T-302-HONN: Project 2 Microservices

Þórður Friðriksson
thordurf@ru.is

Háskólinn í Reykjavík — Reykjavík University

Introduction

Attention Please read over all the following points before you start the project

1. **Groups**
   - This assignment can be done in groups of 1-4 people
   - Only one member of each group needs to hand in the assignment
   - Remember to add each group members name and email to the front page of the handed-in assignment
2. **Rules about cheating**
   - The solution needs to be you own work, if it is discovered that cheating has occured then all participating parties will receive a 0 for this assignment if this is their first offence on the other hand if this is a repeating occurance then more severe measures can be expected. See the school's policy and rules on assignments ru.is/namid/reglur/reglur-um-verkefnavinnu
3. **Assignment hand-in**
   - the assignments need to be handed-in on Canvas as a .zip file for the code
   - zip file must be named: `{student1@ru.is}-{student2@ru.is}-project2.zip`
   - **If the instructions for the handin are not follwed you can expect a grade deduction**
4. **Late submissions**
   - See late submission policy

1 Overview

In this assignment you are supposed to build a ordering system with *event-driven microservice architecture*. The system should be able to create and charge an order as well as send an email for specific steps in the workflow.

The system will consist of a few small microservices where each service is isolated and decoupled from the other services and will only communicate with them through networks calls, some network calls will be event-driven while others will be request/response based.

The goal of this assignment is to get a feeling for how we can develop highly scalable, decoupled and fault tolerant software with events and distributed services.

While developing the system you will use REST(framework of your choice), RabbitMQ, Python, Docker and databases of your choice.

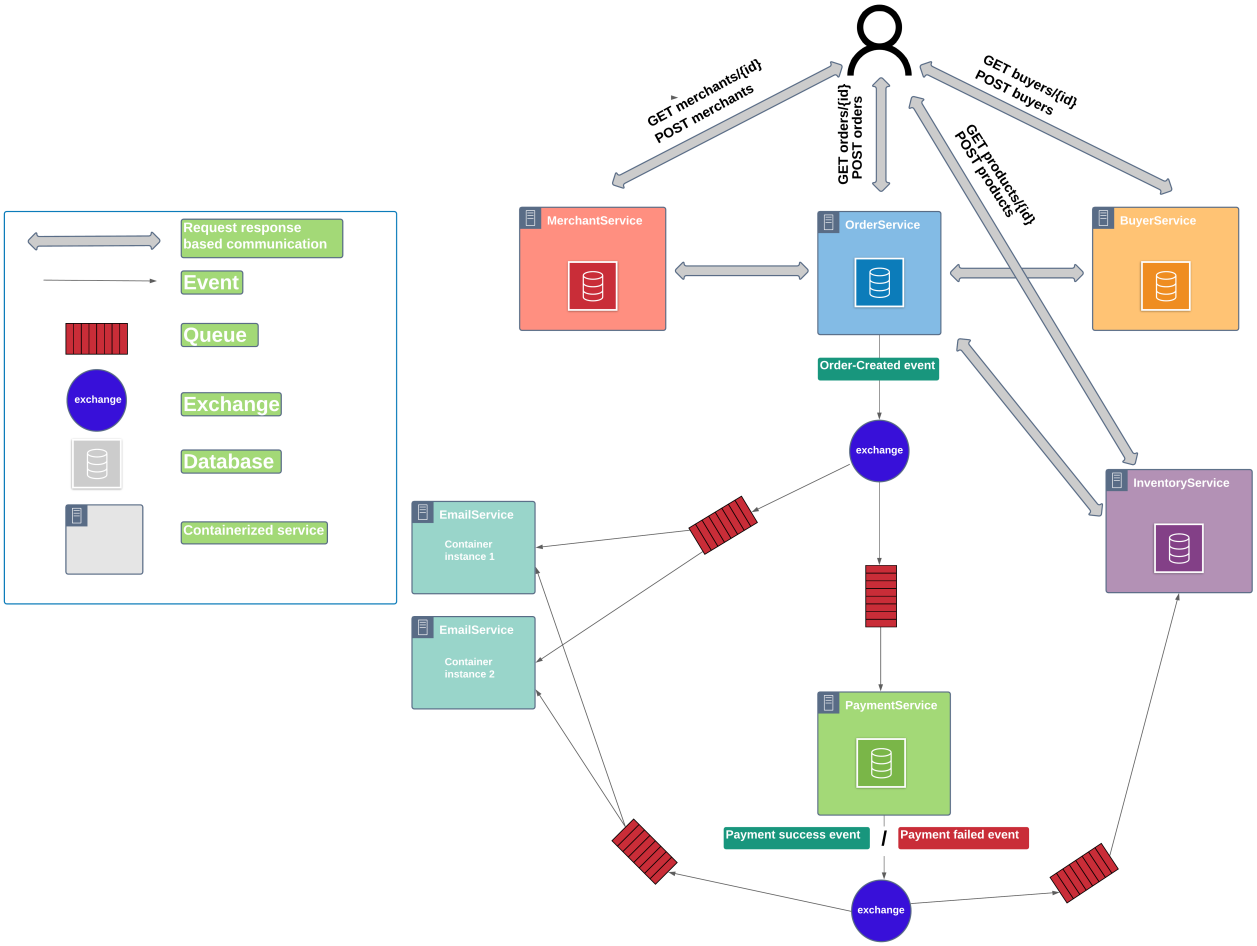
Notice. Keep in mind that this system that you will build is oversimplified and does not follow all the best-practices of developing an event-driven microservice system. Security is missing, error handling is lacking, monitoring is missing, logging is missing, the workflow is lacking, there is no container orchestration and the microservices themselves are simple and bare bones.

The assignment is mostly designed to give you a glimpse into how to work with microservices, event-driven architecture, distributed systems and Docker.

2 Overview of the workflow in the system

The system needs to be able to handle requests to create orders, the process of creating an order has multiple steps:

1. First the use of the system sends a POST request to `OrderService` to create an order, this POST request for example contains `buyer_id`, `merchant_id`, `product_id`, `credi_card` and other information about the order.
2. Next, when `OrderService` gets the request to create an order, it communicates with `MerchantService` and `BuyerService` with Request/Response based communication (for example REST) and checks if there is a buyer and merchanr for the correspondent `buyer_id` og `merchant_id`. If so then OrderService communicates next with the `InventoryService` with request/response based communication and reserves a product with `product_id`. If OrderServicec was successful in reserving the product(the product wasn't sold out / the product exists) then OrderServices stores the order in it's database and sends out an event that the order has been created.
3. This event that the order has been created is picked up by both EmailService and PaymentService. EmailService handles sending out an email that some specific order has been created and PaymentService handles the charge of the order. PaymetnService then sends out and event regarding the results of the charge.
4. This event about the result of the charge is then picked up by the EmailService as well as the InventoryService. EmailService sends out and email about the results, InventoryService marks the product that was reserved as either sold if the charge was successful otherwise it stops reserving the product.

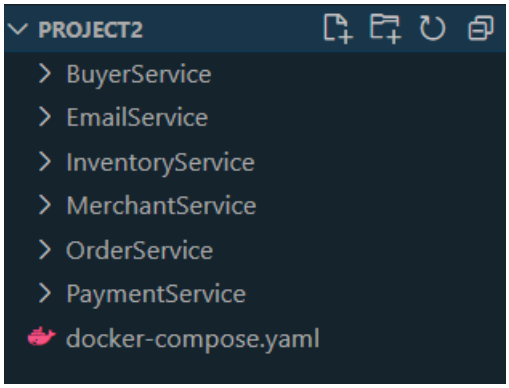


3 Requirements

Notice I recommend that you read the whole project description before you start the implementation.

3.1 Project Structure

- You should have a folder for each microservice where each service folder has the same name as the service iteslf, for example the folder that holds the OrderService should be called OrderService etc.
- On the same folder level as t he microservice folders sthere should be a docker-compose.yaml file that starts up the whole system.
- Ættuð að hafa sér folder fyrir hvert microservice þar sem hvert service heitir sama nafni og service-ið sjálft, t.d. ætti folder-inn sem hýsir `OrderService` að heita OrderService etc.
- Á sama folder level-i og microservice folder-arnir ætti að vera `docker-compose.yaml` file sem start-ar upp öllu kerfinu.



3.2 Databases

All databases that are youseed in the microservices can take whatever for you wnt, this could be an SQL database like Postgres or it could be an NOSQL database like MongoDB og this could simply be a simple textfile that your read and write to.

The only requirement for the databases is that it is **persistent** between container instances.

3.3 Service APIs

`OrderService`, `BuyerService`, `MerchantService` and `InventoryService` all have some "External" Api that the users of the system can call to create or get a certain order, merchant, buyer or product.

The External APIs for the microservices should we made with REST. The Framework used to implement the External API is in you hands and can be whatever you want for example Django, FastApi, Flask

It should be possible to call these external APIs locally when the system is run with docker-compose.yaml

Internal request-based communication (that is to say service-to-service communication) can be implemented however you like, it can be made with REST(for example a service can use the external api-s of other services to communicate with them) or it can be done with pseudo-synchronous communication with RabbitMQ or it could be done with gRPC or GraphQL or whatever you like.

Notice Keep in mind that often we wouldn't want the users of the system to be able to directly call our microservices, we would rather want to encapuslate our services with some sort of an API-gateway. With things like kubernetes it is also possible to specify exactly which service APIs are exposed to the outside world Kubernetes Ingress

We do it like this now for the sake of simplicity.

3.4 RabbitMQ

Some communication between microservices should be done with events, to do so we will use the RabbitMQ message broker.

3.5 Docker

The whole system should be containerized with Docker.

- Each service should have a `Dockerfile` and a `.dockerignore` file
- All databases should be containerized and persistent.
- The RabbitMQ server should be containerized
- The whole system should be able to be started up with a `docker-compose.yaml` file

Notice In the real world it is not necessarily the norm to start the entire system with docker-compose, although it is often common. Other ways are to just run the service you are working on locally and communicate to other services running in the cloud in a development environment (e.g. with kubernetes you can connect to a locally running service in the cluster with Bridge to kubernetes or with Telepresence). An alternative to compose are things like Tye

3.6 OrderService

3.6.1 API

OrderService needs to have an external API that the users of the system can use to create and retrieve orders.

1. `POST /orders`
    - The request should create an order
    - An example of the request body is:

```
{
  "productId": 123,
  "merchantId": 123,
  "buyerId": 123,
  "creditCard": {
    "cardNumber": "12341234123412341234",
    "expirationMonth": 8,
    "expirationYear": 2025,
    "cvc": 123
  },
  "discount": 0.2
}
```

    - OrderService should return **400 HTTP Status Code** with the error message "Merchant does not exist" if there is no merchant with the specific id
    - OrderService should return **400 HTTP Status Code** with the error message "Merchant does not exist" if there is no merchant with the specific `merchantId`.
    - OrderService should return **400 HTTP Status Code** with the error message "Buyer does not exist" if there is not buyer with the specific `buyerId`
    - OrderService should return **400 HTTP Status Code** with the error message "Product does not exist" if there is no product with the specific `productId`
    - OrderService should return **400 HTTP Status Code** with the error message "Product is sold out" if a product with the specific `productId` is sold out.
    - OrderService should return **400 HTTP Status Code** with the error message "Product does not belong to merchant" if product with `productId` does not belong to merchant with `merchantId`.
    - OrderService should return **400 HTTP Status Code** with the error message "Merchant does not allow discount" if merchant with `merchantId` does not allow discounts and the specified discount is something other then null or 0.
    - If all the validations are successful then the OrderSErvie should reserve the product, store it in the database, send an event that the order has been created and return **201 HTTP Status Code** með order id-i sem response message.

2. `GET /orders/{id}`

   - Should return **200 HTTP Status Code** with an order as the response body for the specific `id`. The order should be returned int he following json format:

```
{
  "productId": 123,
  "merchantId": 123,
  "buyerId": 123,
  "cardNumber": "************1234"
  "totalPrice": 123.0
}
```

     `CardNumber` is of the same length as the orignal cardnumber but only shows the 4 last digits and `totalPrice` is the price of the product with `productId` with the discount included that was defined when the order was created, for example wit the price of a particular product is 100 and the discount is 0.2 the then totalPrice is price * (1 - 0.2)

   - OrderService should return **404 HTTP Status Code** with the error message "Order does not exist" if orders with the specific `id` does not exist.

3.6.2 Database

`OrderService` should have a persistent database to store the orders.

3.6.3 RabbitMQ

When an order is created OrderService should send out an event that the order has been created, the event should include all the information about the created order that the event processors need to process the event.

3.6.4 Docker

- OrderService should be containerized
- The order database should be containerized
- The RabbitMQ server should be containerized
- It should be possible to call the OrderService API locally on port 8000. For example it should be possible to send the http request `GET http://localhost:8000/orders/5` to retrieve order with id 5.
- `OrderService`, the Order database and `RabbitMQ` shoud be part of a common docker-compose.yaml file that starts the whole system up.

3.7 MerchantService

3.7.1 API

1. `POST /merchants`
   - The request should create a merchant
   - An example of a request body is:

```
{
  "name": "John Johnson",
  "ssn": "4000000000",
  "email": "email@email.com",
  "phoneNumber": "1234567",
  "allowsDiscount": true
}
```

   - MerchantService should save the merchant to the database.
   - Should return **201 HTTP Status Code** with merchant the merchant id as the response message.

2. `GET /merchants/{id}`
   - Should return **200 HTTP Status Code** with a response body of the seller with the specific id. The seller should be returned in the following json format:

```
{
    "name": "John Johnson",
    "ssn": "4000000000",
    "email": "email@email.com",
    "phoneNumber": "1234567",
    "allowsDiscount": true
}
```

   - Merchantservice should return **404 HTTP Status Code** if the merchant with the corresponding id does not exist

3. It is allowed to define more endpoints for the internal network calls (for example OrderService calling MerchantService) if you are willing to do so but other microservices can also call the External endpoints that are defined above. Thesse extra endpoitns can be made with whatever network communication method that you prefer (REST, GraphQL, gRPC, RabbitMQ etc).

3.7.2 Database

`MerchantService` should have a persistent database to store the merchants.

3.7.3 Docker

- MerchantService should be containerized
- The Merchant database should be containerized
- It should be possible to call the MerchantService API locally on port 8001. For example it should be possible to send the http request `GET http://localhost:8001/merchants/5` to retrieve order with id 5.
- `MerchantService` and the merchant database should be part of a common docker-compose.yaml file to start up the whole system.

3.8 BuyerService

3.8.1 API

1. `POST /buyers`
   - Thre request should create a buyer
   - An example of the request body is:

```
{
  "name": "John Johnson",
  "ssn": "4000000000",
  "email": "email@email.com",
  "phoneNumber": "1234567",
}
```

   - BuyerService should save the buyer to the database.
   - Should return **201 HTTP Status Code** with the buyer id as the response message.

2. `GET /buyers/{id}`
   - Should return **200 HTTP Status Code** with response body of the buyer with the corresponding id. The buyer should be returned in the following json format:

```
{
  "name": "John Johnson",
  "ssn": "4000000000",
  "email": "email@email.com",
  "phoneNumber": "1234567",
}
```

   - BuyerService should return **404 HTTP Status Code** if the buyer with the corresponding `id` does not exist.

3. It is allowed to define more endpoints for the internal network calls (for example OrderService calling BuyerService) if you are willing to do so but other microservices can also call the External endpoints that are defined above. Thesse extra endpoitns can be made with whatever network communication method that you prefer (REST, GraphQL, gRPC, RabbitMQ etc).

3.8.2 Database

`BuyerService` should have a persistent database to store the buyers.

3.8.3 Docker

- BuyerService should be containerized
- The Buyer database should be containerized
- It should be possible to call the `BuyerService` API locally on port 8002. fore example it should be possible to send an http request to `GET http://localhost:8002/buyers/5` to retrieve buyer with id 5
- `BuyerService` and the Buyer databse should be part of a common docker-compose.yaml fiel to start up the whole system.

3.9 PaymentService

3.9.1 Payment Functionality

The PaymentSercice will of course not have any real charging logic, instead it will only validate the credit card information and if they are valid then PaymentService will send out a payment success event else it will send out a payment failed event.

- You will use luhn algorithm-ann to validate the card number
- Month expiration value should be a number within the range 1 til 12
- Year expiration should be a four digit number
- CVC value should be a three digit number.

3.9.2 Database

PaymentService should have a persistent database to store the payment results. What needs to be stored is the order id and the payment result.

3.9.3 RabbitMQ

PaymentService picks up an Order-Created event and vlaidates the card information for the order, if the information is valid then PaymentServie will send out a Payment-Successful event, else it will send out Payment-Failure event.

3.9.4 Docker

- PaymentService should be containerized
- The PaymentService database should be containerized
- The RabbitMQ server-inn should be containerized
- PaymentService should be a part of a common `docker-compose.yaml` file to start up the whole system.

3.10 EmailService

3.10.1 Email Functionality

EmailService should be able to send out an email for certain events to both the buyer and the merchant.

- To send emails we will use Yagmail
- to use Yagmail we will need to create a new
- Til að Yagmail virki rétt þá þurfum við að búa til nýtt gmail account that we will use to send the email.
- To be able to send emails we will need to lower the security of your newly created gmail account
- Store the email and password to the email in a .env file in the root of the EmailService folder and load it into the python code to start using Yagmail.

Notice These are not best practices on how we should be sending emails in a real application, we should not be lowering the security on our account and it would be better to find another way then to store our password in an .env file that is loaded into the python program.

3.10.2 RabbitMQ

1. `Order-Created event`
    - EmailService should be able to pick up an Order-Created event that the OrderService sends when orders are created.
    - When EmailService picks up this event then we send and email to both the buyer and the merchant.
    - The email should have the subject as "Order has been created"
    - The Email body should include the id of the order, the name of the product, the price of the order(the price of the product with the discount).
2. `Payment-Success event`
    - EmailService should be able to pick up a Payment-Success event that the PaymentService sends when a payment succeeded.
    - When EmailService picks up this event then we send and email to both the buyer and the merchant.
    - The eamil should have the subject as "Order has been purchased"
    - The Email body should say "Order {order id} has been successfully purchased" (exchange {order id} for the actual order id)
3. `Payment-Failure event`
    - EmailService should be able to pickup a Payment-Failure event that Paymentservice sends when a payment fails.
    - When EmailService picks up this event then we send and email to both the buyer and the merchant.
    - The eamil should have the subject as"Order purchase failed"
    - The Email body should say "Order {order id} purchase has failed" (exchange {order id} for the actual order id)
4. EmailService should function in a way so that when there are more then one instance of a EmailService container running then the EmailService instances should take turns processing the events on the queue, that is to say that two EmailService instances never process the same event because we don't want to send two emails for the same order.

3.10.3 Docker

- EmailService should be containerized
- RabbitMQ server-inn should be containerized
- EmailService should be a part of a common docker-compose.yaml file that starts up the whole system.
- The system should be started up with docker-compose up –scale EmailService=2 so that the system runs with two instances of the EmailService.

3.11 InventoryService

3.11.1 API

InventoryService needs to have an external AI that the user of the system uses to create and retrieve products.

1. `POST /products`
    - The request creates a product
    - An example of the request body is:

```
{
  "merchantId": 123,
  "productName": "some product name",
  "price": 123.0
  "quantity": 20
}
```

    - InventoryService should store the product in a database.
    - Should return **201 HTTP Status Code** with the product id as the response message.
2. `GET /products/{id}`
    - Should return **200 HTTP Status Code** with the response body as the product for the corresponding `id`. The product should be returned in the following json format:

```
{
  "merchantId": 123,
  "productName": "some product name",
  "price": 123.0
  "quantity": 20,
  "reserved": 5
}
```

      `quantity` is the number of the specific products that exist, `reserved` er is the amount of the specific product that have been reserved.
    - InventoryService should return **404 HTTP Status Code** with the error message "Product does not exist" if a product with the corresponding `id` does not exist.
3.
4. It is allowed to define more endpoints for the internal network calls (for example OrderService calling InventoryService) if you are willing to do so but other microservices can also call the External endpoints that are defined above. These extra endpoitns can be made with whatever network communication method that you prefer (REST, GraphQL, gRPC, RabbitMQ etc).
    - For example there needs to be some functionality so that the OrderService can reserve a product item.

3.11.2 Database

`InventoryService` should have a persistent database to store the products, the amount of product items in stock and the amount of product items that have been reserved.

3.11.3 RabbitMQ

1. `Payment-Success event`
    - When InventoryService picks up a Payment-Success evnet then it should stop reserving the specific product and decrease the amount of products in stock.
2. `Payment-Failure event`
    - When InventoryService picks up the Payment-Failure event then it should stop reserving the specific product.

3.11.4 Docker

- InventoryService should be containerized
- The Inventory database should be containerized
- The RabbitMQ server should be containerized
- it should be possible to call the `InventoryService` API locally on port 8003. For example it should be possible to send an http request to `GET http://localhost:8003/products/5` to retrieve a product with id 5.
- `InventoryService`, the Inventory databse and `RabbitMQ` should be part of a common `docker-compose.yaml` to start up the whole system.

3.12 Bonus API-Layer

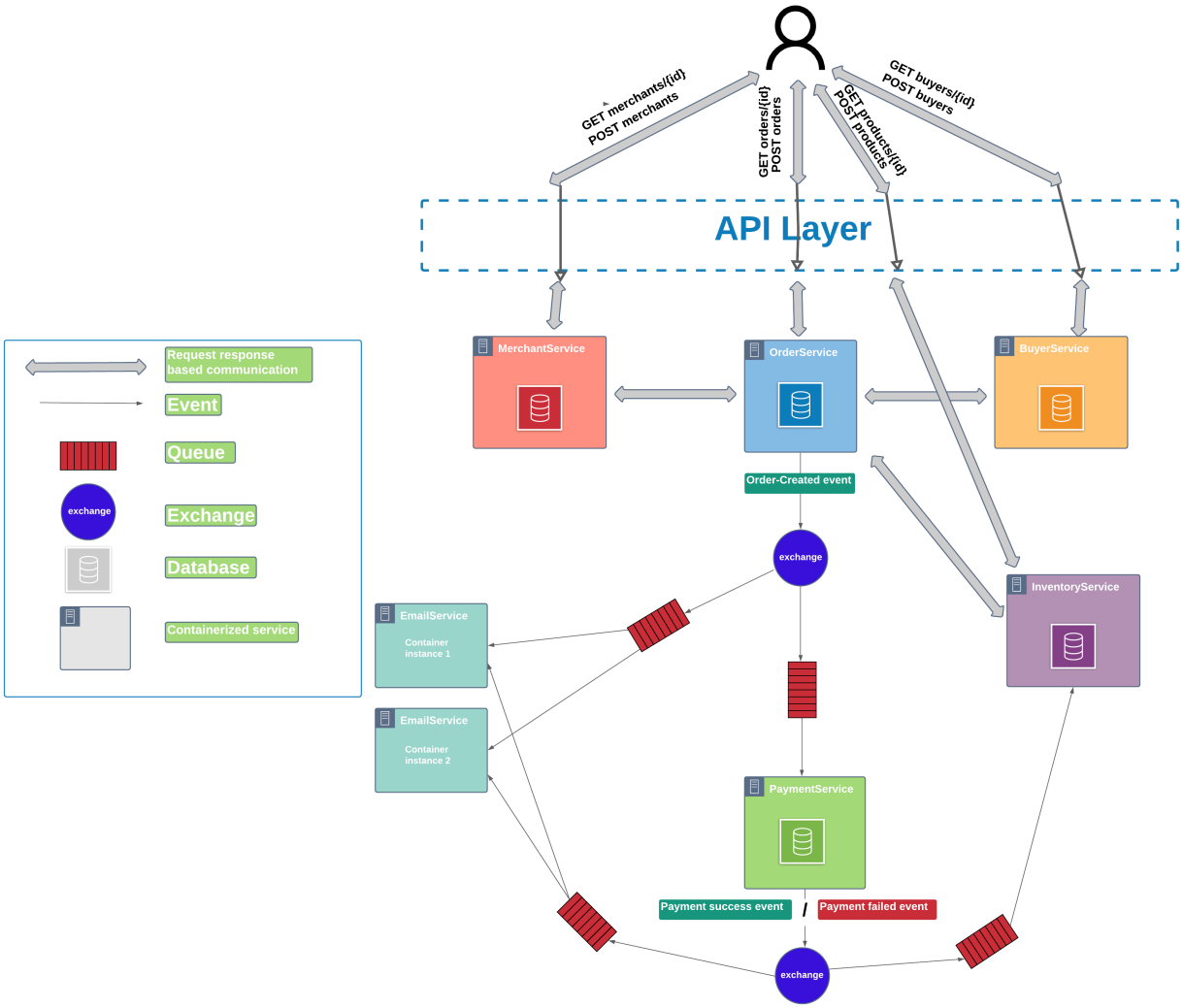
As of now the user of the system directly calls the microservices, add an API-layer between the microservice system and the user.

The API layer should be implemented as REST endpoints, what framework you use is up to your. The Layer should have ther following endpoints:

- `GET api/orders/id`
- `POST api/orders`
- `GET api/merchants/id`
- `POST api/merchants`
- `GET api/buyers/id`
- `POST api/buyers`
- `GET api/products/id`
- `POST api/products`

The endpoints should return and receive the same data as the corresponding endpoints on the microservices themselves.

The API layer should be containerized and it should be part of the docker-compose.yaml file. It should now only be possible to call the API layer locally on port 8000. for example `GET http://localhost:8000/api/orders/5` returns order with id 5. We should no longer be able to call the microservices locally.



4 Grading

1. 20 points for a correct implementation of OrderService
2. 20 points for a correct implementation of InventoryService
3. 20 points for a correct implementation of PaymentService
4. 20 points for a correct implementation of EmailService
5. 10 points for a correct implementation of MerchantService
6. 10 points for a correct implementation of BuyerService
7. 10 bonus points for a correct implementation of the API Layer