Exercise 7: Asynchronous Communication using a Message Broker

Submission in AULIS by 08.01.26, 9:45am

Assignment: Asynchronous Communication with Message Broker RabbitMQ

The goal of this assignment is to implement asynchronous communication between the Order REST Service developed in assignment 6 and a new Spring application. The platform to be used for the exchange of messages (events) between these two applications is *RabbitMQ* (http://rabbitmq.com/):

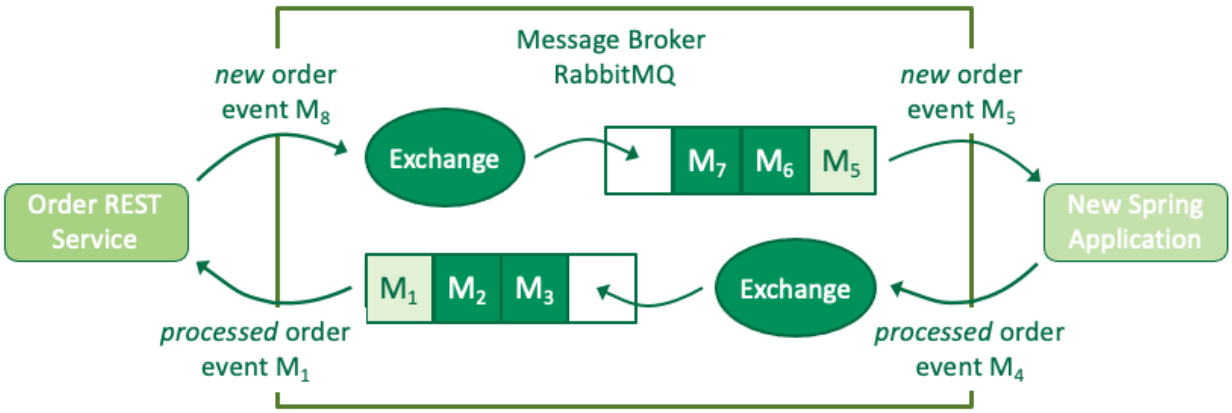


Figure 1: Overview

The configuration of RabbitMQ depicted in Figure *1* is only for illustrational purposes. Decisions on the used exchanges and queues are to be made by you!

The assignment is subdivided into four subtasks.
The results of the subtasks shall be submitted as *one* upload to AULIS. Please make sure that the Postman collections from the previous assignment are included in your submission.

Preparation:

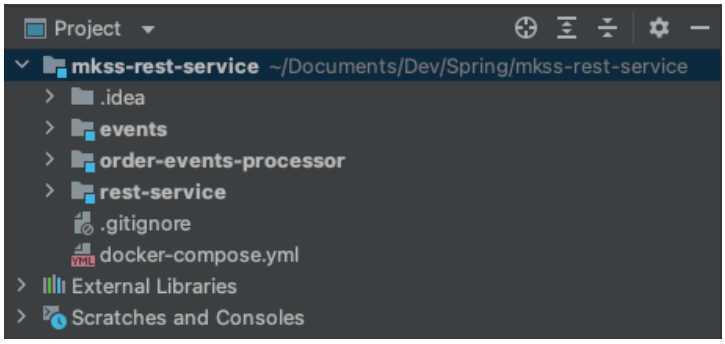
Since RabbitMQ can be easily run from a *Docker* image, you'll have to install Docker on your computer: https://www.docker.com/products/personal ("Docker Personal" should do for our purposes.)

Subtask 1: Emitting Order Events from the Order Service

Your task is to emit an order event to the message broker RabbitMQ whenever an order has been purchased ("committed") in the Order Service. In order to make things easier for you, I have provided some template code. You are free to adapt and use this code or to write your own code.

Template code:

- Events project: IntelliJ Maven project (actually a module of my project) which contains …
  - … class *EventWithPayload<T>* where T is the type of the payload you want transfer using the event. The class uses the *Lombok* library for generating getters, setters, and a builder method.
  - … interface *CrudEventProducer<PayloadOfYourEvent>* which declares methods for emitting create/update/delete events. This interface *can* be implemented and used in your Order Service.
  - You are free to define your own events class!
  - Remark: my test project is made up of three submodules (Order REST service, Events Processor (subtasks 2 and 3), and the Events module). You may also define two individual projects (or three, if you are using my Events project). Have a look at the *pom.xml* file and adapt it to your type of project.



- Additional files:
  - Dependencies: the file *pom.xml* contains dependencies for RabbitMQ which you should add to your Order REST Service's *pom.xml* file. The file additionally defines a dependency on the *Events library* produced by the *Events project* (the Events library has to be generated and installed in your Maven repository; you may use other ways of including the code in the Events project).
  - Docker file: *docker-compose.yml* for installing and running a docker image of RabbitMQ.
  - RabbitMQ configuration: the template file *RabbitMQConfig* contains code and comments on the configuration of exchanges and queues. The file *application.properties* contains configuration values used in class *RabbitMQConfig* (naming of exchanges and queues) as well as for configuration of host/port/login/password used by RabbitMQ. Copy these values into the *application.properties* file of your Order service.
  - Emitting events: the class *XYZEventsProducer* represents a component for emitting events and (incompletely) implements the interface *CrudEventProducer*. The @Component annotation makes it a candidate for Spring's dependency injection.

Steps:

1. Install / run RabbitMQ using Docker image, either using
   - terminal command: navigate to the folder containing the docker compose file and execute command `docker compose up`
   - IntelliJ: simply select *docker-compose.yml* and execute "run" from context menu.
2. Configuration of RabbitMQ:
   - For *emitting* events, only the definition of an *exchange* is required. Queues are a receiver-side concept which, however, is influenced by your choice of exchange(s).
     - A *direct exchange* delivers messages to queues based on a message routing key.
     - A *fanout exchange* ignores the routing key and routes messages to all queues that are bound to it.

     For further information see https://www.rabbitmq.com/tutorials/amqp-concepts.html.
   - Implement your configuration by adapting the file *RabbitMQConfig* and the corresponding values defined in *application.properties*. Assign understandable ("speaking") names to attributes in *RabbitMQConfig* and properties / values in *application.properties*. Can you design a configuration that uses only one exchange?
3. Emitting events:
   - Implement an events producer class. The class should use the names of the exchange(s) and (if defined) routing keys of your configuration as well as an *AmqpTemplate* instance which you configured in *RabbitMQConfig.*
   - Cf. template in file *XYZEventsProducer* which (incompletely) implements methods for sending create/update/delete events using the events class implemented in the supplied *Events* project. "XYZ" is, of course, just a placeholder; assign your class an adequate name!
   - Hint: the statement for sending events to RabbitMQ is

     amqpTemplate.convertAndSend(exchangeName, routingKeyName, event);

     where *routingKeyName* is an empty String, if you use a fanout exchange.
4. Emit an event using the (renamed) *XYZEventsProducer* whenever an order is committed ("new order" event). The committed order instance should be transmitted to RabbitMQ with this event.
5. Test your solution:
   - Create and commit orders in your Order service using Postman.
   - Monitor the arrival of events in the *exchange* using the RabbitMQ console. The console can be accessed via http://localhost:15672/ using the login and password defined in *application.properties*: in the exchange tab you should see new events coming in whenever you commit an order.

Subtask 2: Receiving Order Events in a Consumer Application

Your second task is to implement the reception of order events from the message broker RabbitMQ in a very simple Spring Boot application.

Steps:

1. Setup a new Spring Boot application for consuming events. Just like the Order service in subtask 1, the new application depends on RabbitMQ and – if used – my Events project (cf. template *pom.xml*).

   Remark: In my test project, I just implemented an application with a @Service component *XYZEventsConsumer* (→ step 3).
2. Configuration: Specify the exchange you want to read events from and define a *binding*, i.e. bind a queue for incoming events to that exchange in *RabbitMQConfig* of your consumer application; if you use a direct exchange, you also have to define a routing key. Remember to define RabbitMQ configuration values in *application.properties* file (similar to subtask 1).
3. Implement a @Service consumer class (e.g. *XYZEventsConsumer*) for processing incoming *create* events. Receiving events should be implemented in a method like this:

   ```
   @RabbitListener(queues="${my.rabbitmq.a.queue}")
   public void receiveMessage(EventWithPayload<YourPayloadClass> event) {
       // YOUR CODE
   }
   ```

   and print out the received event to the console.
4. Test your solution:
   - Start your consumer application, commit orders in the Order Service and monitor the output of received events in your consumer application.
   - In addition, you should be able to monitor the arrival of events in the configured *queue* using the RabbitMQ console.

Subtask 3: Sending Reply Events from Consumer Application to Order Service

In addition to just printing received events to the console, the consumer application should now *randomly* choose whether it accepts or rejects an order, set the order's status accordingly and send an *update* event to RabbitMQ containing the updated order instance.

For this subtask, you may reuse results from subtask 1 – at least on a conceptual level: depending on your choice of exchange(s), you may have to define a new exchange or a new routing key.

Steps (in brief):

1. Configuration of RabbitMQ: Specify the exchange to be used and, if required, a routing key in *RabbitMQConfig,* and define corresponding values in *application.properties*.
2. Emitting events: Implement an *XYZEventsProducer* (or copy it from subtask 1) for emitting update events.
3. Make the *XYZEventsConsumer* emit an event using an (injected) *XYZEventsProducer* whenever an order has been received and processed (processing means throwing dice and randomly assigning an *accepted* or *rejected* status to the order instance). The processed order instance should be transmitted to RabbitMQ with this event.
4. Test your solution:
    - Create and commit orders in your Order Service using Postman.
    - Monitor the arrival of events in the *exchange* using the RabbitMQ console. You should now see new order events from *both* the Order Service *and* your consumer application.

Subtask 4: Receiving Reply Events in the Order Service

The last subtask is about receiving and persisting processed orders in the Order Service.

For this subtask, you may reuse results from subtask 2 – at least on a conceptual level: depending on your choice of exchange(s), you may have to define a new exchange or a new routing key.

Steps (in brief):

1. Configuration of RabbitMQ: Specify the exchange to be used, a binding, and (if required) a routing key in *RabbitMQConfig,* and define corresponding values in *application.properties*.
2. Receiving events: Implement an *XYZEventsConsumer* (or copy relevant parts of it from subtask 2) for receiving update events. The *XYZEventsConsumer* should check the type of event and persist the orders contained in *update* events to the repository. Events of other types should not be processed by the Order Service, i.e. they are ignored.
3. Test your solution:
    - Create and commit orders in your Order Service using Postman.
    - Monitor the arrival of events in the *exchange* using the RabbitMQ console. You should now see new order events from the Order Service and your consumer application.
    - Retrieve orders from your Order Service using Postman. Orders that have been purchased ("committed") before should now be either accepted or rejected.