# Mini Project: B2B Retailer

**Borja Velez**

**Ruben Martin**

**Manuel Rico**

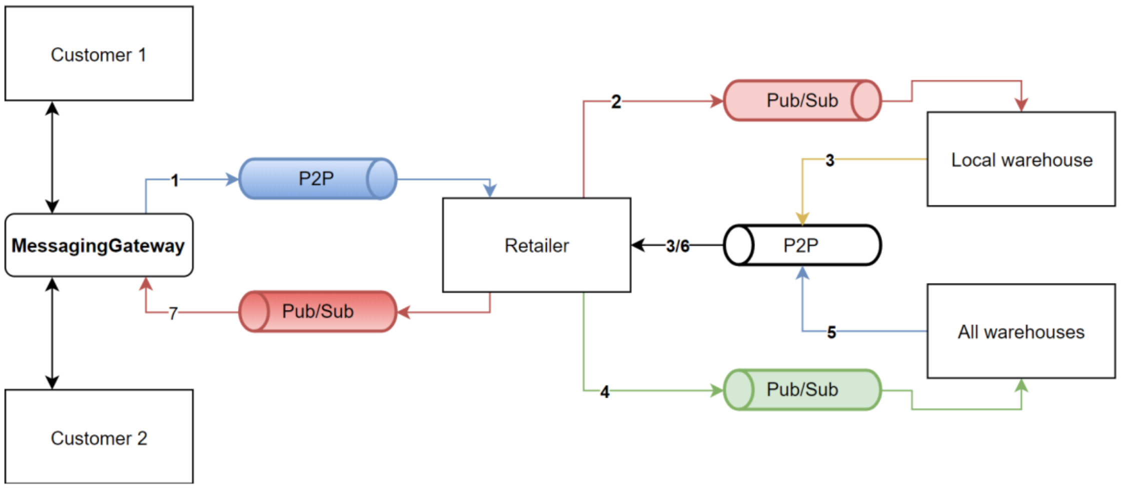
PBA Software Development

System Integration

*08/03/2018*

# Introduction

According to the given instructions, we accomplished a solution described in the following diagram:



Our solution consists on 2 different customers for the sake of simplicity, but making it possible to create many more different customers in case it is needed.

Each client will contact the *MessagingGateway*, with the information of the desired order and their country of origin.

This component will send via P2P these data to the Retailer.

The retailer will collect these data, and publish it along with a topic, in this case, the country of origin.

Only the warehouse whose country identification equals the order's country of origin is going to receive the given order, because only the Local warehouse is going to be expecting a message with its country ID as a topic.

If the product is available, the warehouse will return the total cost (including the shipping cost) along with the delivery days (2 days for local warehouses).

In case the Local Warehouse does not have the asked product, it will send this information to the Retailer. With this information, the Retailer knows that it must send the order to all the other warehouses.

To do so, it will publish a new message which will include the order along with a new topic (“all”).

When the topic “all” is included in the message, all the warehouses will look for the asked product at the same time.

When a warehouse finds the desired item, it will send back the pricing and delivering information via P2P to the Retailer.

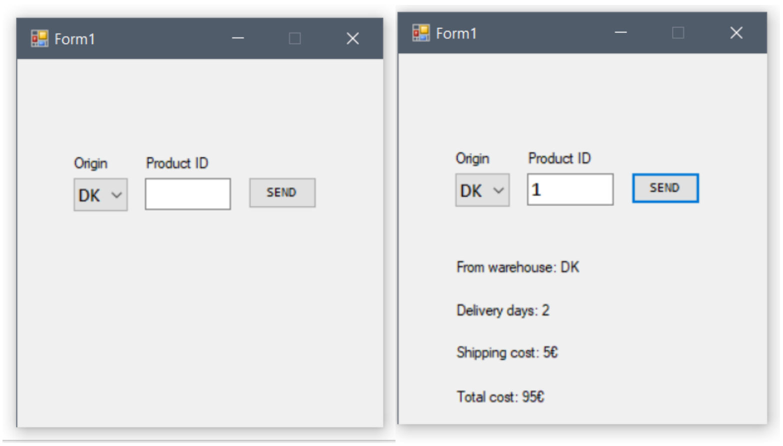
The retailer will now publish these data along with a topic (the customer id).

The customer who made the order, whose ID is defined randomly at the beginning of the execution, will be waiting (subscribed) for a message containing its customer ID.

Finally, the customer will receive the information and will be able to see on the UI the chosen warehouse, the total cost, the shipping cost, and the delivery days.

# Customer

This Project consists on a User Interface where the Customer is able to select its country of origin and the desired Product ID.



From warehouse: DK

Delivery days: 2

Shipping cost: 5€

Total cost: 95€

This class will make sure that the input values are correct, and it will collect all these data and send it to the Messaging Gateway.

When the asked information is received, it will show on screen the information regarding the choosing warehouse, the shipping cost, the total cost and the delivery time.

# Messaging Gateway

The messaging gateway is going to be the logical connector between the Customer UI and the Retailer.

Using the method Send(), it is going to receive the information input by the Customer on the interface.

With the received information, it is going to create and initialize a new Order.

This order is going to be stored in a queue and sent to the Retailer, which will handle it and communicate with the warehouses.

After sending the order, a lock is going to protect the current process from other background processes.

The class will continue running if a message is received through the subscribe queue, or if the process time exceeds a chosen amount of seconds.

Once we receive a response from the Retailer, the class will continue running and the Send() method will return an Order instance with all the information received from the warehouse, prices and delivery date, to the Customer.

## Retailer

The retailer is going to receive an object Order from the Messaging Gateway, using P2P, through a Queue previously created called "*customerSendProductQueue*".

The received objects are going to be handled by a method called *HandleOrderReplyMessage*.

That method is going to handle the messages differently, depending on their origin.

If the message comes from the Customer, the Retailer is going to publish the order along with the origin country as a topic to the warehouses and right after it is going to wait for a response from any of those warehouses.

If the message received comes from one warehouse and not from the customer, the retailer is going to process that message accordingly.

In case the received message comes from the local warehouse saying that it does not have the desired product, the retailer is going publish again the mentioned order, but this time with the keyword "*all*" as a topic, so the message will be received by all the warehouses.

Once the message from the warehouse has been processed, the information will be sent back to the Messaging Gateway to be delivered to the customer and showed on the screen.

## Warehouse

In the project, warehouses are the responsible of handling the requests from retailers and reply if they have or not certain product.

We can differentiate between local warehouses and the rest of warehouses. Depending on the origin of the customer that makes the order request, a warehouse might be local or not.

The local warehouse is always asked first about if it has that product, only if this warehouse does not have it the rest of warehouses are also asked.

In our application there is a set of warehouses created by default with a list of products, prices and quantities in the class *WarehouseProgram.cs*. Some of these products are available only in some countries.

In the class *Warehouse.cs* we define the logic of this component and the format of the responses (price, delivery days…) according to if it is a local warehouse or not in the request that is being processed.

## Patterns, channels and explanation

In the project we have two different kind of channels. On the one hand, we have some Point-to-Point channels and on the other hand we have Publish-Subscribe channels.

However, the decision of choosing them in the connection between the distinct components has not been random, there are reasons beyond this structure that allow us to fulfil the requirements of the mini-project.

First, to allow different customers to ask for a concrete product in a same period of time along other customers placed in other countries, we have implemented a Publish-Subscribe channel in the response of the retailer, making possible that several customers can be waiting for their response while their requests are being put into queues and being processed one by one.

The retailer can just process at one time one single request. However, once the retailer sends a message to a warehouse asynchronously (using a Publish-Subscribe channel) and until it receives a response from a warehouse, it will have the opportunity to process the following request in queue.

If we had used the Point-to-Point channel in the response from the retailer to the customer, then it would not have been possible to manage different requests from different customers and put them into queues. A second customer would have to wait until the first customer received the response from the retailer and only then would be possible to request a second product in a distinct order. Thanks to the topic property of the Publish-Subscribe channels we can identify different customers when handling the response of the requests.

In the communication between the customer and the retailer it would have been possible also to apply another pattern offered by RabbitMQ, the Request-Reply channel. This implementation offers some advantages in comparison to the other kind of channels that are the configuration of a time-out that allows to close the connection and delete the queue if no response has been received in the time that has been set (it also would do the same if it receives the reply before the time-out expires). However, it has a considerable disadvantage, it doesn't allow the processing of multiple requests from different customers at the same time. The time-out feature could be implemented manually in the other kind of channels.

So, as it happened with the P2P pattern, the application is then less scalable in those approaches. With Publish-Subscribe we gain in scalability because with the help of queues we can manage more than one request at the same time.

## Environment setup

Before executing the code, we recommend to check that no queues are already created at the moment of launching the application either for first time or for consecutive times. The behavior might be a little different when returning the results on the screen.

We have experimented that not always when closing the application the queues are deleted, so we recommend to delete them manually from the Administration Panel of RabbitMQ before every test of the application.

## GitHub repository

The code can be downloaded and tested from the following repository:

https://github.com/borjavelez/B2BRetailer.git