JOHN BRYCE
Leading in IT Education
a matrix company

# SPRING CLOUD SRTEAMING

# Spring Cloud Streaming

JOHN BRYCE
Leading in IT Education
a matrix company

## Introduction

- Abstracts and simplifies building an event-driven messaging systems
- Supports both Kafka & RabbitMO
- Provides default boot configuration that can be specified via properties
- Built on SpringBoot & Spring Integration
- Allows creating event-driven Microservices

# Spring Cloud Streaming



## Introduction

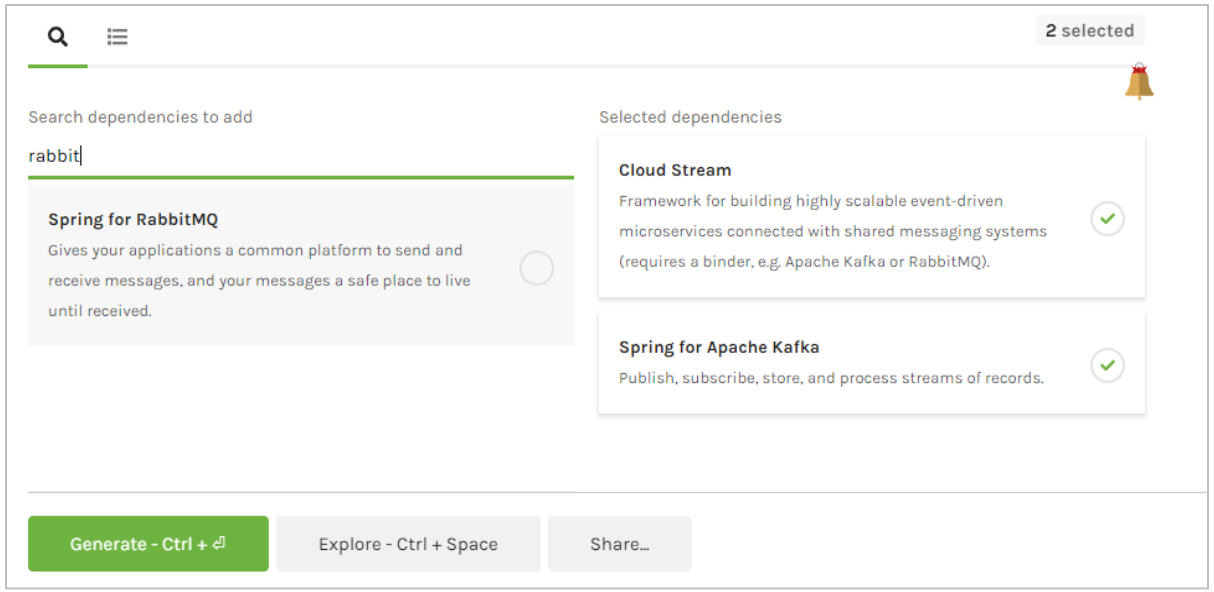
- Provides the following event/messaging oriented implementations:
  - **Producers** - uses SOURCE to publish messages to distribution queues
  - **Suppliers** - automated Producers which generates events on a periodic manner
  - **Consumers** - uses SINK to consumes messages from queues. Are event driven
  - **Processors** - triggered via event, processes its content and publishes the result

# Spring Cloud Streaming

JOHN BRYCE
Leading in IT Education
a matrix company

## Creating Project

- Spring Initializer
  - Spring Cloud Stream must have Kafka or RabbitMQ binder



## Creating Project

- Maven Dependencies

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

<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-stream-test-support</artifactId>
   <scope>test</scope>
</dependency>
```

## Producers

- Producers initialize messages and distributes it to queue
- First, we need to configure output SINK:

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        output:
          binder: kafka
          destination: test
```

- application.yml configuration is used for default output channel
- Alternative configuration can be set via annotations

## Producers

- Producers uses Source bean which is initialized with default output configuration
- Allowing different components to use Source is made simple via custom Producer:
  - @EnableBindings(Source.class) – creates the binding configuration with the broker. In this case, we have only output settings which are applied to the Source bean

```
@EnableBinding(Source.class)
public class Producer {
   private Source mySource;

   public Producer(Source mySource) {
       this.mySource = mySource;
   }

   public Source getMysource() {
       return mySource;
   }

    public void setMysource(Source mysource) {
        this.mySource = mySource;
    }
}
```

# Producers

- Using custom Producer to send messages:

```
@Autowired
private Producer producer;

public String send(String data) {
   producer.getMysource()
   .output()
   .send(MessageBuilder.withPayload(data)
   .build());
   return "Success";
}
```

  - Message Headers - Producers may put custom header values which can be filtered by consumer using expressions

  - Any custom message must be a valid JavaBean class

```
public String send(MyMessage data) {
   producer.getMysource()
   .output()
   .send(MessageBuilder.withPayload(data)
   .setHeader("type", "custom")
   .build());
   return "Success";
}
```

## Consumers

- Consumers receives messages from queues
- May be members of consumer-groups for sharing event distribution
- First, we need to configure input SOURCE:

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        input:
          binder: kafka
          destination: test
```

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        input:
          binder: kafka
          group: cons-group-1
          destination: test
```

- application.yml configuration is used for default input channel

## Consumers

- Producers uses Source bean which is initialized with default output configuration
- Allowing different components to use Source is made simple via custom Producer:

```
@EnableBinding(Sink.class)
public class Producer {

   @StreamListener(target=Sink.INPUT)
   public void handle(String data) {
       //handle data
   }
   @StreamListener(target=Sink.INPUT ,condition= ”headers[‘type’]==‘custom’”)
   public void handle(MyMessage data) {
      //handle data
   }
}
```

- **@StreamListener** – sets the target endpoints and allows filtering via message headers

## Spring Cloud Streaming Functional Support

- Spring Cloud Streaming simplifies coding with Java 8 Functional interfaces support
  - **Suppliers** – generates events on a periodic manner & by invoking get() method
    - Requires outbound configuration
  - **Consumers** – accepts T as an event from a queue
    - Requires inbound configuration
  - **Functions** – acts as a processors which consumes T and publishes R
    - Requires both outbound & inbound configuration

## Spring Cloud Streaming Functional Support

- Input & output configuration
  - Specifying queue names are done with relation to the function name
  - Naming convention for destination binding is:
    - Inbound: spring.cloud.stream.bindings.<function-name>-in-<index>
    - Outbound: spring.cloud.stream.bindings.<function-name>-out-<index>
  - This fact makes it easy to define multiple event-oriented functions mapped to different destinations
  - Index starts from 0 and is useful when more than single destination is used by a function
    - This is relevant for Reactive Consumers only

## Spring Cloud Streaming Functional Support

- Examples:

```
public Supplier<String> handleA() {...}
```

```
public Consumer<String> handleB() {...}
```

```
public Function<String,String> handleC() {...}
```

application.yml

```
...
handleA-out-0:
  destination: test
```

```
...
handleB-in-0:
  destination: test
```

```
...
handleC-in-0:
  destination: test
handleC-out-0:
  destination: test2
```

# Spring Cloud Streaming Functional Support

- Supplier

```
@SpringBootApplication
public class SpringCloudStreamSupplierApplication {

   public static void main(String[] args) {
      ApplicationContext ctx=SpringApplication.run(SpringCloudStreamSupplierApplication.class, args);
   }

   @Bean
   public Supplier<String> date() {
     return ()-> (new Date()).toString();
   }
}
```

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        date-out-0:
          destination: test
      poller:
        fixed-rate: 2000
```

- Poller configuration sets the delay between each sent message default 1000 millis

## Spring Cloud Streaming Functional Support

- Supplier
  - Suppliers generates messages automatically
  - Additionally, it can be triggered programmatically when calling Supplier.get() method
  - If a fixed-rate is set on the poller – it will delay get() publishing accordingly

```
@Autowired
private Supplier supplier;

public void send(){
    supplier.get();
}
```

## Spring Cloud Streaming Functional Support

- Consumer

```
@SpringBootApplication
public class SpringCloudStreamConsumerApplication {

   public static void main(String[] args) {
      ApplicationContext ctx=SpringApplication.run(SpringCloudStreamConsumerApplication.class, args);
   }

   @Bean
   public Consumer<String> handle() {
      return System.out::println;
   }
}
```

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        handle-in-0:
          destination: test
```

# Spring Cloud Streaming Functional Support

- Function (Processor)

```
@SpringBootApplication
public class SpringCloudStreamProcessApplication {

   public static void main(String[] args) {
      ApplicationContext ctx=SpringApplication.run(SpringCloudStreamProcessApplication.class, args);
   }

   @Bean
   public Function<String, String> uppercase() {
      return value -> { System.out.println(value);
                        return value.toUpperCase();
                      };
   }
}
```

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        uppercase-in-0:
          destination: test
        uppercase-out-0:
          destination: test2
```

## Spring Cloud Streaming Functional Support

- Function Flow
  - Spring supports functional programming model by defining function flow
    - Define a ‘function chain’
    - First function input is consumed from a queue
    - Last function output – published to a queue
    - Basically, function code is composed into a single function
  - When using multiple functions with no dedicated destinations it must be chained – otherwise a conflict is expressed

# Spring Cloud Streaming

JOHN BRYCE
Leading in IT Education
a matrix company

## Spring Cloud Streaming Functional Support

- Function Flow

```
@Bean
public Function<String, String> uppercase() {
   return value -> { System.out.println(value);
                     return value.toUpperCase();
                   };
}
@Bean
public Function<String, String> addQuotes() {
  return s ->"\"" + s +"\"";
}
```

application.yml

```
spring:
  cloud:
    function:
        definition: uppercase|addQuotes
    stream:
      bindings:
        uppercase-in-0:
           destination: test
        uppercase-out-0:
           destination: test2
        addQoutes-in-0:
           destination: test
        addQoutes-out-0:
           destination: test2
```

- spring.cloud.function.definition sets the function order by name, separated with pipe (|)

## Handling Errors

- By default, failed messages are dropped
- Spring allows to configure DLQ (Dead Letter Queue)
- Failed messages are stored in DLQ if enabled
- Spring properties relevant for failed consumption:
  - consumer.enableDlq – Boolean (false by default)
  - consumer.dlqName
  - consumer.maxAttempts – number of attempts before redirecting to DLQ

application.yml

```
spring:
  cloud:
    function:
        definition: uppercase
    stream:
      bindings:
        uppercase-in-0:
          destination: test
          consumer:
            enableDlq: true
            dlqName: in-dlq
            maxAttempts: 3
        uppercase-out-0:
          destination: test2
          consumer:
            enableDlq: true
            dlqName: out-dlq
            maxAttempts: 4
```

## Spring Cloud Streaming Reactive Support

- Suppliers & Functions are candidate for reactive way of streaming
  - Supplier<Flux<T>> wraps Flux emitters which bounded with
    - Event generator
    - Reactive Stream
  - Function<Flux<T>,Flux<R>> may consume events and emit it as Flux

  - Nature of Reactive Stream along with Flux API allows functions to work with multiple sources and sinks
  - Flux API provides Tuples to merge different streams

## Spring Cloud Streaming Reactive Support

- Reactive Supplier

```
@Bean
public Supplier<Flux<String>> randomValues(EventGenerator eg) {
   return () -> Flux.create(emitter -> {
                        eg.addListener(e -> {
                        emitter.next(e.getValue() + "");
                  });
   });
}
```

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        randomValues-out-0:
          destination: test
```

# Spring Cloud Streaming

JOHN BRYCE
Leading in IT Education
a matrix company

## Spring Cloud Streaming Reactive Support

- Reactive Function

```
@Bean
public Function<Flux<String>,Flux<String>> reactiveUppercase() {
   return flux-> flux.map(s -> s.toUpperCase());
}
```

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        reactiveUppercase-in-0:
          destination: test
        reactiveUppercase-out-0:
          destination: test2
```

## Spring Cloud Streaming Reactive Support

- Function with multiple inputs and outputs
- BigData scenarios:
  - Receiving different events related to the same logical context (session, tx, sso)
  - Data aggregation

  - Tuples
    - Contains a set of Flux, from 2-8
    - zips/reduces the outputs  of all participating streams via merge function
  - When a function consumes or produces from/to multiple queues it uses the relevant TupleX class

## Spring Cloud Streaming Reactive Support

- Function with 2 inputs and a single output

```
@Bean
public Function<Tuple2<flux<Integer>,Flux<String>>,Flux<String>> fuse() {
   return tuple-> {
           Flux<String> numbers= tuple.getT1().map(i->I + " checked");
           Flux<String> words= tuple.getT2().map(s->s + " checked");
           return Flux.merge(numbers,words);
   };
}
```

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        fuse-in-0:
          destination: test
        fuse-in-1:
          destination: test2
        fuse-out-0:
          destination: test3
```

- Application.yml sets 2 inputs: 'test' is the source of the first type in Tuple2 – Flux<Integer>, while 'test2' is the source for the second Tuple type – Flux<String>

## Spring Cloud Streaming Reactive Support

- Function with 2 outputs and a single input

```
@Bean
public Function<Flux<String>,Tuple2<flux<String>,Flux<String>>> divideByLength() {
   return flux-> {
           Flux<String> short= flux.filter(s->s.length()<=5);
           Flux<String> longer= flux.filter(s->s.length()>5);
           return Tuples.of(short,longer);
   };
}
```

- Application.yml sets 2 outputs: ‘test2’ & ‘test3’

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        divideByLength-in-0:
          destination: test
        divideByLength-out-0:
          destination: test2
        divideByLength-out-1:
          destination: test3
```