[Event Driven Microservices with Spring Cloud Stream](https://blog.codecentric.de/en/2016/04/event-driven-microservices-spring-cloud-stream/" \o "Event Driven Microservices with Spring Cloud Stream)

05/04/16 by [Tobias Flohre](https://blog.codecentric.de/en/author/tobias-flohre/)

[No Comments](https://blog.codecentric.de/en/2016/04/event-driven-microservices-spring-cloud-stream/#comments)

Lately I’ve been much into event driven architectures because I believe it’s the best approach for microservices, allowing for much more decoupled services than point-to-point communication. There are two main approaches for event driven communication:

* **Feed**: Each application has a (synchronous) endpoint anybody may pull domain events from in a feed fashion.
* **Broker**: There is a dedicated broker responsible for distributing the events, like Kafka.

Each approach has its up- and downsides. With a broker you have more infrastructure to handle, but you also have a central place where your events are stored. Feeds are not accessible when the producing application is down. Scaling is easier with a broker – what happens if you suddenly need to double your consuming applications because of load? Who subcribes to the feed? If both subscribe, events are processed twice. With a broker like Kafka you easily create consumer groups, and each event is only processed by one application of this group. So we preferred the broker way, and we decided to use Kafka.  
So far so good – but we were impatient. We wanted to learn about event driven architectures, we didn’t want to spend weeks fighting with Kafka. And there came Spring Cloud Stream to the rescue.  
  
Yes, we spent a little time setting up our own little playground with docker-compose, including Kafka and Zookeeper of course, but also Spring Cloud Config, Spring Boot Admin and an integrated Continuous Delivery setup with Jenkins, Nexus and Sonar. You can find it here: <https://github.com/codecentric/event-driven-microservices-platform>. Then we thought that the tough part would come – connecting to and using Kafka. We stumbled over Spring Cloud Stream – and using Kafka was a matter of minutes.

Dependencies

You need to add one dependency to your pom:

|  |
| --- |
| **<dependency>**  **<groupId>**org.springframework.cloud**</groupId>**  **<artifactId>**spring-cloud-starter-stream-kafka**</artifactId>**  **</dependency>** |

As parent I use the spring-cloud-starter-parent in the most current version (at time of writing Brixton.RC1). It solves all the version management for me.

|  |
| --- |
| **<parent>**  **<groupId>**org.springframework.cloud**</groupId>**  **<artifactId>**spring-cloud-starter-parent**</artifactId>**  **<version>**Brixton.RC1**</version>**  **</parent>** |

When using Actuator, Spring Cloud Stream automatically adds a HealthIndicator for the Kafka binder, and a new actuator endpoint /channels with all the channels used in the application.

Producing events

In our sample application we produce one event every 10 seconds with a Poller.

|  |
| --- |
| @SpringBootApplication  @EnableBinding(Source.**class**)  **public** **class** EdmpSampleStreamApplication {    **public** **static** **void** main(String[] args) {  SpringApplication.run(EdmpSampleStreamApplication.**class**, args);  }    @Bean  @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay = "10000", maxMessagesPerPoll = "1"))  **public** MessageSource<TimeInfo> timerMessageSource() {  **return** () -> MessageBuilder.  withPayload(**new** TimeInfo(**new** Date().getTime()+"","Label")).build();  }    **public** **static** **class** TimeInfo{    **private** String time;  **private** String label;    **public** TimeInfo(String time, String label) {  **super**();  **this**.time = time;  **this**.label = label;  }    **public** String getTime() {  **return** time;  }    **public** String getLabel() {  **return** label;  }    }    } |

When using @EnableBinding(Source.class) Spring Cloud Stream automatically creates a message channel with the name output which is used by the @InboundChannelAdapter. You may also autowire this message channel and write messages to it manually. Our application.properties looks like this:

spring.cloud.stream.bindings.output.destination=timerTopic

spring.cloud.stream.bindings.output.content-type=application/json

spring.cloud.stream.kafka.binder.zkNodes=kafka

spring.cloud.stream.kafka.binder.brokers=kafka

It basically says that we want to bind the output message channel to the Kafka timerTopic, and it says that we want to serialize the payload into JSON. And then we need to tell Spring Cloud Stream the host name where Kafka and Zookeeper are running – defaults are localhost, we are running them in one Docker container named kafka.

Consuming events

Our sample application for consuming events looks like this:

|  |
| --- |
| @SpringBootApplication  @EnableBinding(Sink.**class**)  **public** **class** EdmpSampleStreamSinkApplication {    **private** **static** Logger logger = LoggerFactory.  getLogger(EdmpSampleStreamSinkApplication.**class**);    **public** **static** **void** main(String[] args) {  SpringApplication.run(EdmpSampleStreamSinkApplication.**class**, args);  }    @StreamListener(Sink.INPUT)  **public** **void** loggerSink(SinkTimeInfo sinkTimeInfo) {  logger.info("Received: " + sinkTimeInfo.toString());  }    **public** **static** **class** SinkTimeInfo{    **private** String time;  **private** String label;    **public** String getTime() {  **return** time;  }    **public** **void** setTime(String time) {  **this**.time = time;  }    **public** **void** setSinkLabel(String label) {  **this**.label = label;  }    **public** String getLabel() {  **return** label;  }    @Override  **public** String toString() {  **return** "SinkTimeInfo [time=" + time + ", label=" + label + "]";  }    }    } |

When using @EnableBinding(Sink.class) Spring Cloud Stream automatically creates a message channel with the name input which is used by the @StreamListener above. Our application.properties look like this:

spring.cloud.stream.bindings.input.destination=timerTopic

spring.cloud.stream.bindings.input.content-type=application/json

spring.cloud.stream.bindings.input.group=timerGroup

spring.cloud.stream.kafka.bindings.input.consumer.resetOffsets=true

spring.cloud.stream.kafka.binder.zkNodes=kafka

spring.cloud.stream.kafka.binder.brokers=kafka

We see the binding of input to timerTopic, then we see the content-type we expect. Note that we don’t share the class with the producing application – we just deserialize the content in a class of our own.  
Then we specify the consumer group this application belongs to – so if another instance of this application is deployed, events are distributed among all instances.  
For development purposes we set resetOffsets of the channel input to true which means that on new deployment, all events are processed again because the Kafka offset is reset. It could also be a strategy to do that on every startup – having all the state just in memory – and in Kafka. Then, of course, consumer groups don’t make sense, and processing the events should not create other events – consuming the events is just used to create an internal state.

Conclusion

What can I say? Spring Cloud Stream was really easy to use, and I will certainly do that in the future. If you want to try it out for yourself with a real Kafka, I can point you again to<https://github.com/codecentric/event-driven-microservices-platform>.  
Install [Docker Toolbox](https://www.docker.com/toolbox" \t "_blank), then do this:

$ docker-machine create -d virtualbox --virtualbox-memory "6000" --virtualbox-disk-size "40000" default

$ eval "$(docker-machine env default)"

$ git clone git@github.com:codecentric/event-driven-microservices-platform.git

$ cd event-driven-microservices-platform

$ docker-compose up

Now get a coffee, have a chat with the colleagues, or surf around the internet while Docker is downloading it. Then go to http://${docker-machine ip default}:18080/ and you should see something like this:  
  
Then go to Spring Boot Admin at http://${docker-machine ip default}:10001/ and you should see something like this:  
  
And if you take a look at the logs of edmp-sample-stream-sink you’ll see the events coming in.