Università di Genova | DIBRIS DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

Smart Jugs - Project Documentation

Internet of Things - University of Genova - a.a 2023/24

Authors:
Cattaneo Kevin - S4944382
Isola Riccardo - S4943369

Index

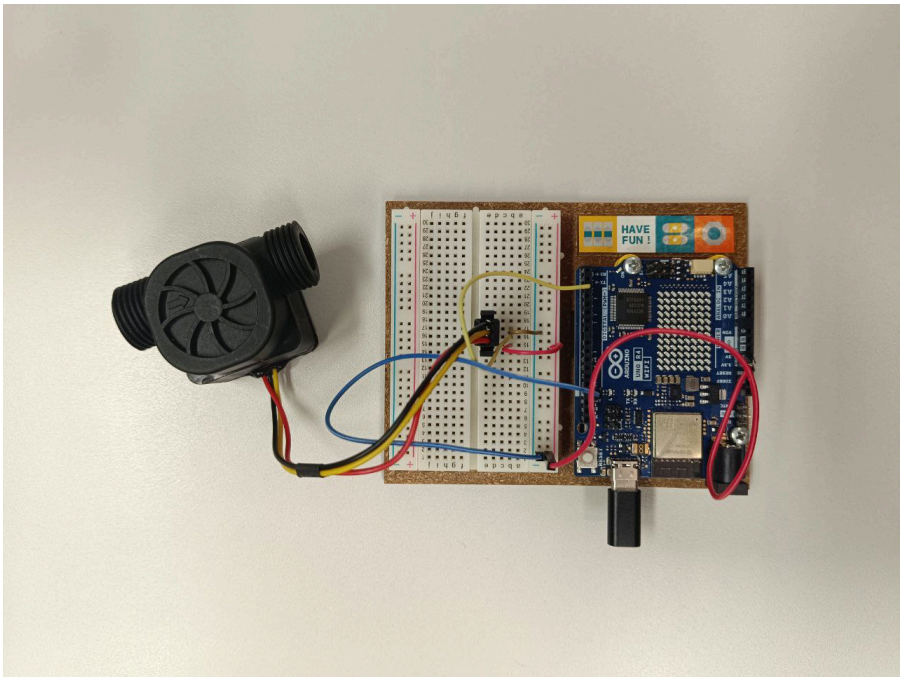
Overview

Our project involves the connectivity of “Smart (Water) Jugs” and the tracking of the fluid filtered each second. We also compute some aggregate information like the total number of liters filtered, the amount of water consumed each day and the quality of the filter. Our implementation interfaces with both a dashboard from an enterprise view and one from a client view; for the latter case we use an application developed in parallel with the Mobile Development course, that provides the same amount of information gathered and personalized per each jug and also a localization.

Sensors

Arduino

We implemented a version of the sensor via Arduino hardware, and a script that allows us to pair the jug to the phone to link the jug to a user and then data via MQTT protocol to ThingWorx.



Simulator in TypeScript

We implemented a version of the sensor via a simulator that allows us to scale freely the number of sensors and so the amount of data that they send. This simulator performs different calls just like the arduino does. Just like that, the data that are simulated (randomly) are also sent on the ThingWorx platform.
We also developed using Bun instead of NodeJS, a popular JavaScript runtime that provides faster performance, simpler APIs, and better development experience.
To see how it works, just install bun and locate into the project folder; after that type:
`bun simulator/src/index.ts --help`
to see the possible commands to perform.
To do a complete simulation, just type:
`bun simulator/src/index.ts simulator <number of sensor devices>`
to initialize them and to make them send data over MQTT.
Please notice that the method calls are of type POST even for some GET requests: this has been done for an easier implementation application-side.

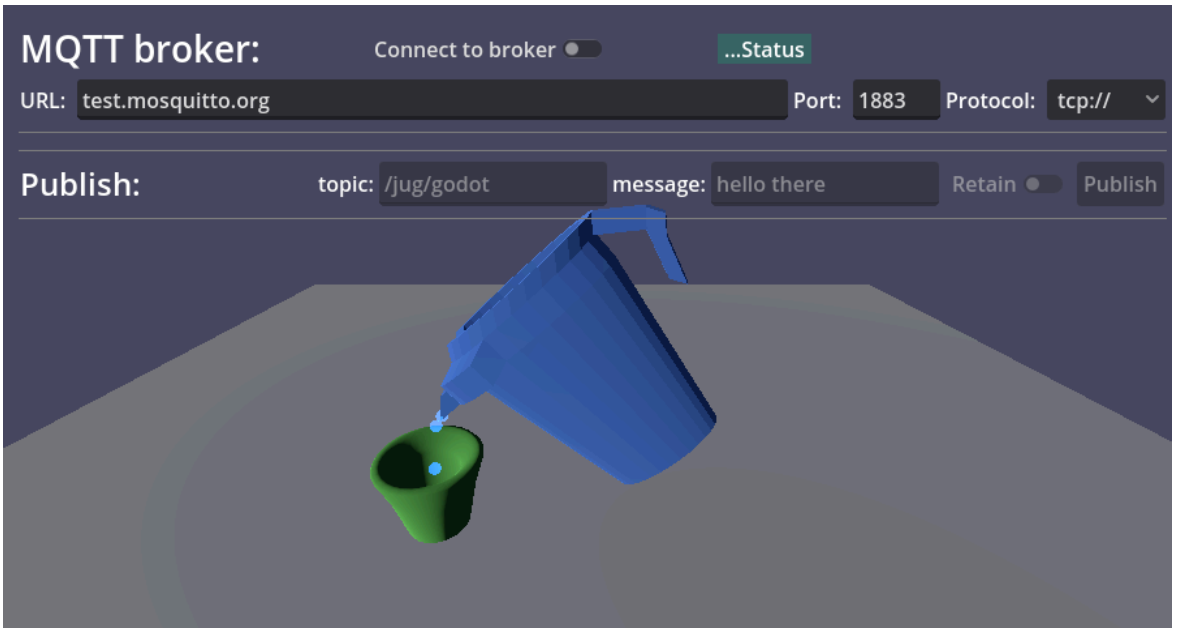
In the following we list all the function we implemented:

- **register <username> <password>**: simulate a user registration
- **login <username> <password>**: simulate a user login; here a JWT token is returned to authenticate future jug operations
- **pair <id> <token>**: add a jug (from id) to a certain user (from token)
- **send-data <id> <data**>: simulate the pouring of water, that is a liter into litresPerSecond property, data is the value we want to send of liters filtered in that second for a certain jug (from id)
- **send-loop <id>**: send random data for litresPerSecond, just like send-data, but in loop, for a certain jug (from id)
- **simulator-single <username> <password> <id>**: simulate the entire process of registration, login, pairing and sending data, for just one user and one jug
- **simulator <n>**: do the same of simulator-single but for n user and jugs
- **filter <n> <token> <capacity>**: retrieve the current filter max capacity value for the jug n
- **set-filter <token> <id> <value>**: simulate the change of filter for a certain jug (from id n) to set a certain capacity
- **arduino-client <ssid> <pw> <token>**: simulate the Arduino hardware sending the pair request to the mobile device from the provided WiFi credentials
- **arduino-server <id>**: simulate the listening of the Arduino device for receiving credentials and pair
- **arduino-simulator <id>**: performs the arduino-server action and then send-loop
- **get-jugs <token>**: get all the current jug owned from the user (from token)
- **change-password <token> <oldPassword> <newPassword>**: simulate the changing password for the user
- **delete-jug <token> <id>**: delete a jug for a provided user (from token)
- **rename-jug <token> <id> <name>**: rename a jug for a provided user (from token)
- **delete-account <token>**: delete account of the provided user (from token)

- **change-email <token> <email>**: change the email with the inserted one of the provided user
- **total-litres <token> <id>**: retrieve the total liters filtered (across different filters) from a jug owned by a certain user (from token)
- **litres-per-second <token> <id>**: retrieve the current litresPerSecond value for the provided jug (from id) of a certain user (from token)
- **total-litres-filter <token> <id>**: retrieve the current totalLitresFiltered, that is the liters filtered from the current filter
- **daily-litres <token> <id>**: retrieve the liters filtered in the last three days from a certain jug (from id)
- **hour-litres <token> <id>**: retrieve the liters filtered in the last hour from a certain jug (from id)
- **week-litres <token> <id>**: retrieve the liters filtered in the last seven days from a certain jug (from id)
- **create-redis-data <id>**: simulate the creation of entries of different data (e.g. litresPerSecond, totalLitresFiltered etc.) in Redis cache for a certain jug (from id)
- **set-location <token> <id> <lat> <lon>**: set a location of a certain jug (from id) of a certain user (from token)
- **watch-data <token> <id>**: for debugging purposes only, will log in console all the values of a certain jug (from id) of a user (from token)

Simulator in Godot - Gamification process

We implemented a version of the sensor also via Godot Software. In the spirit of gamification we tried to see how the MQTT works also in this case. The main idea is to simulate the pouring of water via 3D Models done by us and to see the data that are sent. Notice that those data will not be part of the pool included in ThingWorx and are just listened for testing purposes on Node-Red; but surely can be shifted to ThingWorx.
Notice also that the implementation of MQTT script in GDScript (Godot Script) is not made by us, we used the following add-on by https://github.com/goatchurchprime/godot-mqtt.git. We then personalized it following our needings, without delving too much or refining the original implementation.



Sensor Data

The main sensor we use is a fluxometer, some data is also taken from the mobile device. The data we retrieve and show in the NodeRED dashboard represents the following parameters:

- Total consumption of water from all the jugs
- Quantity of plastic saved (by using a jug)
- Counting the number of low-filter alerts (notification sent to mobile device)
- Positions of jugs (taken via mobile device)

Platforms

Node-RED - “Fog”

The “fog” of the backend is implemented entirely via Node-RED via three different flows. Here we handle the request of pairing done by the jug to the application and the other APIs that interfaces with it. For some data, e.g. user data and their jugs, we save those data into a PostGres database. We pushed Node-RED to its extremes, by implementing also all the APIs needed for interfacing with the mobile devices.
The enterprise dashboard is implemented here and polls directly from the ThingWorx platform.

Authentication

For any operation that involves a check of owning the jug, we verify a JWT Token that is given upon a successful login. This verification is done server-side by Node-RED.

API Flow

Here the main APIs are implemented. Those APIs serve as dialogue between the sensor device and the mobile device (e.g. for pairing) and to retrieve data to display on the dashboard.

Redis Flow

This flow polls at pre-specified intervals ThingWorx to update the values in Redis cache for the active users (so users that used the jug in the last hour).

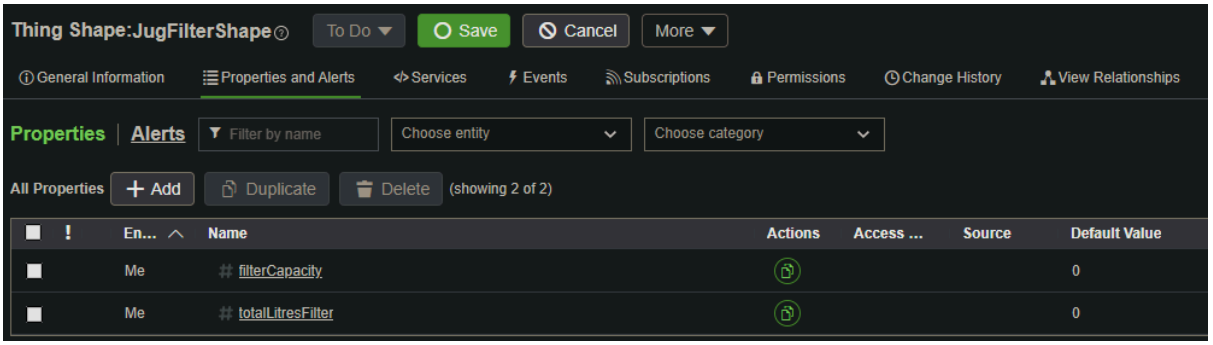
Firebase Flow

This flow handles the notification flow for the mobile device when the filter usage has reached a certain limit.

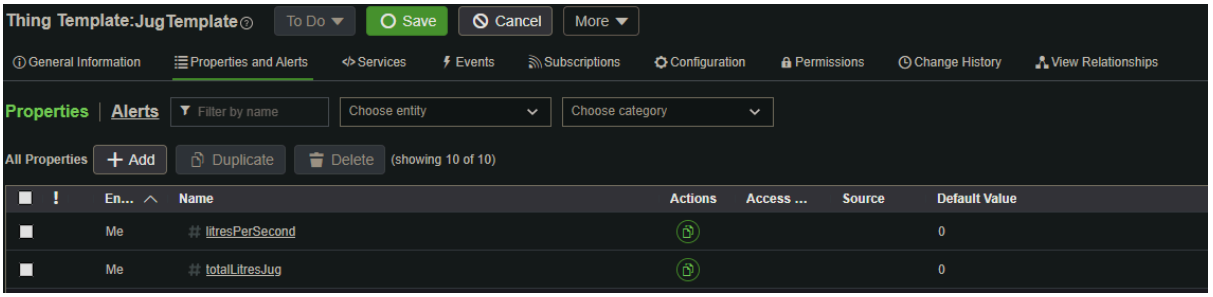
ThingWorx

We use ThingWorx mainly as a database to store things (jugs) data that arrive each second. This platform presents the properties of each jug, implemented as a Template for the properties about the fluid (e.g. total liters filtered) that extends a Shape that holds the filter properties (e.g. filter capacity).
We also have a Value Stream that sends the amount of total liters filtered correlated with a timestamp that we use to implement a time series in the mobile application.

Shape



Template (implements the shape)



INSERIRE VALUE STREAM
INSERIRE THING DI ESEMPIO

PostgreSQL Database

We use PostgreSQL as a database to store information. We treated it as a company database that stores user credentials (received from the mobile application), device tokens (for Firebase notification) and jug details retrieved from Thingworx.
Any sensible information (password) is saved in hash with bcrypt (from Node-RED side).

Redis

We use Redis as a cache service for storing jugs data without requesting each time the entire stream of data from the remote server of ThingWorx. Doing so we speed up the process by contacting the Redis server (in our self-made enterprise). After a certain amount of time via Node-RED we update the Redis data for each jug by polling ThingWorx.

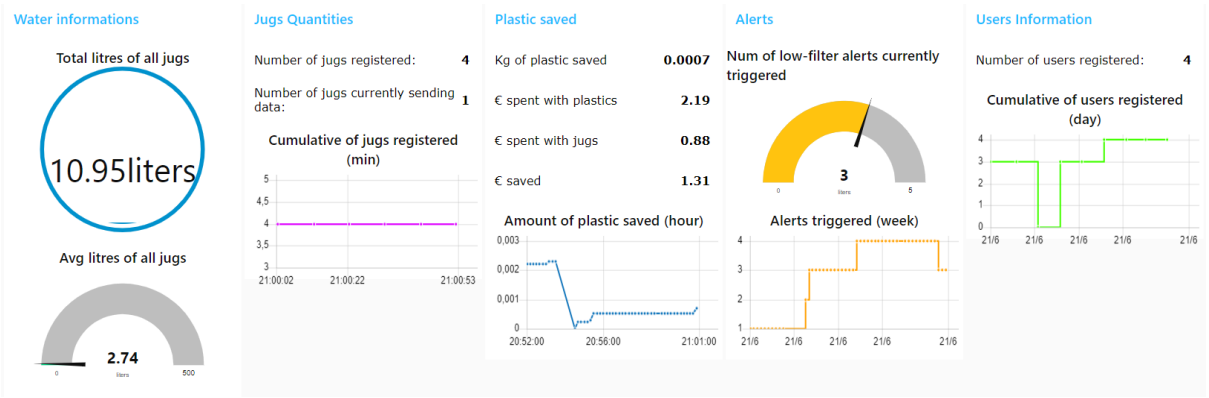
Dashboards

Client-Side (Mobile application)

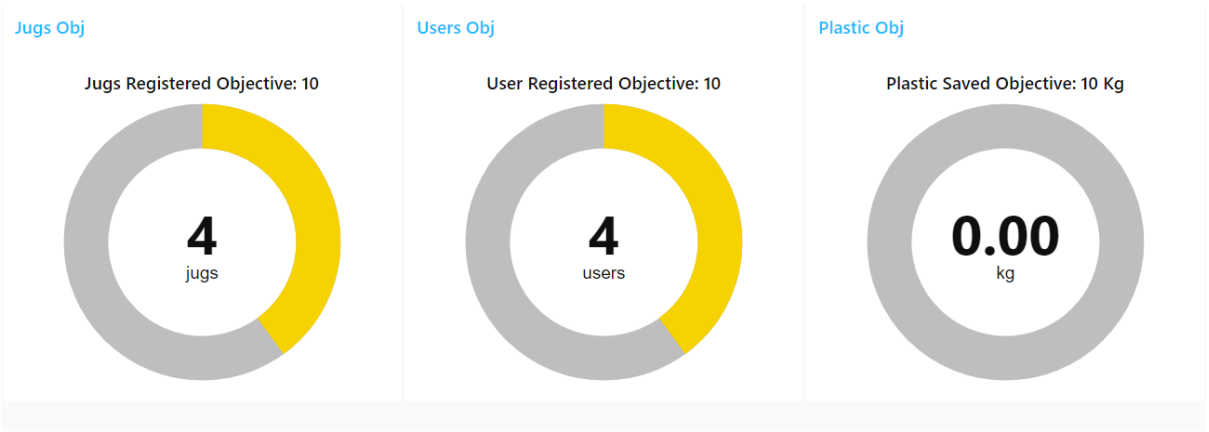
<photo>

Enterprise-Side (Node-RED)

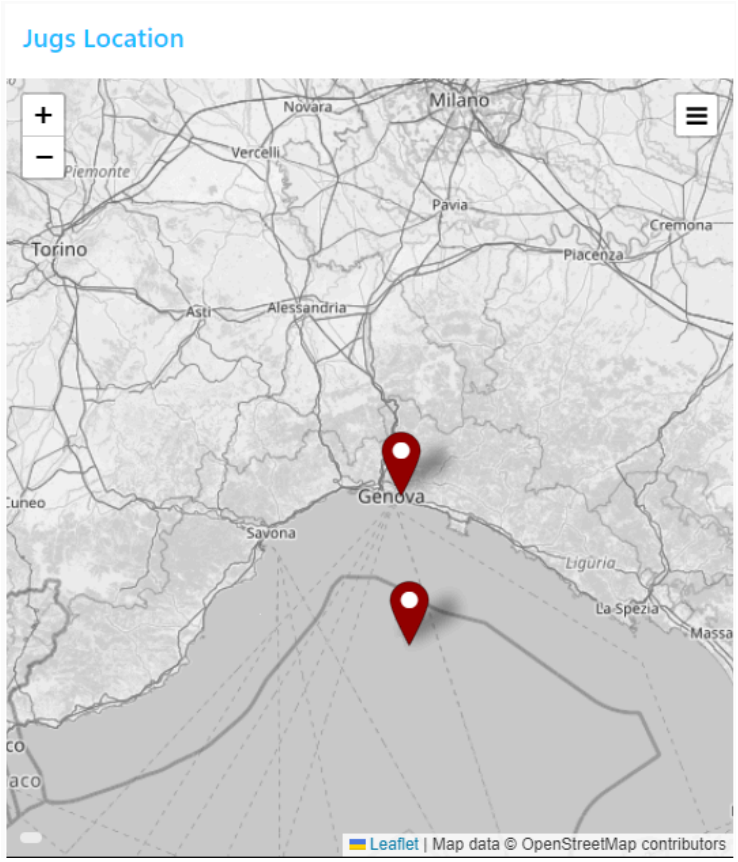
Overview



Company Objectives

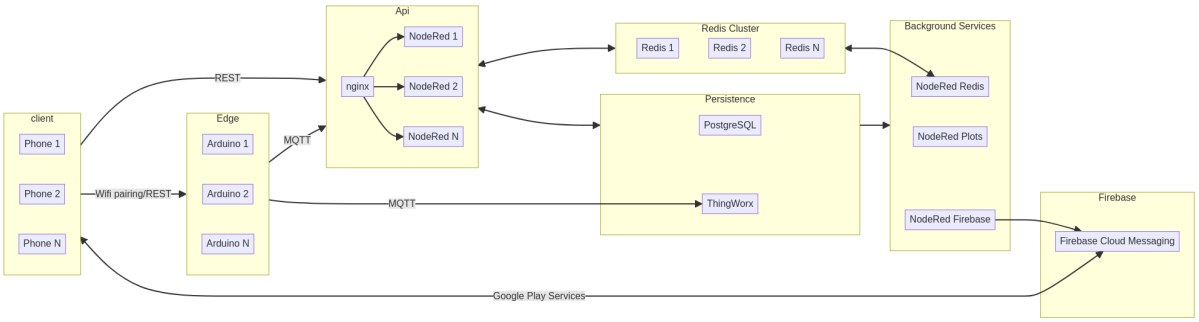


Location



The real position of the device was located in the University of Genova (upper).
We will not hold that precise position in our implementation, just the **approximate** one is taken from the mobile device and shown on the map. As we can see the approximate position has some distance from the precise one.

Architecture



SPECIFICARE HOT / COLD STUFF

Opinions

- NodeRED is an easy platform to use until the system we want to represent grows in size and complexity: this makes even some steps of computation represented into a great amount of blocks.

- ThingWorx documentation usually was poor or inaccessible since it was necessary to pay a license.

- ThingWorx provided server was usually down, so it slowed up the implementation process. On the other hand, it gave us the idea of implementing a Redis cache to limit the continuous traffic to ThingWorx.

- Arduino libraries usually were abandoned: there is a case of needing to modify the header of an arduino library to make it work (see comment in the script).