Reaktive Architekturen

Neue Ansätze für komplexe Datenflüsse

Leonard Thiele

18. Dezember 2018

Master Grundseminar, HAW Hamburg, 18.12.2018

1. Motivation

2. Reaktiver Ansatz

3. Reaktive Architekturen in der Praxis

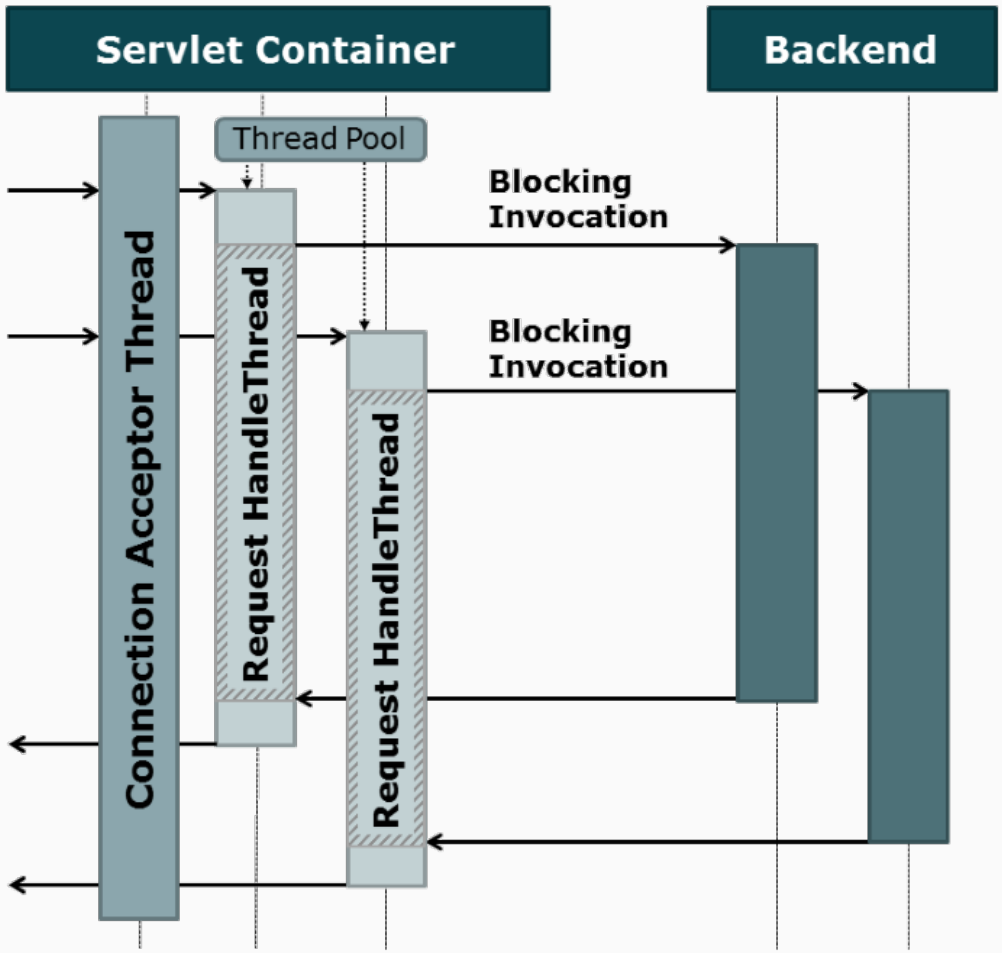
Motivation

Anforderungen an moderne Software

- konstante Verfügbarkeit
- hohe Reaktionsfähigkeit
- gute Skalierbarkeit

Synchroner Ansatz

- direkte, blockierende Aufrufe
- Thread-basierte Verarbeitung



https://blog.senacor.com/content/images/2018/04/ServletModel700.png

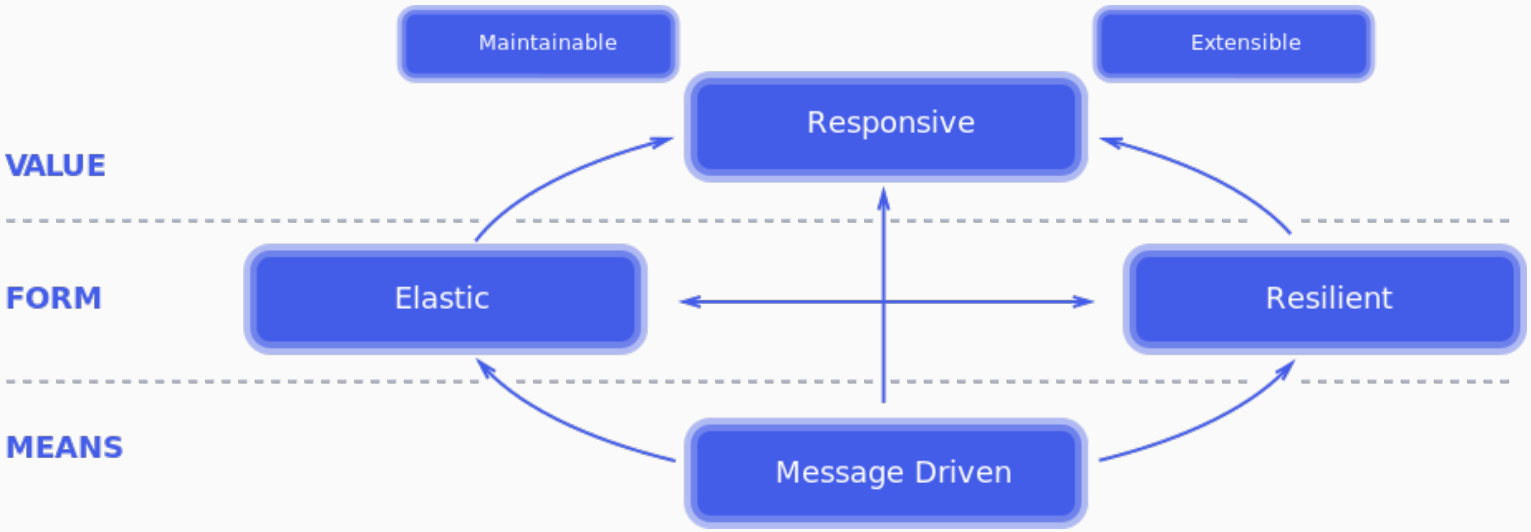
Probleme bei synchroner Architektur

- jeder Request bekommt eigenen Thread/State
  ↪ Overhead
- Threadpools können das Problem lösen...
- ...skalieren aber nicht gut

Was tun wir, wenn 10 000 Nachrichten gleichzeitig verarbeitet werden sollen?

Reaktiver Ansatz

Das Reaktive Manifest

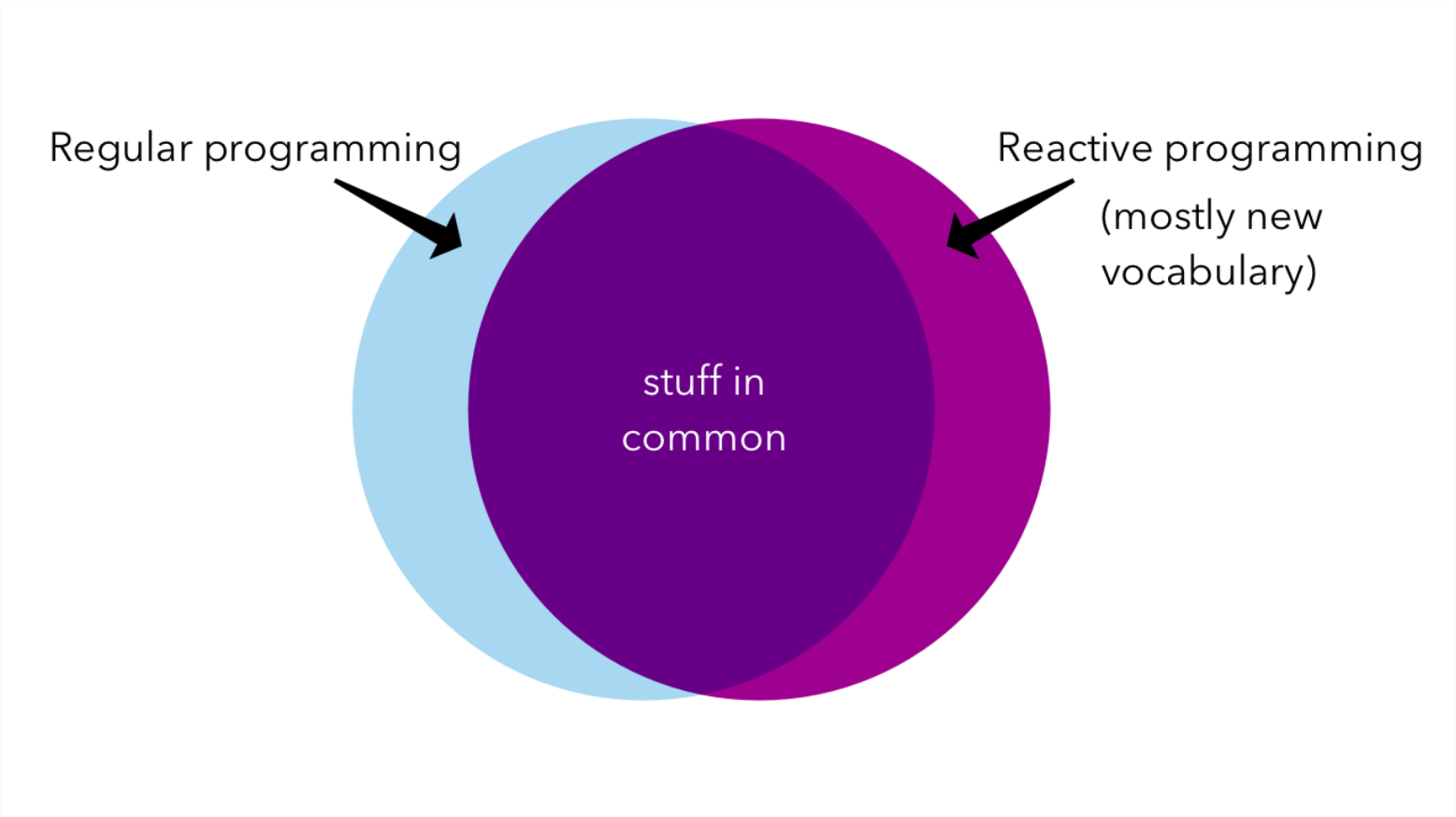


https://www.reactivemanifesto.org/images/reactive-traits.svg

Reaktive Systeme sind...

- **responsive:** Antworten kommen auf jeden Fall innerhalb festgelegter Zeiträume
- **resilient:** Stabilität auch bei Ausfall
- **elastic:** Anpassung an die aktuelle Last
- **message driven:** Effiziente, asynchrone Nachrichtenübermittlung zwischen Komponenten

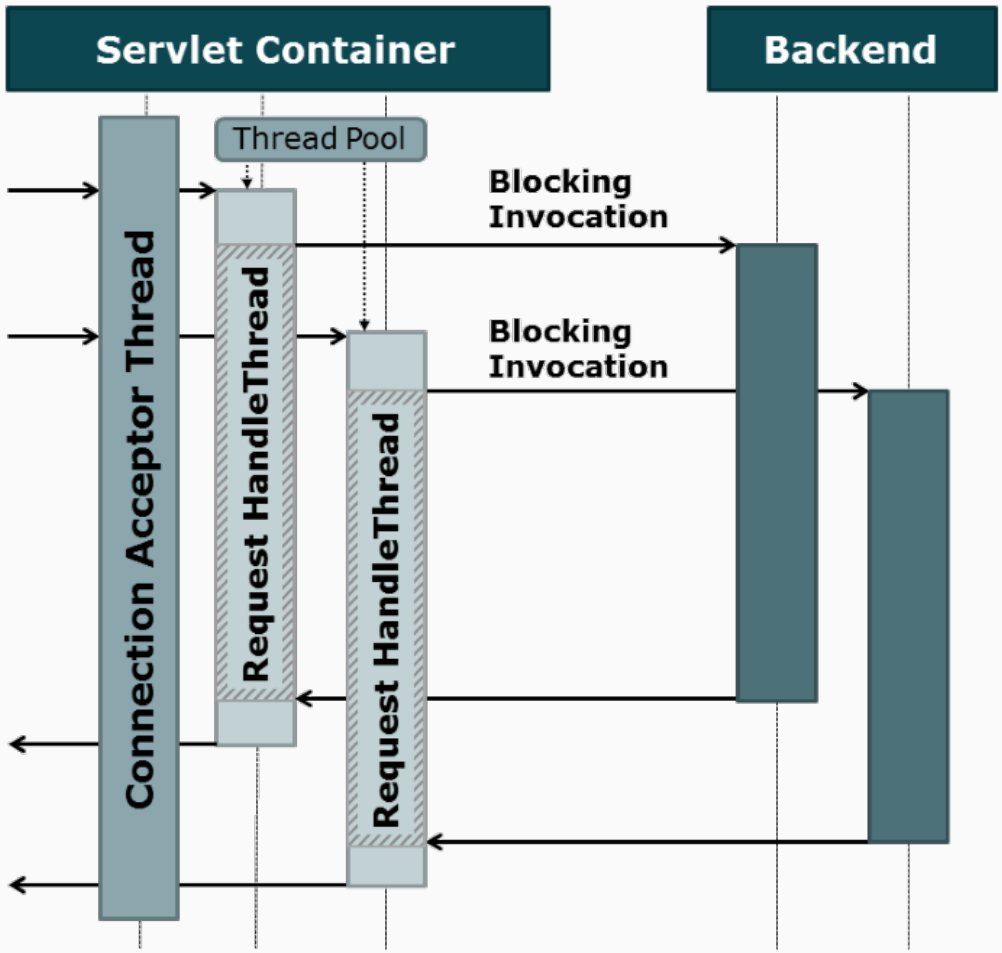
Ist “reaktiv” wirklich neu?



https://gregheo.com/images/blog/rx-venn.png

Synchroner Ansatz

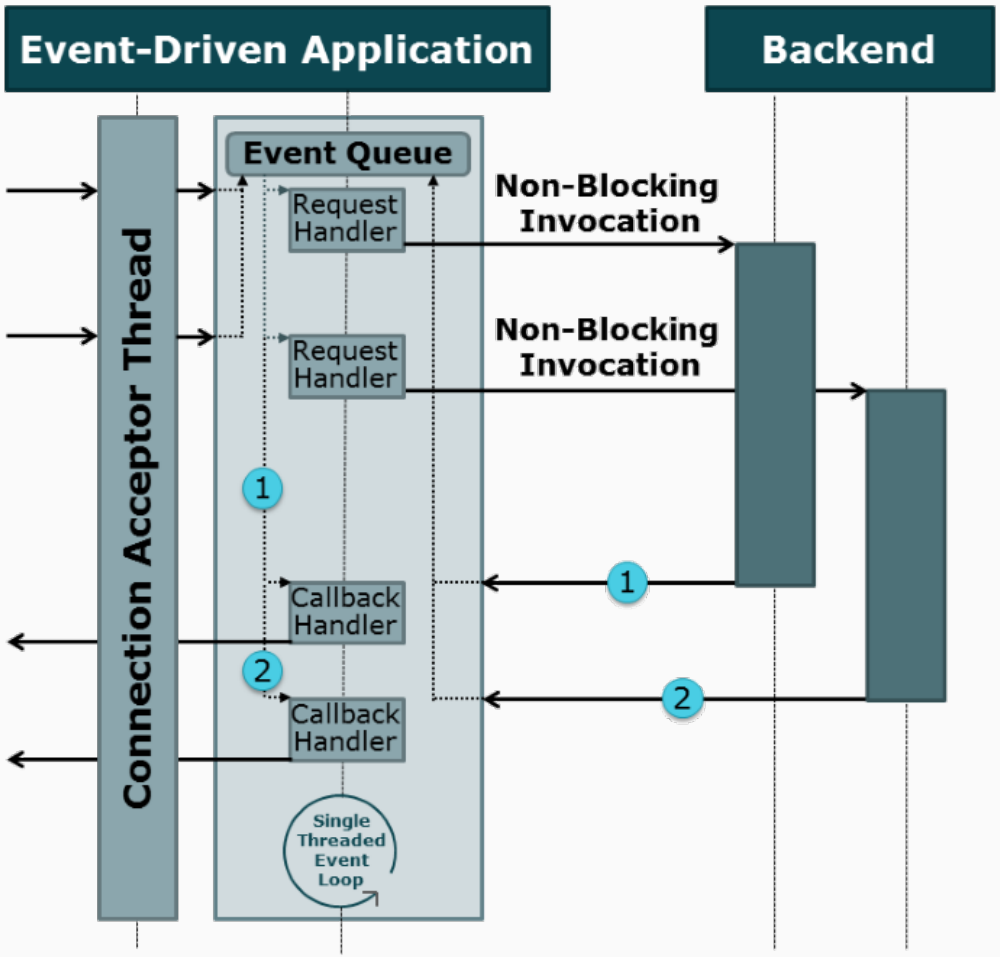
- direkte, blockierende Aufrufe
- zustandsbehaftet
- Thread-basierte Verarbeitung



https://blog.senacor.com/content/images/2018/04/ServletModel700.png

Reaktiver Ansatz

- indirekte, asynchrone Aufrufe
- zustandslos
- Event-basierte Verarbeitung



https://blog.senacor.com/content/images/2018/04/EventLoop700.png

Frameworks

- Eclipse Vert.x
- ReactiveX (RxJava, RxJS...)
- Spring Reactor (seit Spring 5)
- Akka

Reaktive Architekturen in der Praxis

Synchrone Verarbeitung

```
try {
  //lade Kundendaten
  Customer customer = customerService.findById (customerId);
  //aktualisere Kundendaten
  customer.setAddress( ... );
  //speichere Kundendaten
  customerService.update(customer);
}
catch (RuntimeException ex) {
    //verarbeite Fehler
}
```

Asynchrone Verarbeitung mit Callbacks

```
//lade Kundendaten
eventBus.send(
  "CustomerService.findById", customerId, reply ⇒ {
  //prüfe auf Fehler
  if (reply.succeeded()) {
      //Customer aus der Antwort holen
    Customer customer = (Customer) reply.result().body();
    //aktualisere Kundendaten
    customer.setAddress( ... );
    //Speichere Kundendaten
    eventBus.send(
      "CustomerService.update", customer, reply ⇒ {
        //Fehlerprüfung und Ergebnisverarbeitung
    });
  } else {
    //Verarbeite Fehler
  }});
```

Asynchrone Verarbeitung mit Streams

```
//asynchrone Methode liefert ein Observable zurück
Observable<Customer> obs =  customerService.findById(id);

//Anmeldung durch den Observer
obs.subscribe(
  customer ⇒ System.out.println(customer),
  error ⇒ System.out.println(error),
  () ⇒ System.out.println("completed")
);
```

Ausrichtung im Master

- HAWAI-Projekt
- Flutter
- Microservices

Literatur

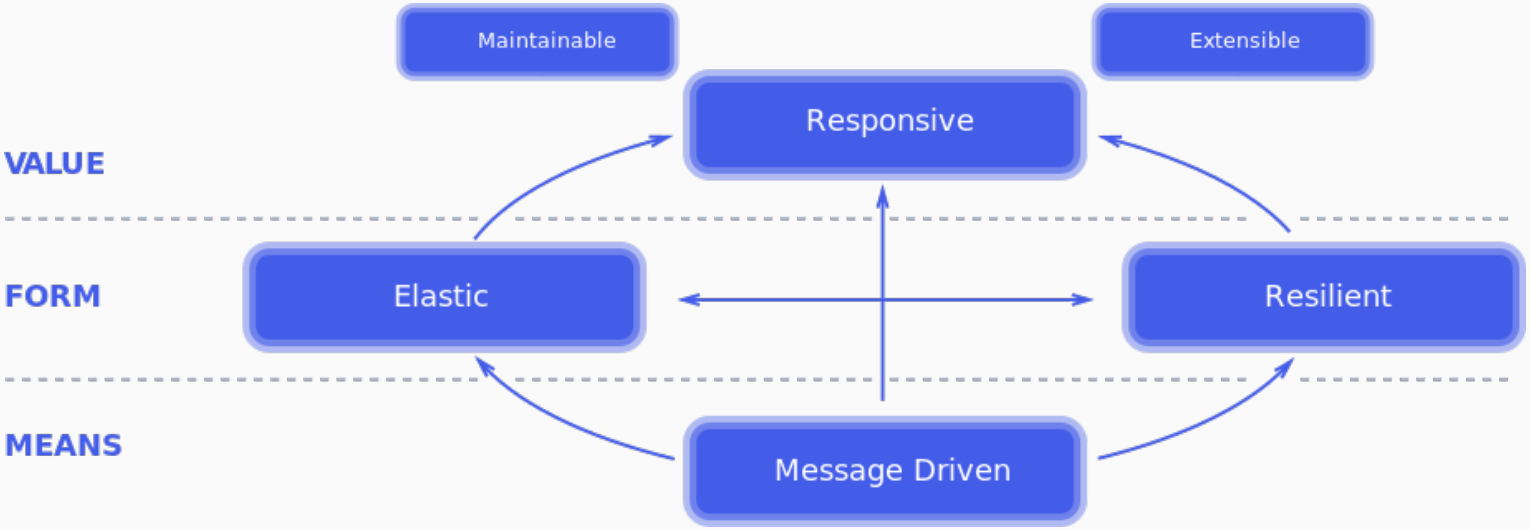
Uwe Friedrichsen, Stefan Toth und Eberhard Wolff. *Resilience - Wie Netflix sein System schützt*. 2015.

Debasish Ghosh. *Functional and Reactive Domain Modeling*. 2016.

Roland Kuhn, Brian Hanafee und Jamie Allen. *Reactive Design Patterns*. 2016.

Michael Menzel. *Reaktive Architekturen mit RxJava*. 18. Dez. 2018. URL: `https://blog.senacor.com/reaktive-architekturen-mit-rxjava/`.

reactive-streams.org. *The introduction to Reactive Programming you've been missing*. 18. Dez. 2018. URL: `http://www.reactive-streams.org/`.



https://www.reactivemanifesto.org/images/reactive-traits.svg

Asynchrone Verarbeitung mit Streams — II

```
eventBus.sendObservable(
  "CustomerService.findById", customerId)
  //Customer aus der Antwort holen
  .map(asyncResult ⇒ (Customer) asyncResult.body())
  //aktualisere Kundendaten
  .map(customer ⇒  ... )
  //Speichere Kundendaten
  .flatMap(customer ⇒ eventBus.sendObservable(
    "CustomerService.update", customer))
  //Konsumiere Ergebnis und Fehler
  .subscribe(customer ⇒ ..., error ⇒ ...);
```

Das BLoC-Pattern