

Sistemi Orientati ad Internet

A.A. 2022-2023

Daniele Porta daniele.porta@mapsgroup.it
Fabio Strozzi fabio.strozzi@mapsgroup.it

Course Project

Reactive Home Automation System

Project objective

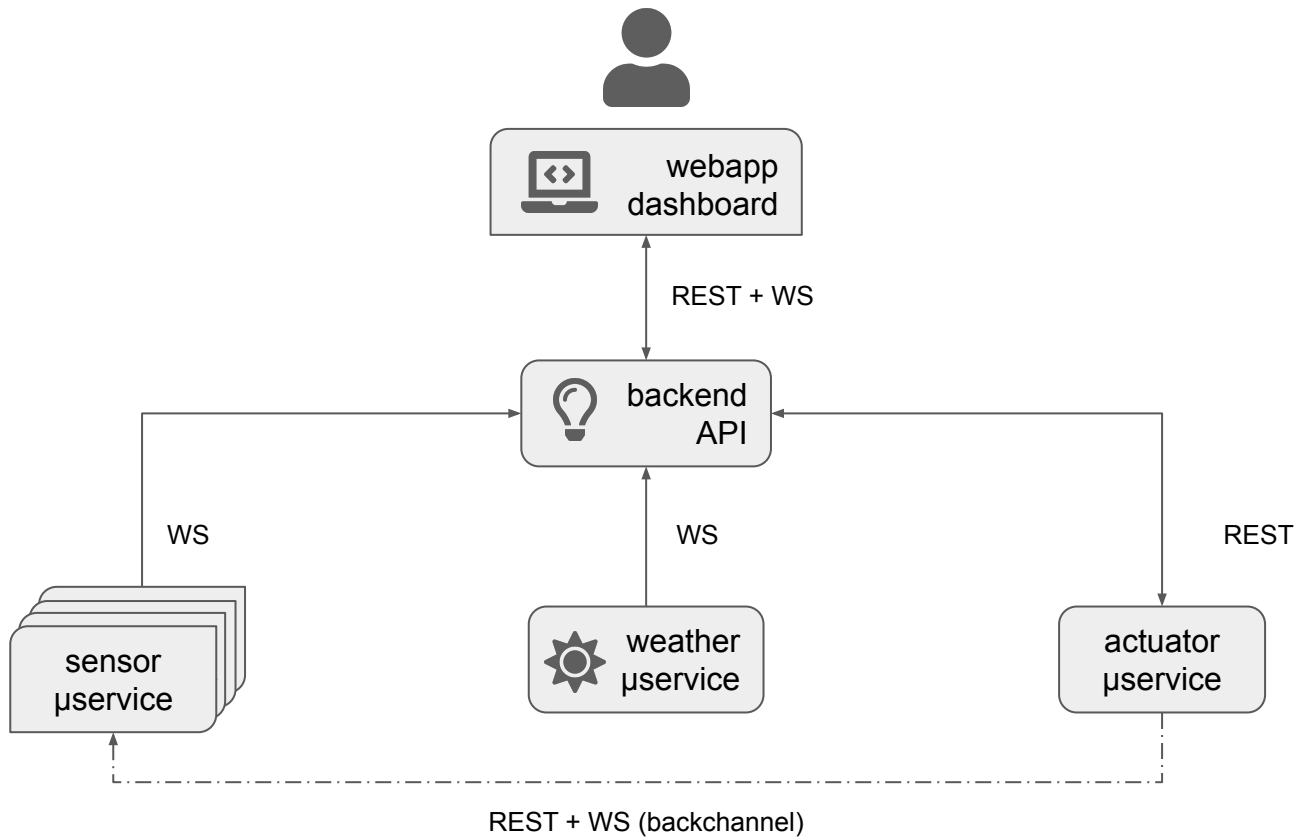
Develop a simulated reactive home automation system made up of the following components:

1. a reactive webapp
2. a backend for the webapp
3. one or more μ service of simulated home sensors
4. a μ service that streams the current weather conditions (provided)
<https://github.com/SOI-Unipr/weather-service>
5. an actuator μ service
6. a backchannel to simulate the actuation

Prove that the solution is robust enough to tolerate (that is, backend and frontend still work):

- simulated downtimes of the μ services
- simulated wrong data being transmitted
- new sensors being added at runtime
- simulated actuation failures on the backchannel

Overview



Functional requirements – Sensors 1/2

You're going to develop some sensors for the following “devices” of a single room:

- two or more **windows**
each sensor has its own state “open”, “closed” or “error”
- a **door** (with the same states)
- a **heat pump**
 - state can be “on”, “off” and “error”
 - operating temperature (in °C)
- a **thermometer** for the internal room temperature (in °C)

All sensors expose a WS API to implement a publish-subscribe pattern; subscribers are updated whenever something changes (state, temperature, etc.).

They also have a REST API to receive commands from the actuator on the simulated backchannel (e.g, change door state to “open”, set temperature, etc.).


Functional requirements – Sensors 2/2

The thermometer sensor must simulate the room temperature according to the state of the other sensors and the current weather.

Example

If a window is opened and the external temperature is lower than the current room temperature, then the thermometer will simulate a linear decrease of temperature.

Sensors can also simulate changes to their state (e.g., a window is opened, the heat pump is in error).

 Don't spend too much time on simulation accuracy and realism.

Functional requirements – WebApp

The WebApp must have two main sections:

- a read-only dashboard of the live and historical data using graphs and anything else that be useful to understand the state of the systems (sensors, weather, liveness, etc.) in time
- several control panels that allow the user to send commands to the actuator (e.g., open the door, stop the heat pump, change the operating temperature of the pump, etc.)

Users are not allowed to use the WebApp without authentication.

Functional requirements – Backend

Backend provides APIs for the WebApp and centralizes data streaming from different sources. The WebApp will only actually speak with the backend (and never with the actuator or any sensor).

The backend will be responsible to:

- provide current simulation properties (e.g., the list of the sensors and their states)
- aggregate data, according to subscription requests from the WebApp (e.g., subscribe to sensors A and B)
- user authentication
- forward commands to the actuator

Functional requirements – Actuator

The actuator must accept commands like:

- open the door
- close the window N
- turn the heat pump off

The actuator must validate incoming requests and might decide to return an error if some are inconsistent with the current system state (e.g., a closed door is asked to be closed again).

Using a dedicated backchannel to each sensor, the actuator must inform sensors to change their state and their operating values (e.g., heat pump temperature is changed to 40°C).

Technical requirements

Every service (webapp, backend, sensors, etc.) must run as an independent **Docker** container.

Project must be initialized and run using **docker-compose** with a single instruction:

```
cd project-folder/  
docker-compose up --build
```

If that means that ad-hoc images are to be built, then use docker-compose for that as well.

Technical requirements

Project must be accessible via HTTP using this URL template:

<http://<id>.soi2223.unipr.it/>

Where <id> is made up of the dash separated list of surnames (in alphabetical order).

For instance, Bianchi and Rossi will go with:

<http://bianchi-rossi.soi2223.unipr.it/>

A port other than 80 can optionally be used.

Caveat: virtual hosting and /etc/hosts must be properly configured in the hosting machine.

Technical requirements

3/4

All backend services

- API developed using Node and Express v4: <https://expressjs.com/>
- Backend API protected with JWT token validation (social login via OpenID Connect is ok)
- No need to protect actuator and sensors with authentication
- Both REST and WebSocket must be used:
 - REST for actuation, configuration and fire-and-forget calls
 - WebSocket for data streaming and publish/subscribe
- If you believe a DB is required, then use SQLite: <https://www.sqlite.org/>
- Website static contents can be hosted using either Express, Apache or Nginx
- Virtual hosting with reverse proxy should be properly configured to route incoming API calls

Sample backend to get started with: <https://github.com/SOI-Unipr/todo-server>

Technical requirements

Frontend

- Single Page Application (SPA)
- Use of RxJS is mandatory
- Feel free to use any CSS open source style library (Bootstrap, Semantic UI, etc.)
- Graphic components from aforementioned libraries that require JS are welcome
- For graphs and diagrams use ChartJS: <https://www.chartjs.org/>
- JQuery is also fine
- Angular, Vue, Svelte and React are banned, sorry about that!
- Please, support at least one browser among Firefox and Chrome

Sample app to get started with: <https://github.com/SOI-Unipr/todo-app>

Additional, optional features

1. Liveness and readiness endpoints that are tested within Docker compose
2. UI components specific for each sensor are not hard-coded in the WebApp: a configuration is provided by the backend to describe which sensors are registered and/or active and how to control them; configuration can of course change in time.

Deliverables

1. Source code must be published on a **private** [GitHub](#) repository
2. On the README.md file of the repository, please specify user credentials in case no social login is provided
3. A PDF document with the [architectural description](#) of the solution that focuses on the aspects discussed during the lessons
4. Please write clean and comprehensible code

Please subscribe ASAP to GitHub and communicate the accounts before discussing the project: one student, one account.



Project **must be working** during the presentation (!) and for us (!!).

Constraints

Teams

- 1 or 2 students, no exceptions
- Each student must prove to have worked on the whole stack of the project (we also look into the Git commits).
- Work organization should be based on functional requirements (e.g., authentication), not architectural ones (e.g., backend).

Caveat: project must fully satisfy all the functional and technical requirements to be evaluated.

Project evaluation criteria (by descending importance)

1. Architectural design, e.g.:
 - a. separation of concerns among services
 - b. how much coupled services are
 - c. how much robust the system is as a whole
 - d. to what extent the system is dynamic (new sensors can be announced when they're run)
 - e. if a DB is used, then a persistence layer is desirable
 - f. for HTTP API, a client class to encapsulate the I/O might be a good choice
 - g. etc.
2. API quality: consistency, granularity, etc.
3. How project meets requirements
4. Additional, optional features being implemented
5. Code quality
6. Usability and user experience