Exercise 9

Event driven approaches with NATS

Prior Knowledge
Previous exercises

Objectives
Understanding queuing and pub-sub models

Software Requirements

- Typescript, Node, Yarn, etc
- NATS
- Python

Overview
We will update the Purchase service to publish every new order to a NATS server. We will use both a CLI and a Python client to subscribe to this feed.

Steps

1. NATS is a high performance broker that supports a variety of event driven scenarios including pub/sub, request-reply and streaming. It is very simple and lightweight, written in Go, and has a very effective and innovative security model. For more information take a look at https://nats.io/

2. Before we write any code, lets interact with nats in a purely CLI based way. Firstly, let's use docker to run nats:

   docker run -p 4222:4222 nats

3. I have previously installed go and the nats publisher and subscriber CLI tools.

   If you want to do this *elsewhere* you can do:

   ```
   sudo apt install golang
   mkdir go
   go get github.com/nats-io/go-nats-examples/tools/nats-pub
   go get github.com/nats-io/go-nats-examples/tools/nats-sub
   ```

4. Open another 3 terminal windows or tabs.

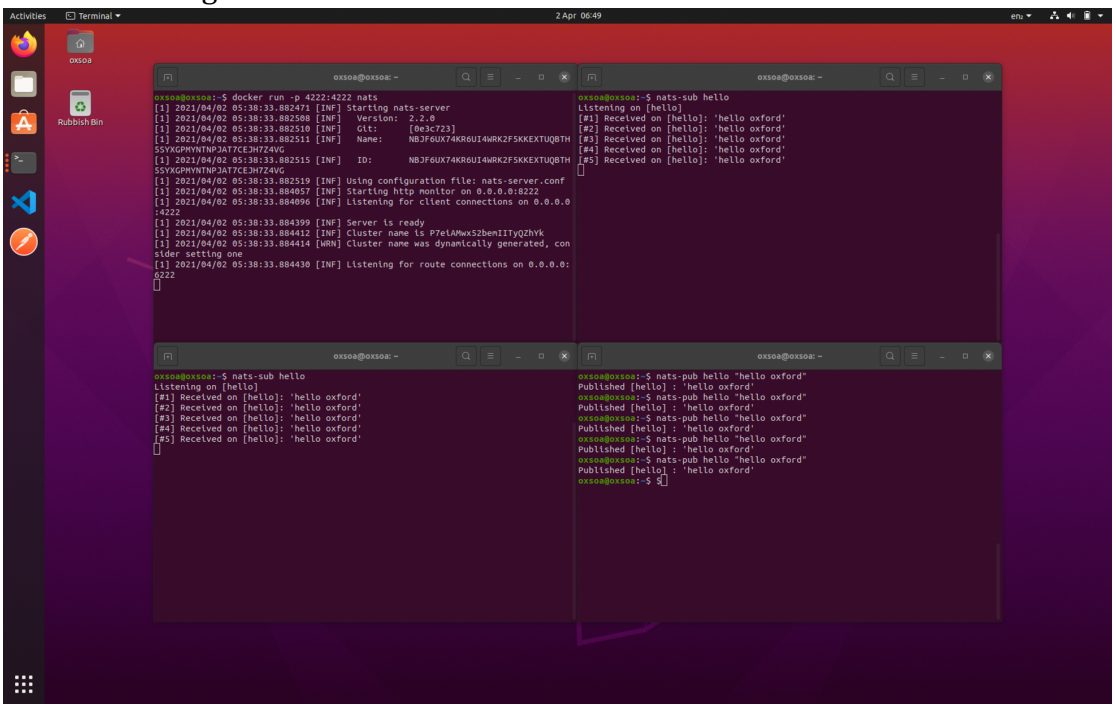
5. In two of them type:

   nats-sub hello

6. In the third type

   nats-pub hello “Hello Oxford”

7. You can repeat the publish step a few times.
   Your screen might look like this:



   As you can see, the CLI clients connect to localhost:4222 by default, which is what our docker container instance is providing.

   This simple steps shows two key aspects of an event driven architecture:
   - decoupling the publishers and subscribers
   - fan out (multiple subscribers to a single topic/subject)

8. Leave the nats docker instance running but you can kill the subscribers.

9. We are going to add extra code to our PurchaseOrder app to publish events to a nats broker, and then subscribe to them elsewhere. For example, it could be very useful to build a simple sales intelligence tool for the organisation by collecting data about all the orders (and cancellations) into a view for the sales director.

An alternative approach would be to tie the sales intelligence dashboard directly to the postgres database we are already using.

As you go through this exercise try to think about:

- Why does the event-based approach have advantages over the database connection approach?
- What are the disadvantages?

10. Back to the code. You can either start from the existing purchase app or clone purchase-complete and start from there.

    If you are cloning purchase complete:

    ```
    git clone https://github.com/pzfreo/purchase-complete.git
    cd purchase-complete
    yarn install
    ```

    Firstly, add the client library we need to interact with nats:

    ```
    yarn add ts-nats
    ```

    ```
    yarn add v1.22.10
    [1/4] Resolving packages...
    warning ts-nats@1.2.15: nats now offers the async functionality
    directly
    [2/4] Fetching packages...
    info fsevents@2.3.2: The platform "linux" is incompatible with this
    module.
    info "fsevents@2.3.2" is an optional dependency and failed
    compatibility check. Excluding it from installation.
    [3/4] Linking dependencies...
    [4/4] Building fresh packages...

    success Saved lockfile.
    success Saved 4 new dependencies.
    info Direct dependencies
    └─ ts-nats@1.2.15
    info All dependencies
    ├─ nuid@1.1.4
    ├─ ts-nats@1.2.15
    ├─ ts-nkeys@1.0.16
    └─ tweetnacl@1.0.3
    Done in 4.71s.
    ```

11. Create a new file:

    ```
    touch src/publish.ts
    ```

12. Edit the file:

    ```
    code .
    ```

13. Here is the code you need to use. It is available from http://freo.me/nats-publish-ts

```
import { PurchaseOrder } from "./purchaseOrder";
import { connect, Client }  from "ts-nats";

let nats = null;

async function getConnection() : Promise<Client> {
    if (nats) return nats;
    const NATS_SERVER = process.env.NATS_SERVER || "localhost";
    nats = await connect({url: NATS_SERVER });
    return nats;
}

export async function publishCreate(p:PurchaseOrder):Promise<void> {
    const connection = await getConnection()
    connection.publish("po.create", JSON.stringify(p) );
}
```

14. This is basically repeating the singleton pattern to create a connection and then the key line is "connection.publish".

    Note that we are using environment variables to connect to the nats server.

    The subject (topic) we are using is `po.create`

15. Now edit your poService.ts to publish the created object once it is saved to the database:

```
public async create(poCreationParams: POCreationParams): Promise<PurchaseOrder> {

  try {
    const repo = await getRepository();
    const po : PurchaseOrder = await repo.create(poCreationParams);
    await repo.save(po);
    publishCreate(po);
    return po;
  } catch (error) {
    console.error(error);
    throw new Error("invalid input");
  }
}
```

16. We will fix up the docker-compose in a minute, but first we will run this in "local" mode.

    Make sure any docker compose stacks are not running.

    Start postgres:

    ```
    ./start-postgres.sh
    ```

17. In a new window, set up a nats subscriber to subscribe to the whole po subject space:

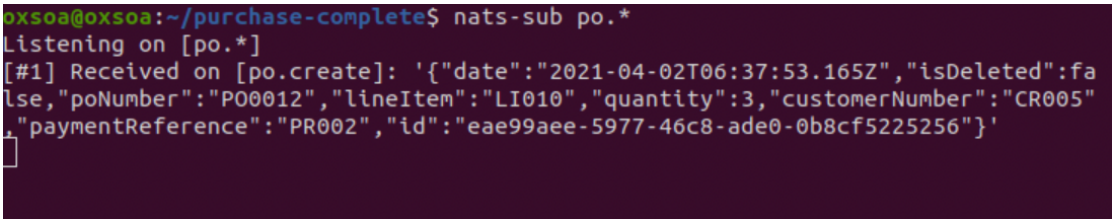
    ```
    nats-sub po.*
    ```

18. Start your app:

    ```
    yarn start
    ```

19. Use newman to run the test collection as before.

20. You should now see po.create events happening in the subscriber window:



21. Let's get the docker-compose file to make all this happen.

    First stop the postgres and nats container instances and your local purchase app instance.

22. Now we need to add some lines into docker-compose.yaml

```
1   version: '3'
2   services:
3     purchase:
4       image:
5         pzfreo/purchase-ts:0.0.1
6       build:
7         context: .
8         dockerfile: Dockerfile
9       depends_on:
10        - postgres
11        - nats
12      ports:
13        - "8000:8000"
14      restart: unless-stopped
15      command: ["./wait-for-it.sh", "postgres:5432", "--","npm", "run","start"]
16      networks:
17        - backend
18      environment:
19        # DEBUG: "*:*"
20        DBHOST: "postgres"
21        DBUSER: "postgres"
22        DBDATABASE: "postgres"
23        DBPASSWORD: "mypass"
24        DBPORT: 5432
25        NATS_SERVER: "nats"
26        PORT: 8000
27    postgres:
28      image: postgres
29      restart: always
30      environment:
31        POSTGRES_USER: "postgres"
32        POSTGRES_PASSWORD: "mypass"
33        POSTGRES_DATABASE: "postgres"
34      networks:
35        - backend
36    nats:
37      image: nats
38      ports:
39        - 4222:4222
40      networks:
41        - backend
42  networks:
43    backend:
44      driver: bridge
45
```

You can see the changes highlighted in green.

On line 11, you can see that we have added the dependency between the purchase service and the nats service.

On line 25, we configure the environment so the publish code can find the nats service.

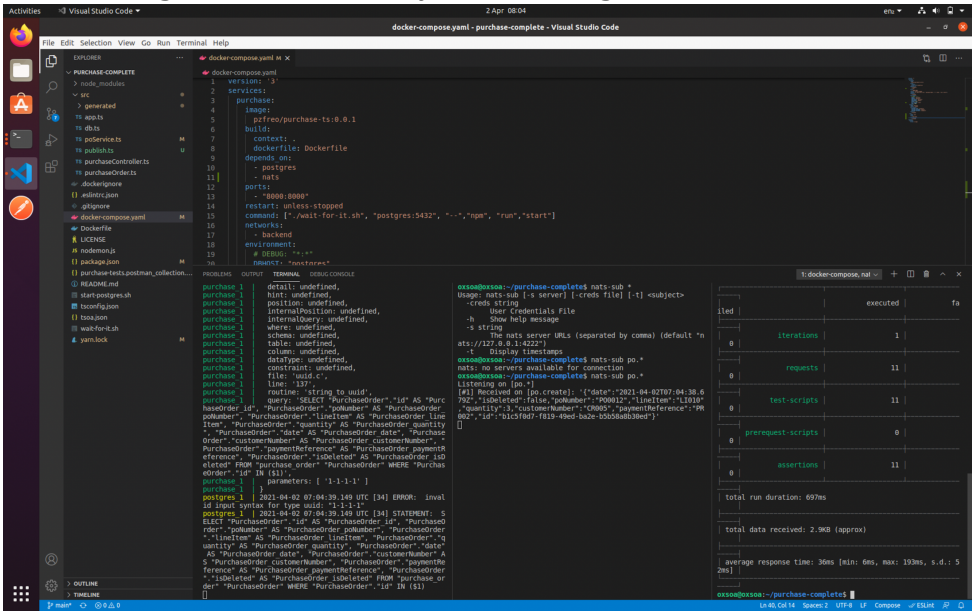
On lines 36-41 we configure the nats container instance, including setting up the backend network but also exposing port 4222 so we can connect from outside the composition.

23. Notice we didn't configure wait-for-it to worry about nats. This is purely pragmatic - nats starts up much much faster than postgres, so by the time postgres is up, nats will be working.

24. Start the docker compose:

    docker-compose up --build

25. Test it out again. You can see my test working here:



26. Let's now build a simple python client to subscribe to the feed of data.

27. Firstly, create a directory for your code and install the nats-python client:

```
pip install nats-python
mkdir ~/pnats
cd ~/pnats
```

28. Note there are two clients for nats in Python - the synchronous one we are using (https://github.com/Gr1N/nats-python) and a more clever (but more complex) async one (https://github.com/nats-io/nats.py)

29. Create a new python program:

```
code subs.py
```

30. And use this code (available here: http://freo.me/nats-py-subs):

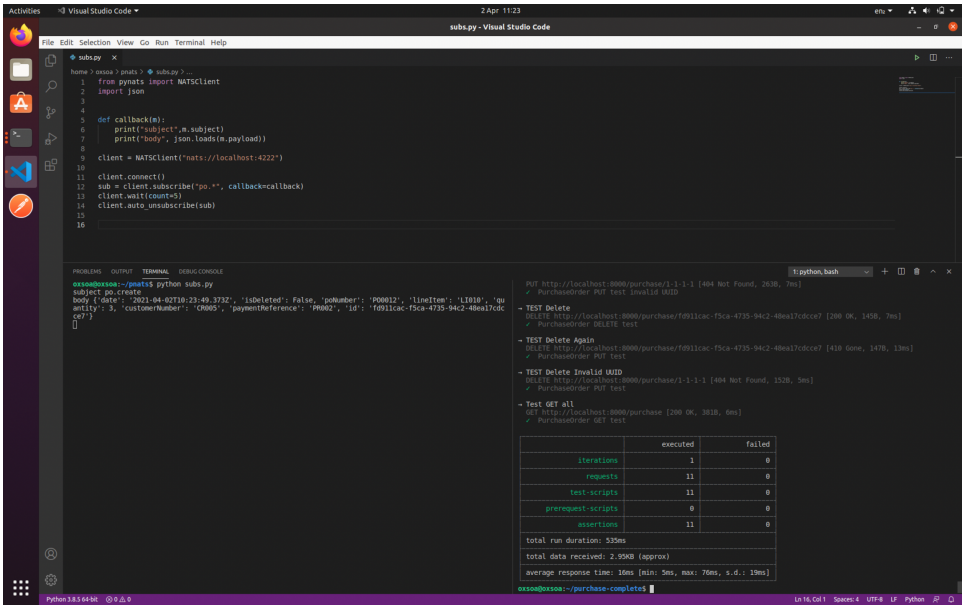
```
1   from pynats import NATSClient
2   import json
3
4   def callback(m):
5       print("subject",m.subject)
6       print("body", json.loads(m.payload))
7
8   client = NATSClient("nats://localhost:4222")
9
10  client.connect()
11  sub = client.subscribe("po.*", callback=callback)
12  client.wait(count=5)
13  client.auto_unsubscribe(sub)
```

    It should be fairly self explanatory!

31. Save and run:

    ```
    python subs.py
    ```

    Here is my running version (note that I have docker-compose running in the background).



32. Subscribe from the CLI and verify that you can subscribe from multiple places.
33. That's the main lab done!

Extensions:

1. Add behaviour to send nats events for delete and update as well.

2. Take a look at nats Jetstream, especially this page
   https://docs.nats.io/jetstream/concepts

3. Take a look at the NATS cloud service - ngs:
   https://synadia.com/ngs/signup