten implementiert werden.

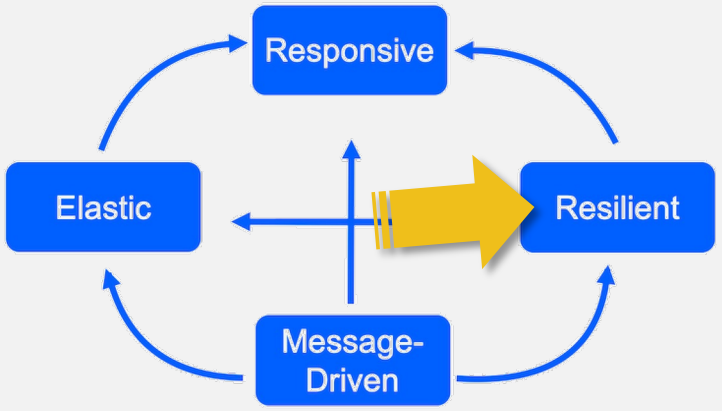


Quelle: https://www.reactivemanifesto.org/de

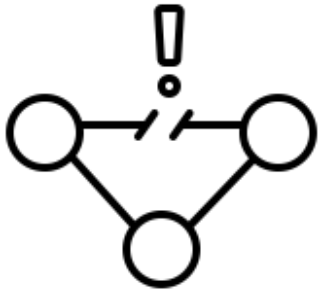
REAKTIVE SYSTEME

Sind widerstandsfähig (engl.: reslient)

- Widerstandsfähigkeit ist erreichbar durch
  - **Replizieren** von Funktionalität
  - **Isolation** von Komponenten
  - **Delegation** der Wiederherstellungs-Verantwortung



- **Replikation:** Komponenten können mehrfach ausgeprägt werden
- **Isolation:** Der Ausfall eines Teilsystems soll auf dieses begrenzt bleiben, um andere Teilsysteme nicht in ihrer Funktion zu beeinträchtigen
- **Delegation:** Die Wiederherstellung des Normalzustandes wird einer übergeordneten Komponente übertragen, die die geforderte Verfügbarkeit sicherstellt

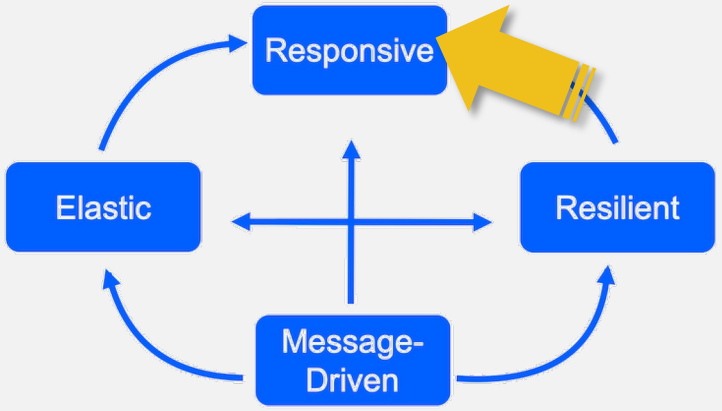


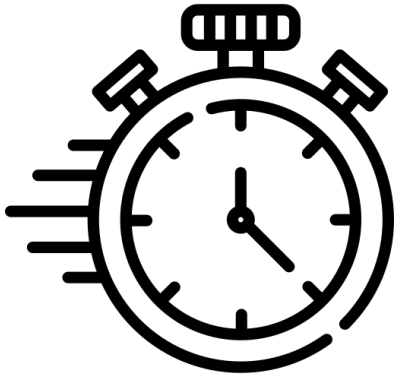
Quelle: https://www.reactivemanifesto.org/de

REAKTIVE SYSTEME

Sind antwortbereit (engl.: responsive)

- Antwortbereitschaft ist die **Grundlage** für Funktion, Benutzbarkeit und Zuverlässigkeit eines Systems
- Ziel ist also das ein System **zeitgerecht** antwortet
- Fehler können in verteilten Systemen nur durch die **Abwesenheit einer Antwort** sicher festgestellt werden
- Ohne vereinbarte **Timeouts** ist die Erkennung und Behandlung von Fehlern nicht möglich





Quelle: https://www.reactivemanifesto.org/de

REACTIVE PROGRAMMING (RX)

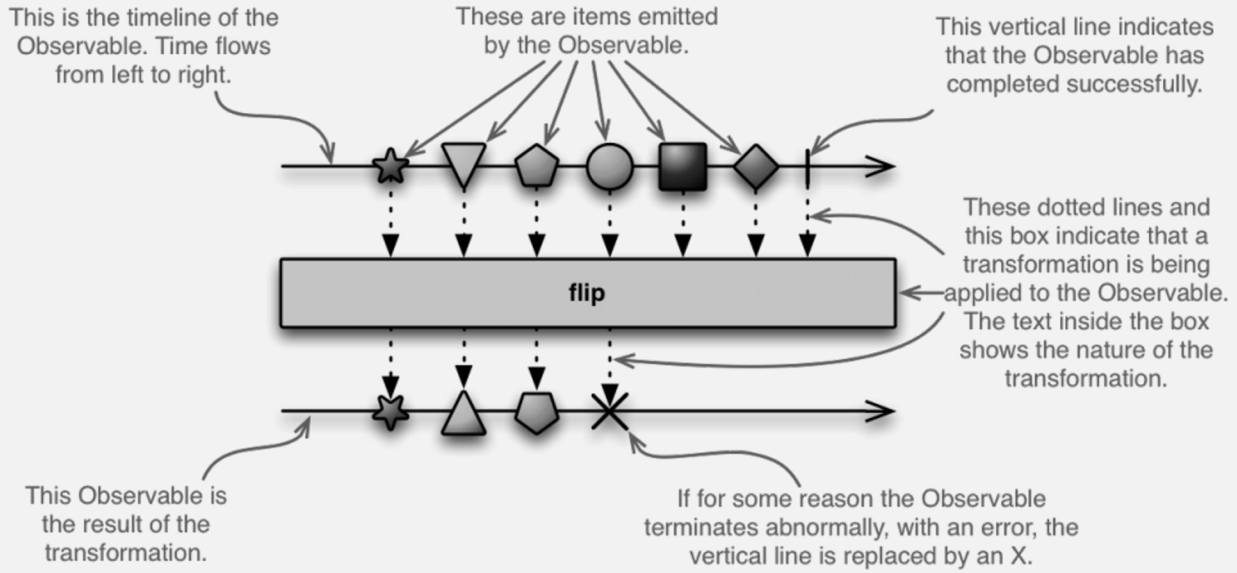
Reaktive Erweiterungen von Programmiersprachen (RxPython / Java / JS / ...)

Reaktive Erweiterungen von Programmiersprachen (Rx) sind ein Mechanismus, mit dem die Ergebnisse mehrerer Ereignisströme zusammengeführt und Operationen auf diesem Event-Stream reaktiv ausgeführt werden können.

Im Kern kehrt Rx traditionelle Kontrollflüsse um.

Anstatt nach Daten zu fragen (und auf die Antwort zu warten), beobachtet man das Ergebnis einer Operation und reagiert, wenn sich etwas ändert. Mittels Rx-Bibliotheken können Funktionen (wie filter(), map(), reduce(), uvm.) auf solchen Observables ausgeführt werden.

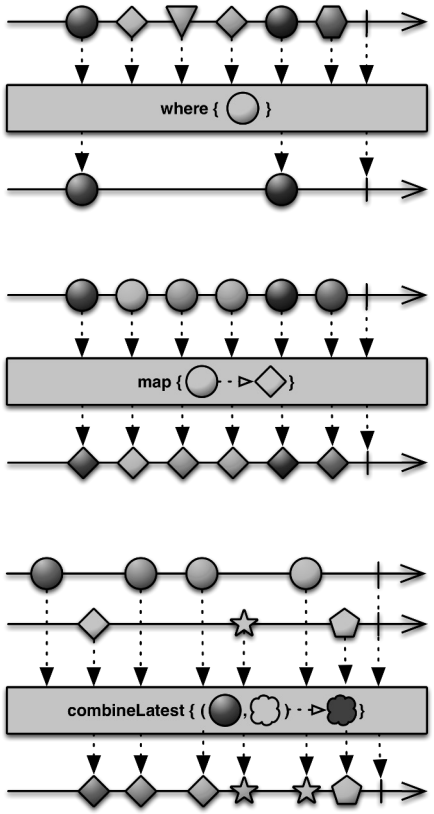
Verschiedenste Rx-Implementierungen erfahren im Rahmen Event-Driven Architectures steigende Beliebtheit, da es diese Bibliotheken ermöglichen, die Details der asynchronen Ereignisabwicklung zu vereinfachen.



Quelle der Diagramme: https://rxpy.readthedocs.io

CLOUD NATIVE COMPUTING

Beispiele reaktiver Operatoren in RxPY (Marble Diagrams)



AUSBLICK

Microservices

- Was ist das für ein Architekturstil?
- Vor- und Nachteile von Microservices

Randbedingungen

- Lose Kopplung und hohe Kohäsion
- Bounded Context und Domain Driven Design
- Conways Law
- Service Ownership und Team-Strukturen

Integration

- Shared Databases, Sync vs. Async, RPC
- REST, Services as State Machines (HATEOS)
- Versioning

Skalierung von Microservices

- Architectural Safety Measures
- Scaling
- Caching

7 Prinzipien von Microservices

- Model Around Business Concepts
- Adopt a Culture of Automation
- Hide Internal Implementation Details
- Decentralize All the Things
- Independently Deployable
- Isolate Failure
- Highly Observable

KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de