Description

The goal of this repository is to implement message routing with topics named as

- my.o
- my.i using Docker.

Functionality

IMED(Intermediate)

- Subscribes for messages from my.o
- Publishes message to my.io

OBSE(Observer)

- Subscribes for all messages within the network, therefore receiving from both my.o and my.i
- Stores the messages into a file

HTTPSERV

- When requested, returns content of the file created by OBSE

Additional notes

- The project uses RabbitMQ as the message broker

Getting started

Running the project

```
docker-compose up --build
```

you should be able to get the system output served by httpserv module at

```
curl localhost:8080
```

Project structure

The project is split into self contained micro services with their of package.json and dependency management. This way they can change independently from each other. This is a bit inconvenient because we need to run yarn or npm install separately in each package. However this is not needed when development happens in the Docker environment.

- Root includes assigment.pdf which includes detailed instructions on the project and our docker-compose config file
- Each aforementioned module is listed under source with their own package.json, env, README and so on

Development

If you want to develop individual files against rabbit, you can initialize the service separately with `docker run -d -p 15672:15672 -p 5672:5672 --name rabbit-test-for-medium rabbitmq:3-management` and access it on the browser at `localhost:15672` with password and username = guest

Using the services is however not supported out of the box without a Docker environment as of now, but this might be added later

Learnings and reflection

In my experience topic based queue communication is a great way to improve the robustness and reliability of a system. For instance what I've used queues for in the past was to ensure that audio buffers were uploaded, processed and saved in a safe manner. This was important, because this datatype represented a non-repeating, unique and irreplaceable piece of information. Whereas in a HTTP based system might fail a request and the data would be lost forever with a queue system (in this case Kue running on Redis) allowed me to

1. Make sure that the queue system received the message, and re-send if necessary.
2. Make sure that the items don't get lost via redundancy of queues and data replication in case of a redis service crash
3. Make sure that items are removed from the queue only after the item has been marked completed In addition most queue systems allow to set both timed events and TTL events, where an event would be fired at, or after some time, and where an item might have a specific time to be consumed before it was removed from the queue or possibly moved to a separate dead-letter queue for later manual or automatic processing. None of these features are baked in HTTP based communication.

What sets RabbitMQ apart for Redis, or for instance AWS SQS, is it's topic based communication, which makes it easy to consume the same events in multiple places and also multiple times if necessary. In addition the documentation and getting started guides of the software are excellent. However even with this, if I had to pick a queue system I would go for SQS in the future, this is because rabbitMQ need to be managed by you, or you have to pay someone to manage it. At this point when you are paying for a service SQS will be a cheaper option. If you decide to self-host, you need to start worrying about redundancy and clustering of the service, which is a pain of itself. In addition some traditional limitations of SQS, such as no FIFO support have been solved already.

As a takeaway RabbitMQ does everything what a queue system needs to do and does it well. However for most uses cases it seems that a better option would be to opt for a hosted service like SQS.

Notes on docker

Up & Down

```
docker-compose up --build -d
```

remove -d if you want to see logs from the instances

when you want to take the service run in root

```
docker-compose down
```

On volumes

Volumes can be shared between containers, there are unnamed and named containers. Containers can be initialized with data or they can be filled with data afterwards. For instance in this project we are sharing the rabbitmq/index.js file with a volume but just initialing an empty data volume to be shared in runtime.

Volumes are stored on the host machine and can , but normally should not, be accessed as sudo

```
$ sudo su
$ cd /var/lib/docker/volumes
$ ls
```

Useful commands

NOTE! /bin/ash for alpine /bin/bash for other linux distros

Ssh into a running container with `docker exec -it <container name> /bin/bash`

Run an existing image in interactive mode `docker run -it <container name> /bin/ash`

Inpect anything `docker inspect <container name>` `docker inspect <image name>`

List available things running containers `docker ps` all containers `docker container ls --all` all images `docker image ls --all` all volumes `docker volume ls`

Remove non essential stuff remove unused containers `docker container prune` remove unused images `docker image prune` remove unused volumes `docker volume prune` if you want to remove absolutely all of one category add the `--all` flag

Sources

https://stackoverflow.com/questions/31746182/docker-compose-wait-for-container-x-before-starting-y https://stackoverflow.com/questions/47710767/what-is-the-alternative-to-condition-form-of-depends-on-in-docker-compose-versio https://stackoverflow.com/questions/57065750/docker-compose-volume-overrides-copy-in-multi-stage-build https://docs.docker.com/config/containers/start-containers-automatically/#use-a-restart-policy https://docs.docker.com/engine/reference/commandline/container_prune/

RabbitMQ

RabbitMQ is one of the most popular open source message brokers including message queuing, topic based communication with included support for data safety. It is used by small to large enterprises, it is highly performant, scalable, lightweight and easy to deploy both on premises and to the cloud.

Publisher

Sends messages to an exchange or directly to a queue

Consumer

Initializes a named or unnamed queue which receives either direct messages from a producer or messages matching the subscribed topic from an exchange

Queue

a client can be part of a queue and receive messages from it.

Receiving messages

How we actually receive a message depends on how we have subscribed, but given that this is how we receive and handle a message

```
channel.consume(queue,((message) => {
    if (message !== null) {
      channel.ack(message) // https://www.rabbitmq.com/confirms.html
      messageCallback(null, message.content.toString())
    }
}))
```

Sending messages

You can send a message to a queue with

```
const options = {
    persistent:true // The message will survive broker restarts provided
it’s in a queue that also survives restarts.
  }
channel.sendToQueue(queue,Buffer.from(message),options)
```

Publishing a message

You can publish a message in an exchange with.

```
channel.publish('exchange-name','', Buffer.from('something to do'))
```

instead of the empty string, you can define a routing key to deliver the message only to queues that are subscribed to the route

Exchange

a queue can be part of an exchange, the exchange can be in different modes.

Fanout

In this mode the exchange sends all messages to all queues

```
// initialize what exchange we are part of
channel.assertExchange('name-of-exchange', 'fanout', {
  durable: false
});

// if you leave the queue name empty, the server generates one on random
// setting exclusive = true scopes the queue to the connection
channel.assertQueue('', { exclusive: true }, (err, q) => {
  // we leave the last value empty so that we will receive all the messages
  channel.bindQueue(q.queue, 'name-of-exchange', '');
})
```

Direct

In this mode we send messages only directly to queues that have bound themselves to listen to the messages:

```
// initialize what exchange we are part of
channel.assertExchange('name-of-exchange', 'direct', {
  durable: false
});

// if you leave the queue name empty, the server generates one on random
channel.assertQueue('', { exclusive: true }, (err, q) => {
  channel.bindQueue(q.queue, 'name-of-exchange', route);
})
```

Topic

Sometimes a simple route is not enough, for instance if we are routing logs to different queues depending on the severity, we might also want to differentiate on the origin. A warning log from a critical system could for instance want to be routed to a queue that normally handles only errors.

Sources

https://www.rabbitmq.com/tutorials/tutorial-one-javascript.html
http://www.squaremobius.net/amqp.node/channel_api.html#channel

TODO

- [ ] And hot code reloading to all scripts
- [ ] Allow development without a Docker environment
- [ ] Consider using Lerna monorepo and changing the modules to be published as NPM packages