Lab 18 - Streams for Reactive Microservices

In this example, we will use RabbitMQ as the message broker:

* Add the following Maven dependency to Booking, Search, and Check-in, as these are the three modules using messaging:
* <dependency>  
   <groupId>org.springframework.cloud</groupId>  
   <artifactId>spring-cloud-starter-stream-rabbit</artifactId>  
  </dependency>
* Add the following two properties to booking-service.properties. These properties bind the logical queue inventoryQ to physical inventoryQ, and the logical checkinQ to the physical checkinQ:
* spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ  
  spring.cloud.stream.bindings.checkInQ.destination=checkInQ
* Add the following property to search-service.properties. This property binds the logical queue inventoryQ to the physical inventoryQ:
* spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ
* Add the following property to checkin-service.properties. This property binds the logical queue checkinQ to the physical checkinQ:
* spring.cloud.stream.bindings.checkInQ.destination=checkInQ
* Commit all files to the Git repository.
* The next step is to edit the code. The Search microservice consumes a message from the Booking microservice. In this case, Booking is the source and Search is the sink.
* Add @EnableBinding to the Sender class of the Booking service. This enables the Cloud Stream to work on autoconfigurations based on the message broker library available in the class path. In our case, it is RabbitMQ. The parameter BookingSource defines the logical channels to be used for this configuration:
* @EnableBinding(BookingSource.class)  
  public class Sender {
* In this case, BookingSource defines a message channel called inventoryQ, which is physically bound to RabbitMQ's inventoryQ, as configured in the configuration. BookingSource uses an annotation, @Output, to indicate that this is of the output type—a message that is outgoing from a module. This information will be used for autoconfiguration of the message channel:
* interface BookingSource {  
   public static String InventoryQ="inventoryQ";   
   @Output("inventoryQ")  
   public MessageChannel inventoryQ();   
  }
* Instead of defining a custom class, we can also use the default Source class that comes with Spring Cloud Stream if the service has only one source and sink:
* public interface Source {  
   @Output("output")  
   MessageChannel output();  
  }
* Define a message channel in the sender, based on BookingSource. The following code will inject an output message channel with the name inventory, which is already configured in BookingSource:
* @Output (BookingSource.InventoryQ)  
   @Autowired  
   private MessageChannel;
* Reimplement the send message method in BookingSender:
* public void send(Object message){  
   messageChannel.  
   send(MessageBuilder.withPayload(message).  
   build());  
  }
* Now add the following to the SearchReceiver class the same way we did for the Booking service:
* @EnableBinding(SearchSink.class)  
  public class Receiver {
* In this case, the SearchSink interface will look like the following. This will define the logical sink queue it is connected with. The message channel in this case is defined as @Input to indicate that this message channel is to accept messages:
* interface SearchSink {  
   public static String INVENTORYQ="inventoryQ";   
   @Input("inventoryQ")  
   public MessageChannel inventoryQ();  
  }
* Amend the Search service to accept this message:
* @ServiceActivator(inputChannel = SearchSink.INVENTORYQ)  
  public void accept(Map<String,Object> fare){  
   searchComponent.updateInventory((String)fare.  
   get("FLIGHT\_NUMBER"),(String)fare.  
   get("FLIGHT\_DATE"),(int)fare.  
   get("NEW\_INVENTORY"));  
  }
* We will still need the RabbitMQ configurations that we have in our configuration files to connect to the message broker:
* spring.rabbitmq.host=localhost  
  spring.rabbitmq.port=5672  
  spring.rabbitmq.username=guest  
  spring.rabbitmq.password=guest  
  server.port=8090
* Run all services, and run the website project. If everything is fine, the website project successfully executes the Search, Booking, and Check-in functions. The same can also be tested using the browser by pointing to http://localhost:8001.