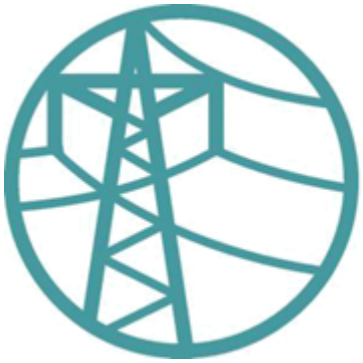


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Usługi sieciowe w biznesie

Projekt

Asynchroniczna komunikacja między serwisami

za pomocą RabbitMQ

4 EF–ZI, L02

Rafał Magryś 161877

Rzeszów, 2021

Wstęp

Celem projektu jest zaprezentowanie prostego asynchronicznego systemu, w którym wymianą informacji będzie zarządzać kolejka. Zastosowanie kolejki jest zupełnie innym podejściem od typowego systemu informatycznego zbudowanego z api i klientów. Jest ono o wiele bardziej wydajne na środowisku produkcyjnym, gdy z systemu korzysta wielu użytkowników. Odpytywanie zwykłego api zapytaniami http z wielu źródeł wiele razy może być bardzo obciążające i można w łatwy sposób spowodować DDoS typu “friendly fire”. Ten problem właśnie rozwiązuje broker wiadomości. Nie ma tutaj typowego podejścia które nie polega na współistnieniu rozmawiających ze sobą procesów tylko, wysłaniu ich i przechowaniu na jakimś agregatorze danych, które zostaną stamtąd odczytane w odpowiednim momencie, np przy zalogowaniu.

Technologie

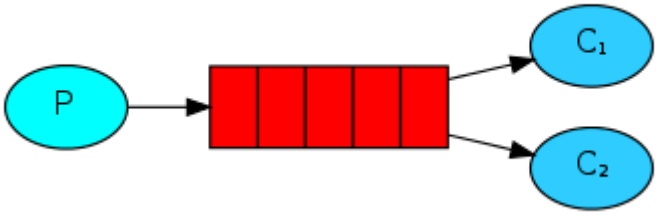
RabbitMQ

RabbitMQ jest wspomnianym wcześniej brokerem wiadomości lub tzw. kolejką. Jest powszechnie używane narzędzie open source. Sami twórcy na swojej stronie się chwalą, że ich technologia została użyta w ponad 45 tysiącach projektach komercyjnych z pozytywnym skutkiem.

Logo RabbitMQ

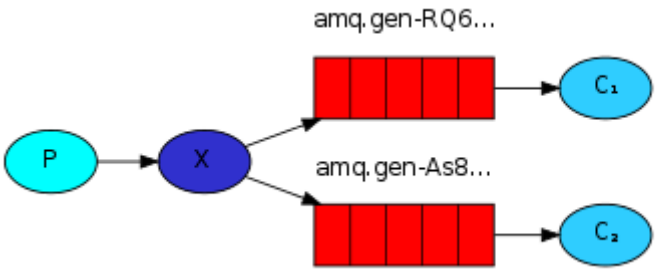
Work Queues

Podstawowymi funkcjami RabbitMQ są tzw “Work Queues”. Polega to na tym, że jeden serwis wysyła dane do kolejki, a serwisy, które nasłuchują zdejmują dane z kolejki w dany sposób, że dane będą równo dostarczone między nimi.



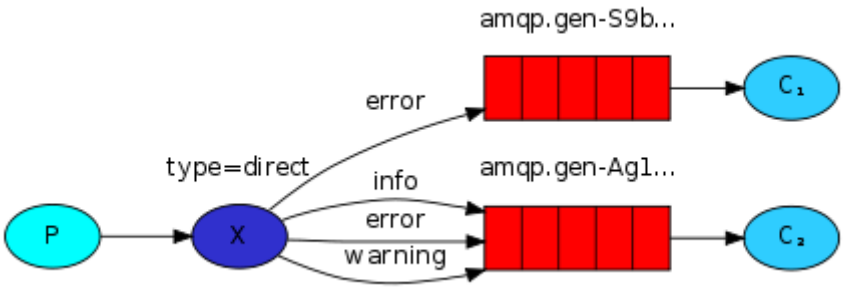
Publish/Subscribe

Jest to kolejna funkcja brokera wiadomości, polega na tym, że jeden serwis jest w stanie generować te same dane dla kilku innych serwisów jednocześnie przy pomocy ‘fanout’



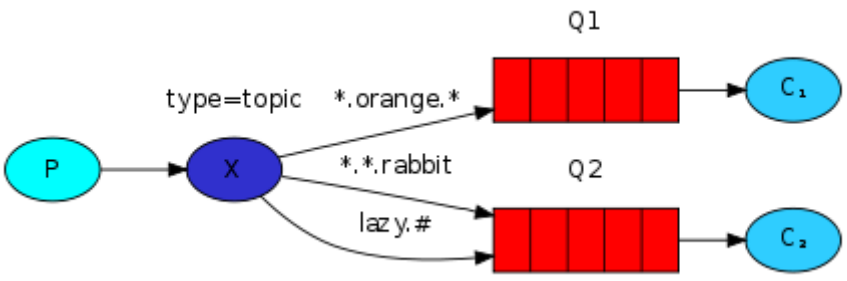
Routing

Tutaj mamy zastosowanie metody publish/subscribe w taki sposób, że możemy określić konkretnie jakie dane powędrują na jaką kolejkę przy pomocy klucza, co poprawia przejrzystość, czytelność jak i wydajność. w tym przypadku na obie kolejki zostaną dodane dane o błędzie, a informacje i ostrzeżenia tylko na kolejkę drugą.



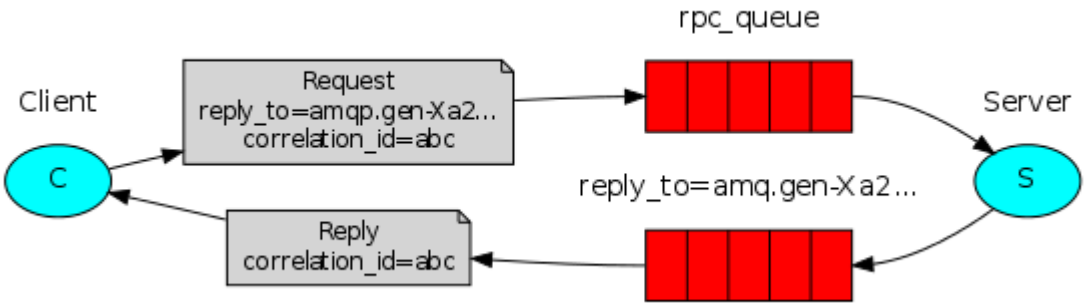
Topics

Zamiast bezpośrednio kluczem jakie informacje mają trafiać na kolejkę można to zrobić przy pomocy “Topic” który jest połączeniem słów oddzielonych kropką, po których moduł (exchange) opowiadający za dodawanie na odpowiednie kolejki, wie gdzie dane mają zostać umieszczone. W tym przypadku na kolejkę pierwszą ląduje wszystko co jest pomarańczowe, a na drugą wszystko co jest królikiem lub jest leniwe.



Remote Procedure Call (RPC)

Jest potrzebne gdy musimy uruchomić jakąś funkcję na zdalnym komputerze i czekać na odpowiedź serwera. Jest to wielce odradzany sposób przez wielu specjalistów, ponieważ generuje ogromną ilość bugów i nieścisłości.



Jako ciekawostkę można dodać, że każdy serwis korzystający z tego samego brokera wiadomości, może być napisany w innym języku programowania, więc daje ogromne możliwości integracji systemów.

Docker

Jest to ważne narzędzie dla każdego programisty i jest niezbędne w wielu rozwiązań chmurowych. Jako jedna z kilku technologii która uruchamia kontenery pozwala na emulowanie części warstwy sprzętowej, dzięki czemu otrzymujemy pełnoprawny mały system operacyjny ze zdefiniowanym procesem lub aplikacją. Ogromną zaletą Dockera jest to, że nie zużywa tyle zasobów co wirtualizacja. Kontenery działają niezależnie od siebie, i by były w stanie się ze sobą komunikować trzeba to określić w ich komendzie/pliku uruchomieniowym. Kolejną zaletą dokera jest to, że nie trzeba

przejmować się kompatybilnością sprzętową, ponieważ docker tworzy swoje własne środowisko.

Pozostałe

JavaScript - skryptowy język programowania, który używam w projekcie

Node.js - środowisko uruchomieniowe JavaScriptu

Reactjs - biblioteka służąca do tworzenia interfejsów użytkownika

TypeScript - podzbiór języka JavaScript pozwalający wprowadzić do niego typy

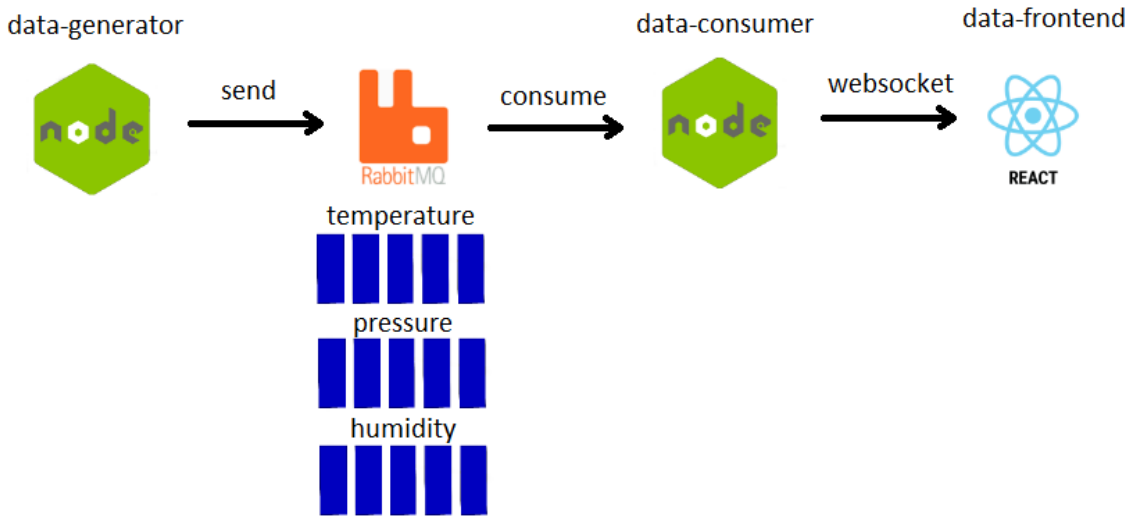
Socket.io - biblioteka JavaScript pozwalająca stworzyć serwer websocket

log4js - biblioteka JavaScript do wypisywania logów

amqplib - biblioteka JavaScript służąca do połączenia z brokerem wiadomości RabbitMQ

Opis działania projektu

Architekturę projektu można w łatwy sposób zrozumieć analizując rysunek poniżej



- serwis data-generator generuje losowe dane temperatury, ciśnienia i wilgotności i w różnych odstępach czasu wysyła je na kolejki z takimi samymi nazwami
- RabbitMQ przechowuje te dane by potem inny proces mógł je odczytać
- serwis data-consumer wyciąga dane z kolejki tworzy z nich obiekt i wysyła poprzez serwer websocket na data-frontend
- data-frontend generuje dane na wykresie

Przykład działania

Każdy serwis opisany powyżej jest w osobnym kontenerze, dla którego został stworzony odpowiedni plik Dockerfile i jest tych plików 3. RabbitMQ nie musi mieć tworzonego własnego pliku dockerowego ponieważ jego obraz jest już dostępny w chmurze na dockerhub i można go łatwo pobrać

```
FROM node:16
WORKDIR /usr/app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
EXPOSE 4000
WORKDIR ./dist
CMD ["node", "index.js"]
```

Dockerfile

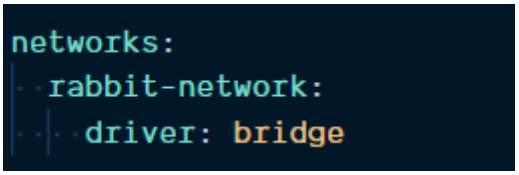
By uruchomić wszystkie obrazy dockerowe za jednym razem użyłem pliku docker-compose.yml w którym dokładnie jest określone jak dany kontener ma się nazywać, jakiej sieci należy oraz jakie porty obsługuje.

```
services:
  rabbitmq:
    container_name: "rabbitmq"
    tty: true
    image: "bitnami/rabbitmq:latest"
    ports:
      - 15672:15672
      - 15671:15671
      - 5672:5672
    volumes:
      - ./config/rabbitmq/rabbitmq.config:/etc/rabbitmq/rabbitmq.config:rw
      - ./config/rabbitmq/definitions.json:/etc/rabbitmq/definitions.json:rw
      - ./config/rabbitmq/enabled_plugins:/etc/rabbitmq/enabled_plugins
    networks:
      - rabbit-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:15672"]
      interval: 30s
      timeout: 10s
      retries: 5
```

Fragment docker-compose dla konfiguracji RabbitMQ

By system poprawnie się uruchomił należało ustawić zależność uruchamiania obrazów dockingowych od uruchomienia kolejki, lecz to nie wystarczyło. Należało jeszcze dodać fragment kodu który uruchamiał pozostałe kontenery dopiero gdy broker zwrócił poprawny “healthcheck”

By kontenery mogły się komunikować ze sobą muszą posiadać jakieś adresy ip. Należało więc stworzyć sieć wewnętrzną między dokerami ze sterownikiem i dodać ją do każdego z nich

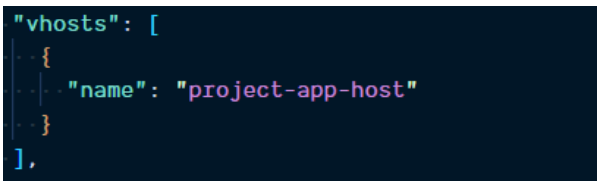


Sieć kontenerów

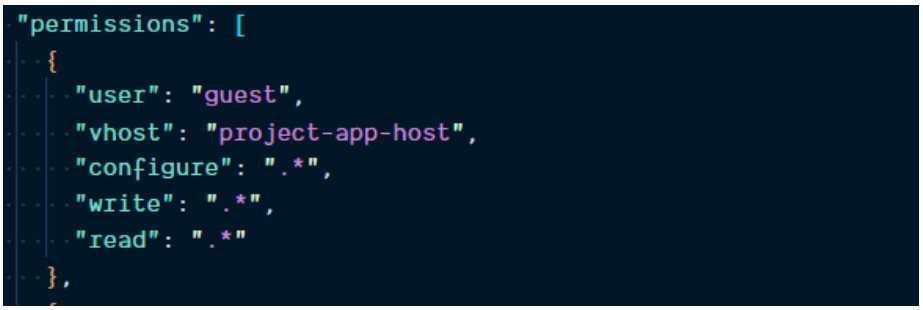
Dzięki temu zamiast localhost kontenery odnajdują się na podstawie nazwy serwisu i mogą się komunikować.

RabbitMQ potrzebował dla działania zdefiniowania konfiguracji w plikach json, w których zawarte są konta użytkowników (serwisów) , wirtualnego hosta, oraz, udzielenia dostępu, co potem dodajemy do kontenera używając przypisu **volumes**

Przykładowy użytkownik



Wirtualny host



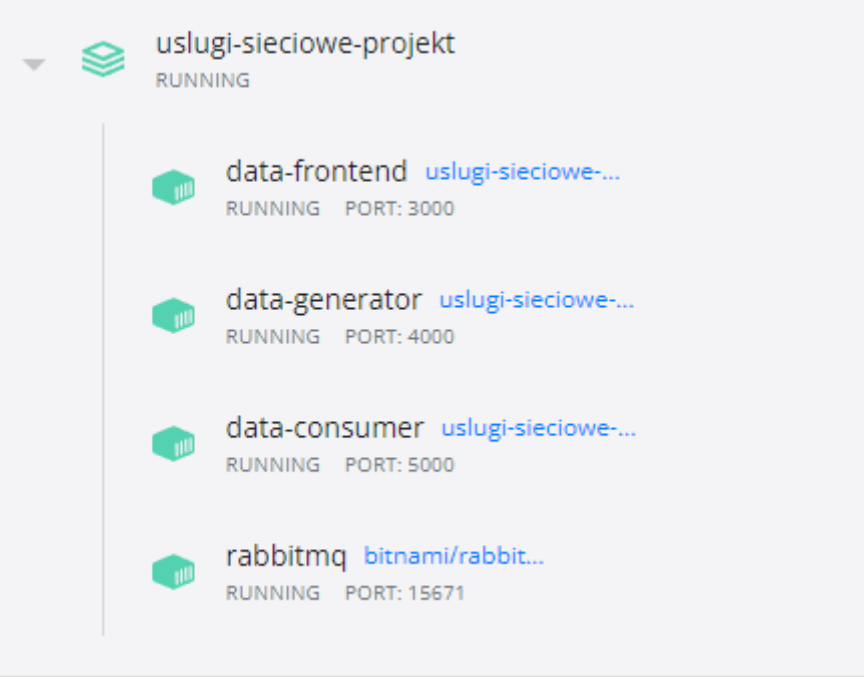
Ustawienia dostępu

By uruchomić cały projekt wystarczy wpisać w konsolę komendę

docker-compose up

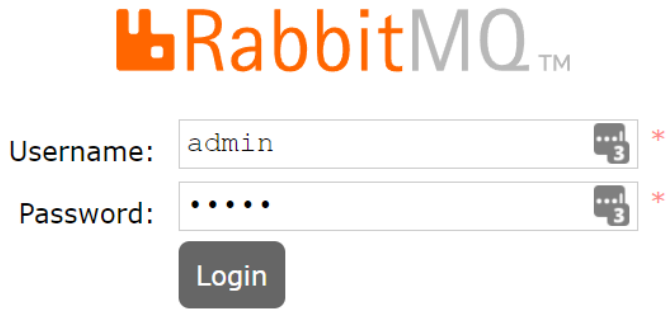
spowoduje ona podniesienie obrazów, a wcześniej pobranie i stworzenie jeśli ich nie ma.

po uruchomieniu systemu w kliencie desktopowym Dockera możemy zobaczyć uruchomione 4 kontenery, jest to również widoczne z Visual Studio Code przy pobraniu odpowiedniego rozszerzenia.



Widok kontenerów z aplikacji

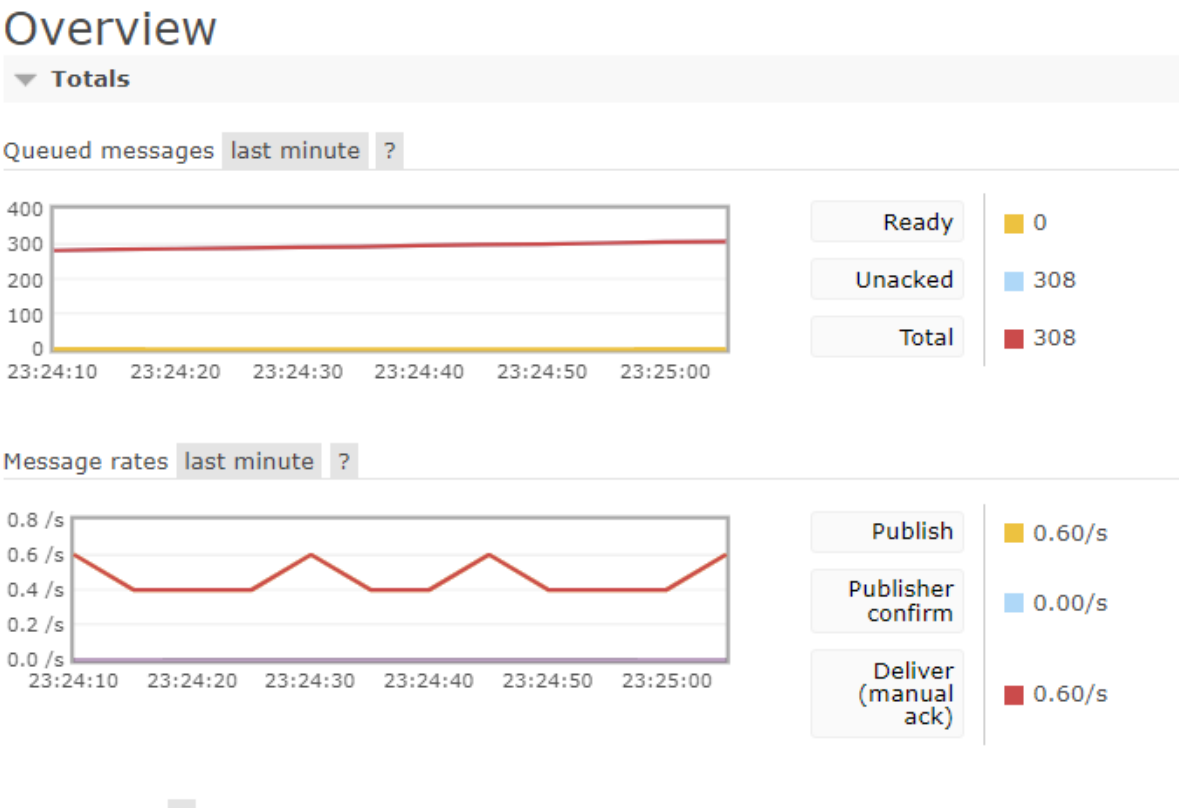
Można teraz wpisując adres http://localhost:15672/#/connections zalogować się do brokera i zobaczyć jak wygląda system



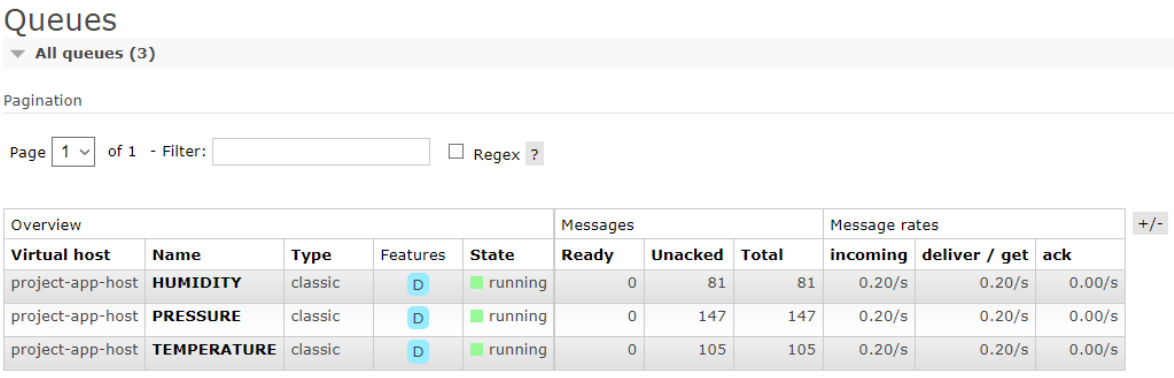
Po zalogowaniu w zakładce connections mozna prześledzić jakie serwisy są zalogowane

Overview				Details			Network	
Virtual host	**Name**	**User name**	**State**	**SSL / TLS**	**Protocol**	**Channels**	**From client**	**To client**
project-app-host	**172.23.0.3:59734** ?	dataConsumer	running	○	AMQP 0-9-1	1	0 B/s	128 iB/s
project-app-host	**172.23.0.4:59768** ?	dataGenerator	running	○	AMQP 0-9-1	1	106 iB/s	0 B/s

Możemy też śledzić jak wygląda stan danych na kolejkach



Oraz że kolejki zostały utworzone i przesyłają dane



Overview					Messages			Message rates			+/-
Virtual host	**Name**	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**	
project-app-host	**HUMIDITY**	classic	D	running	0	81	81	0.20/s	0.20/s	0.00/s	
project-app-host	**PRESSURE**	classic	D	running	0	147	147	0.20/s	0.20/s	0.00/s	
project-app-host	**TEMPERATURE**	classic	D	running	0	105	105	0.20/s	0.20/s	0.00/s	

RabbitMQ poza kolejkami udostępnia jeszcze api, które pozwala ustawiać i zwracać określone informacje o systemie, takie jak np użytkownicy, polityki, kanały itd.

Serwis generujący dane łączy się do kolejki w następujący sposób:

Najpierw trzeba nawiązać połączenie poprzez bibliotekę i stworzyć kanał

```
private async setupConnection() {
    this.connection = await amqp.connect(this.url);
    this.channel = await this.connection.createChannel();
}
```

adres url buduje się podając nazwę użytkownika, hasło, hosta, port i wirtualnego hosta kolejki

```
constructor() {
    this.url = `amqp://${RMQ_USER}:${RMQ_PASSWORD}@${RMQ_HOST}:${RMQ_PORT}/${RMQ_VIRTUAL_HOST}`;
}
```

gdy to się wykona można przystąpić do tworzenia lub dołączenia do kolejki jeśli już istnieją

```
private async createQueues(): Promise<Array<Replies.AssertQueue>> {
    const humQuePromise = await this.channel.assertQueue(Types.HUMIDITY);
    const pressQuePromise = await this.channel.assertQueue(Types.PRESSURE);
    const tempQuePromise = await this.channel.assertQueue(Types.TEMPERATURE);
    return Promise.all([humQuePromise, pressQuePromise, tempQuePromise]);
}
```

Gdy wszystko zostanie wykonane można przejść do generowania danych które powstają w taki sposób, że tworzy się obiekt z danymi które przyjmują losowe wartości w pewnym przedziale i są wysyłane na kolejkę przy użyciu **sendToQueue**

```
private generateData(): void {
  setInterval(() => {
    const temperatureData: Data = this.createRandomData(Types.TEMPERATURE);
    log.log(Types.TEMPERATURE, " sent: ", JSON.stringify(temperatureData));
    this.channel.sendToQueue(
      Types.TEMPERATURE,
      Buffer.from(JSON.stringify(temperatureData))
    );
  }, 7000);
```

Dodatkowo dodany jest log by móc zobaczyć jakie dane zostają wysyłane w konsoli

```
[2022-01-05T22:37:04.536] [INFO] default -  sent:  {"id":"1a71e438-1f77-4660-8e5b-b5c6b81ae13c","value":"20","type":"HUMIDITY","timestamp":1641422224536,"unit":"%"}
[2022-01-05T22:37:05.757] [INFO] default -  sent:  {"id":"8c600c2e-139b-46ba-9535-46edde89ecb7","value":"956","type":"PRESSURE","timestamp":1641422225757,"unit":"hPa"}
[2022-01-05T22:37:07.669] [INFO] default -  sent:  {"id":"346d3fdd-fb58-4523-95d7-97f73b101c97","value":"24.9","type":"TEMPERATURE","timestamp":1641422227669,"unit":"C"}
[2022-01-05T22:37:10.757] [INFO] default -  sent:  {"id":"ccd1dced-c1a8-493c-8621-526e1d2a8b04","value":"976","type":"PRESSURE","timestamp":1641422230757,"unit":"hPa"}
[2022-01-05T22:37:13.540] [INFO] default -  sent:  {"id":"0faa21be-f1fa-4510-82bf-048740feae5a","value":"37","type":"HUMIDITY","timestamp":1641422233540,"unit":"%"}
[2022-01-05T22:37:14.670] [INFO] default -  sent:  {"id":"e928554b-b4c0-42cb-a48f-986af09e4a57","value":"25.4","type":"TEMPERATURE","timestamp":1641422234670,"unit":"C"}
[2022-01-05T22:37:15.759] [INFO] default -  sent:  {"id":"daf18fe7-df0f-44ef-94ae-f3d160121e96","value":"953","type":"PRESSURE","timestamp":1641422235758,"unit":"hPa"}
```

serwis odczytujące dane z kolejki działa podobnie, więc cały proces tworzenia połączenia, kanału i kolejek wygląda dokładnie tak samo, różni się jedynie wywoływaną funkcją przy pobieraniu informacji z kolejki

```
public consumeData(sendingCallback: CallableFunction) {
  this.channel.consume(Types.HUMIDITY, (message) => {
    log.info(message.content.toString());
    const data: Data = JSON.parse(message.content.toString());
    sendingCallback(data.type, data);
  });
```

jest tu dodatkowo użyta funkcja sendingCallback, jest to funkcja callback przekazana jako argument. W tym przypadku jest to funkcja wysyłania danych przez websocket

połączenie do socketu również wygląda podobnie. Podaje się adres url, i czeka na połączenie, zarówno na serwerze jak i kliencie. Metody emit() i on() służą do podstawowej wymiany informacji.

```
public async init() {
    this.io.on("connection", (socket: Socket) => {
        log.info("Connection by socket.io set");
        socket.emit("Successfully connected to server");

        socket.on("disconnect", () => {
            console.log("disconnected");
        });
    });
    log.info(`Server listening on port ${this.port}`);
    this.httpServer.listen(this.port);
}

public sendToChannel = (event: string, eventData: Data): void => {
    this.io.emit(event, eventData);
};
```

funkcja klasy która agreguje websocket i kolejkę i wysyła odebrane dane przez callback wygląda następująco

```
public runDataConsumingAndSending() {
    this.dataConsumer.consumeData(this.socketController.sendToChannel);
}
```

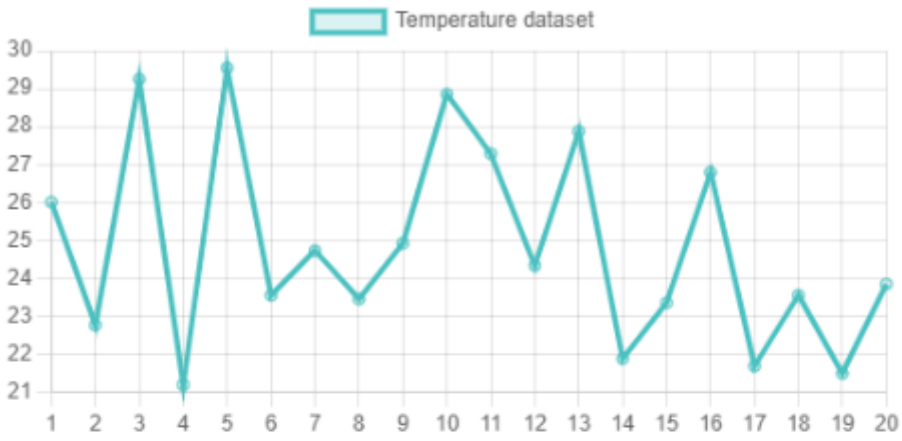
Po wejściu w konsolę obrazu dockerowego, możemy zobaczyć jak przychodzą informacje z kolejki

```
[2022-01-05T23:14:42.357] [INFO] default - {"id":"9db4c216-6cae-4c70-9416-02f6df53aa3a","value":"27.0","type":"TEMPERATURE","timestamp":1641424468334,"unit":"C"}
[2022-01-05T23:14:42.357] [INFO] default - {"id":"f45db2f9-e4d8-42ab-8aa7-2e50e8387828","value":"24.1","type":"TEMPERATURE","timestamp":1641424475334,"unit":"C"}
[2022-01-05T23:14:42.357] [INFO] default - {"id":"8e7eab6e-0ad4-4ce7-82bf-b6841d3eb867","value":"21.5","type":"TEMPERATURE","timestamp":1641424482334,"unit":"C"}
[2022-01-05T23:14:44.008] [INFO] default - {"id":"ffe934da-e733-4421-aff1-0f2a51ba4de3","value":"34","type":"HUMIDITY","timestamp":1641424484007,"unit":"%"}
[2022-01-05T23:14:46.703] [INFO] default - {"id":"b7c0a079-41d2-4f6a-ad35-1a3328cfabcb","value":"975","type":"PRESSURE","timestamp":1641424486702,"unit":"hPa"}
```

Po wejściu na stronę internetową http://localhost:3000

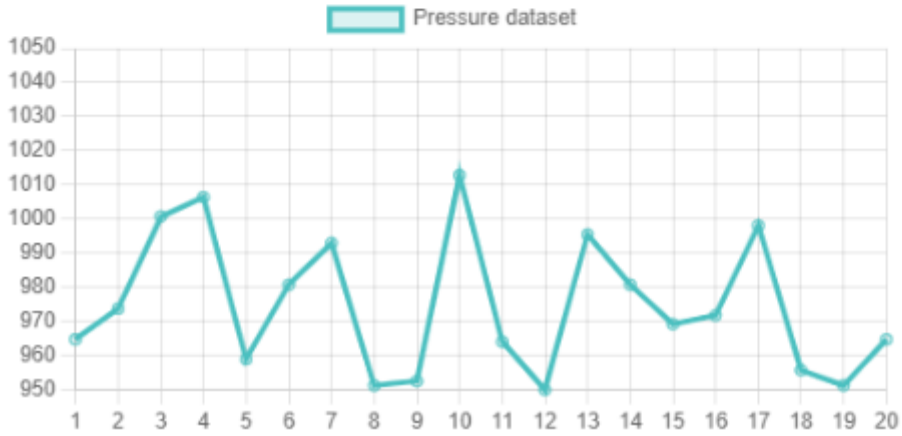
zobaczymy po chwili jak dane pojawiają się powoli na wykresie w różnych odstępach czasowych

Temperature Line Chart



Humidity Line Chart

Pressure Line Chart



Podsumowanie

Celem projektu było pokazanie w prosty sposób zastosowania brokera RabbitMQ oraz asynchronicznej komunikacji między serwisami. Projekt w prawidłowy sposób przesyłał dane między nimi. Jednakże gdyby serwis odczytujący dane się wyłączył, to kolejka rosłaby aż brakłoby pamięci, co jest ogromną wadą brokerów wiadomości i trzeba o tym pamiętać

Brokery dają wiele możliwości i są bardzo popularne przy projektowaniu systemów, gdy potrzebuje się największej dostępności, oraz optymalizacji przesyłania danych, każda osoba która projektuje złożone systemy powinna rozważyć użycie brokerów

Linki

https://github.com/rmagrys/network-services-project

https://www.rabbitmq.com

https://hub.docker.com/r/bitnami/rabbitmq