

Reaktive Programmierung

Vorlesung 11 vom 31.05.17: Reactive Streams II

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2017



Fahrplan

- ▶ Einführung
- ▶ Monaden als Berechnungsmuster
- ▶ Nebenläufigkeit: Futures and Promises
- ▶ Aktoren I: Grundlagen
- ▶ Aktoren II: Implementation
- ▶ Bidirektionale Programmierung
- ▶ Meta-Programmierung
- ▶ Reaktive Ströme I
- ▶ **Reaktive Ströme II**
- ▶ Functional Reactive Programming
- ▶ Software Transactional Memory
- ▶ Eventual Consistency
- ▶ Robustheit und Entwurfsmuster
- ▶ Theorie der Nebenläufigkeit, Abschluss



Rückblick: Observables

- ▶ Observables sind „asynchrone Iterables“
- ▶ Asynchronität wird durch **Inversion of Control** erreicht
- ▶ Es bleiben drei Probleme:
 - ▶ Die Gesetze der Observable können leicht verletzt werden.
 - ▶ Ausnahmen beenden den Strom - **Fehlerbehandlung?**
 - ▶ Ein zu schneller Observable kann den Empfangenden Thread **überfluten**



Datenstromgesetze

- ▶ onNext*(onError|onComplete)
- ▶ Kann leicht verletzt werden:

```
Observable[Int] { observer =>
  observer.onNext(42)
  observer.onCompleted()
  observer.onNext(1000)
  Subscription()
}
```

- ▶ Wir können die Gesetze erzwingen: CODE DEMO



Fehlerbehandlung

- ▶ Wenn Datenströme Fehler produzieren, können wir diese möglicherweise behandeln.
- ▶ Aber: *Observer.onError* beendet den Strom.

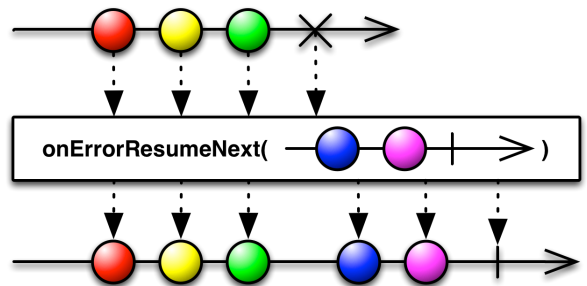
```
observable.subscribe(
  onNext = println,
  onError = ???,
  onCompleted = println("done"))
```

- ▶ *Observer.onError* ist für die Wiederherstellung des Stroms ungeeignet!
- ▶ Idee: Wir brauchen mehr Kombinatoren!



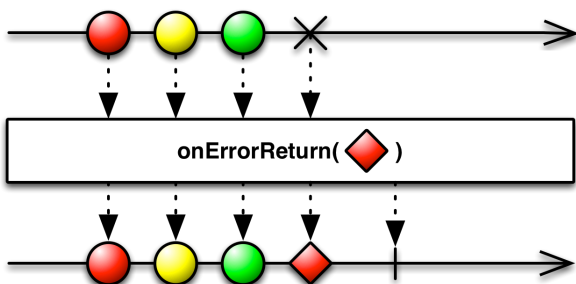
onErrorResumeNext

```
def onErrorResumeNext(f: => Observable[T]): Observable[T]
```



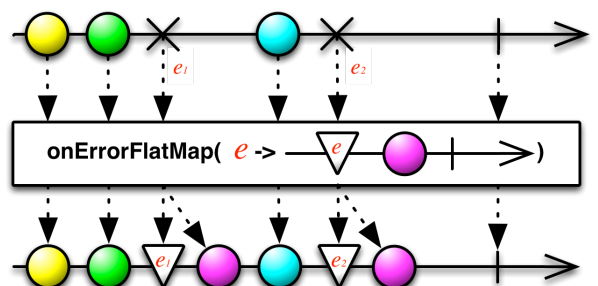
onErrorReturn

```
def onErrorReturn(f: => T): Observable[T]
```



onErrorFlatMap

```
def onErrorFlatMap(f: Throwable => Observable[T]): Observable[T]
```



Schedulers

- ▶ Nebenläufigkeit über Scheduler

```
trait Scheduler {  
  def schedule(work: => Unit): Subscription  
}  
  
trait Observable[T] {  
  ...  
  def observeOn(scheduler: Scheduler): Observable[T]  
}
```

- ▶ CODE DEMO

RP SS 2017

9 [50]



Little's Gesetz

- ▶ In einer stabilen Warteschlange gilt:

$$L = \lambda \times W$$

- ▶ Länge der Warteschlange = Ankunftsrate \times Durchschnittliche Wartezeit
- ▶ Ankunftsrate = $\frac{\text{Länge der Warteschlange}}{\text{Durchschnittliche Wartezeit}}$
- ▶ Wenn ein Datenstrom über einen längeren Zeitraum mit einer Frequenz $> \lambda$ Daten produziert, haben wir ein Problem!

RP SS 2017

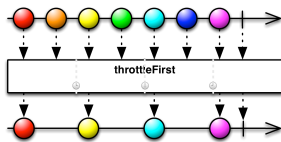
10 [50]



Throttling / Debouncing

- ▶ Wenn wir L und W kennen, können wir λ bestimmen. Wenn λ überschritten wird, müssen wir etwas unternehmen.
- ▶ Idee: Throttling

```
stream.throttleFirst(lambda)
```



- ▶ Problem: Kurzzeitige Überschreitungen von λ sollen nicht zu Throttling führen.

RP SS 2017

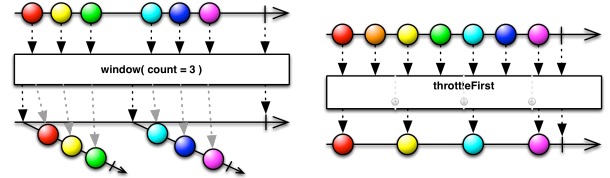
11 [50]



Throttling / Debouncing

- ▶ Besser: Throttling erst bei längerer Überschreitung der Kapazität:

```
stream.window(count = L)  
  .throttleFirst(lambda * L)
```



- ▶ Was ist wenn wir selbst die Daten Produzieren?

RP SS 2017

12 [50]



Back Pressure

- ▶ Wenn wir Kontrolle über die Produktion der Daten haben, ist es unsinnig, sie wegzuerwerfen!
- ▶ Wenn der Konsument keine Daten mehr annehmen kann soll der Produzent aufhören sie zu Produzieren.
- ▶ Erste Idee: Wir können den produzierenden Thread blockieren

```
observable.observeOn(producerThread)  
  .subscribe(onNext = someExpensiveComputation)
```

- ▶ Reaktive Datenströme sollen aber gerade verhindern, dass Threads blockiert werden!

RP SS 2017

13 [50]



Back Pressure

- ▶ Bessere Idee: der Konsument muss mehr Kontrolle bekommen!

```
trait Subscription {  
  def isUnsubscribed: Boolean  
  def unsubscribe(): Unit  
  def requestMore(n: Int): Unit  
}
```

- ▶ Aufwändig in Observables zu implementieren!
- ▶ Siehe <http://www.reactive-streams.org/>

RP SS 2017

14 [50]



Reactive Streams Initiative

- ▶ Ingenieure von Kaazing, Netflix, Pivotal, RedHat, Twitter und Typesafe haben einen offenen Standard für reaktive Ströme entwickelt
- ▶ Minimales Interface (Java + JavaScript)
- ▶ Ausführliche Spezifikation
- ▶ Umfangreiches **Technology Compatibility Kit**
- ▶ Führt unterschiedlichste Bibliotheken zusammen
 - ▶ **JavaRx**
 - ▶ **akka streams**
 - ▶ **Slick 3.0** (Datenbank FRM)
 - ▶ ...
- ▶ Außerdem in Arbeit: Spezifikationen für Netzwerkprotokolle

RP SS 2017

15 [50]



Reactive Streams: Interfaces

- ▶ **Publisher [O]** – Stellt eine potentiell unendliche Sequenz von Elementen zur Verfügung. Die Produktionsrate richtet sich nach der Nachfrage der Subscriber
- ▶ **Subscriber [I]** – Konsumiert Elemente eines Publishers
- ▶ **Subscription** – Repräsentiert ein eins zu eins Abonnement eines Subscribers an einen Publisher
- ▶ **Processor [I, O]** – Ein Verarbeitungsschritt. Gleichzeitig Publisher und Subscriber

RP SS 2017

16 [50]



Reactive Streams: 1. Publisher [T]

`def subscribe(s: Subscriber[T]): Unit`

1. The total number of `onNext` signals sent by a `Publisher` to a `Subscriber` **MUST** be less than or equal to the total number of elements requested by that `Subscriber`'s `Subscription` at all times.
2. A `Publisher` **MAY** signal less `onNext` than requested and terminate the `Subscription` by calling `onComplete` or `onError`.
3. `onSubscribe`, `onNext`, `onError` and `onComplete` signaled to a `Subscriber` **MUST** be signaled sequentially (no concurrent notifications).
4. If a `Publisher` fails it **MUST** signal an `onError`.
5. If a `Publisher` terminates successfully (finite stream) it **MUST** signal an `onComplete`.
6. If a `Publisher` signals either `onError` or `onComplete` on a `Subscriber`, that `Subscriber`'s `Subscription` **MUST** be considered cancelled.

RP SS 2017

17 [50]



Reactive Streams: 1. Publisher [T]

`def subscribe(s: Subscriber[T]): Unit`

7. Once a terminal state has been signaled (`onError`, `onComplete`) it is **REQUIRED** that no further signals occur.
8. If a `Subscription` is cancelled its `Subscriber` **MUST** eventually stop being signaled.
9. `Publisher.subscribe` **MUST** call `onSubscribe` on the provided `Subscriber` prior to any other signals to that `Subscriber` and **MUST** return normally, except when the provided `Subscriber` is null in which case it **MUST** throw a `java.lang.NullPointerException` to the caller, for all other situations the only legal way to signal failure (or reject the `Subscriber`) is by calling `onError` (after calling `onSubscribe`).
10. `Publisher.subscribe` **MAY** be called as many times as wanted but **MUST** be with a different `Subscriber` each time.
11. A `Publisher` **MAY** support multiple `Subscribers` and decides whether each `Subscription` is unicast or multicast.

RP SS 2017

18 [50]



Reactive Streams: 2. Subscriber [T]

`def onComplete(): Unit`

`def onError(t: Throwable): Unit`

`def onNext(t: T): Unit`

`def onSubscribe(s: Subscription): Unit`

1. A `Subscriber` **MUST** signal demand via `Subscription.request(long n)` to receive `onNext` signals.
2. If a `Subscriber` suspects that its processing of signals will negatively impact its `Publisher`'s responsiveness, it is **RECOMMENDED** that it asynchronously dispatches its signals.
3. `Subscriber.onComplete()` and `Subscriber.onError(Throwable t)` **MUST NOT** call any methods on the `Subscription` or the `Publisher`.
4. `Subscriber.onComplete()` and `Subscriber.onError(Throwable t)` **MUST** consider the `Subscription` cancelled after having received the signal.
5. A `Subscriber` **MUST** call `Subscription.cancel()` on the given `Subscription` after an `onSubscribe` signal if it already has an active `Subscription`.

RP SS 2017

19 [50]



Reactive Streams: 2. Subscriber [T]

`def onComplete(): Unit`

`def onError(t: Throwable): Unit`

`def onNext(t: T): Unit`

`def onSubscribe(s: Subscription): Unit`

6. A `Subscriber` **MUST** call `Subscription.cancel()` if it is no longer valid to the `Publisher` without the `Publisher` having signaled `onError` or `onComplete`.
7. A `Subscriber` **MUST** ensure that all calls on its `Subscription` take place from the same thread or provide for respective external synchronization.
8. A `Subscriber` **MUST** be prepared to receive one or more `onNext` signals after having called `Subscription.cancel()` if there are still requested elements pending. `Subscription.cancel()` does not guarantee to perform the underlying cleaning operations immediately.
9. A `Subscriber` **MUST** be prepared to receive an `onComplete` signal with or without a preceding `Subscription.request(long n)` call.
10. A `Subscriber` **MUST** be prepared to receive an `onError` signal with or without a preceding `Subscription.request(long n)` call.

RP SS 2017

20 [50]



Reactive Streams: 2. Subscriber [T]

`def onComplete(): Unit`

`def onError(t: Throwable): Unit`

`def onNext(t: T): Unit`

`def onSubscribe(s: Subscription): Unit`

11. A `Subscriber` **MUST** make sure that all calls on its `onXXX` methods happen-before the processing of the respective signals. I.e. the `Subscriber` must take care of properly publishing the signal to its processing logic.
12. `Subscriber.onSubscribe` **MUST** be called at most once for a given `Subscriber` (based on object equality).
13. Calling `onSubscribe`, `onNext`, `onError` or `onComplete` **MUST** return normally except when any provided parameter is null in which case it **MUST** throw a `java.lang.NullPointerException` to the caller, for all other situations the only legal way for a `Subscriber` to signal failure is by cancelling its `Subscription`. In the case that this rule is violated, any associated `Subscription` to the `Subscriber` **MUST** be considered as cancelled, and the caller **MUST** raise this error condition in a fashion that is adequate for the runtime environment.

RP SS 2017

21 [50]



Reactive Streams: 3. Subscription

`def cancel(): Unit`

`def request(n: Long): Unit`

1. `Subscription.request` and `Subscription.cancel` **MUST** only be called inside of its `Subscriber` context. A `Subscription` represents the unique relationship between a `Subscriber` and a `Publisher`.
2. The `Subscription` **MUST** allow the `Subscriber` to call `Subscription.request` synchronously from within `onNext` or `onSubscribe`.
3. `Subscription.request` **MUST** place an upper bound on possible synchronous recursion between `Publisher` and `Subscriber`.
4. `Subscription.request` **SHOULD** respect the responsiveness of its caller by returning in a timely manner.
5. `Subscription.cancel` **MUST** respect the responsiveness of its caller by returning in a timely manner, **MUST** be idempotent and **MUST** be thread-safe.
6. After the `Subscription` is cancelled, additional `Subscription.request(long n)` **MUST** be NOPs.

RP SS 2017

22 [50]



Reactive Streams: 3. Subscription

`def cancel(): Unit`

`def request(n: Long): Unit`

7. After the `Subscription` is cancelled, additional `Subscription.cancel()` **MUST** be NOPs.
8. While the `Subscription` is not cancelled, `Subscription.request(long n)` **MUST** register the given number of additional elements to be produced to the respective subscriber.
9. While the `Subscription` is not cancelled, `Subscription.request(long n)` **MUST** signal `onError` with a `java.lang.IllegalArgumentException` if the argument is ≤ 0 . The cause message **MUST** include a reference to this rule and/or quote the full rule.
10. While the `Subscription` is not cancelled, `Subscription.request(long n)` **MAY** synchronously call `onNext` on this (or other) subscriber(s).
11. While the `Subscription` is not cancelled, `Subscription.request(long n)` **MAY** synchronously call `onComplete` or `onError` on this (or other) subscriber(s).

RP SS 2017

23 [50]



Reactive Streams: 3. Subscription

`def cancel(): Unit`

`def request(n: Long): Unit`

12. While the `Subscription` is not cancelled, `Subscription.cancel()` **MUST** request the `Publisher` to eventually stop signaling its `Subscriber`. The operation is **NOT REQUIRED** to affect the `Subscription` immediately.
13. While the `Subscription` is not cancelled, `Subscription.cancel()` **MUST** request the `Publisher` to eventually drop any references to the corresponding subscriber. Re-subscribing with the same `Subscriber` object is discouraged, but this specification does not mandate that it is disallowed since that would mean having to store previously cancelled subscriptions indefinitely.
14. While the `Subscription` is not cancelled, calling `Subscription.cancel` **MAY** cause the `Publisher`, if stateful, to transition into the shut-down state if no other `Subscription` exists at this point.

RP SS 2017

24 [50]



Reactive Streams: 3. Subscription

```
def cancel(): Unit
def request(n: Long): Unit
```

16. Calling `Subscription.cancel` MUST return normally. The only legal way to signal failure to a Subscriber is via the `onError` method.
17. Calling `Subscription.request` MUST return normally. The only legal way to signal failure to a Subscriber is via the `onError` method.
18. A `Subscription` MUST support an unbounded number of calls to `request` and MUST support a demand (sum requested - sum delivered) up to $2^{63} - 1$ (`java.lang.Long.MAX_VALUE`). A demand equal or greater than $2^{63} - 1$ (`java.lang.Long.MAX_VALUE`) MAY be considered by the Publisher as "effectively unbounded".

RP SS 2017

25 [50]



Reactive Streams: 4. Processor[I, O]

```
def onComplete(): Unit
def onError(t: Throwable): Unit
def onNext(t: I): Unit
def onSubscribe(s: Subscription): Unit
def subscribe(s: Subscriber[O]): Unit
```

1. A Processor represents a processing stage — which is both a Subscriber and a Publisher and MUST obey the contracts of both.
2. A Processor MAY choose to recover an `onError` signal. If it chooses to do so, it MUST consider the `Subscription` cancelled, otherwise it MUST propagate the `onError` signal to its Subscribers immediately.

RP SS 2017

26 [50]



Akka Streams

- ▶ Vollständige Implementierung der Reactive Streams Spezifikation
- ▶ Basiert auf Datenflussgraphen und Materialisierern
- ▶ Datenflussgraphen werden als Aktornetzwerk materialisiert

RP SS 2017

27 [50]



Akka Streams - Grundkonzepte

Datenstrom (Stream) – Ein Prozess der Daten überträgt und transformiert
Element – Recheneinheit eines Datenstroms
Back-Pressure – Konsument signalisiert (asynchron) Nachfrage an Produzenten
Verarbeitungsschritt (Processing Stage) – Bezeichnet alle Bausteine aus denen sich ein Datenfluss oder Datenflussgraph zusammensetzt.
Quelle (Source) – Verarbeitungsschritt mit genau einem Ausgang
Abfluss (Sink) – Verarbeitungsschritt mit genau einem Eingang
Datenfluss (Flow) – Verarbeitungsschritt mit jeweils genau einem Ein- und Ausgang
Ausführbarer Datenfluss (RunnableFlow) – Datenfluss der an eine Quelle und einen Abfluss angeschlossen ist

RP SS 2017

28 [50]



Akka Streams - Beispiel

```
implicit val system = ActorSystem("example")
implicit val materializer = ActorFlowMaterializer()

val source = Source(1 to 10)
val sink = Sink.fold[Int, Int](0)(_ + _)
val sum: Future[Int] = source runWith sink
```

RP SS 2017

29 [50]



Datenflussgraphen

- ▶ Operatoren sind Abzweigungen im Graphen
- ▶ z.B. Broadcast (1 Eingang, n Ausgänge) und Merge (n Eingänge, 1 Ausgang)
- ▶ Scala DSL um Graphen darzustellen

```
val g = FlowGraph.closed() { implicit builder =>
  val in = source
  val out = sink
  val bcast = builder.add(Broadcast[Int](2))
  val merge = builder.add(Merge[Int](2))
  val f1, f2, f3, f4 = Flow[Int].map(_ + 10)

  in -> f1 -> bcast -> f2 -> merge -> f3 -> out
    bcast -> f4 -> merge
}
```

RP SS 2017

30 [50]



Operatoren in Datenflussgraphen

- ▶ Auffächern
 - ▶ `Broadcast[T]` – Verteilt eine Eingabe an n Ausgänge
 - ▶ `Balance[T]` – Teilt Eingabe gleichmäßig unter n Ausgängen auf
 - ▶ `UnZip[A,B]` – Macht aus $[(A,B)]$ -Strom zwei Ströme $[A]$ und $[B]$
 - ▶ `FlexiRoute[Int]` – DSL für eigene Fan-Out Operatoren
- ▶ Zusammenführen
 - ▶ `Merge[Int]` – Vereintigt n Ströme in einem
 - ▶ `MergePreferred[Int]` – Wie Merge, hat aber einen präferierten Eingang
 - ▶ `ZipWith[A,B,...,Out]` – Fasst n Eingänge mit einer Funktion f zusammen
 - ▶ `Zip[A,B]` – `ZipWith` mit zwei Eingängen und $f = (_, _)$
 - ▶ `Concat[A]` – Sequentialisiert zwei Ströme
 - ▶ `FlexiMerge[Out]` – DSL für eigene Fan-In Operatoren

RP SS 2017

31 [50]



Partielle Datenflussgraphen

- ▶ Datenflussgraphen können partiell sein:

```
val pickMaxOfThree = FlowGraph.partial() {
  implicit builder =>

  val zip1 = builder.add(ZipWith[Int, Int, Int](math.max))
  val zip2 = builder.add(ZipWith[Int, Int, Int](math.max))

  zip1.out -> zip2.in0

  UniformFanInShape(zip2.out, zip1.in0, zip1.in1, zip2.in1)
}
```

- ▶ Offene Anschlüsse werden später belegt

RP SS 2017

32 [50]



Sources, Sinks und Flows als Datenflussgraphen

- ▶ Source — Graph mit genau einem offenen Ausgang

```
Source(){ implicit builder =>
  outlet
}
```

- ▶ Sink — Graph mit genau einem offenen Eingang

```
Sink() { implicit builder =>
  inlet
}
```

- ▶ Flow — Graph mit jeweils genau einem offenen Ein- und Ausgang

```
Flow() { implicit builder =>
  (inlet, outlet)
}
```

RP SS 2017

33 [50]



Zyklische Datenflussgraphen

- ▶ Zyklen in Datenflussgraphen sind erlaubt:

```
val input = Source(Stream.continually(readLine()))

val flow = FlowGraph.closed() { implicit builder =>
  val merge = builder.add(Merge[String](2))
  val bcast = builder.add(Broadcast[String](2))
  val print = Flow.map{s => println(s); s}

  input -> merge -> print -> bcast -> Sink.ignore
          merge      <-      bcast
}
```

- ▶ Hört nach kurzer Zeit auf etwas zu tun — Wieso?

RP SS 2017

34 [50]



Zyklische Datenflussgraphen

- ▶ Besser:

```
val input = Source(Stream.continually(readLine()))

val flow = FlowGraph.closed() { implicit builder =>
  val merge = builder.add(Merge[String](2))
  val bcast = builder.add(Broadcast[String](2))
  val print = Flow.map{s => println(s); s}
  val buffer = Flow.buffer(10, OverflowStrategy.dropHead)

  input -> merge -> print -> bcast -> Sink.ignore
          merge <- buffer <- bcast
}
```

RP SS 2017

35 [50]



Pufferung

- ▶ Standardmäßig werden bis zu **16 Elemente** gepuffert, um parallele Ausführung von Streams zu erreichen.

- ▶ Danach: Backpressure

```
Source(1 to 3)
  .map( i => println(s"A: $i"); i)
  .map( i => println(s"B: $i"); i)
  .map( i => println(s"C: $i"); i)
  .map( i => println(s"D: $i"); i)
  .runWith(Sink.ignore)
```

- ▶ Ausgabe nicht deterministisch, wegen paralleler Ausführung
- ▶ Puffergrößen können angepasst werden (Systemweit, Materialisierer, Verarbeitungsschritt)

RP SS 2017

36 [50]



Fehlerbehandlung

- ▶ Standardmäßig führen Fehler zum Abbruch:

```
val source = Source(0 to 5).map(100 / _)
val result = source.runWith(Sink.fold(0)(_ + _))
```

- ▶ result = Future(Failure(ArithmeticException))

- ▶ Materialisierer kann mit Supervisor konfiguriert werden:

```
val decider: Supervisor.Decider = {
  case _ : ArithmeticException => Resume
  case _ => Stop
}
implicit val materializer = ActorFlowMaterializer(
  ActorFlowMaterializerSettings(system)
  .withSupervisionStrategy(decider))
```

- ▶ result = Future(Success(228))

RP SS 2017

37 [50]



Integration mit Aktoren - ActorPublisher

- ▶ ActorPublisher ist ein Akteur, der als Source verwendet werden kann.

```
class MyActorPublisher extends ActorPublisher[String] {
  def receive = {
    case Request(n) =>
      for (i <- 1 to n) onNext("Hallo")
    case Cancel =>
      context.stop(self)
  }
}
```

```
Source.actorPublisher(Props[MyActorPublisher])
```

RP SS 2017

38 [50]



Integration mit Aktoren - ActorSubscriber

- ▶ ActorSubscriber ist ein Akteur, der als Sink verwendet werden kann.

```
class MyActorSubscriber extends ActorSubscriber {
  def receive = {
    case OnNext(elem) =>
      log.info("received {}", elem)
    case OnError(e) =>
      throw e
    case OnComplete =>
      context.stop(self)
  }
}
```

```
Source.actorPublisher(Props[MyActorPublisher])
```

RP SS 2017

39 [50]



Integration für einfache Fälle

- ▶ Für einfache Fälle gibt es Source.actorRef und Sink.actorRef

```
val source: Source[Foo, ActorRef] = Source.actorRef[Foo](
  bufferSize = 10,
  overflowStrategy = OverflowStrategy.backpressure)
```

```
val sink: Sink[Foo, Unit] = Sink.actorRef[Foo](
  ref = myActorRef,
  onCompleteMessage = Bar)
```

- ▶ Problem: Sink hat kein Backpressure. Wenn der Akteur nicht schnell genug ist, explodiert alles.

RP SS 2017

40 [50]



Anwendung: akka-http

- ▶ Minimale HTTP-Bibliothek (Client und Server)
- ▶ Basierend auf *akka-streams* — reaktiv
- ▶ From scratch — **keine Altlasten**
- ▶ **Kein Blocking** — Schnell
- ▶ Scala DSL für Routen-Definition
- ▶ Scala DSL für Webaufrufe
- ▶ Umfangreiche Konfigurationsmöglichkeiten

RP SS 2017

41 [50]



Low-Level Server API

- ▶ HTTP-Server wartet auf Anfragen:
`Source[IncomingConnection, Future[ServerBinding]]`

```
val server = Http.bind(interface = "localhost", port = 8080)
```

- ▶ Zu jeder Anfrage gibt es eine Antwort:

```
val requestHandler: HttpRequest => HttpResponse = {  
  case HttpRequest(GET, Uri.Path("/ping"), _, _) =>  
    HttpResponse(entity = "PONG!")  
}  
  
val serverSink =  
  Sink.foreach(_.handleWithSyncHandler(requestHandler))  
  
serverSource.to(serverSink)
```

RP SS 2017

42 [50]



High-Level Server API

- ▶ Minimalbeispiel:

```
implicit val system = ActorSystem("example")  
implicit val materializer = ActorFlowMaterializer()  
  
val routes = path("ping") {  
  get {  
    complete { <h1>PONG!</h1> }  
  }  
}  
  
val binding =  
  Http().bindAndHandle(routes, "localhost", 8080)
```

RP SS 2017

43 [50]



HTTP

- ▶ HTTP ist ein Protokoll aus den frühen 90er Jahren.
- ▶ Grundidee: Client sendet **Anfragen** an Server, Server **antwortet**
- ▶ Verschiedene Arten von Anfragen
 - ▶ GET — Inhalt abrufen
 - ▶ POST — Inhalt zum Server übertragen
 - ▶ PUT — Resource unter bestimmter URI erstellen
 - ▶ DELETE — Resource löschen
 - ▶ ...
- ▶ Antworten mit Statuscode, z.B.:
 - ▶ 200 — Ok
 - ▶ 404 — Not found
 - ▶ 501 — Internal Server Error
 - ▶ ...

RP SS 2017

44 [50]



Das Server-Push Problem

- ▶ HTTP basiert auf der Annahme, dass der Webclient den (statischen) Inhalt **bei Bedarf** anfragt.
- ▶ Moderne Webanwendungen sind alles andere als statisch.
- ▶ Workarounds des letzten Jahrzehnts:
 - ▶ **AJAX** — Eigentlich *Asynchronous JavaScript and XML*, heute eher **AJAJ** — Teile der Seite werden dynamisch ersetzt.
 - ▶ **Polling** — "Gibt's etwas Neues?", "Gibt's etwas Neues?", ...
 - ▶ **Comet** — Anfrage mit langem Timeout wird erst beantwortet, wenn es etwas Neues gibt.
 - ▶ **Chunked Response** — Server antwortet stückchenweise

RP SS 2017

45 [50]



WebSockets

- ▶ TCP-Basiertes **bidirektionales** Protokoll für Webanwendungen
- ▶ Client öffnet nur **einmal** die Verbindung
- ▶ Server und Client können **jederzeit** Daten senden
- ▶ Nachrichten ohne Header (1 Byte)
- ▶ **Ähnlich** wie Aktoren:
 - ▶ JavaScript Client sequentiell mit lokalem Zustand (\approx Actor)
 - ▶ `WebSocket.onmessage` \approx `Actor.receive`
 - ▶ `WebSocket.send(msg)` \approx `sender ! msg`
 - ▶ `WebSocket.onclose` \approx `Actor.postStop`
 - ▶ Außerdem **onerror** für Fehlerbehandlung.

RP SS 2017

46 [50]



WebSockets in akka-http

- ▶ WebSockets ist ein `Flow[Message,Message,Unit]`
- ▶ Können über bidirektional Flows gehandhabt werden
 - ▶ `BidiFlow[-I1, +O1, -I2, +O2, +Mat]` – zwei Eingänge, zwei Ausgänge: Serialisieren und deserialisieren.
- ▶ Beispiel:

```
def routes = get {  
  path("ping")(handleWebsocketMessages(wsFlow))  
}  
  
def wsFlow: Flow[Message,Message,Unit] =  
  BidiFlow.fromFunctions(serialize, deserialize)  
  .join(Flow.collect {  
    case Ping => Pong  
  })
```

RP SS 2017

47 [50]



Zusammenfassung

- ▶ Die Konstruktoren in der Rx Bibliothek wenden viel **Magie** an um Gesetze einzuhalten
- ▶ Fehlerbehandlung durch Kombinatoren ist einfach zu implementieren
- ▶ Observables eignen sich nur bedingt um **Back Pressure** zu implementieren, da Kontrollfluss unidirektional konzipiert.
- ▶ Die *Reactive Streams*-Spezifikation beschreibt ein minimales Interface für Ströme mit Back Pressure
- ▶ Für die Implementierung sind Aktoren sehr gut geeignet \Rightarrow akka streams

RP SS 2017

48 [50]



Zusammenfassung

- ▶ **Datenflussgraphen** repräsentieren reaktive Berechnungen
 - ▶ Geschlossene Datenflussgraphen sind ausführbar
 - ▶ Partielle Datenflussgraphen haben **unbelegte** ein oder ausgänge
 - ▶ **Zyklische** Datenflussgraphen sind erlaubt
- ▶ **Puffer** sorgen für **parallele Ausführung**
- ▶ Supervisor können bestimmte Fehler ignorieren
- ▶ *akka-stream* kann einfach mit *akka-actor* integriert werden
- ▶ Anwendungsbeispiel: *akka-http*
 - ▶ Low-Level API: Request =>Response
 - ▶ HTTP ist **pull basiert**
 - ▶ WebSockets sind **bidirektional** → Flow



Bonusfolie: WebWorkers

- ▶ JavaScript ist singlethreaded.
- ▶ Bibliotheken machen sich keinerlei Gedanken über Race-Conditions.
- ▶ Workaround: Aufwändige Berechnungen werden gestückelt, damit die Seite responsiv bleibt.
- ▶ Lösung: HTML5-WebWorkers (Alle modernen Browser)
 - ▶ `new Worker(file)` startet neuen Worker
 - ▶ Kommunikation über `postMessage`, `onmessage`, `onerror`, `onclose`
 - ▶ Einschränkung: Kein Zugriff auf das DOM — lokaler Zustand
 - ▶ WebWorker können weitere WebWorker erzeugen
 - ▶ "*Poor-Man's Actors*"

