Introduction to Spring Cloud

Cloud Streams

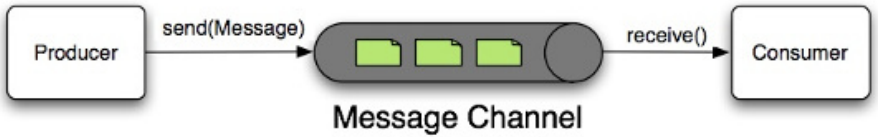
LXFT LISTED NYSE

LUXOFT

Spring Cloud Stream

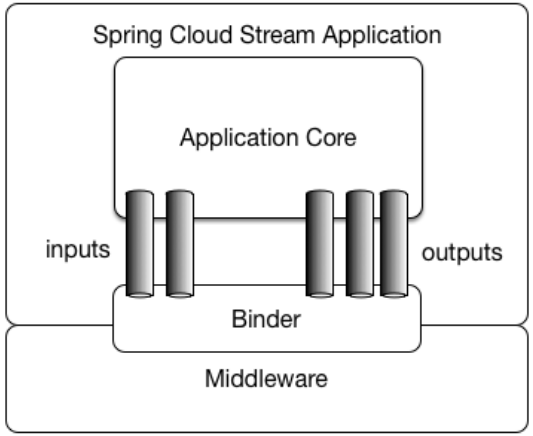
Spring Cloud Stream

- Spring Cloud Stream is a framework for building message-driven microservice applications.
  - Spring Cloud Stream builds upon Spring Boot to create standalone, production-grade Spring applications, and uses Spring Integration to provide connectivity to message brokers.



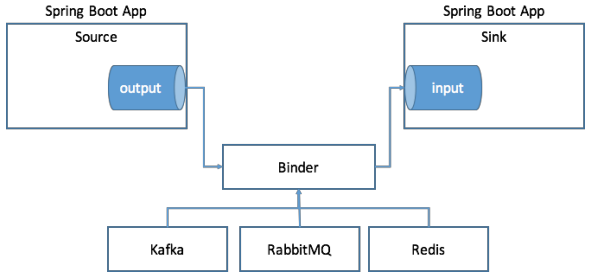
Spring Cloud Stream

- Spring Cloud Stream provides opinionated configuration of middleware from several vendors, introducing the concepts of
  - persistent publish-subscribe semantics,
  - consumer groups,
  - and partitions.



Spring Cloud Stream

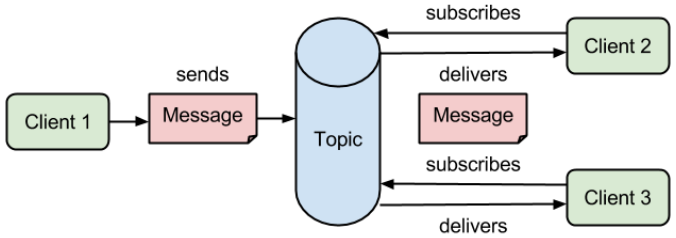
- Spring Cloud Stream simplifies message-stream development down to two simple annotations.
  - By adding @EnableBinding to your main application, you get immediate connectivity to a message broker.
  - By adding @StreamListener to a method, you will receive events for stream processing.



Publish-Subscribe

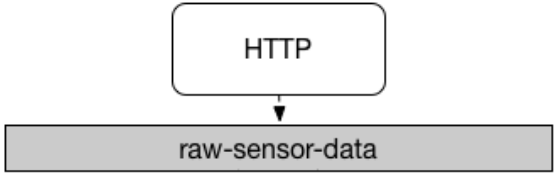
Publish-Subscribe

- Communication between applications follows a publish-subscribe model, where data is broadcast through shared topics.
  - The publish-subscribe communication model reduces the complexity of both the producer and the consumer.
  - It allows new applications to be added to the topology without disruption of the existing flow.



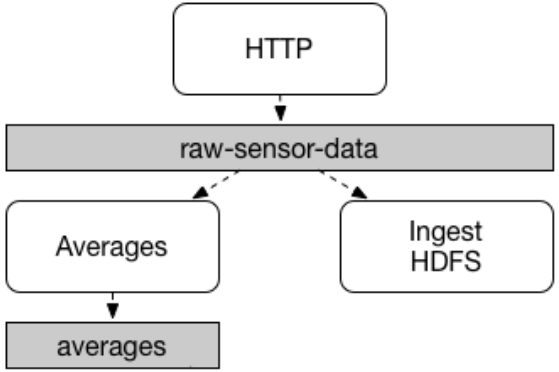
Publish-Subscribe

- Let’s review an example application that analyzes data streamed from some sensors.
  - Sensors stream data to HTTP endpoint which is the Spring Cloud application is running.
  - Data reported by sensors to an HTTP endpoint is sent to a common destination named raw-sensor-data.



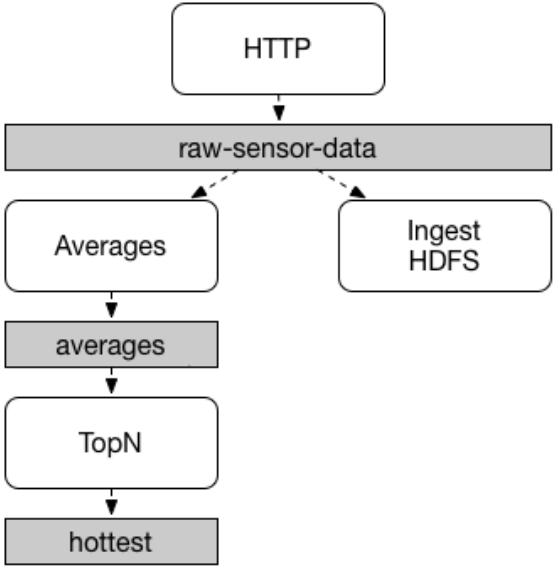
Publish-Subscribe

- In order to process the data, both applications declare the topic as their input at runtime.
  - Data is independently processed from the destination by both, a microservice that computes time-windowed averages and by another microservice that ingests the raw data into HDFS.



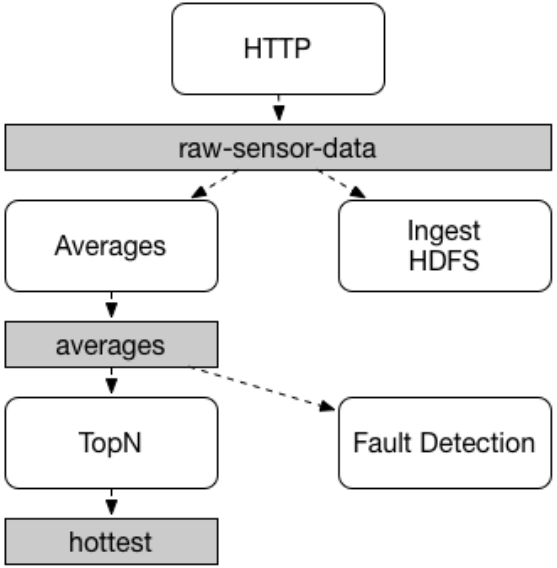
Publish-Subscribe

- New applications can be added to the topology without disruption of the existing flow.
  - For example, downstream from the average-calculating application, you can add an application that calculates the highest temperature values for display and monitoring.



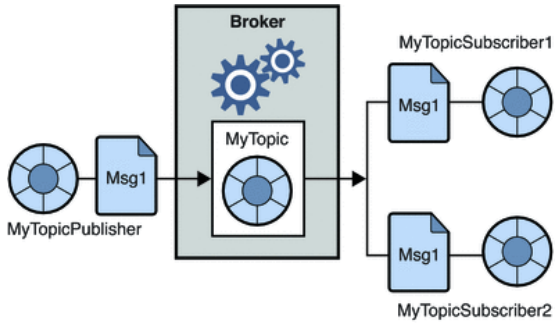
Publish-Subscribe

- Doing all communication through shared topics rather than point-to-point queues reduces coupling between microservices.
  - For example, you can add another application that interprets the same flow of averages for fault detection.



Publish-Subscribe

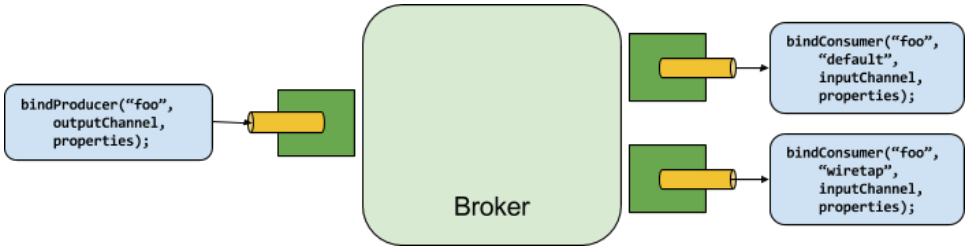
- By using native middleware support, Spring Cloud Stream also simplifies use of the publish-subscribe model across different platforms.
  - While the concept of publish-subscribe messaging is not new, Spring Cloud Stream takes the extra step of making it an opinionated choice for its application model.



Binders

Binders

- Spring Cloud Stream provides a Binder abstraction for use in connecting to physical destinations at the external middleware.
  - A message is sent to a channel by producer and is consumed from a channel by consumer.
  - The channel can be bound to an external message broker via a Binder implementation for that broker.

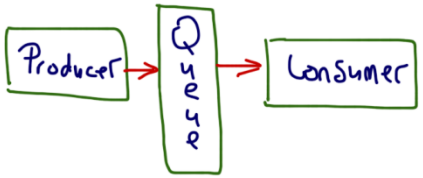


Binders

- A producer is any component that sends messages to a channel.
  - The producer's channel can be bound to an external message broker via a Binder implementation for that broker.
  - Required parameters are
    - The name of the destination within the broker
    - The local channel instance to which the producer will send messages
    - The properties to be used within the adapter that is created for that channel.
      - Such as a partition key expression

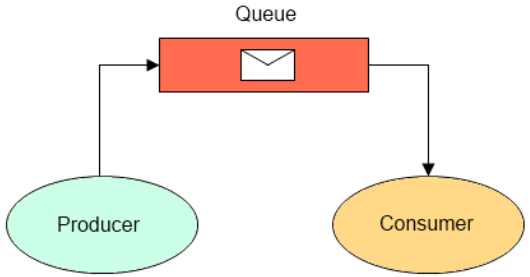
Binders

- A consumer is any component that receives messages from a channel.
  - As with a producer, the consumer's channel can be bound to an external message broker.
  - Required parameters are
    - The destination name
    - The name of a logical group of consumers.



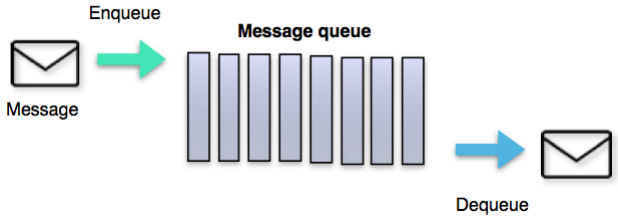
Binders

- Each group that is represented by consumer bindings for a given destination receives a copy of each message that a producer sends to that destination
  - i.e., publish-subscribe semantics.



Binders

- If there are multiple consumer instances bound using the same group name, then messages will be load-balanced across those consumer instances so that each message sent by a producer is consumed by only a single consumer instance within each group
  - i.e. queueing semantics.



Binders

- Spring Cloud Stream provides Binder implementations for Kafka and Rabbit MQ.
  - Spring Cloud Stream also includes a TestSupportBinder, which leaves a channel unmodified so that tests can interact with channels directly and reliably assert on what is received.
- The Binder API is extensible and allows you to write your own Binder implementations.

Binders

- By default, Spring Cloud Stream relies on Spring Boot's auto-configuration to configure the binding process.
  - If a single Binder implementation is found on the classpath, Spring Cloud Stream will use it automatically.
  - For example, a Spring Cloud Stream project that aims to bind only to RabbitMQ can simply add the following dependency:

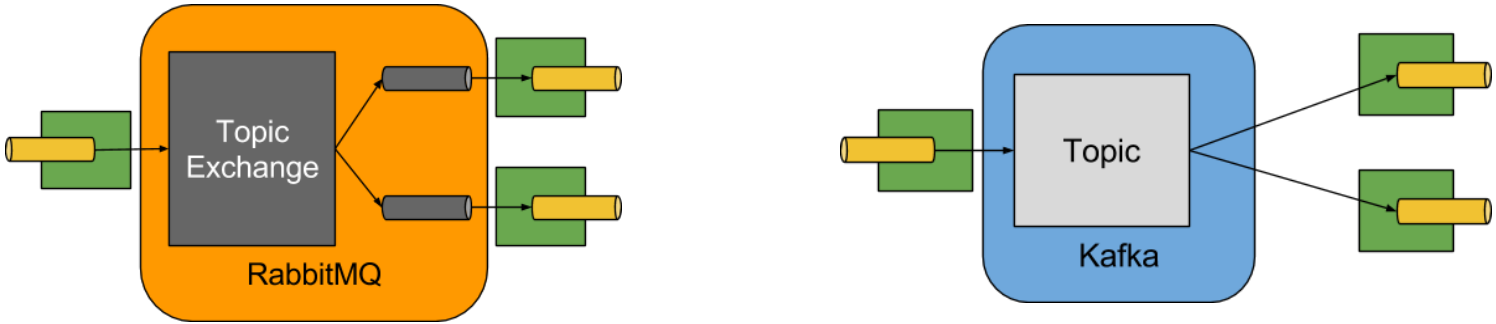
```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

Binders

- Each Binder implementation typically connects to one type of messaging system.
  - You can easily use different types of middleware with the same code including a different binder at build time.
- Each binder comes with a dedicated set of configurable properties as well as makes use of predefined common configuration properties.

Binders

- For complex use cases, you can also package multiple binders with your application and have it choose the binder, and even whether to use different binders for different channels, at runtime.



Binders

- When multiple binders are present on the classpath, the application must indicate which binder is to be used for each channel binding.
- Each binder configuration contains a key that represents an identifying name for the binder implementation.
- For instance,
  - rabbit – RabbitMQ
  - kafka – Kafka

Binders

- Binder selection can either be performed globally, using the defaultBinder property or individually, by configuring the binder on each channel binding.
  - e.g., spring.cloud.stream.defaultBinder: rabbit
  - A processor application that has channels with the names "input" which reads from Kafka and "output" which writes to RabbitMQ can specify the following configuration:
    - spring.cloud.stream.bindings.**input**.binder: **kafka**
    - spring.cloud.stream.bindings.**output**.binder: **rabbit**

Binders

- If your application should connect to more than one broker of the same type, you can specify multiple binder configurations, each with different environment settings.
- Turning on explicit binder configuration will disable the default binder configuration process altogether.
  - If you do this, all binders in use must be included in the configuration.

Binders

- To connect to two RabbitMQ broker instances:

```
spring.cloud.stream:
  bindings:
    input:
      destination: dest1
      binder: rabbit1
    output:
      destination: dest2
      binder: rabbit2
  binders:
    rabbit1:
      type: rabbit
      environment.spring.rabbitmq.host: host1
    rabbit2:
      type: rabbit
      environment.spring.rabbitmq.host: host2
```

Binders

- The spring.cloud.stream.binders.* properties are available when creating custom binder configurations:
  - type – typically references one of the binders found on the classpath.
  - inheritEnvironment – inherit the environment of the application (default true).
  - environment – root for a set of properties that can be used to customize the environment of the binder (default empty).
    - When this is configured, the context in which the binder is being created is not a child of the application context. This allows for complete separation between the binder components and the application components.

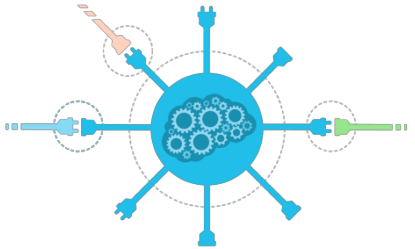
Binders

- The spring.cloud.stream.binders.* properties are available when creating custom binder configurations (continuation):
  - defaultCandidate – indicates if the binder configuration is a candidate for being considered a default binder, or can be used only when explicitly referenced.
    - This allows adding binder configurations without interfering with the default processing(default true).

API

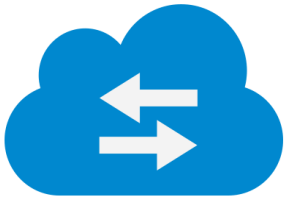
API

- Spring Cloud Stream is enabled by applying the @EnableBinding annotation to application configuration class.
  - The @EnableBinding annotation can take one or more interface classes as parameters.
  - These interface classes should contain methods which represent bindable components, typically message channels.



API

- Supported bindable component interfaces are
  - MessageChannel (the Spring Messaging)
  - SubscribableChannel (extension of MessageChannel)
  - PollableChannel (extension of MessageChannel)



API

- A Spring Cloud Stream application can have an arbitrary number of input and output channels defined in an interface as @Input and @Output methods:

```
public interface Restaurant {
    @Input
    SubscribableChannel orders();

    @Output
    MessageChannel hotDrinks();

    @Output
    MessageChannel coldDrinks();
}
```

  - Using this interface as a parameter to @EnableBinding will trigger the creation of three bound channels named "orders", "hotDrinks", and "coldDrinks", respectively.

API

- The @Input annotation identifies an input channel, through which received messages enter the application.
- The @Output annotation identifies an output channel, through which published messages leave the application.
  - The @Input and @Output annotations can take a channel name as a parameter.
  - If a name is not provided, the name of the annotated method will be used.

API

- Using the @Input and @Output annotations, you can specify a customized channel name for the channel, as shown in the following example:

```
public interface Sink {
    String INPUT = "input";

    @Input(Sink.INPUT)
    SubscribableChannel input();
}
```

- In this example, the created bound channel will be named "input".

API

- For easy addressing of the most common use cases, which involve either an input channel, an output channel, or both, Spring Cloud Stream provides three predefined interfaces out of the box:
  - Source – can be used for an application which has a single outbound channel.
  - Sink – can be used for an application which has a single inbound channel.
  - Processor – can be used for an application which has both an inbound channel and an outbound channel.

API

- Spring Cloud Stream provides no special handling for any of these interfaces, they are only provided out of the box.

```
public interface Source {
    String OUTPUT = "output";

    @Output(Source.OUTPUT)
    MessageChannel output();
}
```

```
public interface Sink {
    String INPUT = "input";

    @Input(Sink.INPUT)
    SubscribableChannel input();
}
```

```
public interface Processor extends Source, Sink {

}
```

API

- For each bound interface, Spring Cloud Stream will generate a bean that implements the interface. Invoking a @Input annotated or @Output annotated method of one of these beans will return the relevant bound channel.

```
@Component
public class MessagePublisher {
    @Autowired
    private Source source;

    public void publish(String s) {
        source.output().send(MessageBuilder.withPayload(s).build());
    }
}
```

API

- Bound channels can be also injected directly:

```
@Component
public class MessagePublisher {
    @Autowired
    private MessageChannel output;

    public void publish(String s) {
        output.send(MessageBuilder.withPayload(s).build());
    }
}
```

API

- If the name of the channel is customized on the declaring annotation, that name should be used instead of the method name. Given the following declaration:

```
@Component
public class MessagePublisher {
    @Autowired
    @Qualifier("customOutput")
    private MessageChannel output;

    public void publish(String s) {
        output.send(MessageBuilder.withPayload(s).build());
    }
}
```

```
public interface CustomOutput {
    @Input("customOutput")
    MessageChannel output();
}
```

API

- Spring Cloud Stream provides a simple model for handling inbound messages with help of the @StreamListener annotation:

```
@Component
public class MessageConsumer {
    @StreamListener(Sink.INPUT)
    public void consume(String s) {
        System.out.println(s);
    }
}
```

API

- For methods which return data, you must use the @SendTo annotation to specify the output binding destination for data returned by the method:

```
@Component
public class MessageProcessor {
    @StreamListener(Sink.INPUT)
    @SendTo(Source.OUTPUT)
    public String process(String s) {
        return s.toUpperCase();
    }
}
```

Lab 11
Cloud Streams

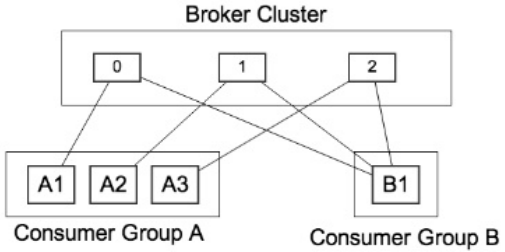
Lab 11 – Cloud Streams

- In this lab we will implement publish-subscribe model as follows:
  - Publisher will publish words to channel.
  - Subscriber will count unique words received from channel.

Consumer Groups

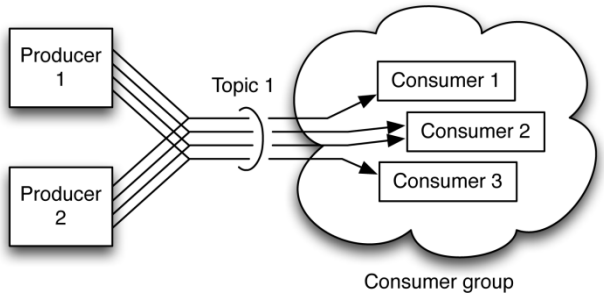
Consumer Groups

- While the publish-subscribe model makes it easy to connect applications through shared topics, the ability to scale up by creating multiple instances of a given application is equally important.
  - When doing this, different instances of an application are placed in a competing consumer relationship, where only one of the instances is expected to handle a given message.



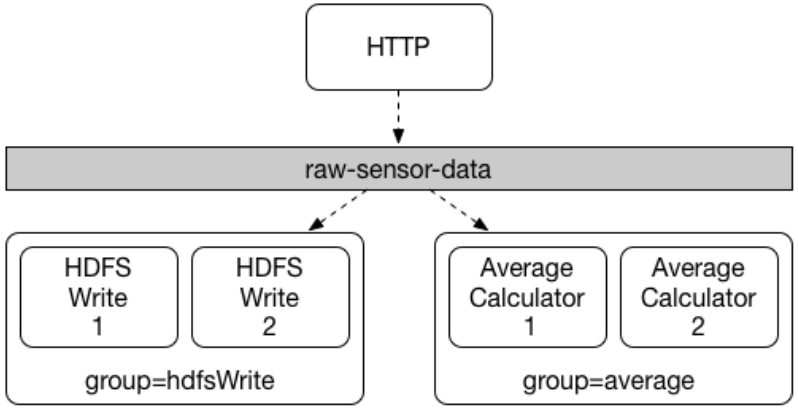
Consumer Groups

- Spring Cloud Stream models this behavior through the concept of a consumer group.
  - Each consumer binding can use the following property to specify a group name:
  - spring.cloud.stream.bindings.channelName.group



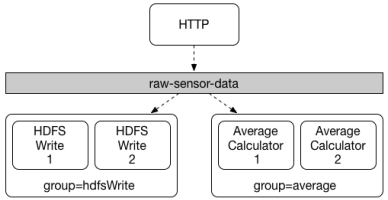
Consumer Groups

- For the consumers shown in the following figure below, group property would be set as:
  - spring.cloud.stream.bindings.channelName.group: hdfsWrite
  - spring.cloud.stream.bindings.channelName.group: average



Consumer Groups

- All groups which subscribe to a given destination receive a copy of published data, but only one member of each group receives a given message from that destination.
  - By default, when a group is not specified, the application is assigned to an anonymous and independent single-member consumer group that is in a publish-subscribe relationship with all other consumer groups.



Durability

Durability

- Consistent with the opinionated application model of Spring Cloud Stream, consumer group subscriptions are durable.
  - That is, a binder implementation ensures that group subscriptions are persistent, and once at least one subscription for a group has been created, the group will receive messages, even if they are sent while all applications in the group are stopped.
  - For some binder implementations (e.g., RabbitMQ), it is possible to have non-durable group subscriptions.
  - Anonymous subscriptions are non-durable by nature.

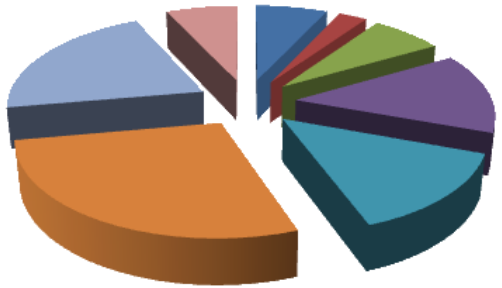
Durability

- In general, it is preferable to always specify a consumer group when binding an application to a given destination.
  - To avoid issues when scaling out.
- When scaling out a Spring Cloud Stream application, you must specify a consumer group for each of its input bindings.
  - This prevents the application's instances from receiving duplicate messages, unless that behavior is desired, which is unusual.

Partitioning

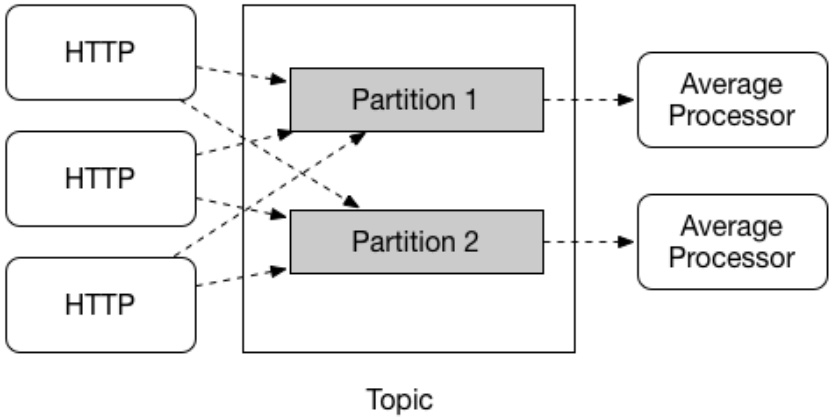
Partitioning

- Spring Cloud Stream provides support for partitioning data between multiple instances of a given application.
  - In a partitioned scenario, the physical communication medium (e.g., the broker topic) is viewed as being structured into multiple partitions.



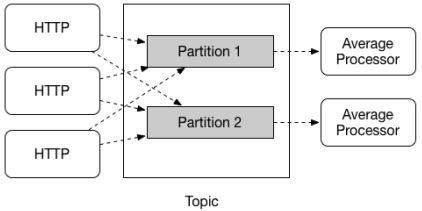
Partitioning

- One or more producer application instances send data to multiple consumer application instances and ensure that data identified by common characteristics are processed by the same consumer instance.



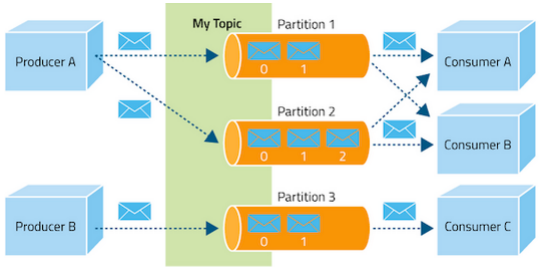
Partitioning

- Spring Cloud Stream provides a common abstraction for implementing partitioned processing use cases in a uniform fashion.
  - Partitioning can thus be used whether the broker itself is naturally partitioned (e.g., Kafka) or not (e.g., RabbitMQ).
  - To set up a partitioned processing scenario, you must configure both the data-producing and the data-consuming ends.



Partitioning

- Partitioning is a critical concept in stateful processing.
  - It is critical, for either performance or consistency reasons, to ensure that all related data is processed together.
  - For example, in the time-windowed average calculation example, it is important that all measurements from any given sensor are processed by the same application instance.



Configuration Properties

Configuration Properties

- Common configurations properties available include:
  - Spring Cloud Stream properties
  - Binding properties
    - Properties for use of Spring Cloud Stream
    - Consumer properties
    - Producer properties

Spring Cloud Stream Properties

Spring Cloud Stream Properties

- spring.cloud.stream.instanceCount: 1
  - The number of deployed instances of an application. Must be set for partitioning and if using Kafka.
- spring.cloud.stream.instanceIndex: 0
  - The instance index of the application: a number from 0 to instanceCount-1.
  - Used for partitioning and with Kafka.
- spring.cloud.stream.defaultBinder: empty
  - The default binder to use, if multiple binders are configured.

Spring Cloud Stream Properties

- spring.cloud.stream.dynamicDestinations: empty
  - A list of destinations that can be bound dynamically.
    - For example, in a dynamic routing scenario.
  - If set, only listed destinations can be bound.
  - If not set, allows any destination to be bound.

Binding Properties

Binding Properties

- Binding properties are supplied using the format
  - spring.cloud.stream.bindings.channelName.*
- The channelName represents the name of the channel being configured (e.g., “output” for a Source).
- Binding properties are available for both input and output bindings and must be prefixed with
  - spring.cloud.stream.bindings.channelName.

Properties for Use of Spring Cloud Stream

Properties for Use of Spring Cloud Stream

- destination: empty
  - The target destination of a channel on the bound middleware
    - e.g. the RabbitMQ exchange or Kafka topic
  - If the channel is bound as a consumer, it could be bound to multiple destinations and the destination names can be specified as comma separated String values.
  - If not set, the channel name is used instead.

Properties for Use of Spring Cloud Stream

- group: null
  - The consumer group of the channel.
  - Applies only to inbound bindings.
  - If not set, indicates an anonymous consumer.
- binder: null
  - The binder used by this binding.
  - If not set, the default binder will be used, if one exists.

Consumer Properties

Consumer Properties

- consumer.concurrency: 1
  - The concurrency of the inbound consumer.
- consumer.partitioned: false
  - Whether the consumer receives data from a partitioned producer.
- consumer.maxAttempts: 3
  - The number of attempts of re-processing an inbound message.

Consumer Properties

- consumer.headerMode: embeddedHeaders
  - When set to raw, disables header parsing on input.
  - Effective only for messaging middleware that does not support message headers natively and requires header embedding.
  - Useful when inbound data is coming from outside Spring Cloud Stream applications.

Consumer Properties

- consumer.instanceIndex: -1
  - When set to a value greater than equal to zero, allows customizing the instance index of this consumer, if different from spring.cloud.stream.instanceIndex.
  - When set to a negative value, it will default to spring.cloud.stream.instanceIndex.
- consumer.instanceCount: -1
  - When set to a value greater than equal to zero, allows customizing the instance count of this consumer, if different from spring.cloud.stream.instanceCount.
  - When set to a negative value, it will default to spring.cloud.stream.instanceCount.

Producer Properties

Producer Properties

- producer.partitionKeyExpression: null
  - A SpEL expression that determines how to partition outbound data.
    - E.g. “payload.id”.
  - If set, or if partitionKeyExtractorClass is set, outbound data on this channel will be partitioned, and partitionCount must be set to a value greater than 1 to be effective.
  - The two options are mutually exclusive.

Producer Properties

- producer.partitionKeyExtractorClass: null
  - A PartitionKeyExtractorStrategy implementation.
  - If set, or if partitionKeyExpression is set, outbound data on this channel will be partitioned, and partitionCount must be set to a value greater than 1 to be effective.
  - The two options are mutually exclusive.

Producer Properties

- producer.partitionSelectorClass: null
  - A PartitionSelectorStrategy implementation.
  - Mutually exclusive with partitionSelectorExpression.
  - If neither is set, the partition will be selected as the hashCode(key) % partitionCount, where key is computed via either partitionKeyExpression or partitionKeyExtractorClass.

Producer Properties

- producer.partitionSelectorExpression: null
  - A SpEL expression for customizing partition selection.
  - Mutually exclusive with partitionSelectorClass.
  - If neither is set, the partition will be selected as the hashCode(key) % partitionCount, where key is computed via either partitionKeyExpression or partitionKeyExtractorClass.

Producer Properties

- producer.partitionCount: 1
  - The number of target partitions for the data, if partitioning is enabled.
  - Must be set to a value greater than 1 if the producer is partitioned.
  - On Kafka, interpreted as a hint; the larger of this and the partition count of the target topic is used instead.

Producer Properties

- producer.requiredGroups
  - A comma-separated list of groups to which the producer must ensure message delivery even if they start after it has been created.
  - E.g. by pre-creating durable queues in RabbitMQ.

Producer Properties

- producer.headerMode: embeddedHeaders
  - When set to raw, disables header embedding on output.
  - Effective only for messaging middleware that does not support message headers natively and requires header embedding.
  - Useful when producing data for non-Spring Cloud Stream applications.

Stream Aggregation

Stream Aggregation

- Spring Cloud Stream provides support for aggregating multiple applications together, connecting their input and output channels directly and avoiding the additional cost of exchanging messages via a broker.

Stream Aggregation

- Aggregation is possible using Sources, Sinks and Processors.
  - They can be aggregated together by creating a sequence of interconnected applications, in which the output channel of an element in the sequence is connected to the input channel of the next element, if it exists.
- A sequence can start with either a Source or a Processor and it can contain an arbitrary number of Processors and must end with either a Processor or a Sink.

Stream Aggregation

- Depending on the nature of the starting and ending element, the sequence may have one or more bindable channels, as follows:
  - if the sequence starts with a Source and ends with a Sink, all communication between the applications is direct and no channels will be bound.
  - if the sequence starts with a Processor, then its input channel will become the input channel of the aggregate and will be bound accordingly.
  - if the sequence ends with a Processor, then its output channel will become the output channel of the aggregate and will be bound accordingly.

Stream Aggregation

- Aggregation is performed using the AggregateApplicationBuilder utility class, as in the following example.
  - Let's consider a project in which we have Source, Processor and a Sink, which may be defined in the project, or may be contained in one of the project's dependencies:

```
@SpringBootApplication
public class SampleAggregateApplication {
    public static void main(String[] args) {
        new AggregateApplicationBuilder()
                .from(SourceApplication.class).args("--fixedDelay=5000")
                .via(ProcessorApplication.class)
                .to(SinkApplication.class).args("--debug=true").run(args);
    }
}
```

Stream Aggregation

- The starting component of the sequence is provided as argument to the from() method.
- The ending component of the sequence is provided as argument to the to() method.
- Intermediate processors are provided as argument to the via() method.

Stream Aggregation

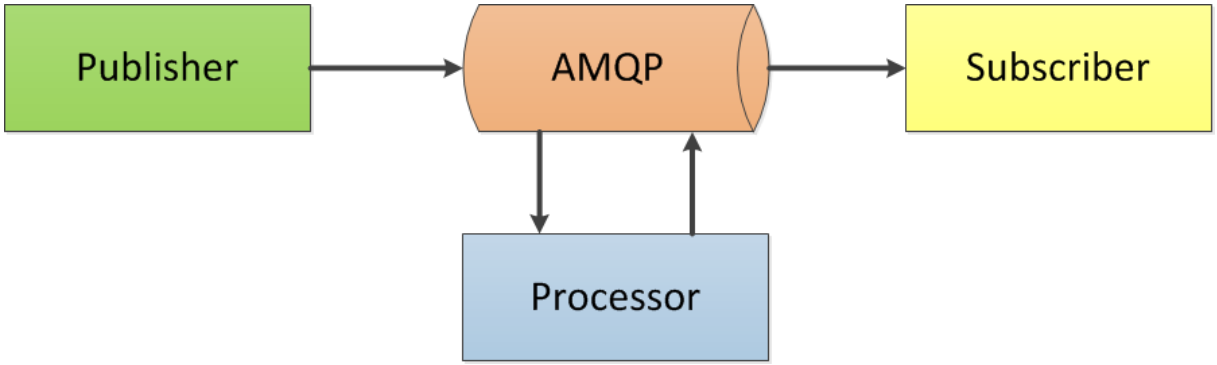
- Multiple processors of the same type can be chained together (e.g. for pipelining transformations with different configurations).
- For each component, the builder can provide runtime arguments for Spring Boot configuration.

```
@SpringBootApplication
public class SampleAggregateApplication {
    public static void main(String[] args) {
        new AggregateApplicationBuilder()
                .from(SourceApplication.class).args("--fixedDelay=5000")
                .via(ProcessorApplication.class)
                .to(SinkApplication.class).args("--debug=true").run(args);
    }
}
```

Lab 12
Advanced Streams

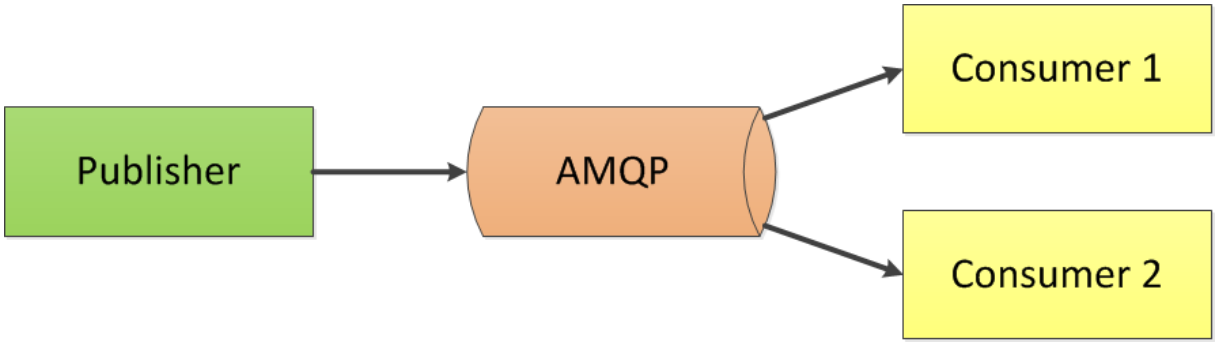
Lab 12 – Advanced Streams

- In this lab we will
  - Introduce a word processor that will process all words submitted by Publisher before they are received by Subscriber



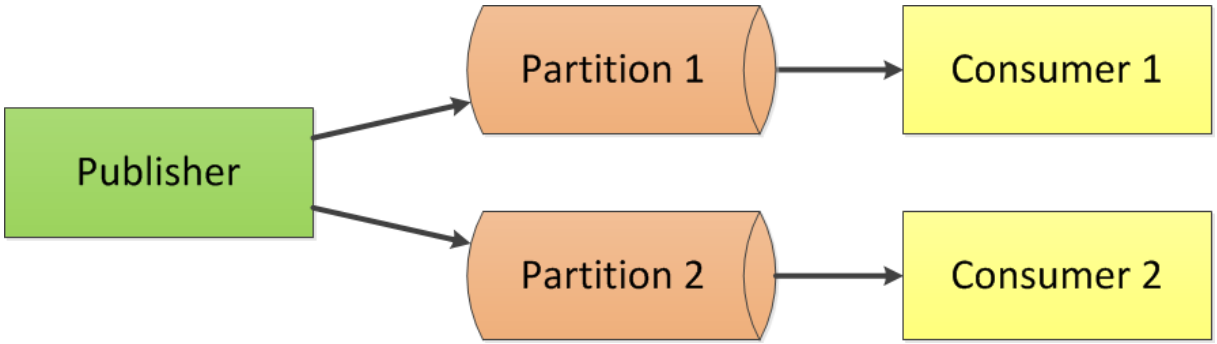
Lab 12 – Advanced Streams

- In this lab we will
  - Introduce consumer groups for load balancing



Lab 12 – Advanced Streams

- In this lab we will
  - Introduce partitioning for parallel processing



Thank You!

LXFT
LISTED
NYSE

LUXOFT