Akka Streams: a match in heaven for Reactive Systems

Henrik Engström, Senior Software Engineer

UJUG presentation 21/09/2017

Lightbend

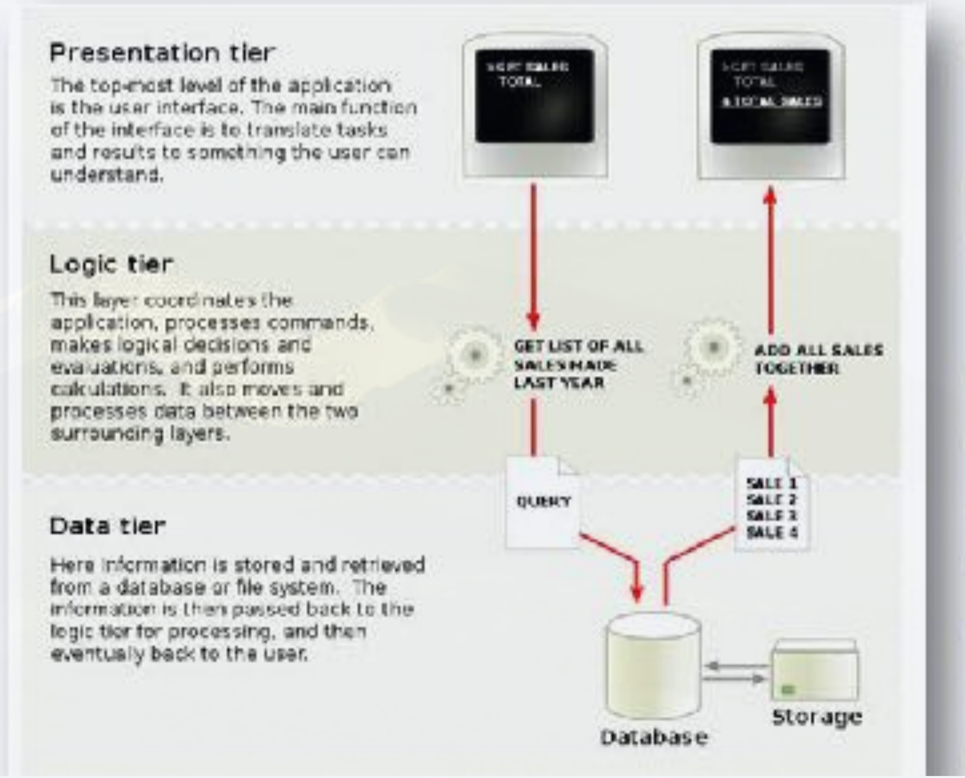
AGENDA - UJUG 21/9/17

- Traditional systems architecture
- Reactive systems architecture
- Akka Actors
- Reactive Streams
- Akka Streams
- Alpakka
- Actors and Streams

PDF/Code found here: **github.com/henrikengstrom/ujug2017**

Traditional systems architecture

Traditional systems architecture



Presentation tier

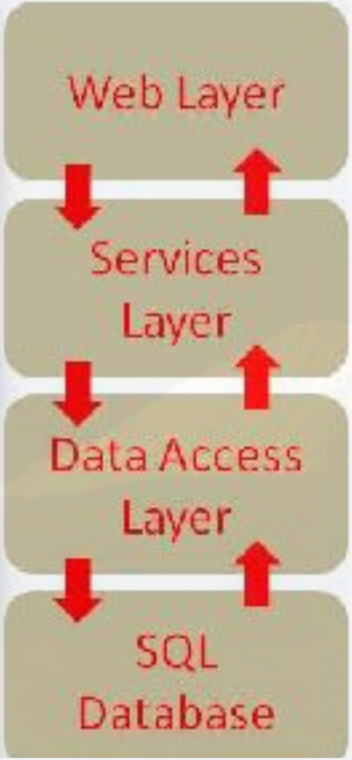
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

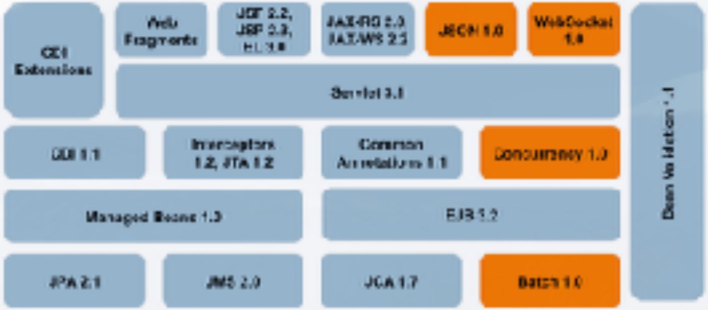
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.



TSA - frameworks and servers



CDI Extensions

Web Fragments

JSF 2.2, JSP 2.3, EL 3.0

JAX-RS 2.0 JAX-WS 2.2

JSON 1.0

WebSocket 1.0

Servlet 3.1

CDI 1.1

Interceptors 1.2, JTA 1.2

Common Annotations 1.1

Concurrency 1.0

Managed Beans 1.0

EJB 3.2

JPA 2.1

JMS 2.0

JCA 1.7

Batch 1.0

Bean Validation 1.1

TSA - cloud

Azure
amazon web services
docker
Google Cloud Platform
kubernetes
DC/OS

TSA - some struggles

- Hard to engineer systems to:
  - withstand load
  - always stay up
  - be performant
  - utilize HW to a maximum

- What can we do about it?

Reactive systems

The Reactive Manifesto

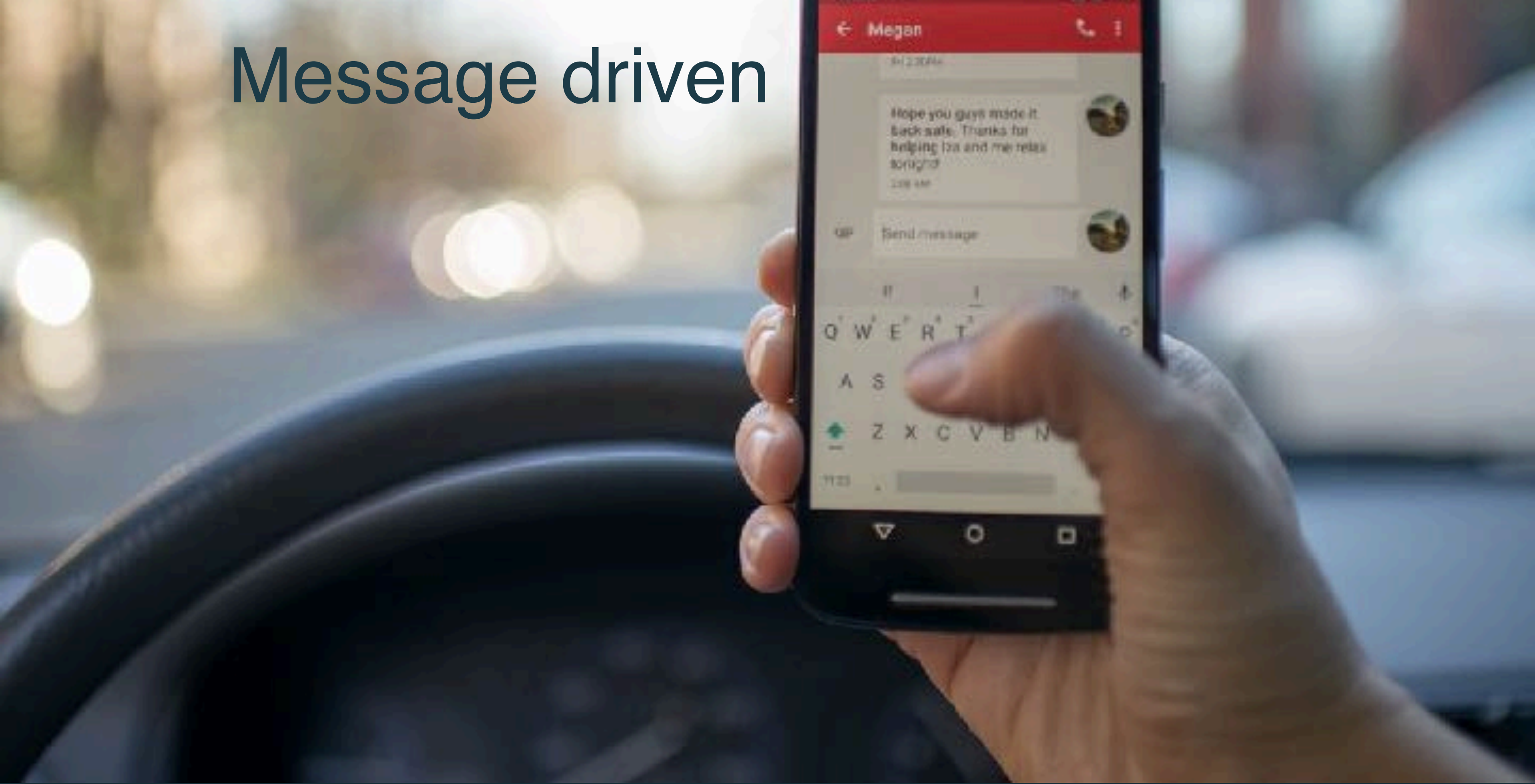
- http://www.reactivemanifesto.org/
- September 16, 2014
- +20k signatures
- Four traits:
  - *Responsive*
  - *Resilient*
  - *Elastic*
  - *Message driven*

Responsive

Resilient

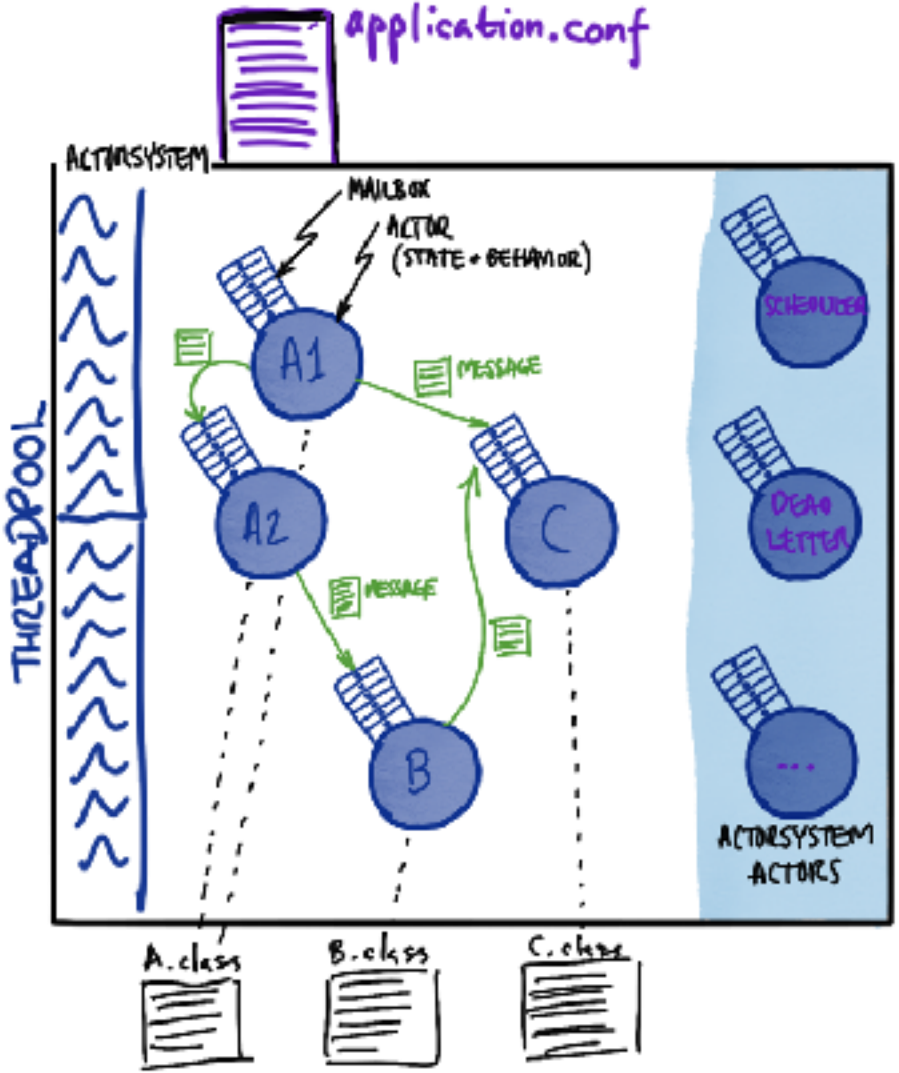
Elastic

Message driven


Megan
Hope you guys made it back safe. Thanks for helping Isa and me relax tonight!
Send message
Q W E R T
A S
Z X C V B N

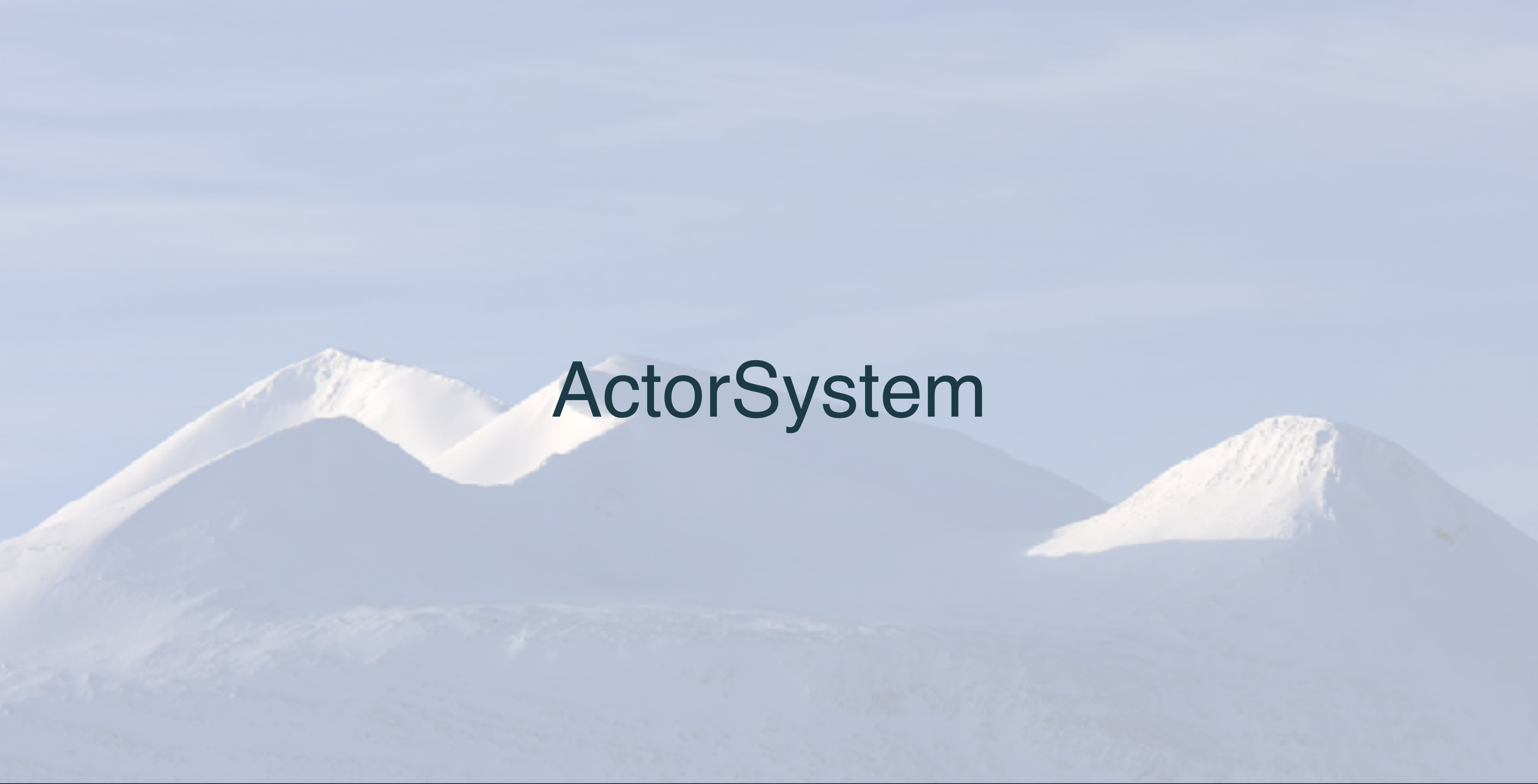
akka Actors

- Actor
  - Mailbox
  - Behavior
  - State
- ActorSystem
  - Thread pools
  - Configuration
  - System actors
- Messages
- JVM



Defining actors

```
import akka.actor.AbstractActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
public class MyActor extends AbstractActor {
   private final LoggingAdapter log = Logging.getLogger(getContext().getSystem(), this);
   private int msgs = 0;
    @Override
     public Receive createReceive() {
          return receiveBuilder()
            .match(String.class, s -> {
                msgs++;
                log.info("Received message: {}, msg number: {}", s, msgs);
          }).build()
        }
}
```



ActorSystem

```
final ActorSystem actorSystem =
    ActorSystem.create("AS");
```

Creating actors

```java
final ActorRef myActor =
 actorSystem.actorOf(Props.create(MyActor.class));
```

```java
final ActorRef myActor =
  actorSystem.actorOf(Props.create(MyActor.class),
        "myActor");
```

```java
final ActorRef myActor =
    actorSystem.actorOf(MyActor.props(),"myActor");
```

Defining messages

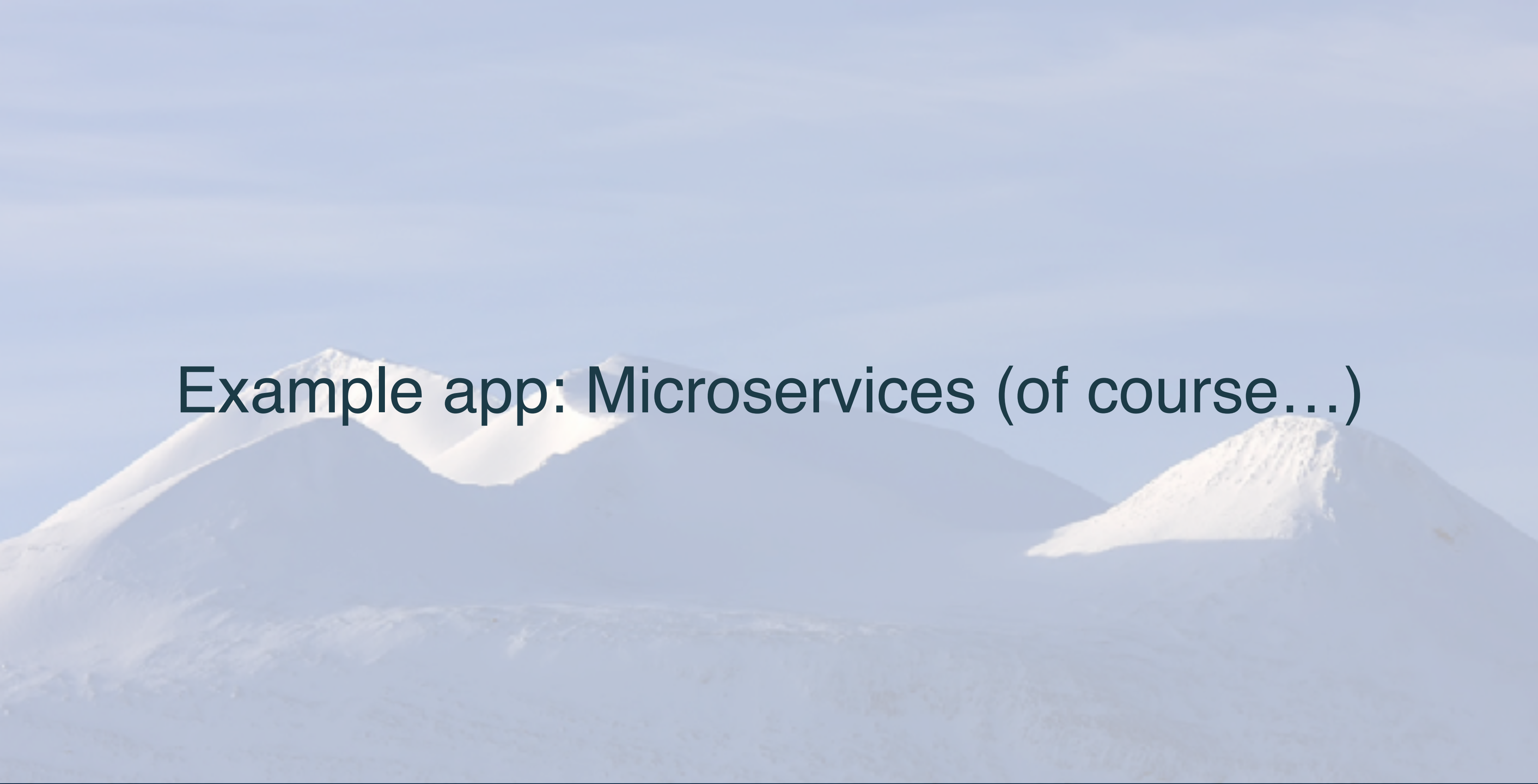
```java
package example;
import akka.actor.AbstractActor;
public class MyActor extends AbstractActor {
   // implement createReceive

    public static class SomeMessage {
        public String someValue;
        public SomeMessage(String someValue) {
            this.someValue = someValue;
        }
       // implement hashCode, equals, toString
    }
}
```

Sending messages

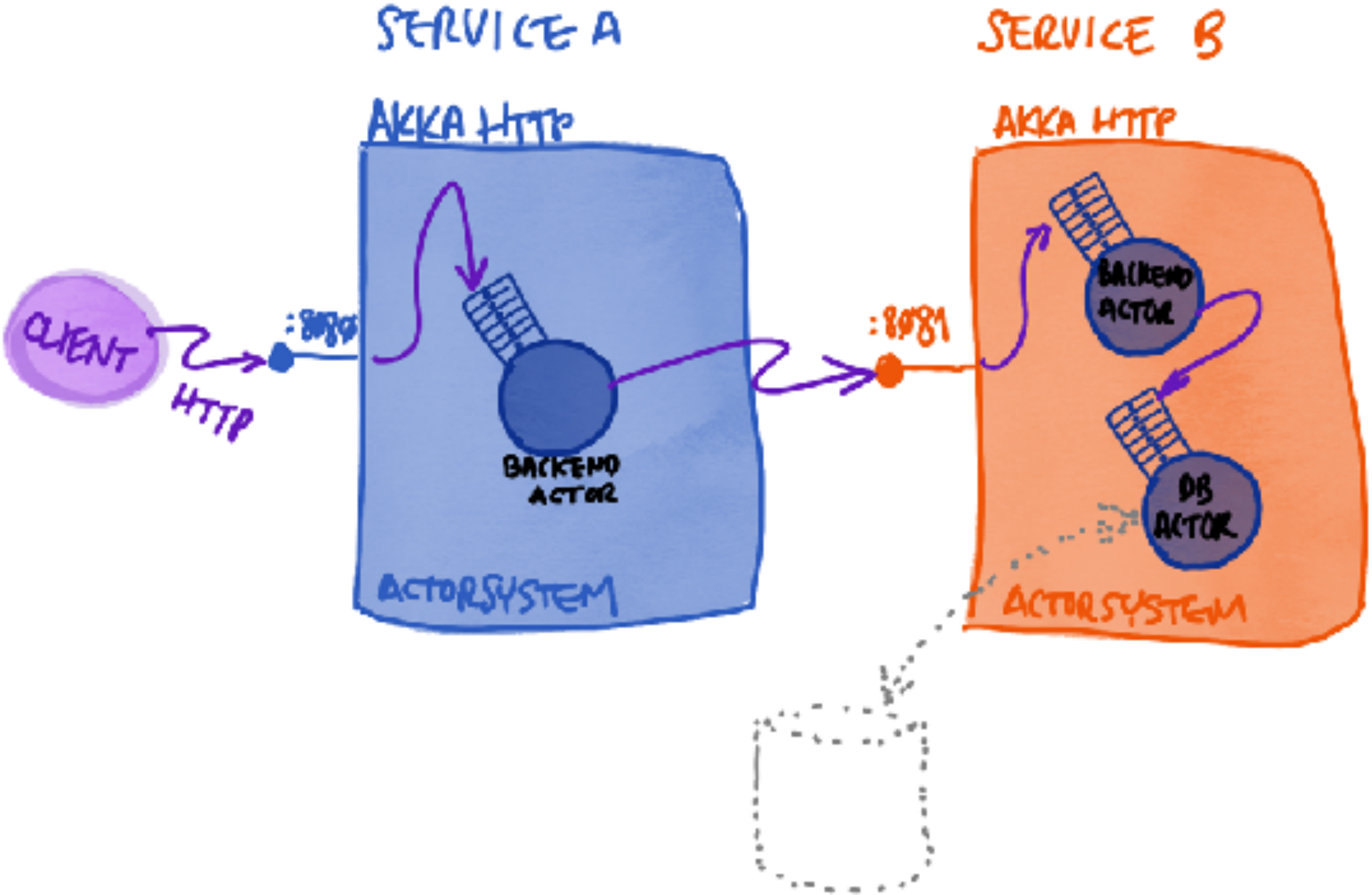
```
final ActorRef myActorRef =
    actorSystem.actorOf(MyActor.props(), "myActor");
myActorRef.tell(
    new MyActor.SomeMessage("something"),
ActorRef.noSender());

// Or if in the context of an actor
final ActorRef myActorRef =
    getContext().actorOf(MyActor.props(), "myActor");
myActorRef.tell(
    new MyActor.SomeMessage("something"), getSender());
```

Other important concepts for this demo

```
// the "ask" pattern
CompletionStage<Object> futureResult =
    ask(myActorRef,
         new MyActor.SomeMessage("something"), 1000);
// configuration - src/main/resources/application.conf
// someContext { a-b-c = 123 }
// HOCON - Typesafe Config
int someContextABC =
   getContext()
        .getSystem()
            .settings()
                .config()
                    .getInt("someContext.a-b-c");
```



Example app: Microservices (of course…)


SERVICE A
SERVICE B
AKKA HTTP
AKKA HTTP
CLIENT
HTTP
:8080
:8081
BACKEND ACTOR
BACKEND ACTOR
DB ACTOR
ACTORSYSTEM
ACTORSYSTEM

Example app - coding time!

Reactive Streams

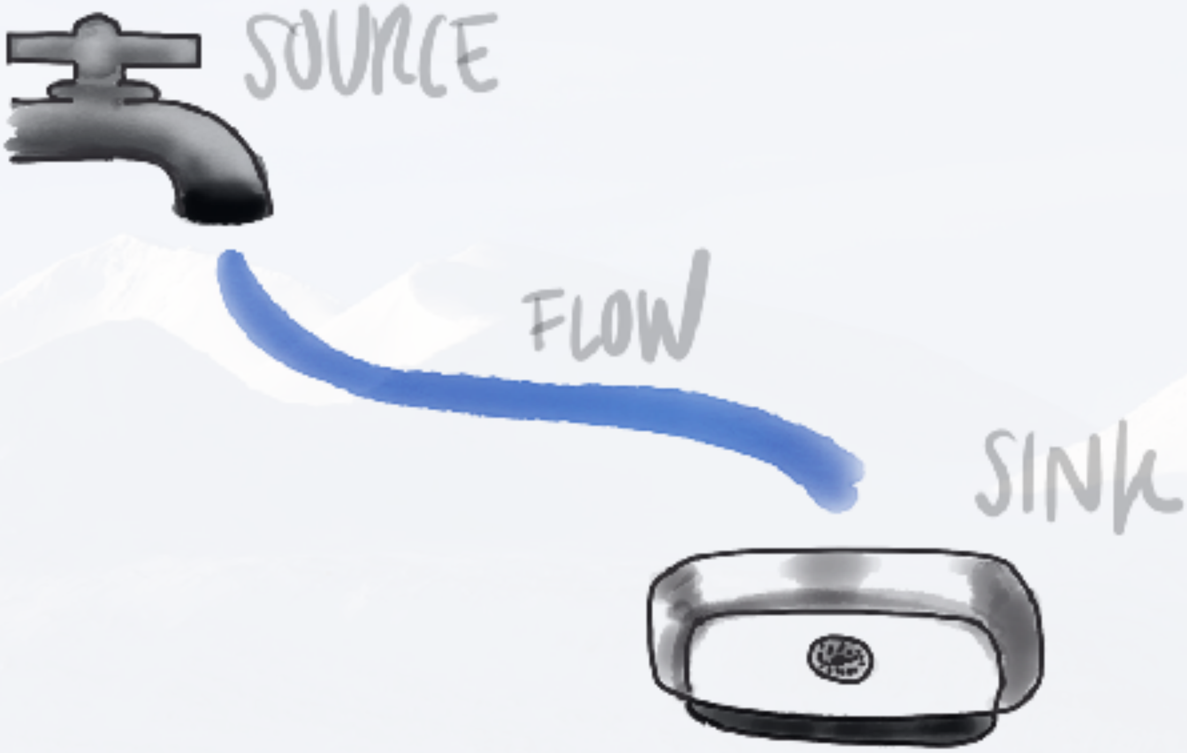
Reactive Streams - reactivestreams.org

"Reactive Streams is an initiative to provide a standard for ***asynchronous stream processing*** with ***non-blocking back pressure***. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols."

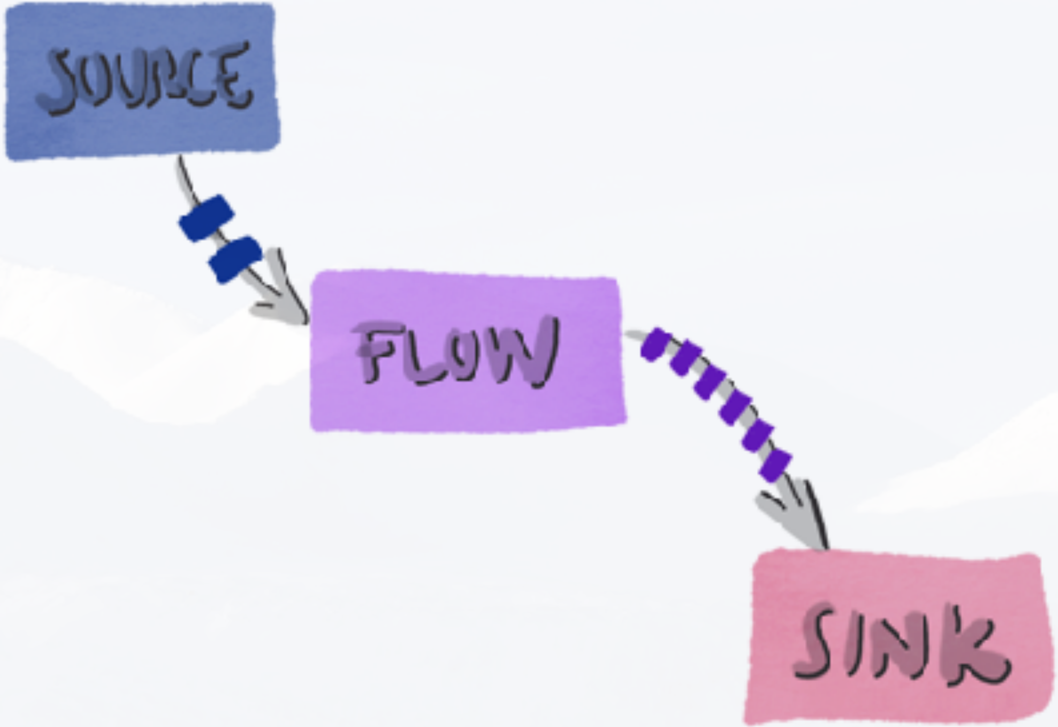
What is a "stream"?

- A possibly infinite set of datum
- Processed element by element
  - Could be *Byte* but more useful *<T>*
- Asynchronous processing
  - Sender and receiver are decoupled
  - Asynchronous boundaries (between *threads*)
  - Network boundaries (between *nodes*)

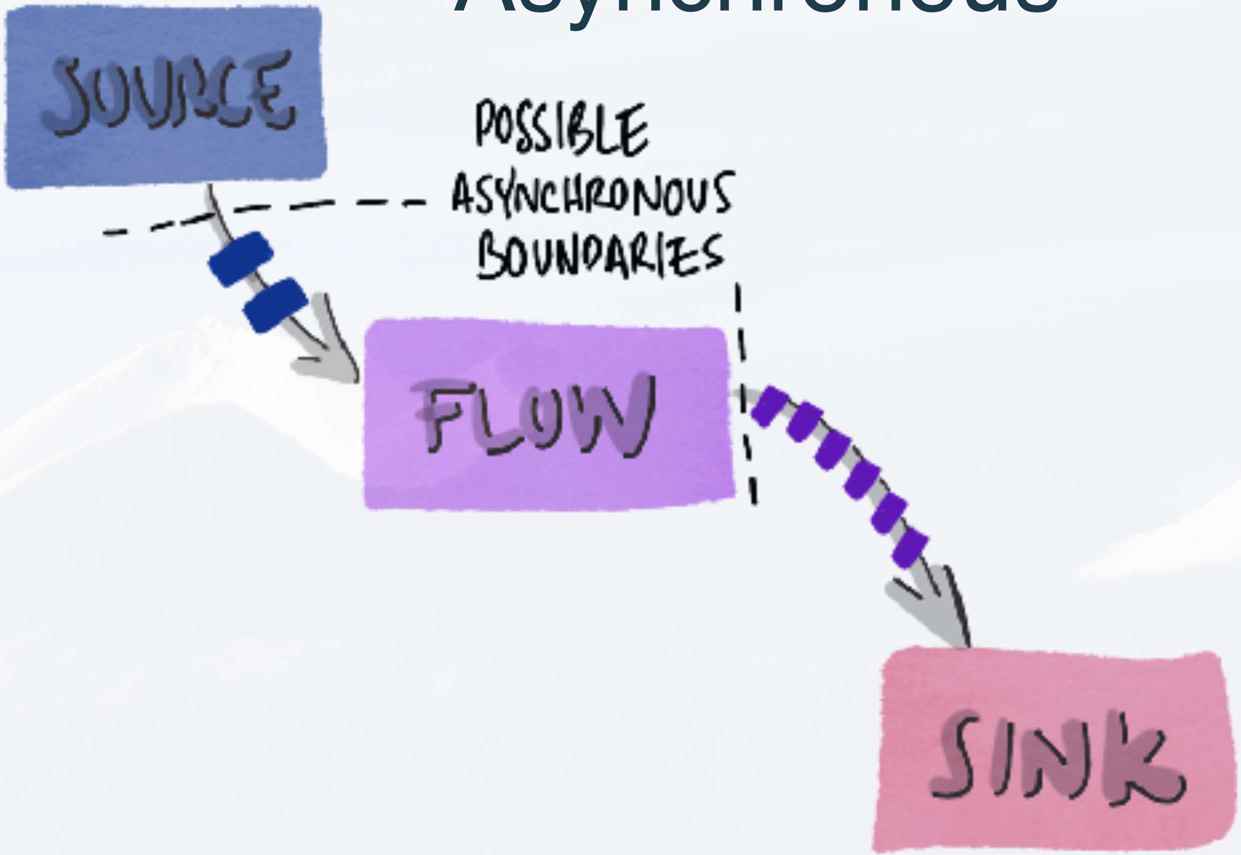
Reactive Streams



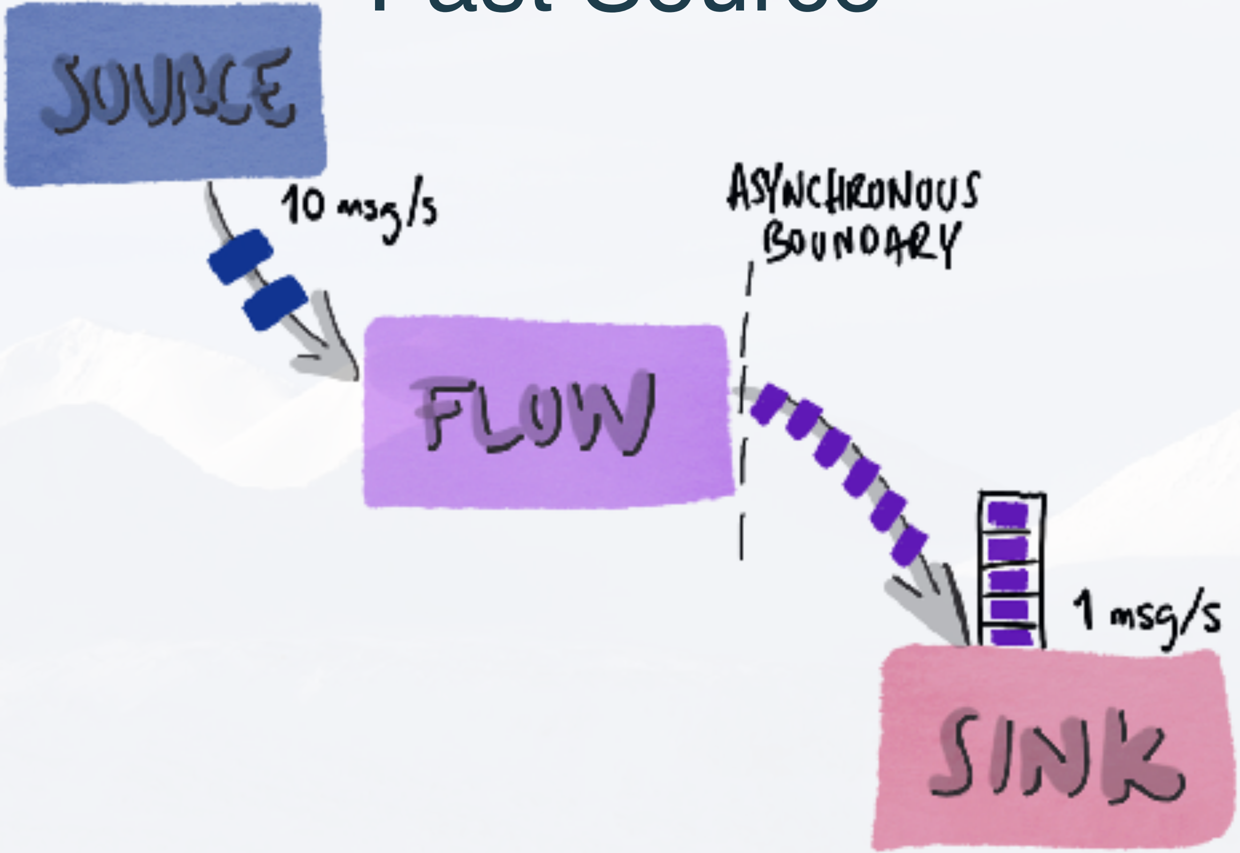
RS: Alternative representation



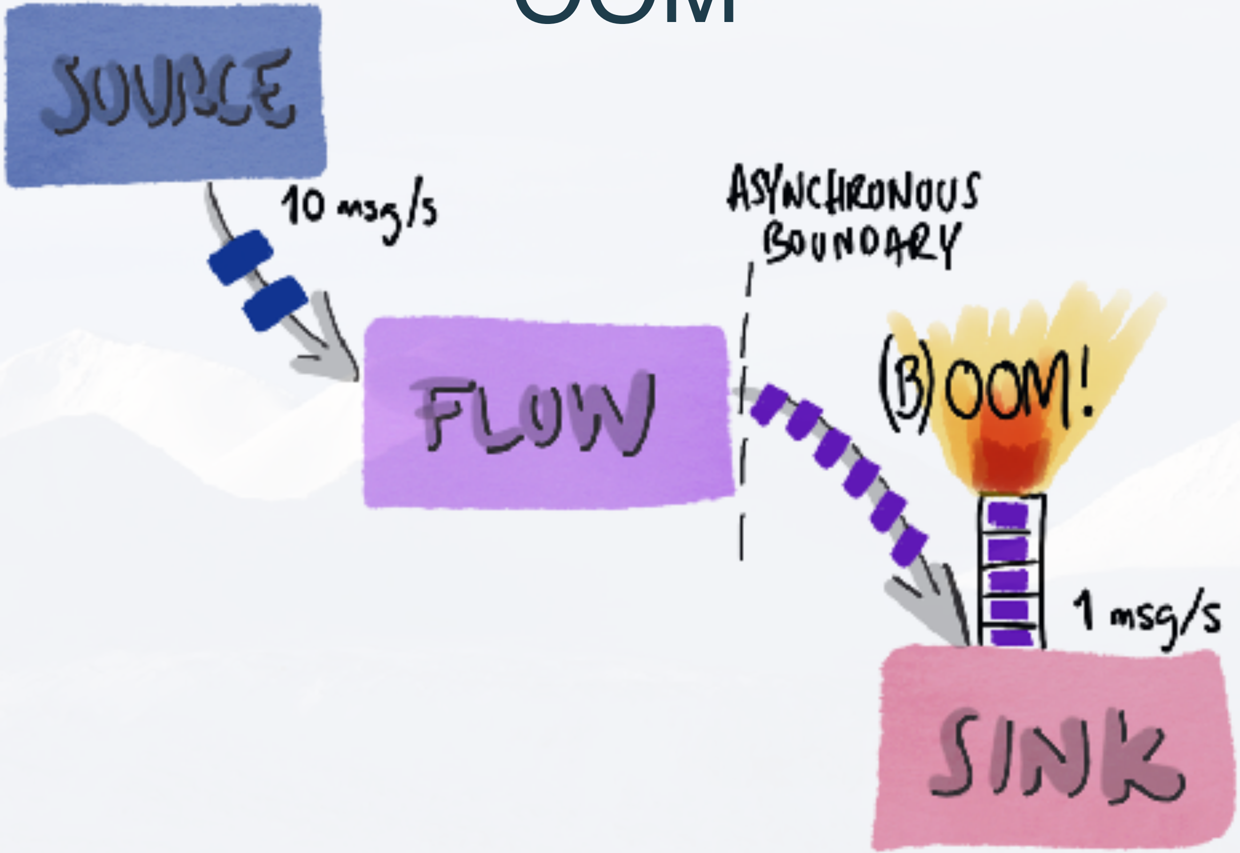
Asynchronous



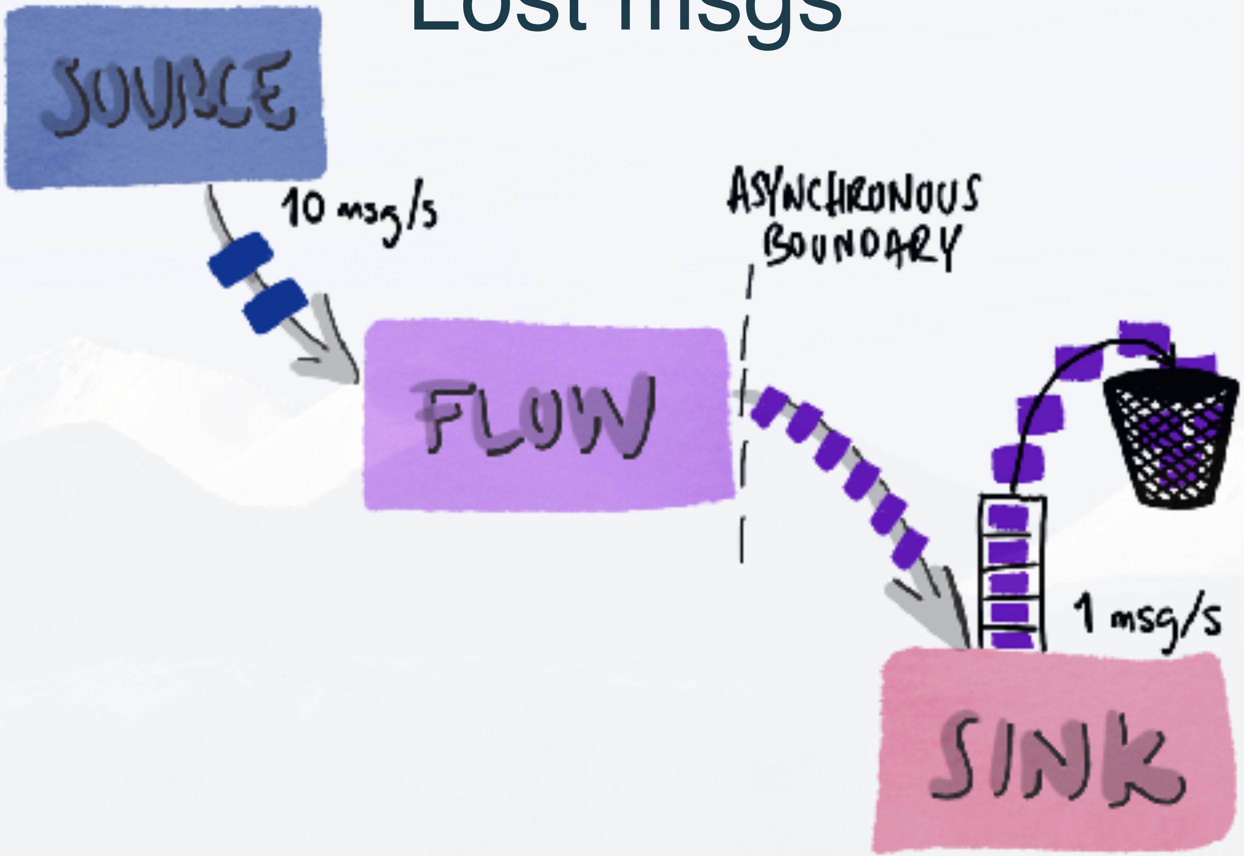
Fast Source



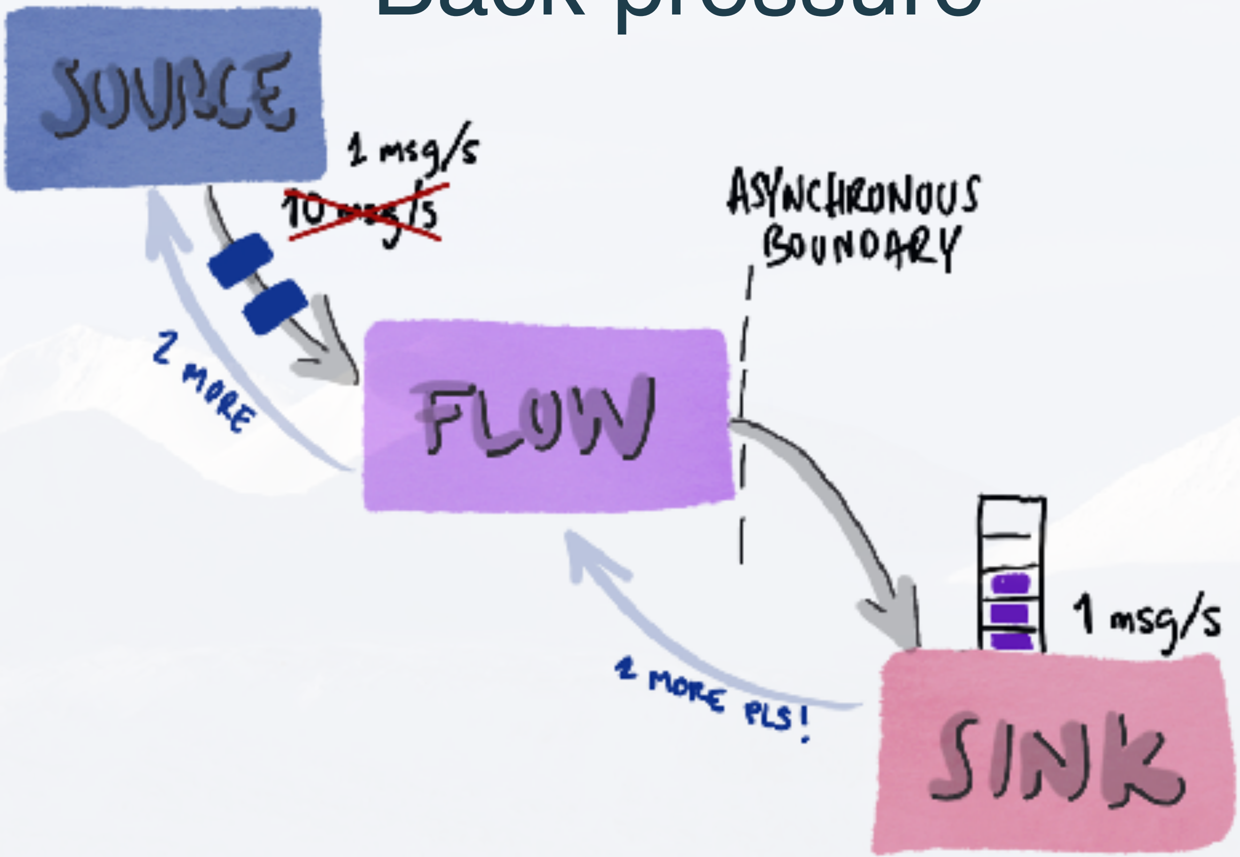
OOM



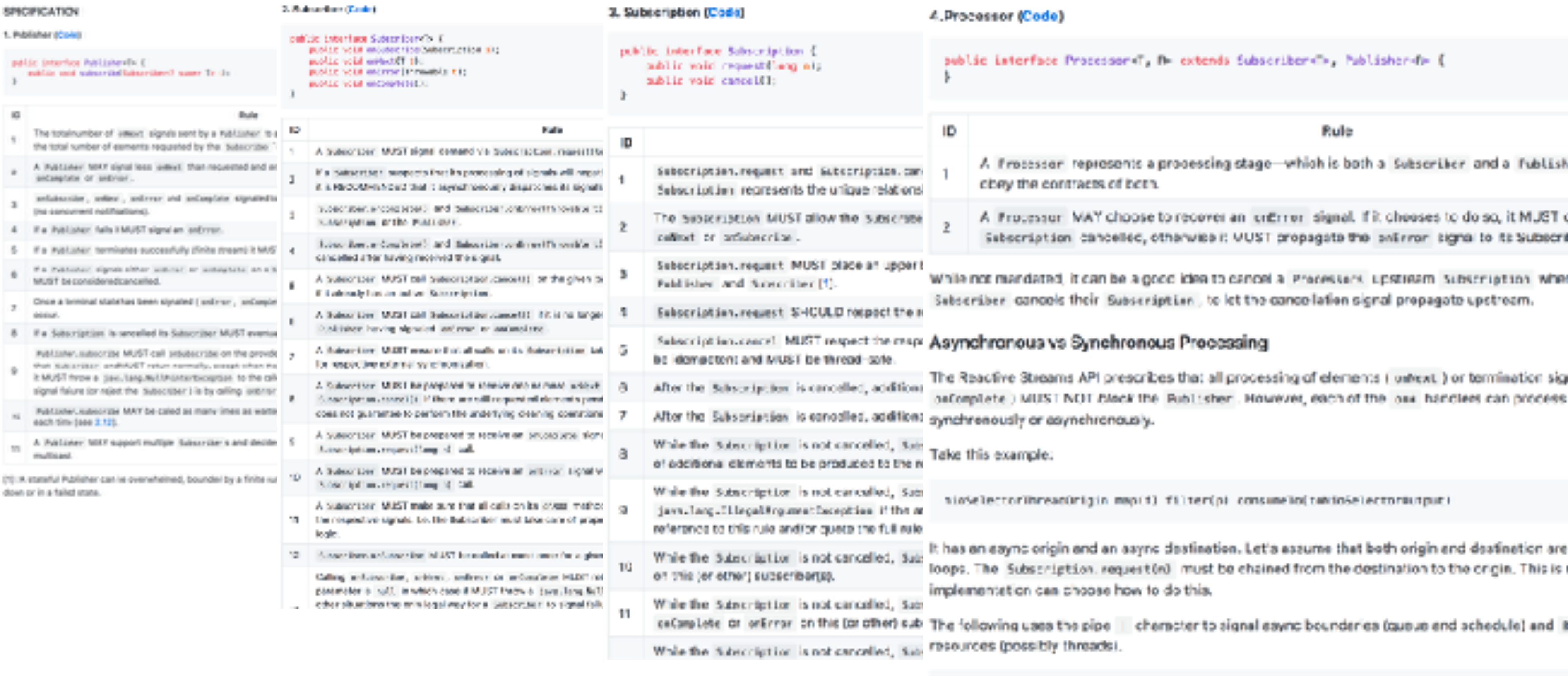
Lost msgs



Back pressure



Reactive streams specification



SPECIFICATION

1. Publisher (Code)

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

ID	Rule
1	The total number of onNext signals sent by a Publisher to a the total number of elements requested by the Subscriber
2	A Publisher MAY signal less onNext than requested and a onComplete or onError.
3	onSubscribe, onNext, onError and onComplete signaled to (no concurrent notifications).
4	If a Publisher fails it MUST signal an onError.
5	If a Publisher terminates successfully (finite stream) it MUS
6	If a Publisher signals either onError or onComplete on a MUST be considered cancelled.
7	Once a terminal state has been signaled (onError, onComple occur.
8	If a Subscription is cancelled its Subscriber MUST eventua
9	Publisher.subscribe MUST call onSubscribe on the provide than Subscriber and MUST return normally, except when the it MUST throw a java.lang.NullPointerException to the cal signal failure (or reject the Subscriber) is by calling onError
10	Publisher.subscribe MAY be called as many times as wante each time (see 2.12).
11	A Publisher MAY support multiple Subscribers and decide multicast.

[1]: A stateful Publisher can be overwhelmed, bounded by a finite nu down or in a failed state.

2. Subscriber (Code)

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

ID	Rule
1	A Subscriber MUST signal demand via Subscription.request(long n)
2	If a Subscriber suspects that its processing of signals will negat it is RECOMMENDED that it asynchronously dispatches its signals
3	Subscriber.onComplete() and Subscriber.onError(Throwable t) Subscription or the Publisher.
4	Subscriber.onComplete() and Subscriber.onError(Throwable t) cancelled after having received the signal.
5	A Subscriber MUST call Subscription.cancel() on the given Su it already has an active Subscription.
6	A Subscriber MUST call Subscription.cancel() if it is no longer Publisher having signaled onError or onComplete.
7	A Subscriber MUST ensure that all calls on its Subscription tak for respective external synchronization.
8	A Subscriber MUST be prepared to receive one or more onNext Subscription.cancel() if there are still requested elements pend does not guarantee to perform the underlying cleaning operations
9	A Subscriber MUST be prepared to receive an onComplete signal Subscription.request(long n) call.
10	A Subscriber MUST be prepared to receive an onError signal w Subscription.request(long n) call.
11	A Subscriber MUST make sure that all calls on its signal method the respective signals. I.e. the Subscriber must take care of proper logic.
12	Subscriber.onSubscribe MUST be called at most once for a given
13	Calling onSubscribe, onNext, onError or onComplete MUST ret parameter is null in which case it MUST throw a java.lang.NullP other situations the only legal way for a Subscriber to signal failu

3. Subscription (Code)

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

ID	
1	Subscription.request and Subscription.can Subscription represents the unique relations
2	The Subscription MUST allow the Subscribe onNext or onSubscribe.
3	Subscription.request MUST place an upper b Publisher and Subscriber [1].
4	Subscription.request SHOULD respect the r
5	Subscription.cancel MUST respect the respo be idempotent and MUST be thread-safe.
6	After the Subscription is cancelled, additional
7	After the Subscription is cancelled, additional
8	While the Subscription is not cancelled, Sub of additional elements to be produced to the re
9	While the Subscription is not cancelled, Sub java.lang.IllegalArgumentException if the ar reference to this rule and/or quote the full rule
10	While the Subscription is not cancelled, Sub on this (or other) subscriber(s).
11	While the Subscription is not cancelled, Sub onComplete or onError on this (or other) sub
	While the Subscription is not cancelled, Sub

4. Processor (Code)

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

ID	Rule
1	A Processor represents a processing stage—which is both a Subscriber and a Publish obey the contracts of both.
2	A Processor MAY choose to recover an onError signal. If it chooses to do so, it MUST c Subscription cancelled, otherwise it MUST propagate the onError signal to its Subscri

While not mandated, it can be a good idea to cancel a Processor's upstream Subscription whe Subscriber cancels their Subscription, to let the cancellation signal propagate upstream.

Asynchronous vs Synchronous Processing

The Reactive Streams API prescribes that all processing of elements (onNext) or termination sig onComplete) MUST NOT block the Publisher. However, each of the on* handlers can process synchronously or asynchronously.

Take this example:

```
nioSelectorThreadOrigin map(f) filter(p) consumeTo(toNioSelectorOutput)
```

It has an async origin and an async destination. Let's assume that both origin and destination are loops. The Subscription.request(n) must be chained from the destination to the origin. This is implementation can choose how to do this.

The following uses the pipe | character to signal async boundaries (queue and schedule) and R resources (possibly threads).

Reactive Streams interoperability

- Vert.x
- RxJava
- Reactor
- Akka Streams

RxJava <-> Vert.x <-> Akka Streams

JDK9 - *java.util.concurrent.Flow*

- Flow.Publisher<T> : *Source*
- Flow.Processor<T, R> : *Flow*
- Flow.Subscriber<T> : *Sink*

Akka Streams

Akka Streams in 60s

```
final ActorSystem actorSystem = ActorSystem.create();
final Materializer materializer =
    ActorMaterializer.create(actorSystem);
final Source<Integer, NotUsed> source =
    Source.range(0, 10);
final Flow<Integer, String, NotUsed> flow =
    Flow.fromFunction((Integer i) -> i.toString());
final Sink<String, CompletionStage<Done>> sink =
    Sink.foreach(s -> System.out.println("Number: " + s));
final RunnableGraph runnable =
    source.via(flow).to(sink);
runnable.run(materializer);
```

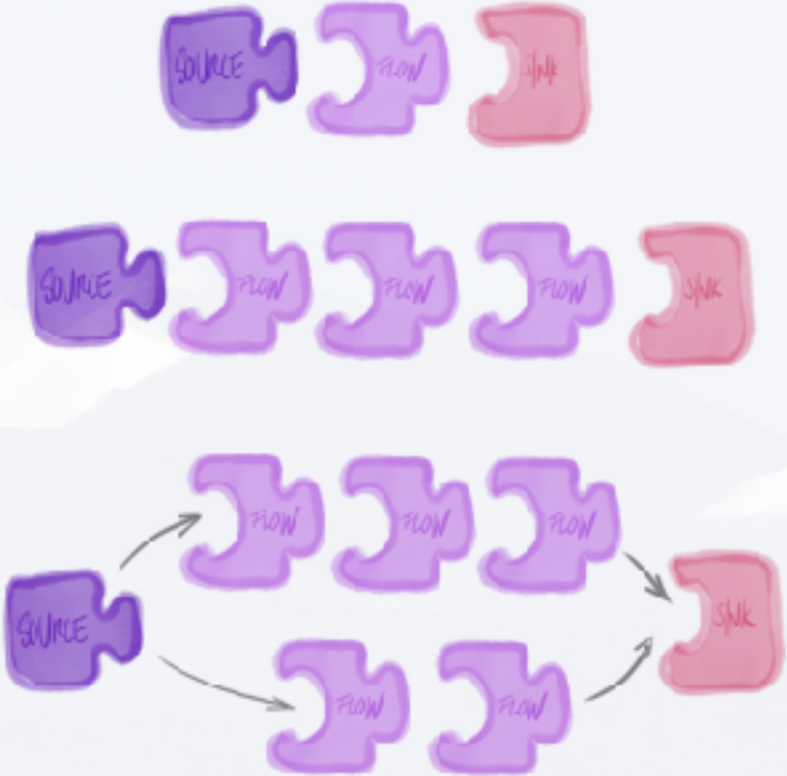
Akka Streams in 60s

```scala
implicit val actorSystem = ActorSystem()
implicit val materializer = ActorMaterializer()

val source = Source(0 to 10)
val flow = Flow[Int].map(_.toString)
val sink =
  Sink.foreach[String](s => println(s"Number: $s"))
val runnable = source.via(flow).to(sink)
runnable.run()
```

Akka Streams in **20s**

```
final ActorSystem actorSystem = ActorSystem.create();
final Materializer materializer =
    ActorMaterializer.create(actorSystem);

Source.range(0, 10)
        .map(Object::toString)
        .runForeach(s ->
            System.out.println("Number: " + s),
                materializer);
```

Graph stages



Materialization

Blueprint
(Graph)

Internal representation
(Actors)

Creating akka.stream.javadsl.Source

```
static <T> Source<T,ActorRef> actorRef(int bufferSize,
        OverflowStrategy overflowStrategy);
static <O> Source<O,NotUsed> empty();
static <O> Source<O,NotUsed> from(java.lang.Iterable<O> iterable);
static <O> Source<O,NotUsed> fromFuture(scala.concurrent.Future<O> future);
static <O> Source<O,NotUsed> fromIterator(Creator<java.util.Iterator<O>> f);
static Source<java.lang.Integer,NotUsed> range(int start, int end);
static <T> Source<T,NotUsed> single(T element);
// etc.
```

Using akka.streams.javadsl.Flow

```
static <T> Flow<T,T,NotUsed> create();
Flow<In,Out,Mat> drop(long n);
Flow<In,Out,Mat> dropWhile(Predicate<Out> p);
Flow<In,Out,Mat> filter(Predicate<Out> p);
Flow<In,Out,Mat> filterNot(Predicate<Out> p);
<T> Flow<In,T,Mat> fold(T zero, Function2<T,Out,T> f);
<T> Flow<In,T,Mat> foldAsync(T zero,
    Function2<T,Out,java.util.concurrent.CompletionStage<T>> f);
Flow<In,java.util.List<Out>,Mat> groupedWithin(int n,
    scala.concurrent.duration.FiniteDuration d);
Flow<In,Out,Mat> log(java.lang.String name);
<T> Flow<In,T,Mat> map(Function<Out,T> f);
```

Using akka.streams.javadsl.Sink

```
static <T> Sink<T,java.util.concurrent.CompletionStage<Done>>
      foreach(Procedure<T> f);
static <U,In> Sink<In,java.util.concurrent.CompletionStage<U>>
      fold(U zero, Function2<U,In,U> f);
static <In> Sink<In,java.util.concurrent.CompletionStage<In>> head();
static <T> Sink<T,java.util.concurrent.CompletionStage<Done>> ignore();
static <In> Sink<In,java.util.concurrent.CompletionStage<In>> last();
// etc.
```

Example app - take two

Alpakka

Alpakka: Akka Streams connectors

- http://developer.lightbend.com/docs/alpakka/current/
- Example connectors: AMQP, AWS DynamoDB, AWS Kinesis, AWS Lambda, Cassandra, JMS, SSE, File IO, Azure, Camel, Kafka, TCP, etc.

Akka Actors & Streams

Akka Actors and Akka Streams

- Why actors?
  - Managing state.
- Why Akka Streams?
  - Process handling of "flowing" data.

Side note: Akka Streams uses Akka Actors under the hood, i.e. it is possible to implement "anything" in actors but it is more low level.

Stream -> Actor: *ask* with *mapAsync*

```
public static class MultiplierActor extends AbstractActor {
    @Override
    public Receive createReceive() {
        return receiveBuilder().match(Integer.class, i -> {
            getSender().tell(i * 2, getSelf());
        }).build();
    }
}
```

Stream -> Actor: *ask* with *mapAsync*

```
final ActorSystem actorSystem = ActorSystem.create();
final Materializer materializer =
    ActorMaterializer.create(actorSystem);
final ActorRef multiplier =
    actorSystem.actorOf(MultiplierActor.props());
Source.range(0, 10)
        .mapAsync(1, x -> ask(multiplier, x, 1000L))
        .map(e -> (Integer) e)
        .runWith(
             Sink.foreach(i ->
                 System.out.println("Example 3 - Number: " +
             i)), materializer);
```

Credits

- Colin Breck's inspirational blog post: http://blog.colinbreck.com/akka-streams-a-motivating-example
- My colleagues Björn Antonsson (@bantonsson), Peter Vlugter, Johan Andrén (@apnylle), and Konrad Malawski (@ktoso) for their help with this presentation/code.
- Lightbend for supporting this mission!

Lightbend