Waste Incinerator Service

Sprint info

Sprint name	Sprint 3
Previous sprint	Sprint 2
QAK model	sprint3.qak, monitoringdevice.qak
Developed by	Alessio Benenati Giulia Fattori
Repo Site	WasteIncineratorService

Sprint Starting Condition and Goals

In the previous sprint, we focused on the MonitoringDevice and how to connect its physical components to our system.

In this sprint, we aim to connect our fully working system to a web interface to show users all the steps taken by OpRobot and let them interact directly with it.

On top of that, we will dockerize all of our sprints in different containers for easier activation.

Additional Requirements

In the previous sprint review, to test the virtual environment application more easily, the client asked to add two buttons to the user interface, one to increase the number of RP in WasteStorage, the other to free AshStorage.

PROF The client also asked to search for alternative ways to handle the communication between the MonitoringDevice and the WIS systems in order to achieve a higher degree of decoupling between the components and technological independence from QAK.

Problem Analysis

Graphical User Interface

Following the system requirements, a **ServiceStatusGui** is needed to display and update the status of two storages, the incinerator, and the OpRobot.

Additionally, it must allow the user to increase the number of RP in WasteStorage and empty the AshStorage completely.

Therefore, the GUI needs to both receive and send information to the qak system, requiring a common

communication protocol for real-time data exchange.

Initially, integrating the backend logic for serving the GUI directly into the WIS Project might seem viable.

However, this approach would violate the Single Responsibility Principle (SRP).

A more appropriate solution is to create a separate component, the GUI_Backend, which mediates communication between the user and the WIS.

This introduces new challenges regarding the interaction between the Backend, the WIS, and the User Interface.

To address these, we chose to adopt MQTT for communication between the Backend and WIS, while the GUI is developed as a React application that interacts with the Backend via WebSockets.

MD-WIS and WIS-Backend Communication

All the previous sprints used the qak function *updateResource* to emit updates that are then read by the observers; this method introduces quite a strong coupling between the different components of the system.

Another possibility is using a publish/subscribe protocol, like **MQTT**, that introduces an intermediary between the communicating qak actors and grants a **higher degree of separation** throughout the entire system, as well as more robust communication.
For this reasons we decided to switch all extra-context communication (MD-WIS and WIS-Backend) to MQTT

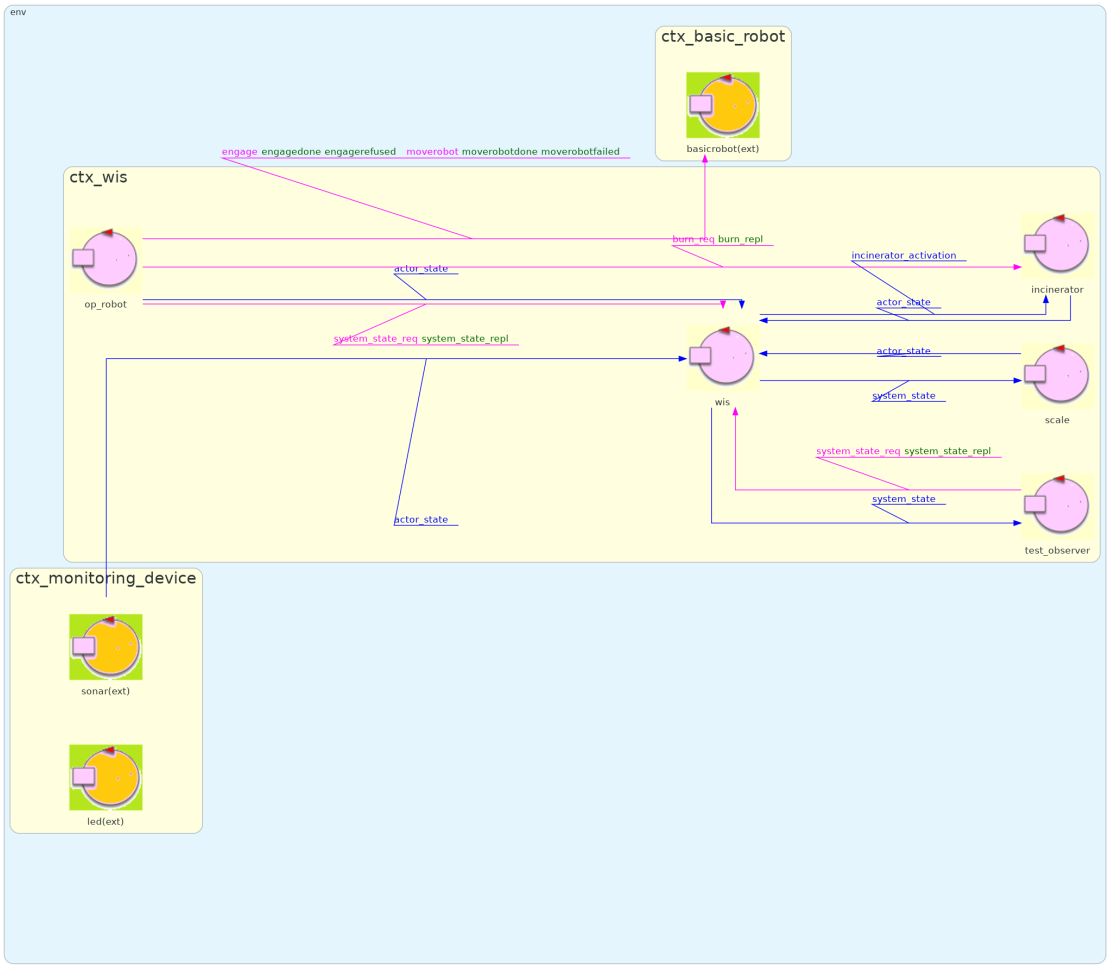
Project

Project Architecture

At the end of the sprint 2 the system architecture was the following:

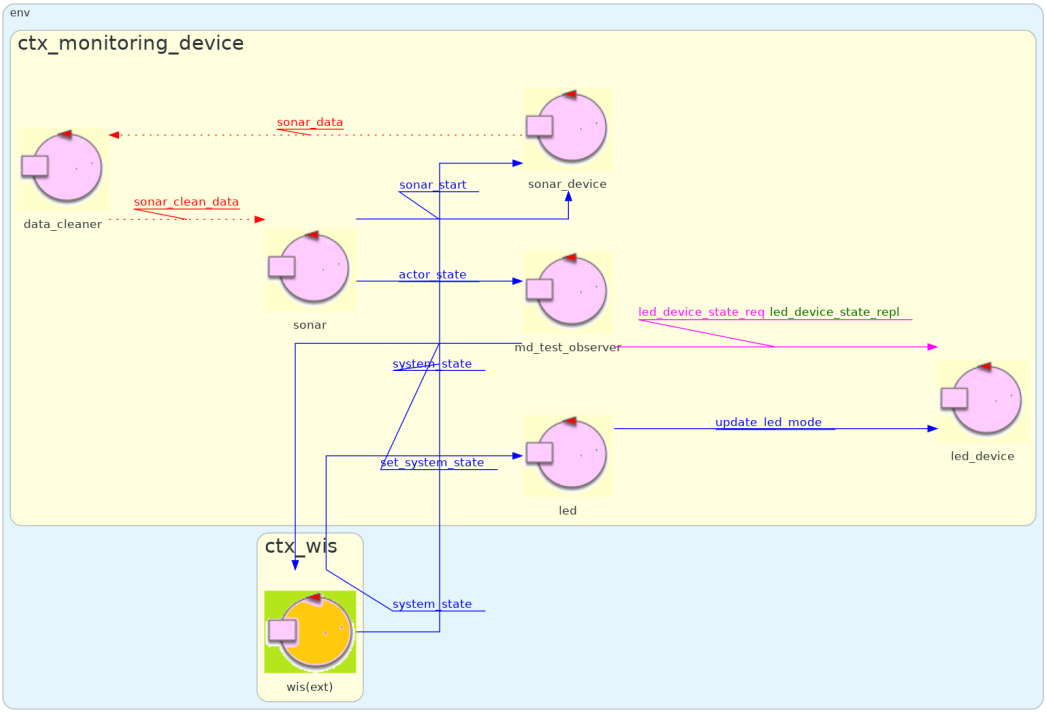
PROF

Sprint 2 Architecture:



wis_systemArch

PROF

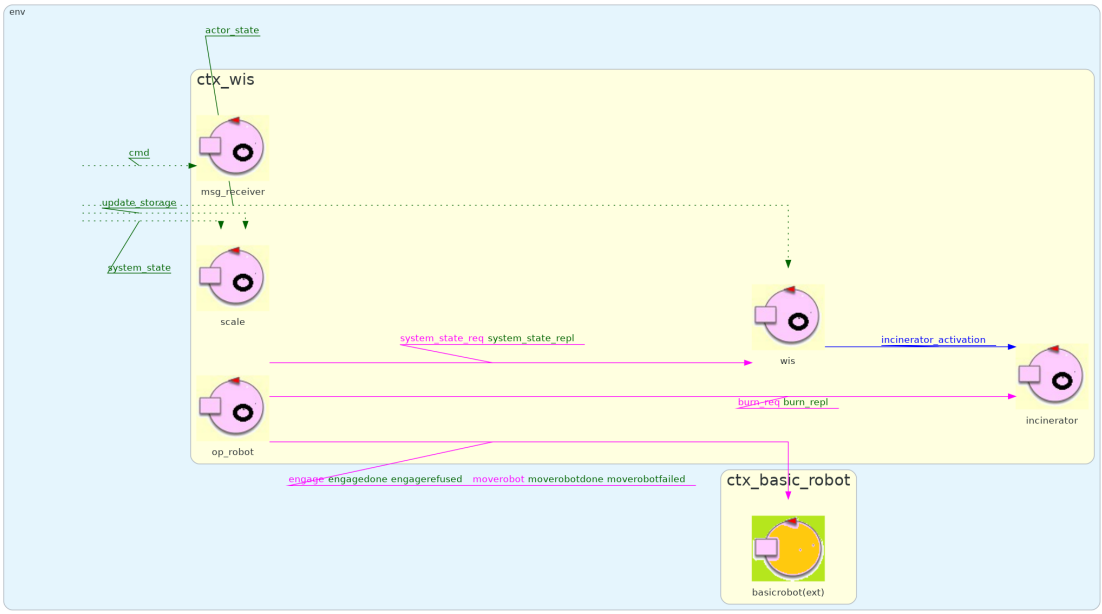


monitoring_deviceArch

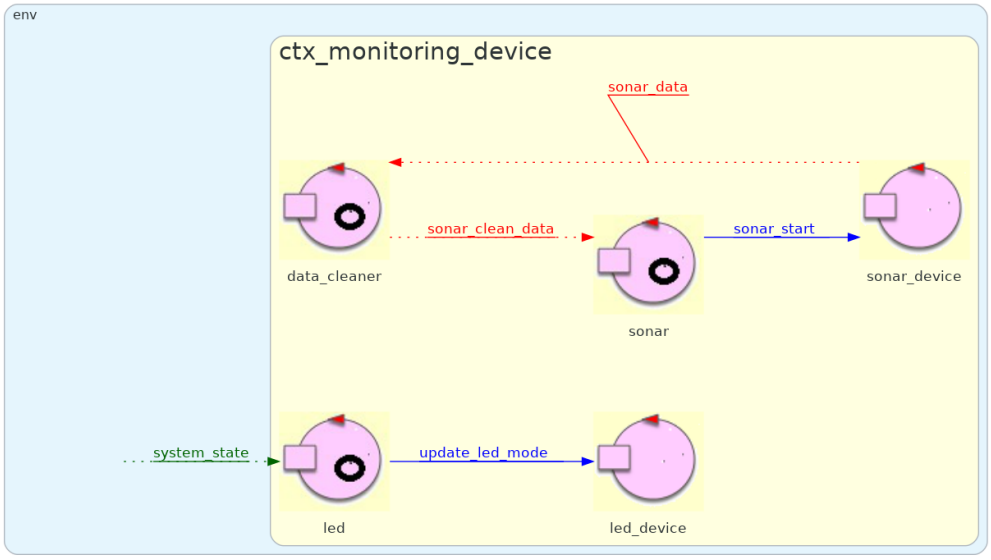
Based on the Problem Analysis carried out previously, we upgraded the executable version of the system covering the discussed features.

We attach here a visual representation of the new system architecture:

Sprint 3 Architecture:



wis_systemArch



monitoring_deviceArch

PROF

As can be seen, the adoption of the MQTT protocol allowed a greater level of independence, in particular, the WIS does not need anymore to know the MonitoringDevice ip address, while the MonitoringDevice still needs the main host ip address to communicate with the MQTT broker, but it can be injected at runtime by the SystemConfigurator using the `monitoring_device.properties` file.

Implementation and Deployment

Docker

To simplify the application deployment, we decided to **dockerize all the components of the system** but the MonitoringDevice, since it needs to run on an embedded device.

MQTT Broker

A **MQTT Broker** is needed to implement the MQTT protocol. We decided to use a Mosquitto local broker, as it is easily integrated alongside our qak architecture; this broker is also used to implement communication between the ServiceStatusGui and the qak system, more details will be given in the next section.

The actors that need to send or receive messages connect to the broker and then either **subscribe to or publish updates on specific topics**:

- **actor_state**: topic used by the actors to publish updates about their own status.
- **system_state**: topic used by WIS to expose the real-time state of the whole system.
- **mock_cmd**: topic used by Backend to send the command (increase RP or empty AshStorage) when a button is pressed.

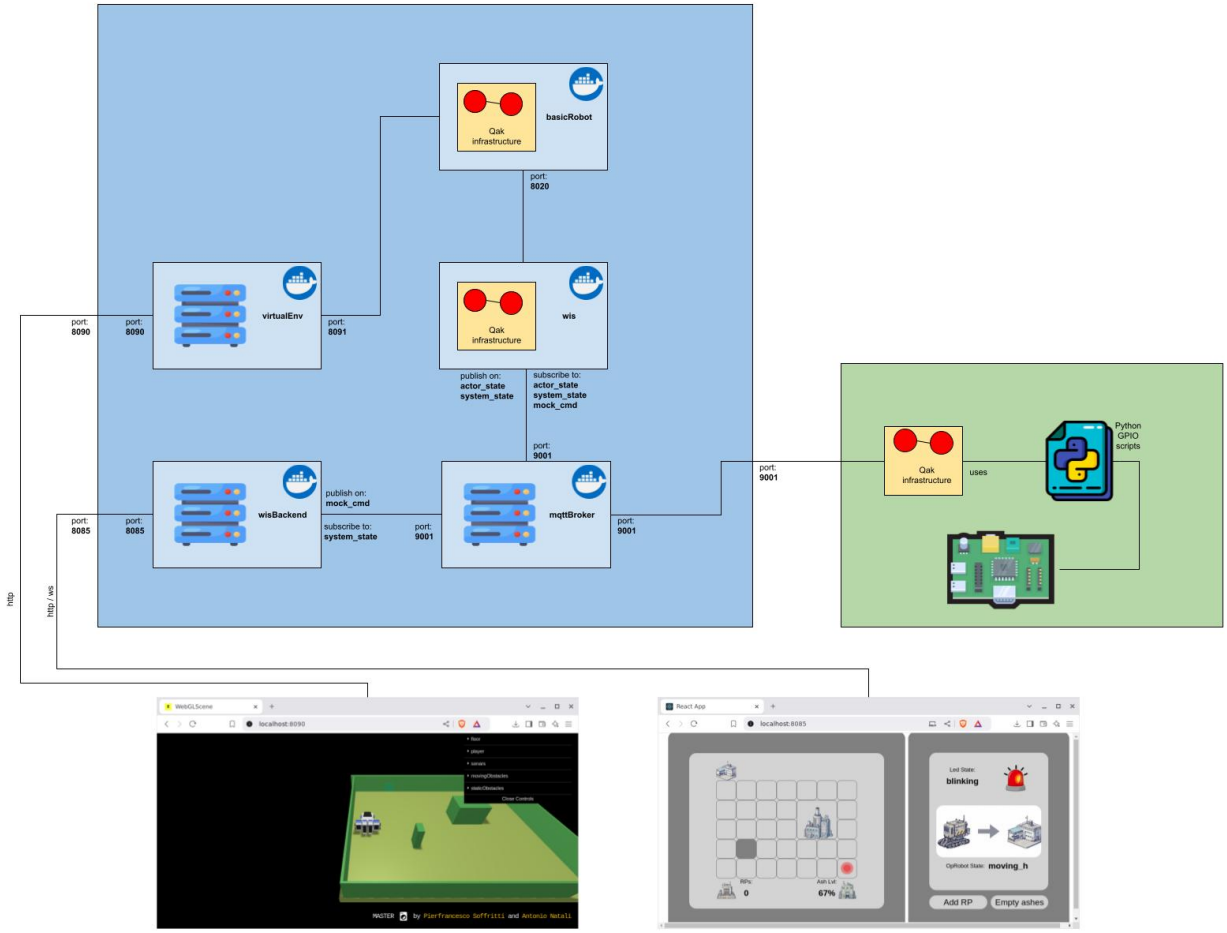
ServiceStatusGui

We decided to split our Gui into two projects:

- **Backend**: it manages all communication between the Gui and the WIS System, using the MQTT protocol.
- **Frontend**: React Application showing the console. It communicates with Backend via WebSocket.

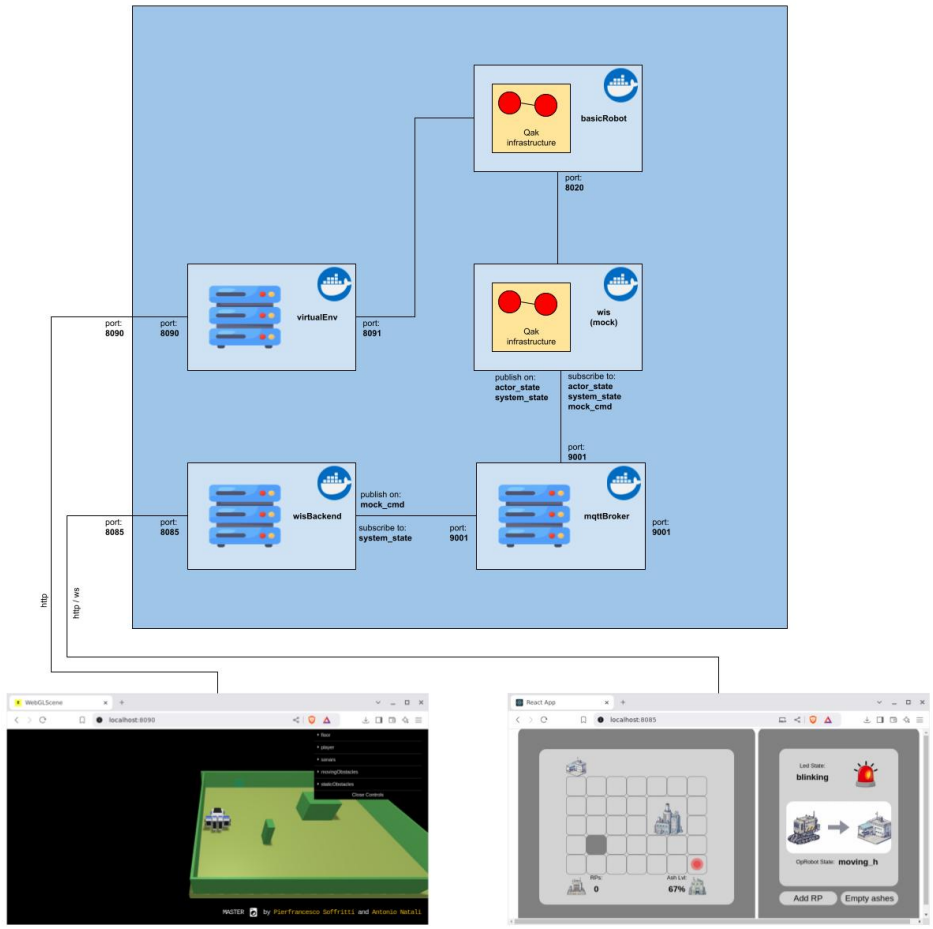
Component Architecture:

Here we attach the deployment architecture resulting from the previous analysis:



PROF

The following schema instead shows the mock deployment architecture:



Test Plan

For this sprint, we switched to a new QAK tester project, TEST_Sprint3, instead of using the JUnit library. This change simplifies communication with all system components.

PROF

Test Project: wis_tester

Test Name	Initial Condition	Expected Behavior
test_check_n1	WasteStorage contains 5 RP, AshStorge is empty, nobody empties AshStorage, Incinerator is inactive	Once the OpRobot check the system state for the first time, the the WasteStorage contains 5RP, the Incinerator is active, and the AshStorage is empty
test_check_n2	test_check_n1 is ok	Once the OpRobot check the system state for the second time, the the WasteStorage contains 4RP, the Incinerator is active, and the AshStorage is 1/3 full
test_check_n3	test_check_n2 is ok	Once the OpRobot check the system state for the second time, the the WasteStorage contains

		3RP, the Incinerator is active, and the AshStorage is 2/3 full
test_check_n4	test_check_n3 is ok	Once the OpRobot check the system state for the second time, the the WasteStorage contains 3RP, the Incinerator is active, and the AshStorage is full

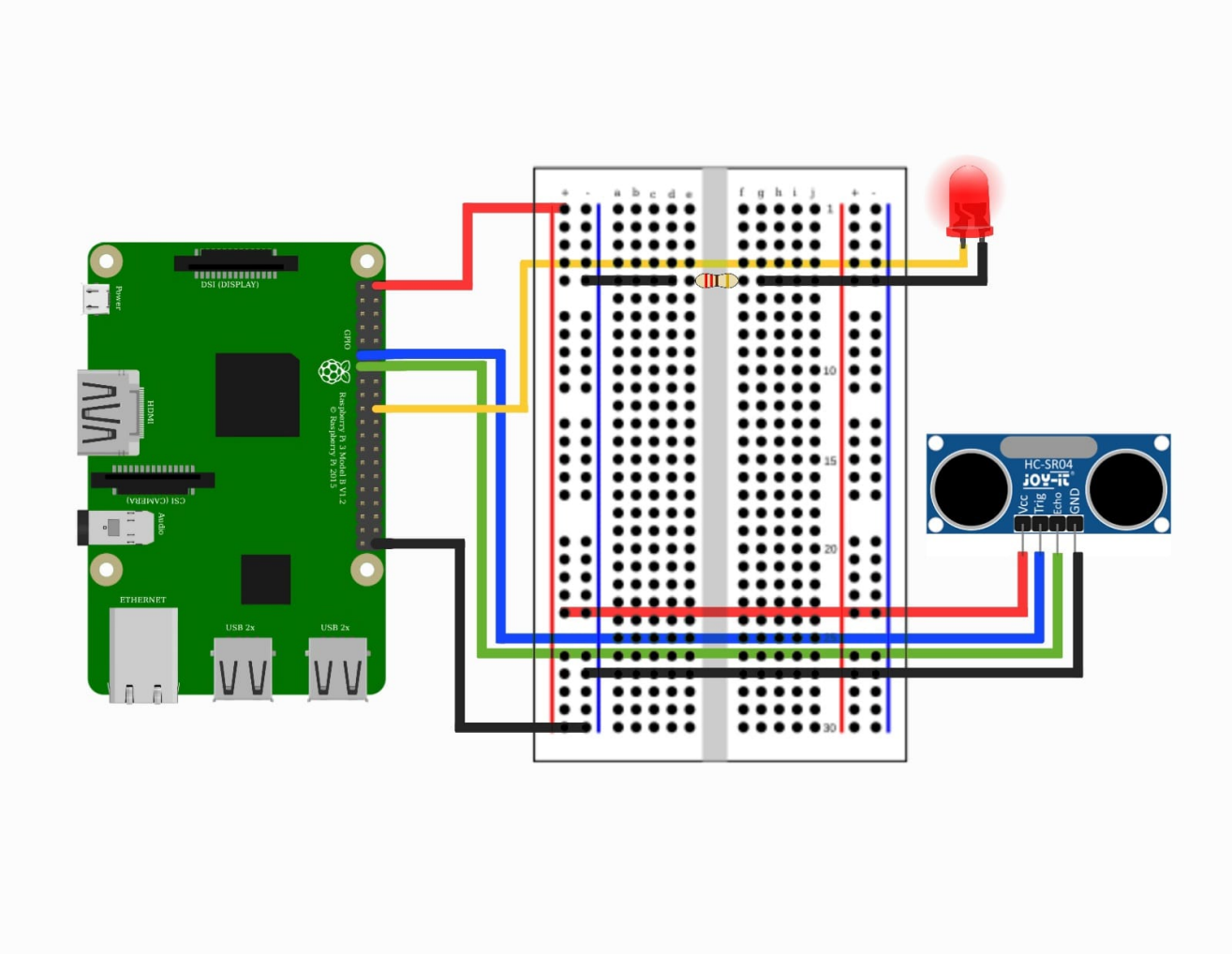
Usage

Monitoring Device

Firstly, you will need:

- a raspberry (we used a raspberry PI 3+)
- a led
- a sonar (HCSR04)
- a 220ohm resistor
- a breadboard

You will have to assemble those elements following this wiring scheme:



Then you will have to deploy the Monitoring Device control software, to do so, open a terminal inside the `Sprint3/MD` folder run

```
gradlew build
```

After that, copy the `Sprint3/MD/build/distributions/monitoring_device-1.0.zip` folder inside the Raspberry (for instance using `scp`) and unzip it

System Activation

Then, you have to activate the monitoring device, to do so connect to your Raspberry via `ssh` then move inside the `monitoring_device-1.0/bin` folder, and run

```
./monitoring_device
```

Lastly, you have to activate the WIS system by opening a third terminal inside the `Sprint3/WIS` folder and running

```
docker compose up
```

N.B. Type `docker compose -f docker-compose-mock.yaml up` if you want to launch the mock version of the application, or `docker compose -f docker-compose-test.yaml up | grep tester-1` if you want to launch the test configured system