# Flink Development Week

**Day 1: Introduction**

- Streaming Fundamentals and History
- Basics of Flink
- My first Flink application

**Day 2: Fundamentals of Flink Streaming**

- Events
- Watermarks
- Completeness and Lateness
- Windows
- Timers

**Day 3: Running Flink in a Cluster**

- Flink as a Streaming Middleware
- Job and Task Managers
- Connecting to Kafka

**Day 4a: Advanced Routing and State**

- Advanced Event Routing
- Accumulation and State

**Day 4b: Stream Joining**

- State & Checkpointing for fault tolerance
- Joining Streams including Fast and Slow-moving streams

**Day 5: Scheduling and Timers**

- Timers and Windowed Streams
- Late Events

**Day 6: Best Practices**

- Testing Flink Functions
- Unit Tests
- Common gotchas

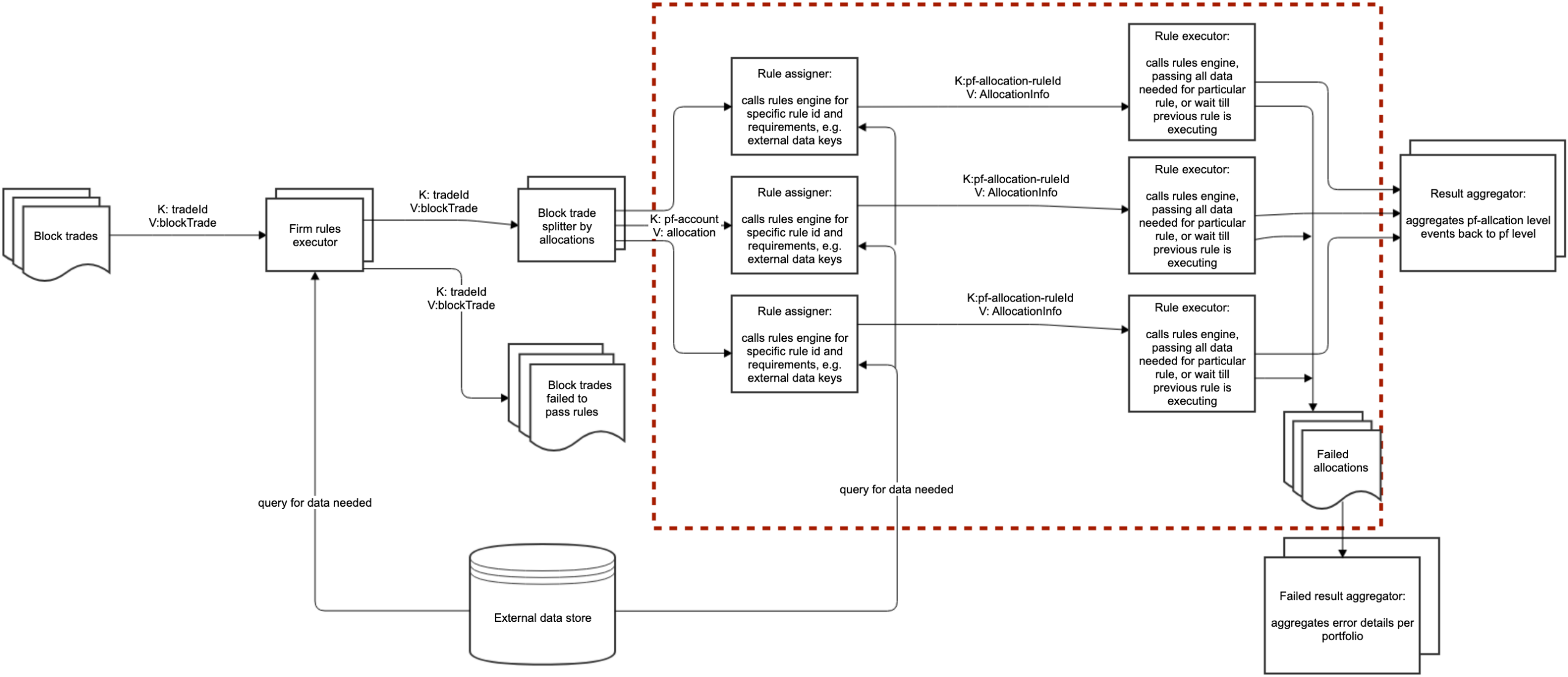
riskfocus

# Flink Training Day 4a: Advanced Routing and State

July 2020

riskfocus

# The Big Picture



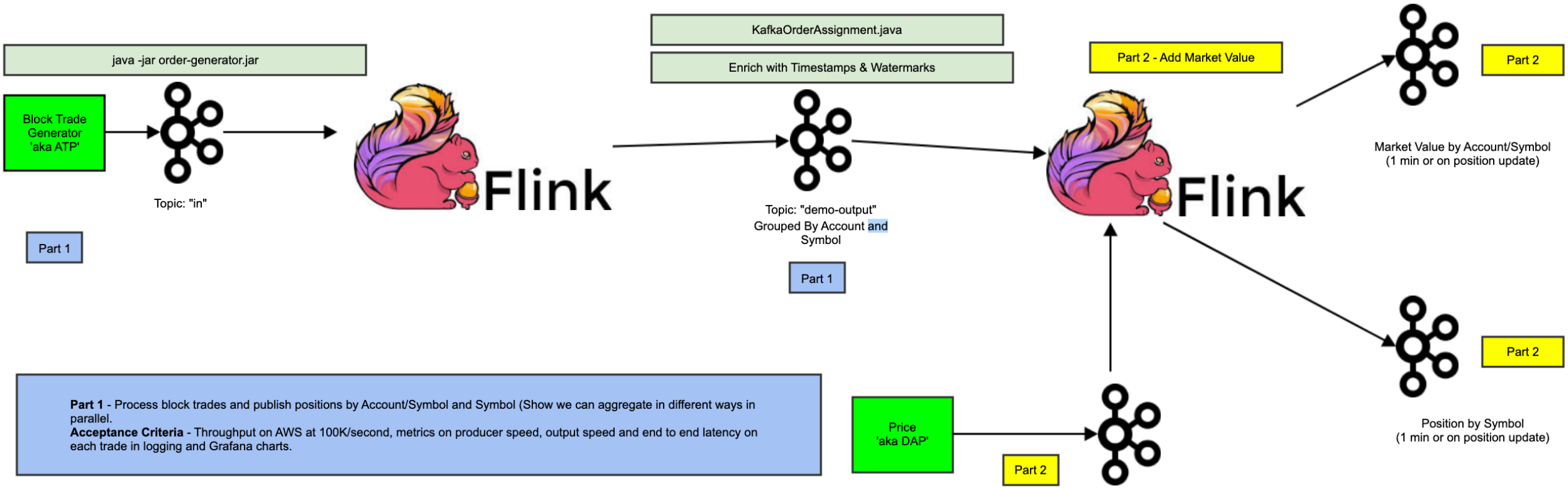
# Day 4 - Agenda

In this training you will learn about creating a data pipeline between Kafka and Flink. Training will focus on these key concepts:

- Kafka as a DataSource
- Watermarks and event time
- Aggregating Data
- Kafka as a DataSync

Knowledge of JAVA 8+ and Flink 1.10 or higher is required and at the end of each training section there will be a hands-on lab.

# Block Orders Pipeline




**Part 1** - Process block trades and publish positions by Account/Symbol and Symbol (Show we can aggregate in different ways in parallel.
**Acceptance Criteria** - Throughput on AWS at 100K/second, metrics on producer speed, output speed and end to end latency on each trade in logging and Grafana charts.

**Part 2**- add joining price to position updates to give market value & add new topic to publish Market value every 1 min.
**Acceptance Criteria** - same as Part 1 (standard metrics inputs, outputs, latency). + scale to maintain performance of Part 1 & use Snapshots in Flink

riskfocus

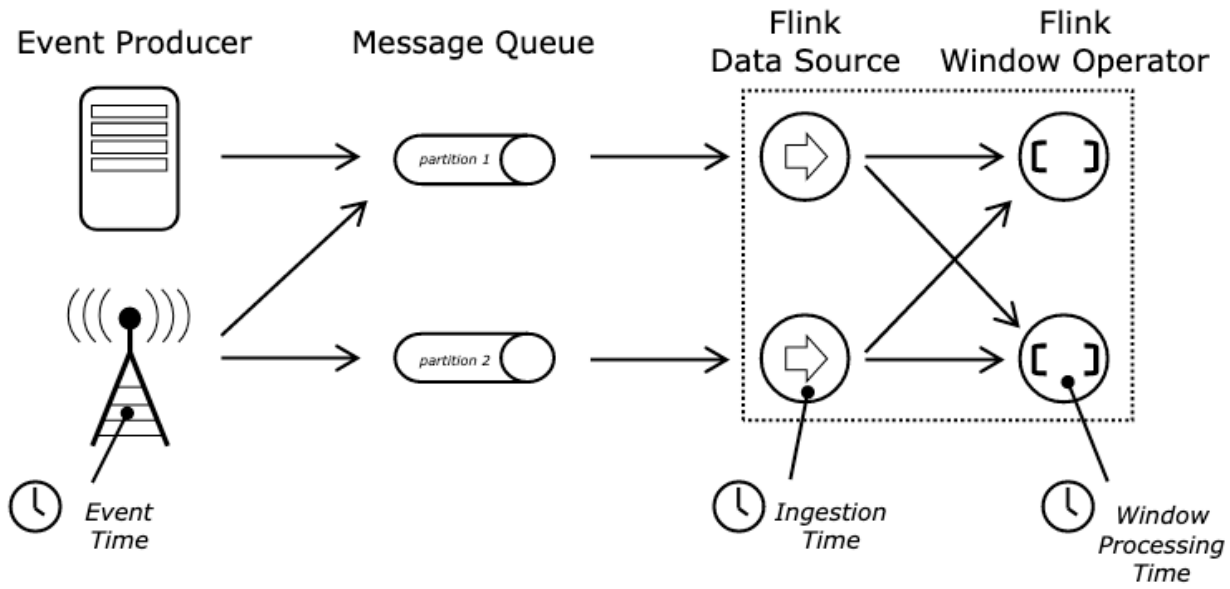
# Day 4a Recap

- DataStream API
  - Stream Sources
  - Stream Sinks
  - Operators
  - Keyed Streams
  - Event Time
  - Watermarks

riskfocus

# Event Time

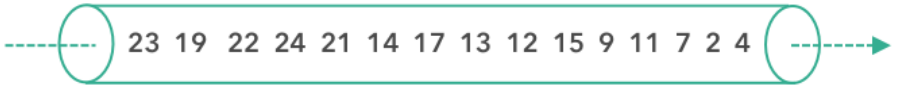
- ***processing time**:* system time of the machine that is executing the respective operation (wall clock time). Processing time is the simplest notion of time and requires no coordination between streams and machines. It provides the best performance and the lowest latency. However, in distributed and asynchronous environments processing time does not provide determinism.
- ***event time**:* time of each individual event occurred on its producing device. This time is typically embedded within the records before they enter Flink, and that *event timestamp* can be extracted from each record. In event time, the progress of time depends on the data, not on any wall clocks.
- ***ingestion time**:* time that events enter Flink. At the source operator each record gets the source's current time as a timestamp, and time-based operations (like time windows) refer to that timestamp. *Ingestion time* sits conceptually in between *event time* and *processing time*. Compared to *processing time*, it is slightly more expensive, but gives more predictable results.




riskfocus

# Watermarks

A stream processor that supports *event time* needs a way to measure the progress of event time. For example, a window operator that builds hourly windows needs to be notified when event time has passed beyond the end of an hour, so that the operator can close the window in progress.
The mechanism in Flink to measure progress in event time is **watermarks**. Watermarks flow as part of the data stream and carry a timestamp *t*. A *Watermark(t)* declares that event time has reached time *t* in that stream, meaning that there should be no more elements from the stream with a timestamp *t'* <= *t* (i.e. events with timestamps older or equal to the watermark)




Event Stream

Note that this is an example of *event-time* processing, meaning that the timestamps reflect when the events took place, and not when they were processed. Event-time processing is a powerful abstraction that makes it possible to create streaming applications that behave consistently whether they are processing live data or re-processing historic data. Now imagine that we are trying to create a stream sorter. This is meant to be an application that processes each event from a stream as it arrives and emits a new stream containing the same events but ordered by their timestamps.
**Observation #1:** *Some buffering, and some delay is necessary.*
**Observation #2:** *Eventually, we have to be courageous and emit the 2 as the start of the sorted stream.*
**Observation #3:** *What we need then is some sort of policy that defines when, for any given timestamped event, to stop waiting for the arrival of earlier events.This is precisely what watermarks do — they define when to stop waiting for earlier events.*
**Observation #4:** *We can imagine different policies for deciding how to generate watermarks.*

riskfocus

# Generating Timestamps / Watermarks

By default, Flink will use processing time. To change this, you can set the Time Characteristic:

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

In order to work with *event time*, Flink needs to know the events' *timestamps*, meaning each element in the stream needs to have its event timestamp *assigned*. This is usually done by accessing/extracting the timestamp from some field in the element. Timestamp assignment goes hand-in-hand with generating watermarks, which tell the system about progress in event time. There are two ways to assign timestamps and generate watermarks:

- Directly in the data stream source
- Via a timestamp assigner / watermark generator: in Flink, timestamp assigners also define the watermarks to be emitted

Both timestamps and watermarks are specified as milliseconds since the Java epoch of 1970-01-01T00:00:00Z.

Timestamp assigners are usually specified immediately after the data source, but it is not strictly required to do so. A common pattern, for example, is to parse (*MapFunction*) and filter (*FilterFunction*) before the timestamp assigner. In any case, the timestamp assigner needs to be specified before the first operation on event time (such as the first window operation)

```
DataStream<MyEvent> withTimestampsAndWatermarks = stream
        .filter(event -> event.severity() == WARNING)
        .assignTimestampsAndWatermarks(new MyTimestampsAndWatermarks());
```

# Periodic and Punctuated assigners

*AssignerWithPeriodicWatermarks* assigns timestamps and generates watermarks periodically (possibly depending on the stream elements, or purely based on processing time).
The interval (every *n* milliseconds) in which the watermark will be generated is defined via *ExecutionConfig.setAutoWatermarkInterval(...)*. The assigner's *getCurrentWatermark()* method will be called each time, and a new watermark will be emitted if the returned watermark is non-null and larger than the previous watermark.

```
@Override
public long extractTimestamp(MyEvent element, long previousElementTimestamp) {
    long timestamp = element.getCreationTime();
    currentMaxTimestamp = Math.max(timestamp, currentMaxTimestamp);
    return timestamp;
}

@Override
public Watermark getCurrentWatermark() {
    // return the watermark as current highest timestamp minus the out-of-orderness bound
    return new Watermark(currentMaxTimestamp - maxOutOfOrderness);
}
```

To generate watermarks whenever a certain event indicates that a new watermark might be generated, use *AssignerWithPunctuatedWatermarks*. For this class Flink will first call the *extractTimestamp(...)* method to assign the element a timestamp, and then immediately call the *checkAndGetNextWatermark(...)* method on that element. **Note**: It is possible to generate a watermark on every single event. However, because each watermark causes some computation downstream, an excessive number of watermarks degrades performance.

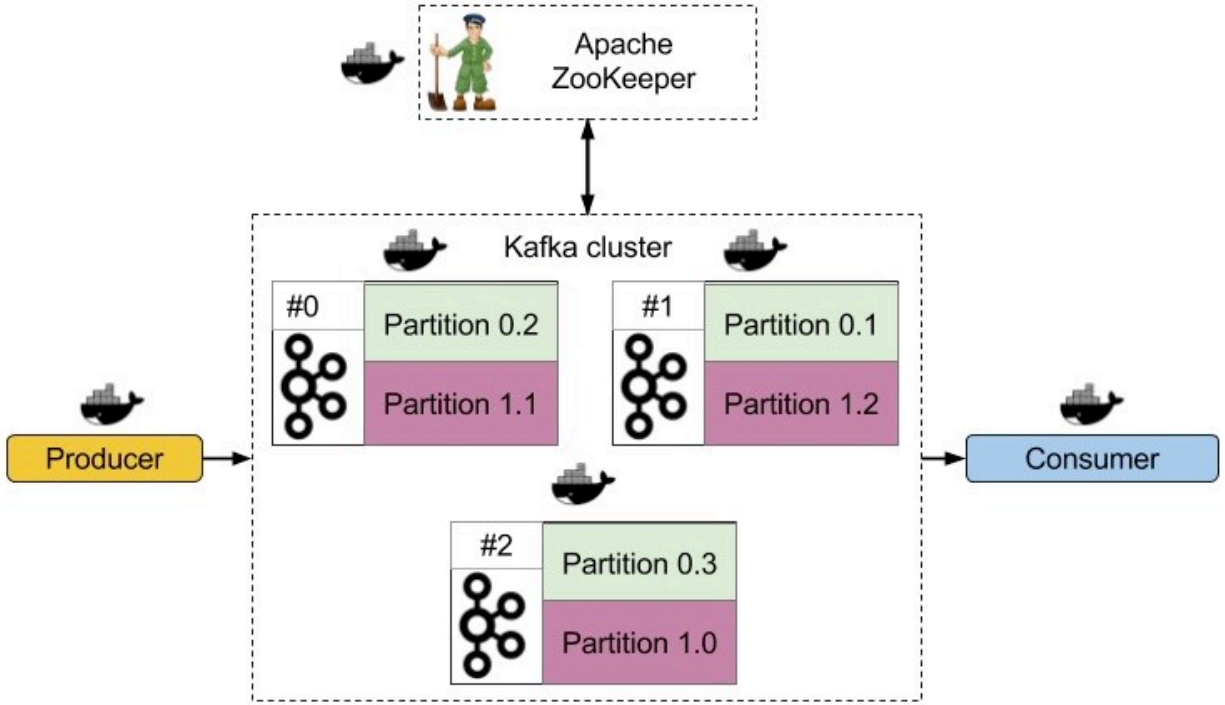
```
@Override
public long extractTimestamp(MyEvent element, long previousElementTimestamp) {
        return element.getCreationTime();
}

@Override
public Watermark checkAndGetNextWatermark(MyEvent lastElement, long extractedTimestamp) {
        return lastElement.hasWatermarkMarker() ? new Watermark(extractedTimestamp) : null;
}
```

riskfocus

# A Brief Overview of Kafka

- Apache Kafka is a distributed publish-subscribe messaging system rethought as a distributed commit log. Kafka stores messages in *topics* that are *partitioned* and *replicated* across multiple *brokers* in a cluster. *Producers* send messages to topics from which *consumers* read.
- Consumers may be grouped in a *consumer group* with multiple consumers. Each consumer in a consumer group will read messages from a unique subset of partitions in each topic they subscribe to. Each message is delivered to one consumer in the group, and all messages with the same key arrive at the same consumer.
- Kafka broker does not track which messages were read by each consumer. Kafka keeps all messages for a finite amount of time, and it is consumers' responsibility to track their location per topic, by committing *offsets*.




riskfocus

# Kafka Connector

- Apache Flink allows for a real-time stream processing technology. **The framework allows using multiple third-party systems as stream sources or sinks**.
- Flink provides an Apache Kafka connector for reading data from and writing data to Kafka topics with exactly-once guarantees.
- Requires an additional dependency in Maven

```
<dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-connector-kafka-0.10_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
</dependency>
```

riskfocus

# Apache Kafka Connector: Consumer

Flink provides an Apache Kafka connector for reading data from and writing data to Kafka topics with exactly-once guarantees.

```
Properties props = new Properties();
props.setProperty("bootstrap.servers", "kafka.dest.<namespace>.wsn.riskfocus.com:9092");
props.setProperty("group.id", "");

var consumer = new FlinkKafkaConsumer010<>(
        topic, new OrderDeserializationSchema(), props);
```

There are couple of additional settings for the Consumer:

- Start Position Configuration
- Timestamp Extraction/Watermark Emission

riskfocus

# Apache Kafka Connector: Producer

```
var producer = new FlinkKafkaProducer010<>("kafka.dest.<namespace>.wsn.riskfocus.com:9092",
        topic, new OrderSerializationSchema());
```

Three different modes of operating chosen by passing appropriate semantic parameter:

- Semantic.*NONE*: Flink will not guarantee anything. Produced records can be lost or they can be duplicated.
- Semantic.*AT_LEAST_ONCE* (default setting): This guarantees that no records will be lost (although they can be duplicated).
- Semantic.*EXACTLY_ONCE*: Kafka transactions will be used to provide exactly-once semantic. Whenever you write to Kafka using transactions, do not forget about setting desired isolation.level (read_committed or read_uncommitted - the latter one is the default value) for any application consuming records from Kafka.

riskfocus

# Deserializing Kafka To Flink

```
public class OrderDeserializationSchema implements DeserializationSchema<Order> {

    static ObjectMapper objectMapper = new ObjectMapper().registerModule(new JavaTimeModule());

    @Override
    public Order deserialize(byte[] bytes) throws IOException {
        return objectMapper.readValue(bytes, Order.class);
    }

    @Override
    public boolean isEndOfStream(Order inputMessage) {
        return false;
    }

    @Override
    public TypeInformation<Order> getProducedType() {
        return TypeInformation.of(Order.class);
    }
}
```

riskfocus

# Serializing Flink To Kafka

```
public class OrderDeserializationSchema implements DeserializationSchema<Order> {

    static ObjectMapper objectMapper = new ObjectMapper().registerModule(new JavaTimeModule());

    @Override
    public Order deserialize(byte[] bytes) throws IOException {
        return objectMapper.readValue(bytes, Order.class);
    }

    @Override
    public boolean isEndOfStream(Order inputMessage) {
        return false;
    }

    @Override
    public TypeInformation<Order> getProducedType() {
        return TypeInformation.of(Order.class);
    }
}
```

riskfocus

# Kafka and Flink Parallelism - Rebalancing Partitions

- In Kafka, each consumer from the same consumer group gets assigned one or more partitions. Note that it is not possible for two consumers to consume from the same partition. The number of Flink consumers depends on the Flink parallelism (defaults to 1).
- There are three possible cases:
  1. **Kafka partitions == Flink parallelism**: this case is ideal, since each consumer takes care of one partition. If your messages are balanced between partitions, the work will be evenly spread across Flink operators;
  2. **Kafka partitions < Flink parallelism**: some Flink instances won't receive any messages. To avoid that, you need to call rebalance on your input stream *before any operation*, which causes data to be re-partitioned:

     ```
     inputStream = env.addSource(new FlinkKafkaConsumer10("topic", new
     SimpleStringSchema(), properties)); inputStream .rebalance() .map(s -> "message"
     + s) .print();
     ```

  3. **Kafka partitions > Flink parallelism**: in this case, some instances will handle multiple partitions. Once again, you can use rebalance to spread messages evenly across workers.

riskfocus

# Hooking up the Pipeline

```
FlinkKafkaConsumer010<Order> flinkKafkaConsumer = …

DataStream<Order> orderStream = env.addSource(flinkKafkaConsumer);

DataStream<Order> outputStream = orderStream.rebalance().map(...);


FlinkKafkaProducer010<Order> flinkKafkaProducer = …

outputStream.addSink(flinkKafkaProducer);
```

riskfocus

# Quiz

1. What are the three parts of Kafka?
2. What Flink functionality enables us to use Kafka?
3. What is required in order to translate Kafka messages to Flink Objects?
4. What's the purpose of using the rebalance operator on my data stream?

riskfocus

# Lab Assignment

1. Using the supplied order-generator.jar populate kafka topic “in” with order data

   ***Note**: Must first edit application.properties and replace bootstrap.servers with your kafka cluster URL:* `kafka.dest.rlewis.wsn.riskfocus.com:9092`
2. Write a Flink application to consume this data
3. Add timestamps and watermarks to this data
4. Group the data by account
5. Write the data to Kafka topic “demo-output”

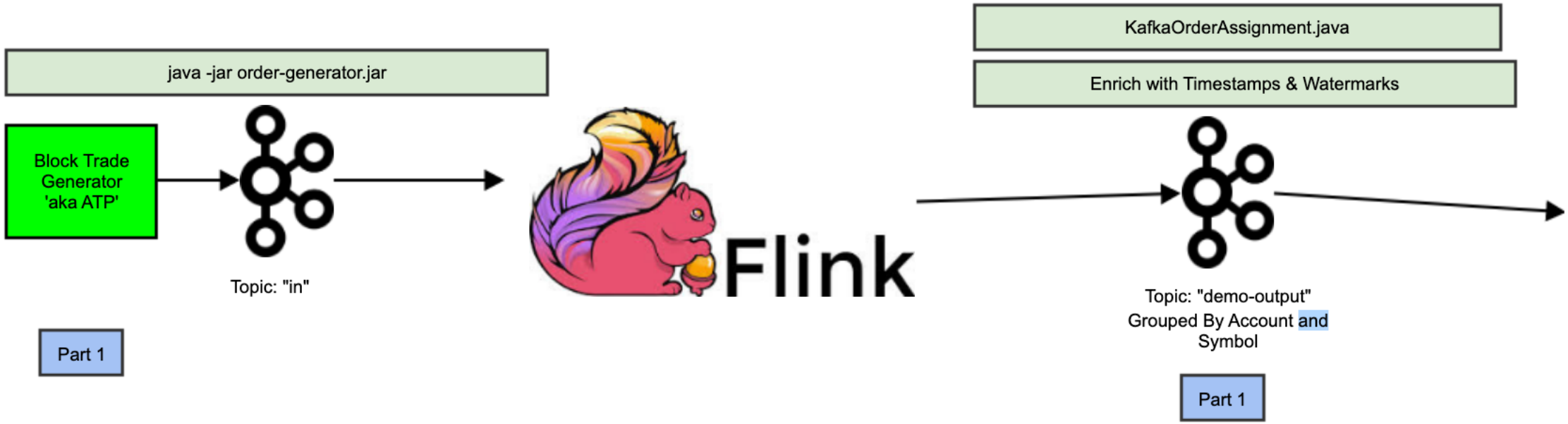
riskfocus

# Lab Assignment – Extra Credit

- Create your own Kafka Topics Jenkins/Cadmium
- Rewrite the lab assignment to use these new topics
- Create a local cluster of Kafta and Flink

riskfocus

# The Pipeline



# Risk Focus – Testimonials

## Ensuring security of Infrastructure and data is foundational

*"[Risk Focus] were committed to our success and were **obsessed with automation, security and the total cost of ownership** of the deliverable."*

Eram Schlegel
CTO, CFRA

## Decades of experience building complex solutions for Capital Mark

*"The key differentiator that Risk Focus brought to the table is the **unique combination of deep domain knowledge** coupled with technical expertise and delivery excellence."*

Brian Lynch,
CEO, RegTek.Solutions
a Bloomberg Company

## An Engagement model that is customer obsessed and outcome based

*"The Risk Focus team created **a strong partnership with my team to deliver the project**. "*

Maja Schwob
CIO, Data Services
Deutsche Börse

## An Agile delivery process to enable our clients to move quickly

*"[Risk Focus] allowed us to **shorten what would have been a two-year project and deliver it in one**."*

Graeme Peacock
CIO Technology Enterprise
Services, TD Bank

riskfocus