# Instructions

This week the point is to add the communication to the project.

## Trivia

You'll be receiving data from the robot and your job will be to process, and save it in the database using the models from the previous week. There is a slight chance, their sensors fail, in which case a specific message is published on a dedicated channel. Faulty sensors will recover after unspecified time, sending relevant info accordingly.

## Sensor communication

All sensors send data via mqtt under the `sensors` topic; connection details & number of sensors will be provided separately.

The basic data format is a pseudo-json string (following `{'data': <str>}` or `{'info': <str>}` formats), e.g.: `{'data': '41D8AC346C2F4A4D40495412306E03594031DDF35233DB02'}` (received from the `sensors/SNR05/location` topic)

Sensors are differenciated by their `sensor_id`, consisting of `SNR` prefix, concatenated with zero-padded, enumerated field (ranging 01..99), e.g. `SNR01`, `SNR99`, or `SNR05`.

### Data format

The sensors send data via the three distinct topics ( `.../location`, `.../telemetry`, and `.../fault_log` ). Data is represented using raw bytes formatted in hexadecimal; fields are concatenated, check specific field lengths

in respective "example" secions below (the field lengths are constant, padded with `0`'s). All data fields are of `double` type unless stated otherwise, the byte order used is big-endian.

**Examples**

`sensors/<sensor_id>/location`

```
41D8AC346C2F4A4D 40495412306E0359 4031DDF35233DB02
timestamp        latitiude        longitude
```

`sensors/<sensor_id>/telemetry`

```
3DD252FB33ACD841 34             11                03DC
timestamp        humidity (int) temperature (int) pressure (int)
```

`sensors/<sensor_id>/fault_log`

```
'fault_detected'
OR
'fault_recovered'
```

Latter uses the second format ( `{'info': <str>}` ).

# Additional information

Due to the asynchronous nature of the robot communication, you will have to use celery for message parsing. Internally, celery requires message backend/queue - in our case it will be redis. Both need to be configured in the docker-compose file. Because this time you'll be dealing with sensitive data - passwords for mqtt broker, you must start your journey with docker compose environmental variables (so-called env). Additionally, we suggest using a

standalone mqtt client to test the connection (MQTTExporer or MQTTBox to name a few).

# Tasks

1. Setup env for docker compose and add environmental variables to it - create .env.template file with empty variables inside and add it to the repo, then copy .env.template and rename it to .env and fill in missing information. .env file must be in .gitignore file, as sensitive data should never be pushed to the repository. Variables that are required:
    1. mqtt client IP
    2. mqtt client port
    3. mqtt client login
    4. mqtt client password
    5. django database name
    6. django database username
    7. django database password
    8. django database host
    9. django database port
    10. django secret key
2. Add celery to docker compose
3. Add redis to docker compose
4. Create MQTT consumer, that will be handling communication described in the previous section. Consumer must be run as a separate container inside docker compose.
5. Consumer must use celery tasks for parsing and saving messages from robots. (8 in useful links)
6. All messages related to telemetry and location must be saved in appropriate models.

# Useful links

1. https://docs.celeryq.dev/en/stable/django/first-steps-with-django.html
2. https://tamerlan.dev/message-queues-with-celery-redis-and-django/
3. https://pypi.org/project/paho-mqtt/
4. http://www.steves-internet-guide.com/into-mqtt-python-client/
5. http://www.steves-internet-guide.com/mqtt/
6. https://docs.docker.com/compose/environment-variables/
7. http://www.steves-internet-guide.com/mqtt-python-callbacks/
8. https://stackoverflow.com/a/19426755