BECOMING A SMOOTH OPERATOR

A LOOK AT THE STREAMOPERATOR API

WHO AM I?

ADDISON HIGHAM

- Cloud/Data Guy at Instructure
- github.com/addisonj
- twitter/addisonjh

HOW I KNEW FLINK WAS GREAT

- Technology
- Community

- ~~Technology~~
- Community

- ~~Technology~~
- ~~Community~~

- ~~Technology~~
- ~~Community~~
- Best Apache Animal Logo

SIMPLE, BUT OKAY

Apache Pig

OFF-BRAND MARIO

NIGHTMARE FUEL

Flink

CLEARLY THE WINNER

INSTRUCTURE

FROM THE FIRST DAY OF SCHOOL TO THE LAST DAY OF WORK

canvas

Learning Management System

BRIDGE

Employment Development

HOW WE USE FLINK

GET MORE DATA TO CUSTOMERS FASTER

- Fast and correct ETLs
- Replace/Augment batch systems
- Still new effort, but growing

REPLACEMENT FOR LAMBDA ARCHICTURES

WE FOUND LAMBDA TO BE:

REPLACEMENT FOR LAMBDA ARCHICTURES

WE FOUND LAMBDA TO BE:

- Difficult to maintain (two stacks, duplicated logic)

REPLACEMENT FOR LAMBDA ARCHICTURES

WE FOUND LAMBDA TO BE:

- Difficult to maintain (two stacks, duplicated logic)
- Hard to coordinate (external state, source of truth)

REPLACEMENT FOR LAMBDA ARCHICTURES

WE FOUND LAMBDA TO BE:

- Difficult to maintain (two stacks, duplicated logic)
- Hard to coordinate (external state, source of truth)
- Hard to debug (erroneous or missing data, bad UX)

REPLACEMENT FOR LAMBDA ARCHICTURES

WE FOUND LAMBDA TO BE:

- Difficult to maintain (two stacks, duplicated logic)
- Hard to coordinate (external state, source of truth)
- Hard to debug (erroneous or missing data, bad UX)
- Not worth doing anymore (with better stream processors, why use lambda?)

OUR EXPERIENCE SO FAR

Flink is a feature-rich and fast stream processor with flexibility to meet a ton of different demands

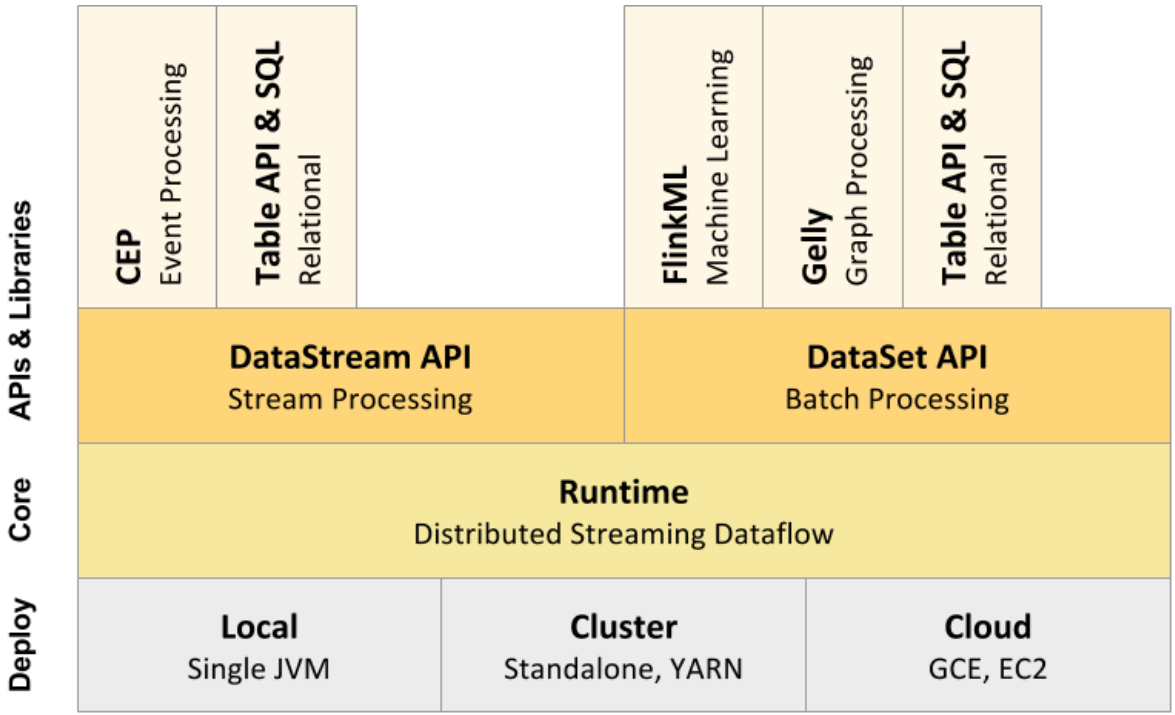
The `StreamOperator` API is one aspect that leads to Flink's flexibility

WHAT WE WILL COVER TODAY

THE `StreamOperator` API

- What it is and how it fits in with the other Flink APIs
- What it enables and how Flink uses it
- Some examples of what advanced functionality you can do with it
- How Instructure uses it to move to Kappa-like Architectures

THE MANY LAYERS OF FLINK


APIs & Libraries
CEP
Event Processing
Table API & SQL
Relational
FlinkML
Machine Learning
Gelly
Graph Processing
Table API & SQL
Relational
DataStream API
Stream Processing
DataSet API
Batch Processing
Core
Runtime
Distributed Streaming Dataflow
Deploy
Local
Single JVM
Cluster
Standalone, YARN
Cloud
GCE, EC2

FLINK SQL

```
case class Student(id: Long, name: String)
case class Grade(id: Long, assignmentId: Long, studentId: Long, score: Int)
...
val students = env.fromCollection(Seq(
  Student(1, "Jane"),
  Student(2, "Bill"),
  Student(3, "Addison")
))
val grades = env.fromCollection(Seq(
  Grade(1, 1, 1, 85),
  Grade(2, 1, 2, 79),
  Grade(3, 1, 3, 42),
  Grade(4, 1, 3, 42)
))
```

FLINK SQL

```
case class Student(id: Long, name: String)
case class Grade(id: Long, assignmentId: Long, studentId: Long, score: Int)
...
val students = env.fromCollection(Seq(
  Student(1, "Jane"),
  Student(2, "Bill"),
  Student(3, "Addison")
))
val grades = env.fromCollection(Seq(
  Grade(1, 1, 1, 85),
  Grade(2, 1, 2, 79),
  Grade(3, 1, 3, 42),
  Grade(4, 1, 3, 42)
))
val tableEnv = TableEnvironment.getTableEnvironment(env)
tableEnv.registerDataStream("Students", students)
tableEnv.registerDataStream("Grades", grades)
```

FLINK SQL

```
case class Student(id: Long, name: String)
case class Grade(id: Long, assignmentId: Long, studentId: Long, score: Int)
...
val students = env.fromCollection(Seq(
  Student(1, "Jane"),
  Student(2, "Bill"),
  Student(3, "Addison")
))
val grades = env.fromCollection(Seq(
  Grade(1, 1, 1, 85),
  Grade(2, 1, 2, 79),
  Grade(3, 1, 3, 42),
  Grade(4, 1, 3, 42)
))
val tableEnv = TableEnvironment.getTableEnvironment(env)
tableEnv.registerDataStream("Students", students)
tableEnv.registerDataStream("Grades", grades)
val fails = tableEnv.sqlQuery("""
SELECT * FROM Grades INNER JOIN Students ON Grades.studentId = Students.id
WHERE Grades.score < 60
""")
```

"HIGH LEVEL" STREAM APIS (MAP, FILTER, WINDOW, ETC)

```
case class Grade(id: Long, assignmentId: Long, studentId: Long, score: Int, attempt: Int)
val grades = env.fromCollection(Seq(
  Grade(1, 1, 1, 85, 1),
  Grade(2, 1, 2, 79, 1),
  Grade(3, 1, 3, 42, 1),
  Grade(4, 1, 3, 42, 2)
))
```

"HIGH LEVEL" STREAM APIS (MAP, FILTER, WINDOW, ETC)

```
case class Grade(id: Long, assignmentId: Long, studentId: Long, score: Int, attempt: Int)
val grades = env.fromCollection(Seq(
  Grade(1, 1, 1, 85, 1),
  Grade(2, 1, 2, 79, 1),
  Grade(3, 1, 3, 42, 1),
  Grade(4, 1, 3, 42, 2)
))

val avgFirstsAttempt = grades
  .keyBy("assignmentId")
  // keep only first attempts
  .filter((grade) => grade.attempt == 0)
```

"HIGH LEVEL" STREAM APIS (MAP, FILTER, WINDOW, ETC)

```
case class Grade(id: Long, assignmentId: Long, studentId: Long, score: Int, attempt: Int)
val grades = env.fromCollection(Seq(
  Grade(1, 1, 1, 85, 1),
  Grade(2, 1, 2, 79, 1),
  Grade(3, 1, 3, 42, 1),
  Grade(4, 1, 3, 42, 2)
))

val avgFirstsAttempt = grades
  .keyBy("assignmentId")
  // keep only first attempts
  .filter((grade) => grade.attempt == 0)
  // just store everything in one window
  .window(GlobalWindows.create())
  // a trigger that fires every n seconds
  .trigger(new PeriodicFiringTrigger())
  // an aggregate function to keep a running average
  .aggregate(new AverageAggregate())
```

"LOW LEVEL" STREAM APIS (PROCESS, ASYNC, BROADCAST, ETC)

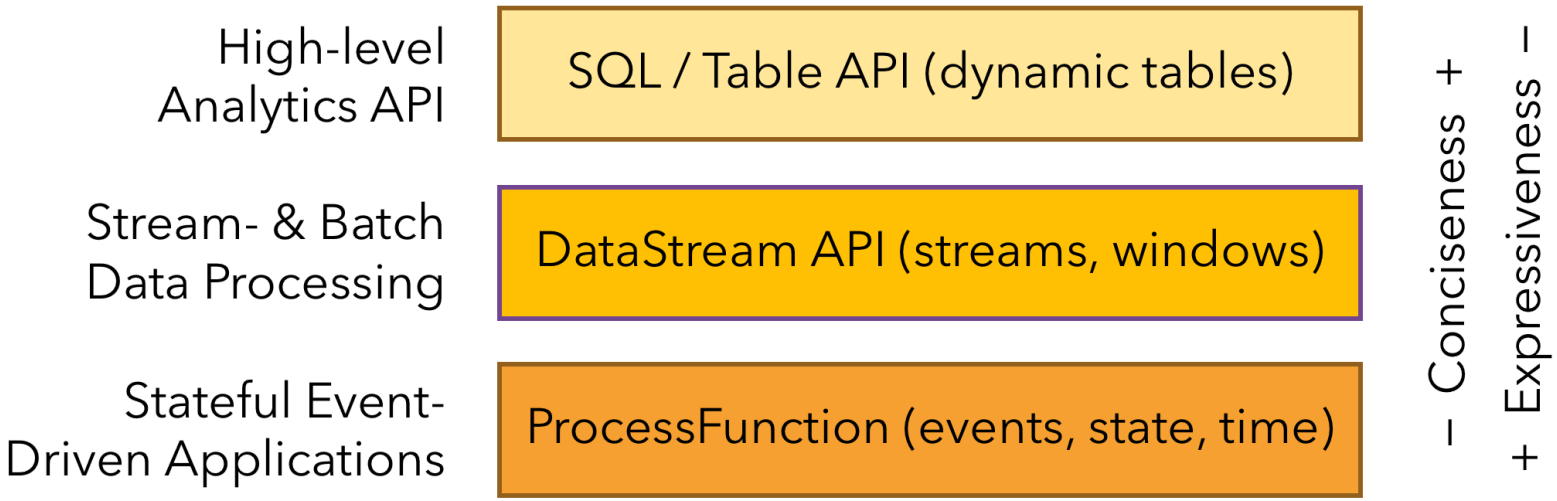
```
case class Grade(id: Long, assignmentId: Long, studentId: Long, score: Int, attempt: Int, dueAt: Instant)
case class NotifyFail(studentId: Long, message: String)
val grades = env.fromCollection(Seq(
  Grade(1, 1, 1, 85, 1, Instant.parse("2019-04-01T23:50:00Z")),
  Grade(2, 1, 2, 79, 1, Instant.parse("2019-04-01T23:50:00Z")),
  Grade(3, 1, 3, 42, 1, Instant.parse("2019-04-01T23:50:00Z")),
  Grade(4, 1, 3, 42, 2, Instant.parse("2019-04-01T23:50:00Z"))
))
```

"LOW LEVEL" STREAM APIS (PROCESS, ASYNC, BROADCAST, ETC)

```
case class Grade(id: Long, assignmentId: Long, studentId: Long, score: Int, attempt: Int, dueAt: Instant)
case class NotifyFail(studentId: Long, message: String)
val grades = env.fromCollection(Seq(
  Grade(1, 1, 1, 85, 1, Instant.parse("2019-04-01T23:50:00Z")),
  Grade(2, 1, 2, 79, 1, Instant.parse("2019-04-01T23:50:00Z")),
  Grade(3, 1, 3, 42, 1, Instant.parse("2019-04-01T23:50:00Z")),
  Grade(4, 1, 3, 42, 2, Instant.parse("2019-04-01T23:50:00Z"))
))

val notifyFailingGrades = grades.
  keyBy("assignmentId", "studentId")
  process(new KeyedProcessFunction[Grade, NotifyFail] with RichFunction {
    private val WarnTime = 1000 * 60 * 60 // one hour
    private val WarnScore = 60 // warn if score is failing
    lazy private val bestGrade: ValueState[Grade] = getRuntimeContext().getState("bestGrade", ...)
    override def processElement(el: Grade, ctx: Context, coll: Collector[NotifyFail]): Unit = {
      if (bestGrade.get == null) {
        val warnAt = el.dueAt.toEpochMilli() - WarnTime
        ctx.timerService.registerProcessingTimeTimer(warnAt)
      }
      if (el.score > Option(bestGrade.get()).map(_.score).getOrElse(-1)) {
        bestGrade.update(el)
      }
    }
```

"LOW LEVEL" STREAM APIS (PROCESS, ASYNC, BROADCAST, ETC)

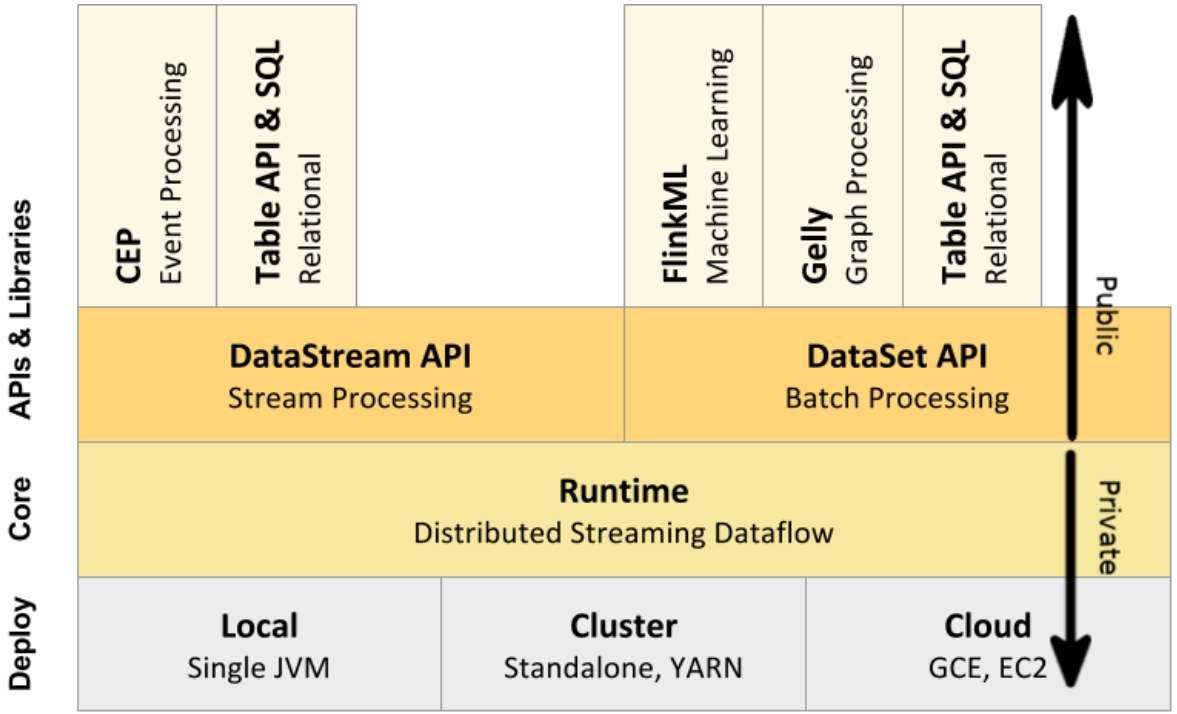
```
case class Grade(id: Long, assignmentId: Long, studentId: Long, score: Int, attempt: Int, dueAt: Instant)
case class NotifyFail(studentId: Long, message: String)
val grades = env.fromCollection(Seq(
  Grade(1, 1, 1, 85, 1, Instant.parse("2019-04-01T23:50:00Z")),
  Grade(2, 1, 2, 79, 1, Instant.parse("2019-04-01T23:50:00Z")),
  Grade(3, 1, 3, 42, 1, Instant.parse("2019-04-01T23:50:00Z")),
  Grade(4, 1, 3, 42, 2, Instant.parse("2019-04-01T23:50:00Z"))
))

val notifyFailingGrades = grades.
  keyBy("assignmentId", "studentId")
  process(new KeyedProcessFunction[Grade, NotifyFail] with RichFunction {
    private val WarnTime = 1000 * 60 * 60 // one hour
    private val WarnScore = 60 // warn if score is failing
    lazy private val bestGrade: ValueState[Grade] = getRuntimeContext().getState("bestGrade", ...)
    override def processElement(el: Grade, ctx: Context, coll: Collector[NotifyFail]): Unit = {
      if (bestGrade.get == null) {
        val warnAt = el.dueAt.toEpochMilli() - WarnTime
        ctx.timerService.registerProcessingTimeTimer(warnAt)
      }
      if (el.score > Option(bestGrade.get()).map(_.score).getOrElse(-1)) {
        bestGrade.update(el)
      }
    }
    override def onTimer(ts Long, ctx: OnTimerContext, coll: Collector[NotifyFail]): Unit = {
      val grade = bestGrade.get()
      if (grade.score < WarnScore) {
        coll.collect(NotifyFail(grade.studentId, s"You are failing assignment ${grade.assignmentId}"))
      }
    }
  })
```


High-level
Analytics API
SQL / Table API (dynamic tables)
Stream- & Batch
Data Processing
DataStream API (streams, windows)
Stateful Event-
Driven Applications
ProcessFunction (events, state, time)
– Conciseness +
+ Expressiveness –

IS THAT ALL THE CONTROL WE GET?

- checkpoint barriers, watermarks, latency marker handling?
- state snapshotting?
- per-key state magic?
- ... and a whole lot more?

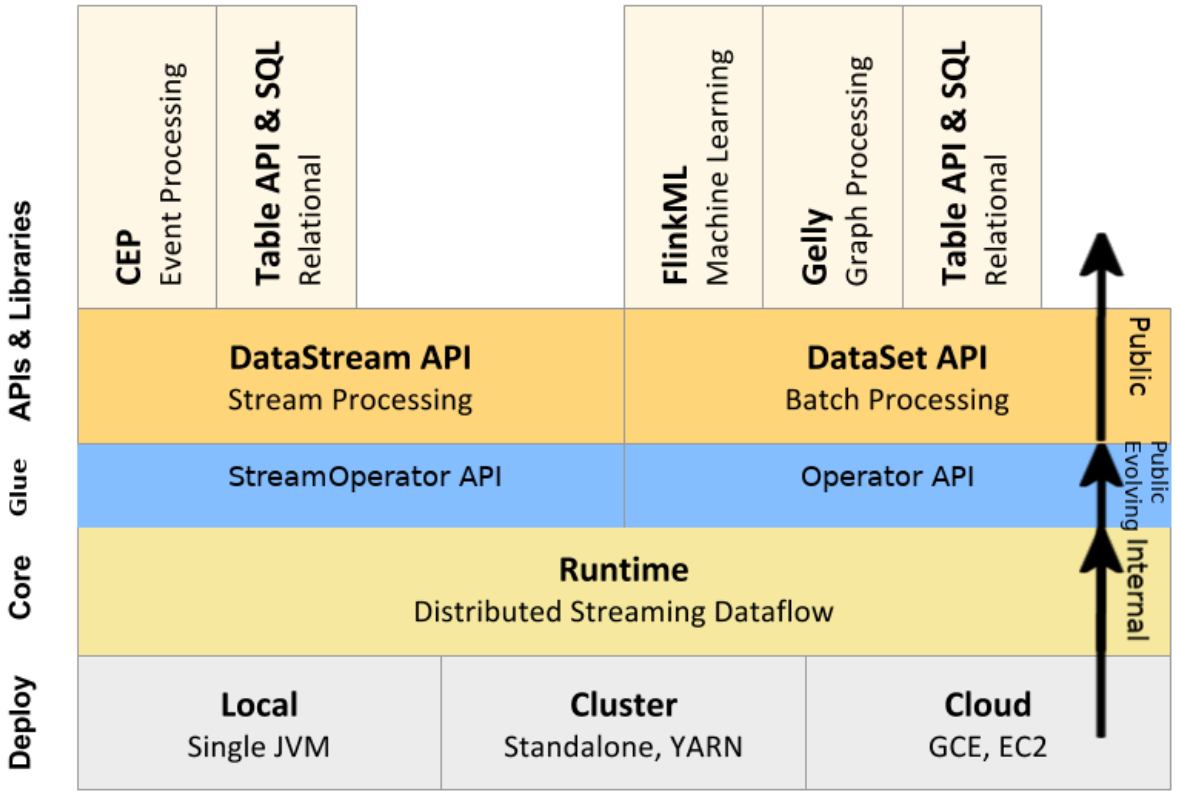
PUBLIC VS PRIVATE APIS?



FLINK'S API DESIGN APPROACH

In my opinion

- Flink marks very few classes/interfaces as private to allow users lots of flexibility
- Instead, guidelines are given via annotations and docs
- This encourages users to experiment and understand, without ossification or constant breakage
- As Flink evolves, more functionality is moved into higher level APIs


APIs & Libraries
Glue
Core
Deploy
CEP
Event Processing
Table API & SQL
Relational
FlinkML
Machine Learning
Gelly
Graph Processing
Table API & SQL
Relational
DataStream API
Stream Processing
DataSet API
Batch Processing
StreamOperator API
Operator API
Runtime
Distributed Streaming Dataflow
Local
Single JVM
Cluster
Standalone, YARN
Cloud
GCE, EC2
Public
Public
Evolving
Internal

StreamOperator API

WARNING

- The `StreamOperator` is marked as `PublicEvolving`, which means that it can change between versions.
- Also, you can totally break your app and do weird things

WARNING

- The `StreamOperator` is marked as `PublicEvolving`, which means that it can change between versions.
- Also, you can totally break your app and do weird things

WARNING

- The `StreamOperator` is marked as `PublicEvolving`, which means that it can change between versions.
- Also, you can totally break your app and do weird things
- Really, you should only use this if you know why

WHAT IT DOES

- is the primary building block of the `DataStream` API
- encapsulates each `DataStream` operation
- handles interaction with runtime and internal APIs
- handle internal messaging (`StreamRecord`, `Watermark`, etc)
- facilitates state snapshotting and managing key contexts

WHAT IT LOOKS LIKE

```
// StreamOperator.java
@PublicEvolving
public interface StreamOperator<OUT> extends CheckpointListener, KeyContext, Disposable, Serial

}
```

WHAT IT LOOKS LIKE

```
// StreamOperator.java
@PublicEvolving
public interface StreamOperator<OUT> extends CheckpointListener, KeyContext, Disposable, Serial
  // lifecycle
  void setup(StreamTask<?, ?> containingTask, StreamConfig config, Output<StreamRecord<OUT>> ou
  void open() throws Exception;
  void close() throws Exception;
  void dispose() throws Exception;









}
```

WHAT IT LOOKS LIKE

```
// StreamOperator.java
@PublicEvolving
public interface StreamOperator<OUT> extends CheckpointListener, KeyContext, Disposable, Seriali
  // lifecycle
  void setup(StreamTask<?, ?> containingTask, StreamConfig config, Output<StreamRecord<OUT>> out
  void open() throws Exception;
  void close() throws Exception;
  void dispose() throws Exception;

  // state
  void prepareSnapshotPreBarrier(long checkpointId) throws Exception;
  OperatorSnapshotFutures snapshotState(...) throws Exception;
  void initializeState() throws Exception;
  // from CheckpointListener interface
  // void notifyCheckpointComplete(long checkpointId) throws Exception;




}
```

WHAT IT LOOKS LIKE

```
// StreamOperator.java
@PublicEvolving
public interface StreamOperator<OUT> extends CheckpointListener, KeyContext, Disposable, Seriali
  // lifecycle
  void setup(StreamTask<?, ?> containingTask, StreamConfig config, Output<StreamRecord<OUT>> out
  void open() throws Exception;
  void close() throws Exception;
  void dispose() throws Exception;

  // state
  void prepareSnapshotPreBarrier(long checkpointId) throws Exception;
  OperatorSnapshotFutures snapshotState(...) throws Exception;
  void initializeState() throws Exception;
  // from CheckpointListener interface
  // void notifyCheckpointComplete(long checkpointId) throws Exception;

  // keys
  // from KeyContext interface
  // void setCurrentKey(Object key);
  void setKeyContextElement1(StreamRecord<?> record) throws Exception;
  void setKeyContextElement2(StreamRecord<?> record) throws Exception;
  ...
}
```

WHAT IT LOOKS LIKE (CONTINUED)

```java
// OneInputStreamOperator.java
@PublicEvolving
public interface OneInputStreamOperator<IN, OUT> extends StreamOperator<OUT> {
  // process messages
  void processElement(StreamRecord<IN> element) throws Exception;
  void processWatermark(Watermark mark) throws Exception;
  void processLatencyMarker(LatencyMarker latencyMarker) throws Exception;
}
```

or

```java
// TwoInputStreamOperator.java
@PublicEvolving
public interface TwoInputStreamOperator<IN1, IN2, OUT> extends StreamOperator<OUT> {
  // process messages x2
  void processElement1(StreamRecord<IN1> element) throws Exception;
  void processElement2(StreamRecord<IN2> element) throws Exception;
  void processWatermark1(Watermark mark) throws Exception;
  void processWatermark2(Watermark mark) throws Exception;
  void processLatencyMarker1(LatencyMarker latencyMarker) throws Exception;
  void processLatencyMarker2(LatencyMarker latencyMarker) throws Exception;
}
```

SOME THINGS LOOK FAMILIAR...

We still have methods for:

- hooking into life-cycle
- dealing with state
- processing messages

HOWEVER, WE SEE A WHOLE LOT MORE

- `void setup(StreamTask<?, ?>, StreamConfig, Output>)`
- `OperatotSnapshotFuture snapshotState(...)`
- Handlers for `StreamRecord`, `Watermark` and other messages
- dealing with keys

AND SOME THINGS AREN'T SO CLEAR...

- How do we send messages downstream?
- How do we perform a snapshot?
- How do we set timers?

A HELPING HAND

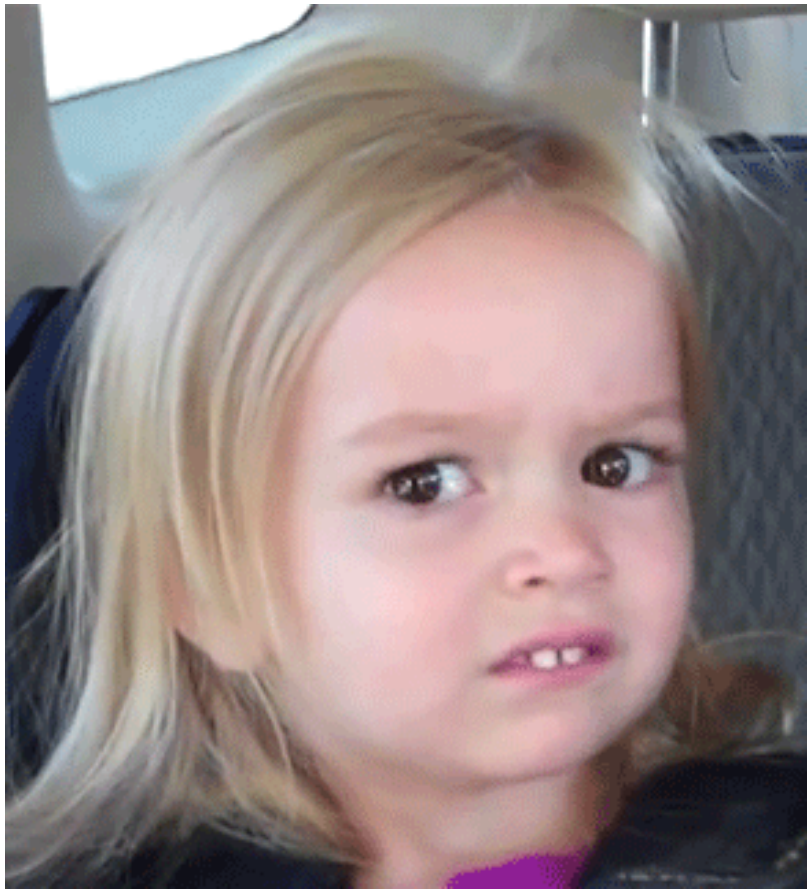
Luckily, we don't have to figure this all out, the `AbstractStreamOperator` gives us some help in using this API

```
class MyFirstOperator
    extends AbstractStreamOperator[String]
    with OneInputStreamOperator[String, String] {




}
```

A HELPING HAND

Luckily, we don't have to figure this all out, the `AbstractStreamOperator` gives us some help in using this API

```
class MyFirstOperator
    extends AbstractStreamOperator[String]
    with OneInputStreamOperator[String, String] {

  override def processElement(element: StreamRecord[String]): Unit = {
    output.collect(element.replace(element.getValue + "!!!!"))
  }
}
```



... THAT'S IT?

WHY WOULD I WANT TO USE THIS?

HOW FLINK USES IT

TimestampsAndPeriodicWatermarksOperator

WHAT DOES IT DO?

When you use any watermark extractor, it gets wrapped in a `TimestampsAndPeriodicWatermarksOperator`

```
class MyWatermarkExtractor extends AssignerWithPeriodicWatermarks[MyEvent] {
  private var maxTimestamp: Long = 0L
  override def extractTimestamp(element: MyEvent, previousElementTimestamp: Long): Long = {
    if (element.timestamp > maxTimestamp) {
      maxTimestamp = element.timestamp
    }
    element.timestamp
  }
  override def getCurrentWatermark(): Watermark = new Watermark(maxTimestamp - 1)
}

dataStream.assignTimestampsAndWatermarks(new MyWatermarkExtractor)
```

THE CODE

```
public class TimestampsAndPeriodicWatermarksOperator<T>
        extends AbstractUdfStreamOperator<T, AssignerWithPeriodicWatermarks<T>>
        implements OneInputStreamOperator<T, T>, ProcessingTimeCallback {
    private transient long watermarkInterval;
    private transient long currentWatermark;

    public TimestampsAndPeriodicWatermarksOperator(AssignerWithPeriodicWatermarks<T> assigner) {
        super(assigner);
        this.chainingStrategy = ChainingStrategy.ALWAYS;
    }
    @Override
    public void open() throws Exception {
        super.open();

        currentWatermark = Long.MIN_VALUE;
        watermarkInterval = getExecutionConfig().getAutoWatermarkInterval();

        if (watermarkInterval > 0) {
            long now = getProcessingTimeService().getCurrentProcessingTime();
            getProcessingTimeService().registerTimer(now + watermarkInterval, this);
        }
    }
  @Override
    public void processElement(StreamRecord<T> element) throws Exception {
        final long newTimestamp = userFunction.extractTimestamp(element.getValue(),
                element.hasTimestamp() ? element.getTimestamp() : Long.MIN_VALUE);

        output.collect(element.replace(element.getValue(), newTimestamp));
    }
```

THE CODE

```
public class TimestampsAndPeriodicWatermarksOperator<T>
        extends AbstractUdfStreamOperator<T, AssignerWithPeriodicWatermarks<T>>
        implements OneInputStreamOperator<T, T>, ProcessingTimeCallback {
    private transient long watermarkInterval;
    private transient long currentWatermark;

    public TimestampsAndPeriodicWatermarksOperator(AssignerWithPeriodicWatermarks<T> assigner) {
        super(assigner);
        this.chainingStrategy = ChainingStrategy.ALWAYS;
    }
    @Override
    public void open() throws Exception {
        super.open();

        currentWatermark = Long.MIN_VALUE;
        watermarkInterval = getExecutionConfig().getAutoWatermarkInterval();

        if (watermarkInterval > 0) {
            long now = getProcessingTimeService().getCurrentProcessingTime();
            getProcessingTimeService().registerTimer(now + watermarkInterval, this);
        }
    }
  @Override
    public void processElement(StreamRecord<T> element) throws Exception {
        final long newTimestamp = userFunction.extractTimestamp(element.getValue(),
                element.hasTimestamp() ? element.getTimestamp() : Long.MIN_VALUE);

        output.collect(element.replace(element.getValue(), newTimestamp));
    }
```

THE CODE

```
public class TimestampsAndPeriodicWatermarksOperator<T>
        extends AbstractUdfStreamOperator<T, AssignerWithPeriodicWatermarks<T>>
        implements OneInputStreamOperator<T, T>, ProcessingTimeCallback {
    private transient long watermarkInterval;
    private transient long currentWatermark;

    public TimestampsAndPeriodicWatermarksOperator(AssignerWithPeriodicWatermarks<T> assigner) {
        super(assigner);
        this.chainingStrategy = ChainingStrategy.ALWAYS;
    }
    @Override
    public void open() throws Exception {
        super.open();

        currentWatermark = Long.MIN_VALUE;
        watermarkInterval = getExecutionConfig().getAutoWatermarkInterval();

        if (watermarkInterval > 0) {
            long now = getProcessingTimeService().getCurrentProcessingTime();
            getProcessingTimeService().registerTimer(now + watermarkInterval, this);
        }
    }
  @Override
    public void processElement(StreamRecord<T> element) throws Exception {
        final long newTimestamp = userFunction.extractTimestamp(element.getValue(),
                element.hasTimestamp() ? element.getTimestamp() : Long.MIN_VALUE);

        output.collect(element.replace(element.getValue(), newTimestamp));
    }
```

THE CODE

```
public class TimestampsAndPeriodicWatermarksOperator<T>
        extends AbstractUdfStreamOperator<T, AssignerWithPeriodicWatermarks<T>>
        implements OneInputStreamOperator<T, T>, ProcessingTimeCallback {
    private transient long watermarkInterval;
    private transient long currentWatermark;

    public TimestampsAndPeriodicWatermarksOperator(AssignerWithPeriodicWatermarks<T> assigner) {
        super(assigner);
        this.chainingStrategy = ChainingStrategy.ALWAYS;
    }
    @Override
    public void open() throws Exception {
        super.open();

        currentWatermark = Long.MIN_VALUE;
        watermarkInterval = getExecutionConfig().getAutoWatermarkInterval();

        if (watermarkInterval > 0) {
            long now = getProcessingTimeService().getCurrentProcessingTime();
            getProcessingTimeService().registerTimer(now + watermarkInterval, this);
        }
    }
  @Override
    public void processElement(StreamRecord<T> element) throws Exception {
        final long newTimestamp = userFunction.extractTimestamp(element.getValue(),
                element.hasTimestamp() ? element.getTimestamp() : Long.MIN_VALUE);

        output.collect(element.replace(element.getValue(), newTimestamp));
    }
```

THE CODE

```
public class TimestampsAndPeriodicWatermarksOperator<T>
        extends AbstractUdfStreamOperator<T, AssignerWithPeriodicWatermarks<T>>
        implements OneInputStreamOperator<T, T>, ProcessingTimeCallback {
    private transient long watermarkInterval;
    private transient long currentWatermark;

    public TimestampsAndPeriodicWatermarksOperator(AssignerWithPeriodicWatermarks<T> assigner) {
        super(assigner);
        this.chainingStrategy = ChainingStrategy.ALWAYS;
    }
    @Override
    public void open() throws Exception {
        super.open();

        currentWatermark = Long.MIN_VALUE;
        watermarkInterval = getExecutionConfig().getAutoWatermarkInterval();

        if (watermarkInterval > 0) {
            long now = getProcessingTimeService().getCurrentProcessingTime();
            getProcessingTimeService().registerTimer(now + watermarkInterval, this);
        }
    }
  @Override
    public void processElement(StreamRecord<T> element) throws Exception {
        final long newTimestamp = userFunction.extractTimestamp(element.getValue(),
                element.hasTimestamp() ? element.getTimestamp() : Long.MIN_VALUE);

        output.collect(element.replace(element.getValue(), newTimestamp));
    }
```

THE CODE (CONTINUED)

```
  @Override
  public void onProcessingTime(long timestamp) throws Exception {
    // register next timer
    Watermark newWatermark = userFunction.getCurrentWatermark();
    if (newWatermark != null && newWatermark.getTimestamp() > currentWatermark) {
      currentWatermark = newWatermark.getTimestamp();
      // emit watermark
      output.emitWatermark(newWatermark);
    }

    long now = getProcessingTimeService().getCurrentProcessingTime();
    getProcessingTimeService().registerTimer(now + watermarkInterval, this);
  }

  /**
   * Override the base implementation to completely ignore watermarks propagated from
   * upstream (we rely only on the {@link AssignerWithPunctuatedWatermarks} to emit
   * watermarks from here).
   */
  @Override
  public void processWatermark(Watermark mark) throws Exception {
    // if we receive a Long.MAX_VALUE watermark we forward it since it is used
    // to signal the end of input and to not block watermark progress downstream
    if (mark.getTimestamp() == Long.MAX_VALUE && currentWatermark != Long.MAX_VALUE) {
      currentWatermark = Long.MAX_VALUE;
      output.emitWatermark(mark);
    }
  }
  ...
}
```

THE CODE (CONTINUED)

```
  @Override
  public void onProcessingTime(long timestamp) throws Exception {
    // register next timer
    Watermark newWatermark = userFunction.getCurrentWatermark();
    if (newWatermark != null && newWatermark.getTimestamp() > currentWatermark) {
      currentWatermark = newWatermark.getTimestamp();
      // emit watermark
      output.emitWatermark(newWatermark);
    }

    long now = getProcessingTimeService().getCurrentProcessingTime();
    getProcessingTimeService().registerTimer(now + watermarkInterval, this);
  }

  /**
   * Override the base implementation to completely ignore watermarks propagated from
   * upstream (we rely only on the {@link AssignerWithPunctuatedWatermarks} to emit
   * watermarks from here).
   */
  @Override
  public void processWatermark(Watermark mark) throws Exception {
    // if we receive a Long.MAX_VALUE watermark we forward it since it is used
    // to signal the end of input and to not block watermark progress downstream
    if (mark.getTimestamp() == Long.MAX_VALUE && currentWatermark != Long.MAX_VALUE) {
      currentWatermark = Long.MAX_VALUE;
      output.emitWatermark(mark);
    }
  }
  ...
}
```

THE CODE (CONTINUED)

```
  @Override
  public void onProcessingTime(long timestamp) throws Exception {
    // register next timer
    Watermark newWatermark = userFunction.getCurrentWatermark();
    if (newWatermark != null && newWatermark.getTimestamp() > currentWatermark) {
      currentWatermark = newWatermark.getTimestamp();
      // emit watermark
      output.emitWatermark(newWatermark);
    }

    long now = getProcessingTimeService().getCurrentProcessingTime();
    getProcessingTimeService().registerTimer(now + watermarkInterval, this);
  }

  /**
   * Override the base implementation to completely ignore watermarks propagated from
   * upstream (we rely only on the {@link AssignerWithPunctuatedWatermarks} to emit
   * watermarks from here).
   */
  @Override
  public void processWatermark(Watermark mark) throws Exception {
    // if we receive a Long.MAX_VALUE watermark we forward it since it is used
    // to signal the end of input and to not block watermark progress downstream
    if (mark.getTimestamp() == Long.MAX_VALUE && currentWatermark != Long.MAX_VALUE) {
      currentWatermark = Long.MAX_VALUE;
      output.emitWatermark(mark);
    }
  }
  ...
}
```

WHAT WE LEARN

- In `processElement` we get the `StreamRecord` which allows us to modify not only the record, but it's timestamp
- In `processWatermark` we can change or completely ignore watermarks
- We have access to lots of APIs, such as timer services, `output` and `chainingStrategy` properties

THE `StreamOperator` PATTERN

As mentioned, all of the `DataStream` API is built on top of and encapsulated by the `StreamOperator` API

THE `StreamOperator` PATTERN

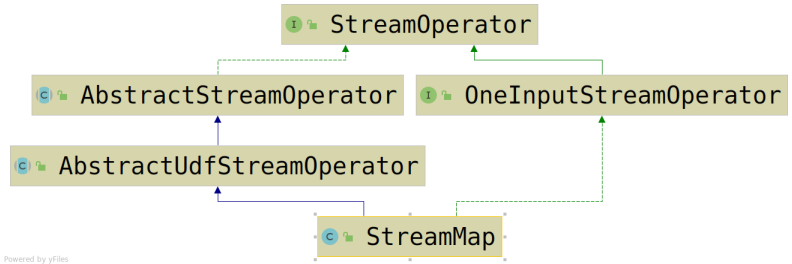
As mentioned, all of the `DataStream` API is built on top of and encapsulated by the `StreamOperator` API

```
public <R> SingleOutputStreamOperator<R> map(MapFunction<T, R> mapper) {
  TypeInformation<R> outType = ...;
  return transform("Map", outType, new StreamMap<>(clean(mapper)));
}
```

THE `StreamOperator` PATTERN

As mentioned, all of the `DataStream` API is built on top of and encapsulated by the `StreamOperator` API

```
public <R> SingleOutputStreamOperator<R> map(MapFunction<T, R> mapper) {
  TypeInformation<R> outType = ...;
  return transform("Map", outType, new StreamMap<>(clean(mapper)));
}
```



OUR FIRST REAL OPERATOR

HANDLING A STREAM OF FILES

WE HAVE A JOB THAT:

- has a single source which produces `FileReadRequest` messages, can send anywhere from 150 to 0.1 msgs/minute

HANDLING A STREAM OF FILES

WE HAVE A JOB THAT:

- has a single source which produces `FileReadRequest` messages, can send anywhere from 150 to 0.1 msgs/minute
- a process function which reads the file and produces roughly ~100k messages per file, these new messages contain our event-time

HANDLING A STREAM OF FILES

WE HAVE A JOB THAT:

- has a single source which produces `FileReadRequest` messages, can send anywhere from 150 to 0.1 msgs/minute
- a process function which reads the file and produces roughly ~100k messages per file, these new messages contain our event-time
- downstream logic that re-keys, aggregates, and sums into 1 hour windows

IN CODE

```
// our message types
case class FileReadRequest(uri: String)
case class FileMessage(sourceUri: String, timestamp: Long, id: String, value: Long)
```

IN CODE

```
// our message types
case class FileReadRequest(uri: String)
case class FileMessage(sourceUri: String, timestamp: Long, id: String, value: Long)
// our functions
class FileRequestSource extends SourceFunction[FileReadRequest] {...}
class FileReaderProcess extends ProcessFunction[FileReadRequest, FileMessage] {...}
class WatermarkAssigner extends AssignerWithPeriodicWatermarks[FileMessage] {...}
```

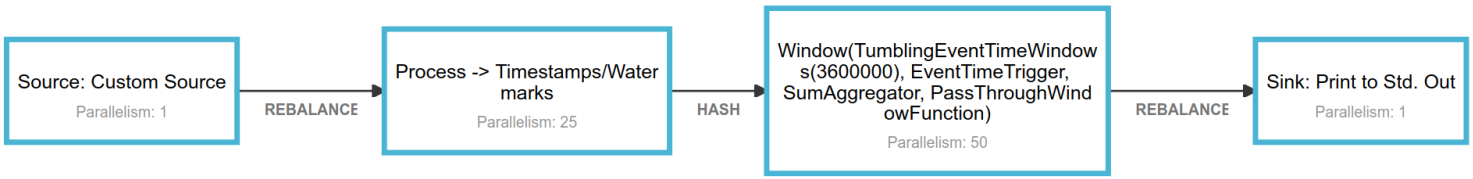
IN CODE

```
// our message types
case class FileReadRequest(uri: String)
case class FileMessage(sourceUri: String, timestamp: Long, id: String, value: Long)
// our functions
class FileRequestSource extends SourceFunction[FileReadRequest] {...}
class FileReaderProcess extends ProcessFunction[FileReadRequest, FileMessage] {...}
class WatermarkAssigner extends AssignerWithPeriodicWatermarks[FileMessage] {...}
// our job
class FileReaderDemo {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    val fileStream    = env.addSource(new FileRequestSource)
    val messageStream = fileStream
      .process(new FileReaderProcess)
      .setParallelism(25)
      .assignTimestampsAndWatermarks(new WatermarkAssigner)
```

IN CODE

```
// our message types
case class FileReadRequest(uri: String)
case class FileMessage(sourceUri: String, timestamp: Long, id: String, value: Long)
// our functions
class FileRequestSource extends SourceFunction[FileReadRequest] {...}
class FileReaderProcess extends ProcessFunction[FileReadRequest, FileMessage] {...}
class WatermarkAssigner extends AssignerWithPeriodicWatermarks[FileMessage] {...}
// our job
class FileReaderDemo {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    val fileStream    = env.addSource(new FileRequestSource)
    val messageStream = fileStream
      .process(new FileReaderProcess)
      .setParallelism(25)
      .assignTimestampsAndWatermarks(new WatermarkAssigner)

    messageStream
      .keyBy("id")
      .window(TumblingEventTimeWindows.of(Time.hours(1)))
      .sum("value")
      .setParallelism(50)
      .addSink(...)
  }
}
```

JOB GRAPH


Source: Custom Source
Parallelism: 1
REBALANCE
Process -> Timestamps/Watermarks
Parallelism: 25
HASH
Window(TumblingEventTimeWindows(3600000), EventTimeTrigger, SumAggregator, PassThroughWindowFunction)
Parallelism: 50
REBALANCE
Sink: Print to Std. Out
Parallelism: 1

PROBLEMS ARISE

At low load, we start seeing no windows closing until load picks back up

Analysis shows the watermark isn't advancing

ROOT CAUSE

- At low load, we get roughly 6 msgs/hour

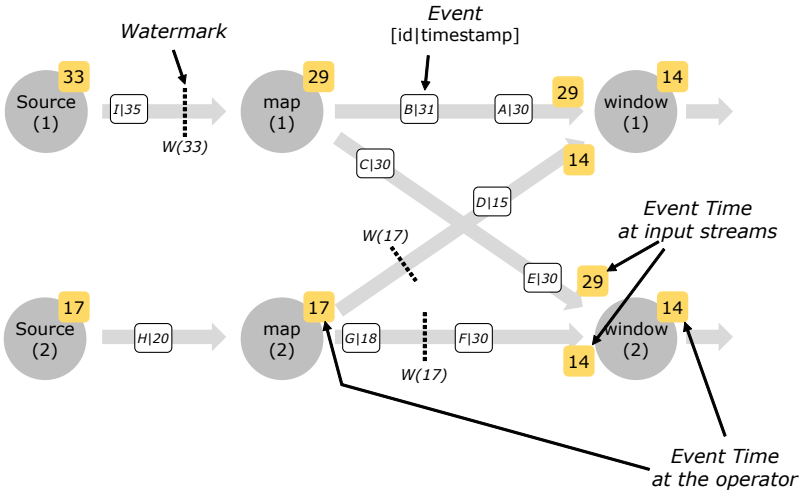
ROOT CAUSE

- At low load, we get roughly 6 msgs/hour
- In an hour, only ~25% of readers will have a message to process, the rest will be idle

ROOT CAUSE

- At low load, we get roughly 6 msgs/hour
- In an hour, only ~25% of readers will have a message to process, the rest will be idle
- Since we assign watermarks after our reader, earlier watermarks are discarded

DEALING WITH IDLE STREAMS



The watermark for an operator is the minimum of all the input watermarks, when we have a single "source" that doesn't output any records, the watermark doesn't advance

Without an advancing watermark, we can't close windows, and we stop getting results

HOW TO FIX IT!

Sources can be marked as idle using SourceFunction.SourceContext #markAsTemporarilyIdle

From https://ci.apache.org/projects/flink/flink-docs-stable/dev/event_time.html#idling-sources

EXCEPT... OUR SOURCE ISN'T WHAT IS IDLE...

WHAT IS A SOURCE REALLY?

In this case, the real "source" of most of our data is **not** the SourceFunction

With the operator API, we can work around this API limitation

OUR NEW ABSTRACTION

```
abstract class MessageableSource[IN, OUT](idleTimeout: Long)
    extends AbstractStreamOperator[OUT]
    with OneInputStreamOperator[IN, OUT] {









}
```

OUR NEW ABSTRACTION

```
abstract class MessageableSource[IN, OUT](idleTimeout: Long)
    extends AbstractStreamOperator[OUT]
    with OneInputStreamOperator[IN, OUT] {
  private var sourceCtx: SourceContext[OUT] = _
  protected def sourceContext: SourceContext[OUT] = {
    if (sourceCtx == null) {
      sourceCtx = StreamSourceContexts.getSourceContext(
        getOperatorConfig.getTimeCharacteristic,
        getContainingTask.getProcessingTimeService,
        getContainingTask.getCheckpointLock,
        getContainingTask.getStreamStatusMaintainer,
        output,
        getRuntimeContext.getExecutionConfig.getAutoWatermarkInterval,
        idleTimeout
      )
    }
    sourceCtx
  }



}
```

OUR NEW ABSTRACTION

```
abstract class MessageableSource[IN, OUT](idleTimeout: Long)
    extends AbstractStreamOperator[OUT]
    with OneInputStreamOperator[IN, OUT] {
  private var sourceCtx: SourceContext[OUT] = _
  protected def sourceContext: SourceContext[OUT] = {
    if (sourceCtx == null) {
      sourceCtx = StreamSourceContexts.getSourceContext(
        getOperatorConfig.getTimeCharacteristic,
        getContainingTask.getProcessingTimeService,
        getContainingTask.getCheckpointLock,
        getContainingTask.getStreamStatusMaintainer,
        output,
        getRuntimeContext.getExecutionConfig.getAutoWatermarkInterval,
        idleTimeout
      )
    }
    sourceCtx
  }

  def processElement(el: IN, sourceCtx: SourceContext[OUT]): Unit


}
```

OUR NEW ABSTRACTION

```
abstract class MessageableSource[IN, OUT](idleTimeout: Long)
    extends AbstractStreamOperator[OUT]
    with OneInputStreamOperator[IN, OUT] {
  private var sourceCtx: SourceContext[OUT] = _
  protected def sourceContext: SourceContext[OUT] = {
    if (sourceCtx == null) {
      sourceCtx = StreamSourceContexts.getSourceContext(
        getOperatorConfig.getTimeCharacteristic,
        getContainingTask.getProcessingTimeService,
        getContainingTask.getCheckpointLock,
        getContainingTask.getStreamStatusMaintainer,
        output,
        getRuntimeContext.getExecutionConfig.getAutoWatermarkInterval,
        idleTimeout
      )
    }
    sourceCtx
  }

  def processElement(el: IN, sourceCtx: SourceContext[OUT]): Unit
  override def processElement(element: StreamRecord[IN]): Unit =
    processElement(element.getValue, sourceContext)
}
```

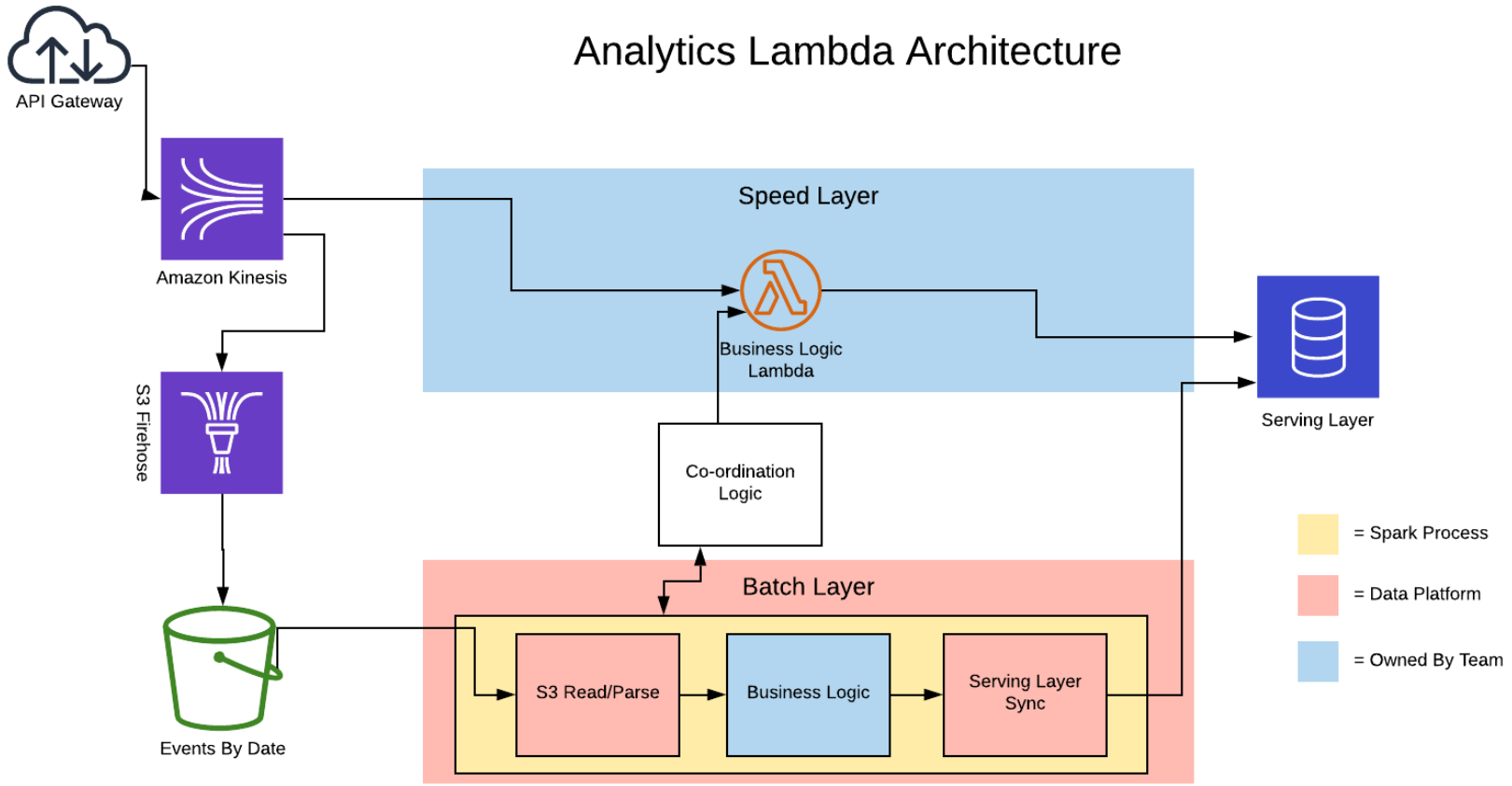
USED IN OUR NEW JOB

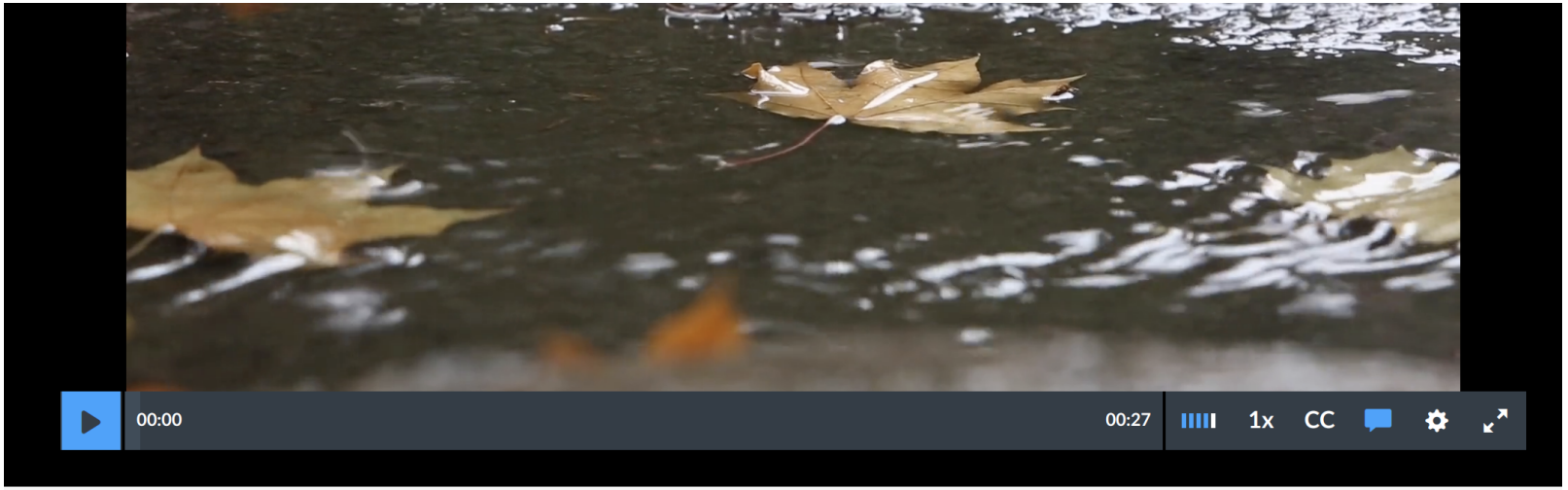
```
// our functions
class FileRequestSource extends SourceFunction[FileReadRequest] {...}
class FileReaderSource extends MessageableSource[FileReadRequest, FileMessage](idleTimeout) {...
class WatermarkAssigner extends AssignerWithPeriodicWatermarks[FileMessage] {...}
// our message types
case class FileReadRequest(uri: String)
case class FileMessage(sourceUri: String, timestamp: Long, id: String, value: Long)
// our job
class FileReaderDemo {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    val fileStream    = env.addSource(new FileRequestSource)
    val messageStream = fileStream
      .transform("fileReader", new FileReaderSource)
      .setParallelism(25)
      .assignTimestampsAndWatermarks(new WatermarkAssigner)

    messageStream
      .keyBy("id")
      .window(TumblingEventTimeWindows.of(Time.hours(1)))
      .sum("value")
      .setParallelism(50)
      .addSink(...)
  }
}
```

USED IN OUR NEW JOB

```
// our functions
class FileRequestSource extends SourceFunction[FileReadRequest] {...}
class FileReaderSource extends MessageableSource[FileReadRequest, FileMessage](idleTimeout) {...
class WatermarkAssigner extends AssignerWithPeriodicWatermarks[FileMessage] {...}
// our message types
case class FileReadRequest(uri: String)
case class FileMessage(sourceUri: String, timestamp: Long, id: String, value: Long)
// our job
class FileReaderDemo {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    val fileStream    = env.addSource(new FileRequestSource)
    val messageStream = fileStream
      .transform("fileReader", new FileReaderSource)
      .setParallelism(25)
      .assignTimestampsAndWatermarks(new WatermarkAssigner)

    messageStream
      .keyBy("id")
      .window(TumblingEventTimeWindows.of(Time.hours(1)))
      .sum("value")
      .setParallelism(50)
      .addSink(...)
  }
}
```

OUT WITH LAMBDA, IN WITH KAPPA


Analytics Lambda Architecture
API Gateway
Amazon Kinesis
S3 Firehose
Events By Date
Speed Layer
Business Logic
Lambda
Co-ordination
Logic
Batch Layer
S3 Read/Parse
Business Logic
Serving Layer
Sync
Serving Layer
= Spark Process
= Data Platform
= Owned By Team

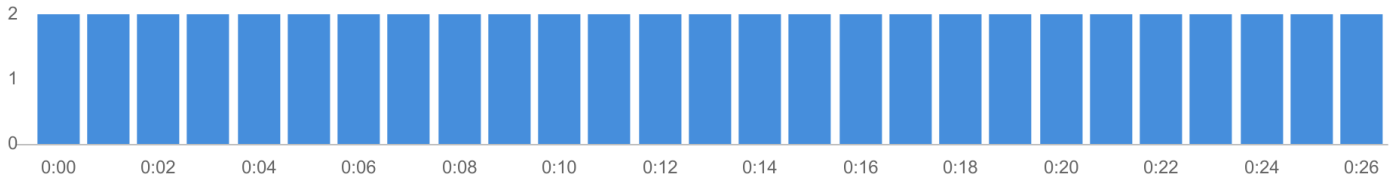


add your two cents

Comment at 0:00

Details Comments Insights Captions

All Viewers




addison


bryant

LAMBDA ARCHITECTURE IS NO GOOD

KAPPA ARCHITECTURE SOLVES EVERYTHING RIGHT?

A STEP IN THE RIGHT DIRECTION, BUT A LOT OF WAYS TO GO ABOUT IT

STREAM RETENTION

- Still a very unsolved problem
- Many message transport layers don't support infinite retention at all (Kinesis) or it can be expensive (Kafka)
- Even with more advanced solutions (Pulsar or Pravega tiered storage), scaling reads is non-trivial

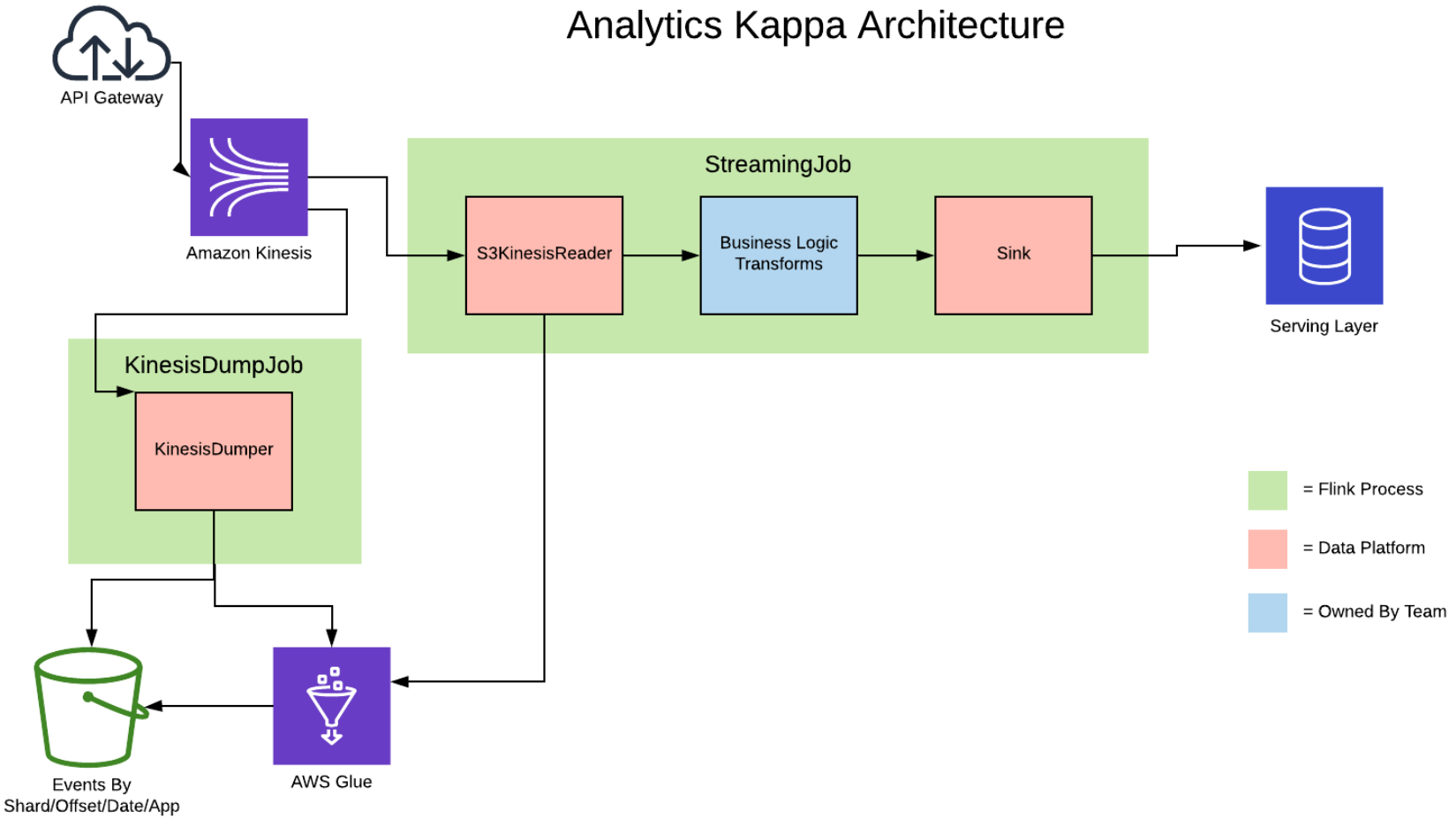
REUSE STREAM LOGIC WITH TWO SOURCES

- Another common pattern is to re-use logic in different jobs with different sources
- This requires external automation and/or external state to co-ordinate between the two different jobs

OUR APPROACH TO KAPPA

- We weren't ready to move to a new streaming transport and Kafka without trim seemed scary
- We also wanted to avoid external automation or state if possible
- Can we do this all inside Flink?

THE FLEXIBILITY OF FLINK


Analytics Kappa Architecture
API Gateway
Amazon Kinesis
StreamingJob
S3KinesisReader
Business Logic Transforms
Sink
Serving Layer
KinesisDumpJob
KinesisDumper
= Flink Process
= Data Platform
= Owned By Team
Events By Shard/Offset/Date/App
AWS Glue

A BETTER SOLUTION FOR ARCHIVING A STREAM

- By controlling our writes from Kinesis Streams to S3, we could read more granularly for back-fill
- Additionally, by writing offset metadata to AWS Glue, we can seamlessly transition to Kinesis

NOT THE ONLY CHALLENGE

- The `FlinkKinesisConsumer` is complex and not obviously re-usable as it is built on a `ParallelSourceFunction`

NOT THE ONLY CHALLENGE

- The `FlinkKinesisConsumer` is complex and not obviously re-usable as it is built on a `ParallelSourceFunction`
- In order to scale reads during back-fill, we need to read multiple files at once and then re-order the results

NOT THE ONLY CHALLENGE

- The `FlinkKinesisConsumer` is complex and not obviously re-usable as it is built on a `ParallelSourceFunction`
- In order to scale reads during back-fill, we need to read multiple files at once and then re-order the results
- We still can't do bi-directional communication, which makes synchronization a challenge

NOT THE ONLY CHALLENGE

- The `FlinkKinesisConsumer` is complex and not obviously re-usable as it is built on a `ParallelSourceFunction`
- In order to scale reads during back-fill, we need to read multiple files at once and then re-order the results
- We still can't do bi-directional communication, which makes synchronization a challenge
- We still have to worry about properly handling idle streams and watermarks to make downstream code work as expected

A MORE FLEXIBLE "SOURCE"

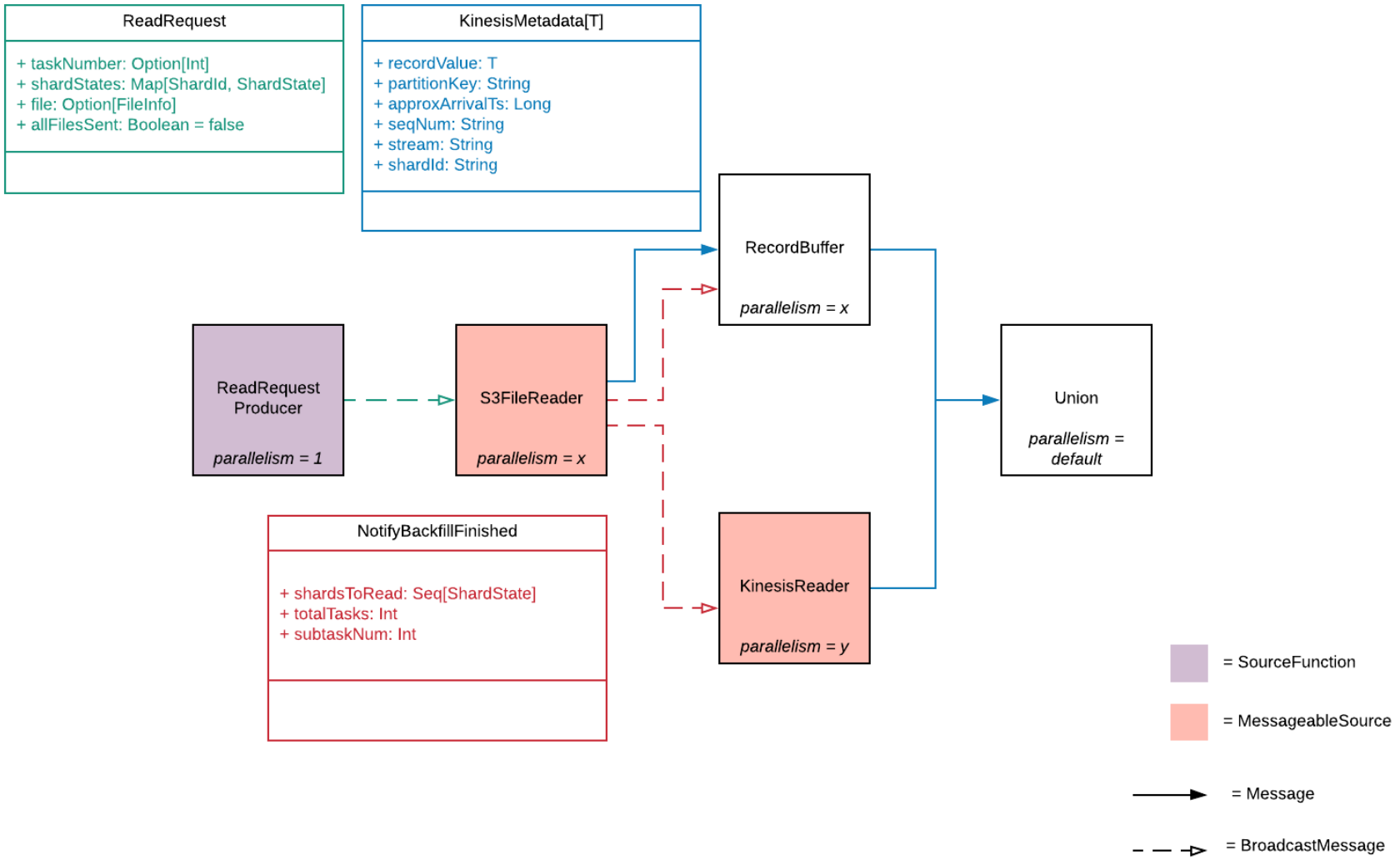
- With the `StreamOperator` API and our own `MessageableSource` abstraction, we could build an adapter for the `KinesisDataFetcher`, the `KinesisReader`

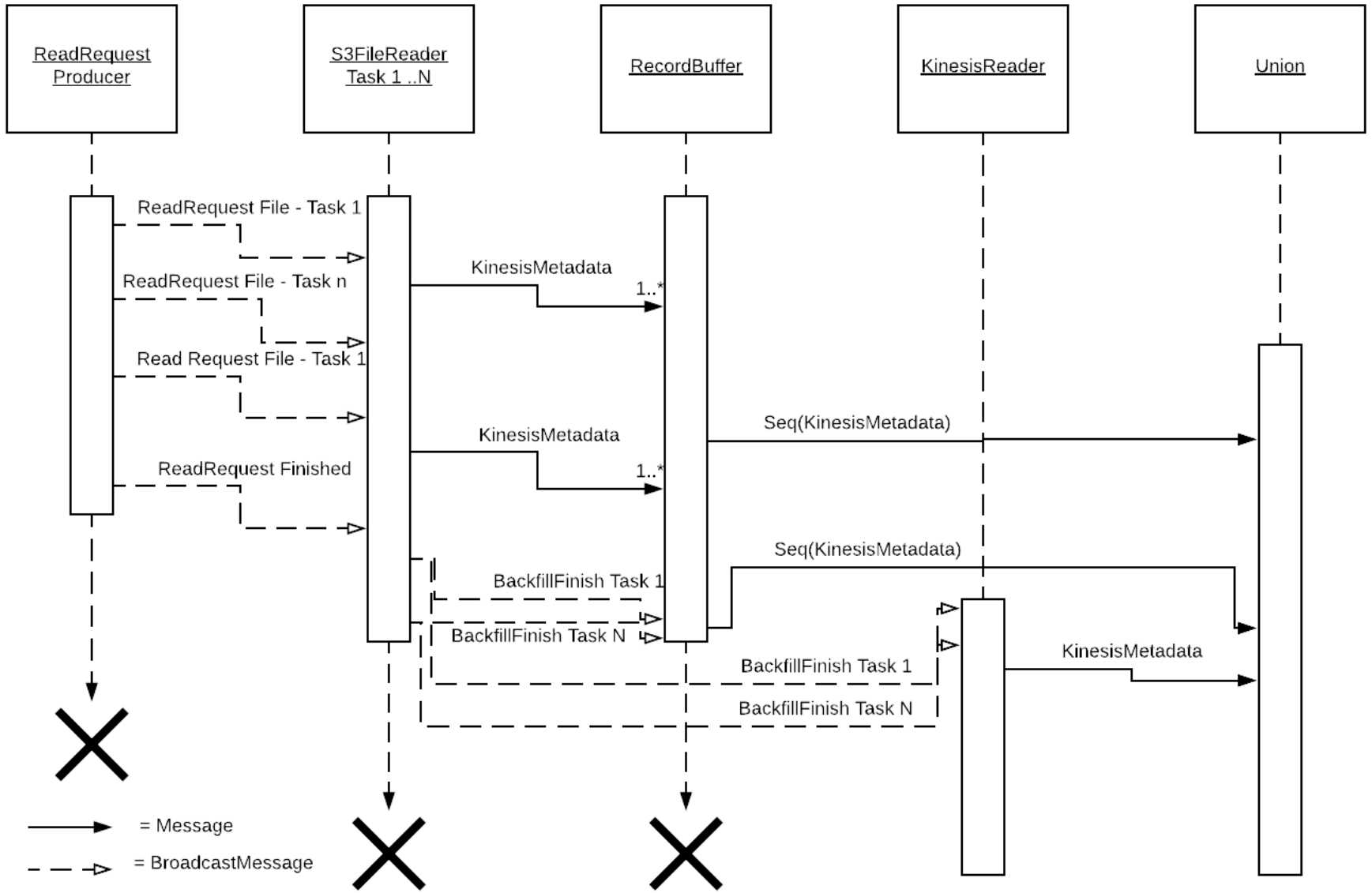
A MORE FLEXIBLE "SOURCE"

- With the `StreamOperator` API and our own `MessageableSource` abstraction, we could build an adapter for the `KinesisDataFetcher`, the `KinesisReader`
- We also use the `MessageableSource` to build a "source" which recieves messages of files to read, the `S3FileReader`

A MORE FLEXIBLE "SOURCE"

- With the `StreamOperator` API and our own `MessageableSource` abstraction, we could build an adapter for the `KinesisDataFetcher`, the `KinesisReader`
- We also use the `MessageableSource` to build a "source" which recieves messages of files to read, the `S3FileReader`
- These sources allowed us to back-fill data from S3, then transition to reading from Kinesis all with the same job graph, which we encapsulate in a single class, the `S3KinesisReader`


ReadRequest
+ taskNumber: Option[Int]
+ shardStates: Map[ShardId, ShardState]
+ file: Option[FileInfo]
+ allFilesSent: Boolean = false
KinesisMetadata[T]
+ recordValue: T
+ partitionKey: String
+ approxArrivalTs: Long
+ seqNum: String
+ stream: String
+ shardId: String
RecordBuffer
parallelism = x
ReadRequest Producer
parallelism = 1
S3FileReader
parallelism = x
Union
parallelism = default
NotifyBackfillFinished
+ shardsToRead: Seq[ShardState]
+ totalTasks: Int
+ subtaskNum: Int
KinesisReader
parallelism = y
= SourceFunction
= MessageableSource
= Message
= BroadcastMessage


ReadRequest Producer
S3FileReader Task 1 ..N
RecordBuffer
KinesisReader
Union
ReadRequest File - Task 1
ReadRequest File - Task n
Read Request File - Task 1
ReadRequest Finished
KinesisMetadata
1..*
KinesisMetadata
1..*
Seq(KinesisMetadata)
Seq(KinesisMetadata)
BackfillFinish Task 1
BackfillFinish Task N
BackfillFinish Task 1
BackfillFinish Task N
KinesisMetadata
= Message
= BroadcastMessage

REMOVING UNNEEDED STUFF

- Once we are done back-filling, we want to eventually scale down batch reading portions
- We can do this either by just scaling down the `S3FileReaders` or conditionally building a different graph

OUR RESULTS SO FAR

- Much less business logic to maintain, first use case reduced from ~500 loc to ~100 loc
- Performance has been good in back-fill, but not strictly ordered
- Figuring out glue partitioning to avoid small partitions is a a challenge
- Still some bugs to iron out

UPCOMING FLINK CHANGES

FLIP-27 - Refactor Source Interface

Aims to separate out sources into two components `SplitEnumerator` and `SplitReader`, which should allow for more flexible sources

It also formalizes `Unbounded` and `Bounded` streams, which should make unified batch and streaming easier

SHOULD YOU DO THIS?

SHOULD YOU DO THIS?

¯\(ツ)/¯

WHY YOU MIGHT WANT TO HOLD OFF

- Flink will get better support for unified batch/streaming
- This is pretty experimental, but working for us

IN CONCLUSION

- The `StreamOperator` API is how Flink achieves a lot of the features we know and love
- Understanding it can make it easier to reason about your Flink applications
- You can use it to do some really powerful things and build your own abstractions (but be careful)

TALKS YOU MIGHT WANT TO SEE TODAY:

- Towards Flink 2.0 - Nikko II & III at 2:00PM
- Moving From Lambda and Kappa to Kappa+ - Nikko II & III at 3:20PM

THANKS!

QUESTIONS?