

[Pervasive Computing exercises] Report

Federico Mazzini
federico.mazzini3@studio.unibo.it

Febbraio 2022

1 Exercise 1: About WoT web thing design

1.1 The problem

We want to develop an interoperable smart coffee machine (smart-cm) for the Campus, as a WoT Thing. The machine has no physical UI, it can be used only via app (smartphone or whatever device). It provides an API for users and for maintainers. In this exercise we focus on users.

Given a specific smart coffee machine (identified in some way), a user with a smartphone should be able to get a coffee (or a tea, or one of the products available from the specific machine), with the specified options (e.g. sugar level), as well as check the availability or state the machine, and the current availability of products as well.

The idea is that in the Campus there could be smart coffee machines produced by different vendors but having a shared smart coffee machine Thing Description, so that they can be used by any app working with that TD.

Given this scenario, then:

- Think about an abstract model of the smart coffee machine with essential capabilities to implement the use case above, and define a Thing Description based on the WoT model. An Open API perspective should be adopted, so that the smart coffee machine web thing should provide a minimal API focused on its capabilities.
- Design and develop a simplified prototype implementation of the smart-cm web thing (using the software stack that you prefer and mocks up where needed) and a basic user app (mobile or desktop) allowing users to interact with a smart-cm, to do test/demo.

1.2 WoT Architecture

To model a coffee machine it was decided to adopt the WoT standard defined by the W3C. Through this standard, a thing offers standard operations to be controlled, in such a way as to facilitate interoperability between various devices. By the use of this standard, we try to make all smart devices homogeneous in their use, so as to be able to carry out operations common to them.

The WoT standard defines three main **affordances** in order to interact with a thing:

- **Property:** An Interaction Affordance that exposes state of the Thing.
- **Action:** An Interaction Affordance that allows to invoke a function of the Thing, which manipulates state or triggers a process on the Thing.
- **Event:** An Interaction Affordance that describes an event source, which asynchronously pushes event data to Consumers.

1.3 Thing Description

A Thing Description is an alive documentation for a smart thing. It defines the modality of interaction of a thing and, with some ontologies, the purposes of it. Inside the Thing Description are specified the **interaction affordances** of a thing, in such a way that a user, or also another thing, knows how to interact with the specific thing.

1.4 Coffe Machine Thing Description

The purpose of the exercise is to define a smart coffee machine thing in order to allow different vendors to develop their own coffee machine thing, but sharing the main thing description to provide the same interaction affordances.

A typical scenario would be the implementation by vendors of different coffee machine, with different interaction affordances. This would lead to the implementation of several user app in order to interact with the different coffee machine. In this scenario it would be no interoperability between machines, forcing the user to use different application.

In the scenario of the exercise instead, a common TD is implemented for all the different vendors. The different vendors must therefore implement their smart coffee machine under a Thing Description, in order to have a single user application for all the smart coffee machines.

The interaction affordances meant for a smart coffee machine are the following:

- **Properties**

- **availableResources**: current level of all available resources given as an integer percentage for each particular resource.
- **possibleDrinks**: the list of possible drinks
- **lastDrink**: date of the last drink produced
- **lastMantainance**: date of last mantainance
- **servedCounter**: the total number of served beverages
- **mantainanceNeeded**: shows if a maintenance is needed

- **Actions**

- **makeDrink**: make a drink from available list of beverages. It's possible to specify the type of drink and the level of sugar by uri variables.

- **Events**

- **outOfResource**: out of resource event. Emitted when the available resource level is not sufficient for a desired drink.
- **needMantainance**: generic problem, need for mantainance.
- **limitedResource**: some resource is about to finish, it's necessary a refill.

Below is reported the Thing Description for a smart coffee machine.

```

{
  "title": "Smart-Coffee-Machine",
  "description": "A smart coffee machine",
  "support": "git://github.com/eclipse/thingweb.node-wot.git",
  "@context": ["https://www.w3.org/2019/wot/td/v1"],
  "properties": {
    "availableResources": {
      "type": "array",
      "description": "Current level of all available resources given as
an integer percentage for each particular resource.",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string"
          },
          "value": {
            "type": "integer",
            "minimum": 0,
            "maximum": 100
          }
        }
      }
    }
  },
  "possibleDrinks": {
    "type": "array",
    "description": "The list of possible drinks",
    "items": {
      "type": "object",
      "properties": {
        "id": {
          "type": "integer",
        },
        "textId": {
          "type": "string"
        },
        "name": {
          "type": "string"
        },
        "ingredients": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {

```

```

        "resource": "string",
        "quantity": "integer"
    }
}
},
"available": {
    "type": "boolean"
}
}
},
"lastDrink": {
    "type": "string",
    "description": "Date of the last drink produced",
    "readOnly": true
},
"lastMaintenance": {
    "type": "string",
    "description": "Date of last maintenance",
    "readOnly": true
},
"servedCounter": {
    "type": "integer",
    "description": "The total number of served beverages",
    "minimum": 0
},
"maintenanceNeeded": {
    "type": "boolean",
    "description": "Shows if a maintenance is needed",
    "observable": true
}
},
"actions": {
    "makeDrink": {
        "description": "Make a drink from available list of beverages.",
        "uriVariables": {
            "drinkId": {
                "type": "string",
                "description": "Defines what drink to make,
                drinkId is one of possibleDrinks property values"
            },
            "sugarLevel": {
                "type": "integer",
                "description": "Defines the level of sugar

```

```

        for the drink",
        "minimum": 0,
        "maximum": 5
    }
},
"output": {
    "type": "object",
    "description": "Returns true/false and a message when
all invoked promises are resolved (asynchronous).",
    "properties": {
        "result": {
            "type": "boolean"
        },
        "message": {
            "type": "string"
        }
    }
}
},
"events": {
    "outOfResource": {
        "description": "Out of resource event.
Emitted when the available resource level
is not sufficient for a desired drink.",
        "data": {
            "type": "string"
        }
    },
    "needMaintenance": {
        "description": "Generic problem, need for maintenance.",
        "data": {
            "type": "String"
        }
    },
    "limitedResource": "Some resource is about to finish,
it's necessary a refill.",
    "data": {
        "type": "string"
    }
}
}
}

```

1.5 Smart coffee machine thing implementation

After the develop of the Thing Description, it's possible to implement a coffe machine under the TD. In this exercise I choose to implement a smart coffee machine with the library node-WoT. node-WoT is a javascript library, using nodeJS, that allow, starting from a TD, to implement the thing and satisfy the rules of the thing description. Through this library it's possible to develop a REST API in order to interact with the thing using properties, actions and events as a first levels of abstraction.

In a first moment, I test the REST API with Postman. Later, i defined a user application in order to interact with the thing.

I initially tested the REST API for the smart thing using Postman. Subsequently I defined an application that can be used by users and maintainers to interact with the coffee machine.

1.6 User app implementation

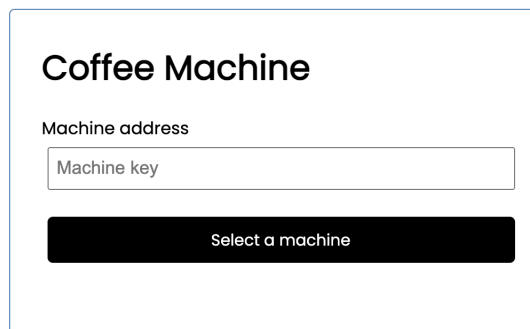
Using a common thing model defined by the Thing Description, allows me to define a single application to control different smart coffee machines implemented by different vendors.

I choose to develop a user web app using React, in order to create a responsive single page application that can be used by any browser and specifically by any smartphone.

1.6.1 Discovery of a smart coffee machine

For demo reasons, the discovery of smart coffee machines wasn't implemented as if the application were to be distributed to the public. The model, not implemented, is the presence of a registry in which to keep all the coffee machine. This registry associates an ID with each machine. Then a QR code is phisically put on the specific machine, which users can scan in order to connect with the specific coffee machine.

For the sake of simplicity in this demo the discovery of a smart coffee machine is done by manually entering the ID.



Coffee Machine

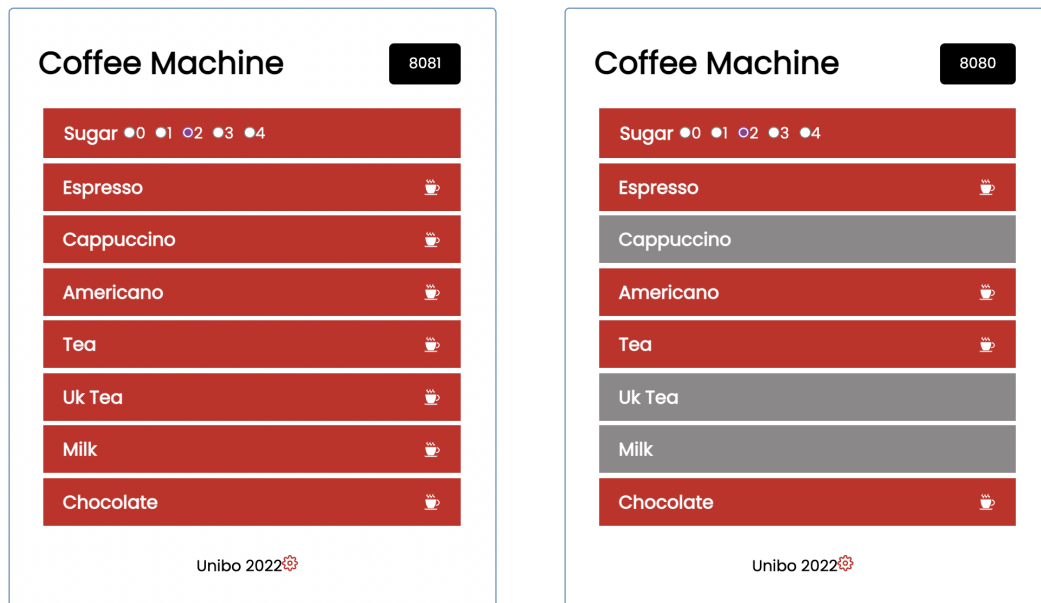
Machine address

Select a machine

Figure 1: Selection page

1.6.2 User side

Once a smart coffee machine has been identified, the user will be redirect to a page where he can view the menu and can choose to buy a drink or not, also specifying the level of the sugar.



(a) All drink available

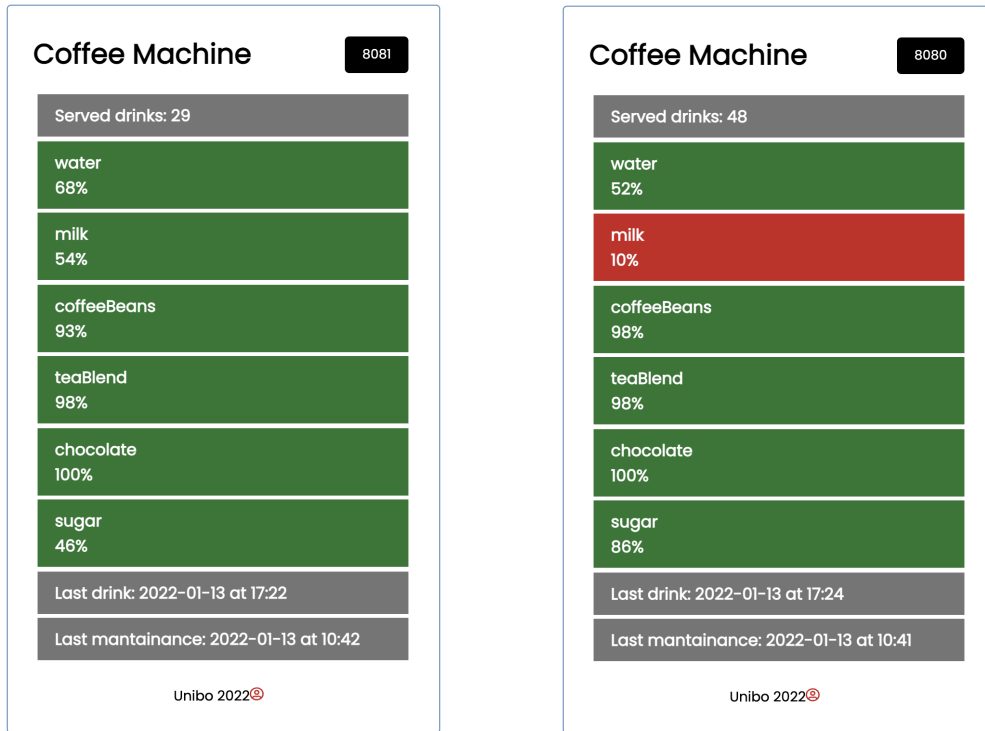
(b) Some drink unavailable

Figure 2: User side

The menu is not static, but is retrieved dynamically by the smart thing. In fact it's the smart coffee machine that decide the availability of a drink, based on the availability of the resources inside the thing. If a drink is not available, it will be highlighted and it will not be possible to order it from the menu.

1.6.3 Maintainer side

By clicking on the gear icon it's possible for a maintainer to view some useful information, like the level of the available resources, the date of the last maintenance or the drink's counter.



(a) Good level of resource

(b) Low resource

Figure 3: Maintainer side

Also in this page, if a resource is limited or is finished it will be marked differently from the others.

This page allow to monitor a single smart thing, but it doesn't yet have the aggregation feature useful for defining an IIOT solution.

2 Exercise 2: About things organisation and mashup

2.1 The problem

In a smart room we have the following smart things/services:

- A light thing
 - providing affordances to control the main light, including increasing/decreasing the intensity.
- A light sensor thing
 - providing affordances to know the current light level inside the room.
- A presence detector thing
 - providing affordances to know if there is someone in the room or not. An event is generated to signal entrance and exit.
- A vocal UI thing
 - providing affordances to get notified of the vocal commands issued by the user in the room.

2.1.1 Goal to achieve for the smart service

1. Don't waste energy
 - When no one is in the room, the light should be off.
 - The light should be off even in the case that the current light level is above a threshold LT1.
2. Keep a proper light level for the user
 - If there is someone in the room, then the light level should be kept not below some level LT2, by properly controlling the light.
3. Listen to the user
 - A user can request to switch on or off the light by issuing a proper vocal command
 - User's requests override policies (1) and (2), that is:
 - if the user requested to switch off the light, the light should be kept off or on, the light should be kept off or on in spite of (1)(2).
 - The user can request to reset overriding by issuing a "reset" vocal command.

2.2 Thing models

2.2.1 Light thing

The affordances designed for a light thing are the following:

- **Properties**
 - **isOn**: return if the light is on.
 - **isOff**: return if the light is off.
 - **intensity**: return the current intensity of the light
 - **status**: return the complete status of the light (state on/off and the intensity).
- **Actions**
 - **increase**: increase light intensity.
 - **decrease**: decrease light intensity.
 - **switchOn**: switch on the light.
 - **switchOff**: switch off the light.
 - **switch**: switch the light with the opposite status.
- **Events**
 - **changeState**: state on/off change.
 - **changeIntensity**: intensity change.

2.2.2 Light sensor thing

The affordances designed for a light sensor thing are the following:

- **Properties**
 - **lightLevel**: return the current light level of the room.

Since the brightness sensor is used only for measuring the ambient brightness, I choose to keep it as free of other functions as possible, leaving the mashup to implement them.

2.2.3 Presence detector thing

The affordances designed for a presence detector thing are the following:

- **Properties**
 - **presence**: return if the light is on
 - **presenceTimer**: the seconds after which, if no presence is detected, it is assumed that there is no one in the room.
- **Actions**

- **setPresenceTimer**: set the presence timer seconds.
- **Events**
 - **detectPresence**: The sensor detect a presence inside the room.
 - **nonDetectPresence**: the presence timer arrive to zero after the last presence detected.

2.2.4 Vocal UI thing

The affordance for a vocal UI thing are the following:

- **Properties**
 - **command**: the last command triggered by the user.
- **Events**
 - **newCommand**: the sensor detect a new command.

2.2.5 Thing implementation

All the previous thing are develop in javascript using node and node-WoT, as in the exercise one. Each thing therefore offers a REST API to interact with it. The choice to use http protocol guarantees interoperability with all applications that support the protocol and follows the WoT standards.

2.3 Static mashup using node-red

Mashup means aggregate different thing, their informations and their functionalities in order to create new control scenarios and new user application.

node-red allows to create static mashup based on a data flow, this development method is called flow programming. In this scenario, the smart service is made up of a series of nodes that produce and consume events. The composition of these nodes and the related data that pass through they, create the flows. The data that pass node by node are processed, modified and used in order to reach different goals.

node-red was used with the [WoT package](#), that allows to exploit the abstraction defined by W3C and interact with the different things implemented before.

The WoT package for node-red offers different node that allow to subscribe to events, read and write properties and trigger actions.

Initially, in a static way, the url was entered to which it was possible for node-red to fetch the Thing Descriptions for the four smart things. In this way the program becomes aware of the affordances offered by the things and makes it easy to perform different actions.

It was decided to develop 4 different flows to achieve the goal in the specification document. The first 3 flows implement the 3 main functions that a smart room must have, the fourth on the other hand, it was used to interact with the system and to perform debug prints, to leave the other main nodes free from non characterizing behaviors.

2.3.1 DontWasteEnergy flow

In this flow there are two main purposes, determined by the two actions **switchOn** and **switchOff**.

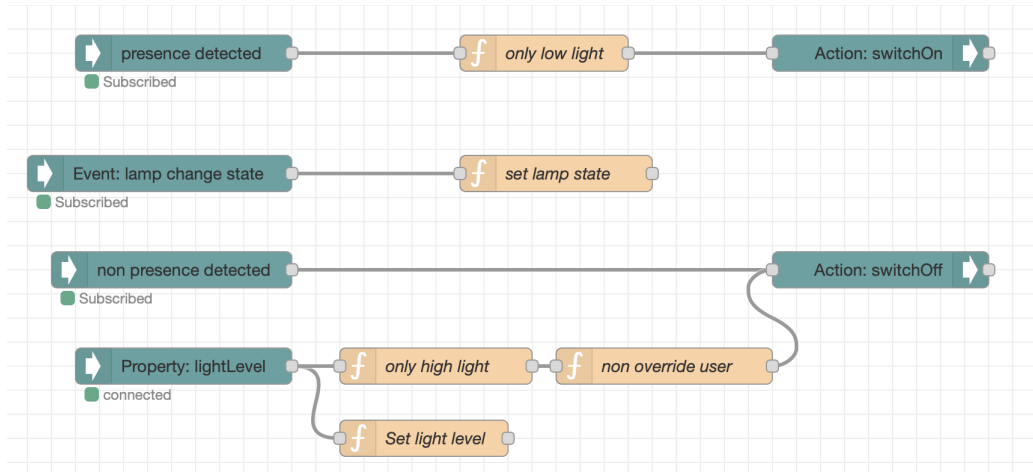


Figure 4: The DontWasteEnergy flow

You therefore want to turn on the light if there is someone inside the room and the light is below a certain threshold and, moreover, keep the light off if there is no one in the room.

Specifically, in this flow there are node that subscribes to the **detectPresence** and **nonDetectPresence** events, in order to detect if there is someone in the room or not.

The flows then continues with the actions **switchOn** and **switchOff**, first passing through a filter that checks, in case of **presenceDetected** event, if the light in the room is low and there is the need to switch on the light.

The property **lightLevel** is also observed. Observing a property means polling it, requesting the value every X seconds. Through this observation, the light in the room is constantly monitored and leads to take a decision on wether to turn on or of the light, also considering the amount of light in the room.

The polling is necessary, even if not wanted, because the node **readProperty** of the WoT package doesn't accept any data in input. It's therefore not possible, not even asynchronously, asking for a property. It is always necessary to do polling in order to do that.

To allow different nodes to know the property, it was necessary to insert the value of the property in a global variable, in order to read it when needed.

2.3.2 ProperLightLevel flow

This flow has the goal of keeping the light on a target light threshold decided by the user. When someone is inside the room and the light is on, the total amount of light in the room is kept at a certain level, by correctly interact with the light thing's intensity.

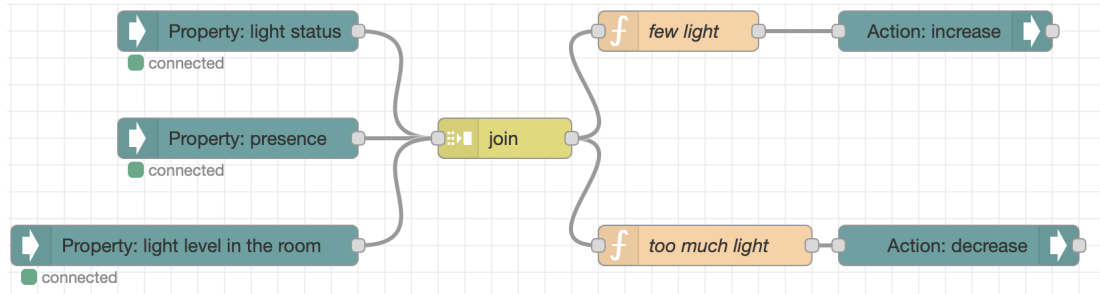


Figure 5: The ProperLightLevel flow

In practice, several pieces of information are joined: the status of the light thing (on / off and intensity), the presence of someone in the room and the level of light inside the room. The flow then continues with two filters **few light** and **too much light**, which take the data from the previous node and trigger the actions **increase** and **decrease** to the light thing.

2.3.3 ListenUser flow

This flow subscribes to the **newCommand** event of the vocal UI thing to intercept the voice commands given by the user. At the reset command, a global variable is reset. At the command *to switch on the light*, the action **switchOn** is triggered on the light thing. At the command *to switch off the light*, the action **switchOff** is triggered on the light thing. After each command, a global variable is set, in order to determine if the user is overriding a command. In the previously described flows it's possible to view this filter and understand that, if the user has given a command, then the other flows have limited effects to the room.

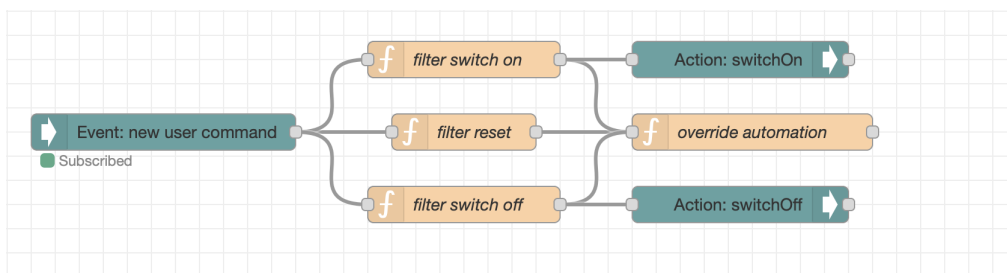


Figure 6: The ListenUser flow

2.3.4 Debug flow

This flow was created to be able to interact with the room, by simulating someone's presence, user's vocal command and different ambient light. Furthermore, various events and properties are collected in order to present data useful for debugging.

Figure 7 shows that it's possible to associate a luminosity level, a presence in the room and a vocal command. Furthermore, the first **inject** allows to set a series of global parameters, like the light threshold.

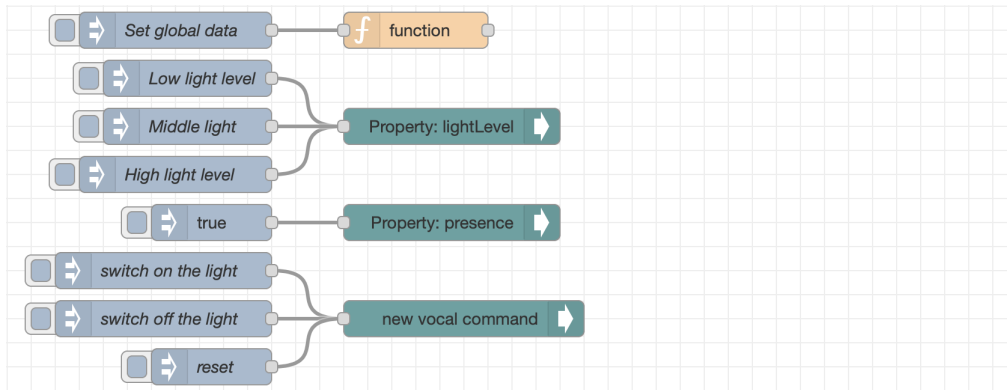


Figure 7: The debug flow: user interaction part

Figure 8 shows the different events observed and the different properties read, in order to show them in console.

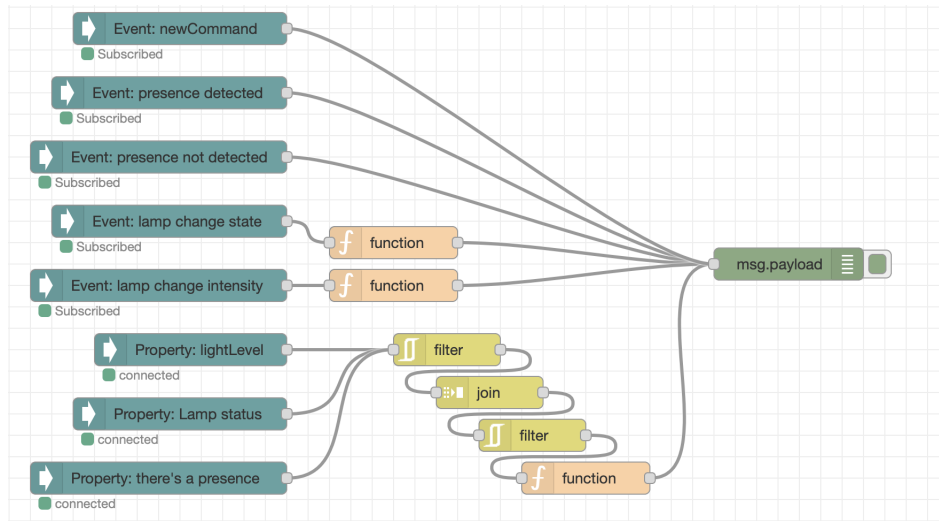


Figure 8: The debug flow: log part

An example of this data is reported in figure 9.

2.3.5 node-red consideration

node-red is a graphical and low code tool that allows to easily compose different things in order to create new applications.

One of the major advantages of node-red is that is event and flow based. On the other hand, one of the more disadvantages is the handle of complexity that node-red has give it to me. Being a low-code development environment doesn't give node-red the flexibility to develop complex apps.

Even for developing a simple application like this, I felt stuck inside the node-red binaries, not allowing me to develop as I wanted.

```
19/1/2022, 17:41:43  node: 984f763f85c28181
msg.payload : string[32]
"The sensor has detected movement"

19/1/2022, 17:41:43  node: 984f763f85c28181
presenceInRoom : msg.payload : Object
▼ object
▼ lampStatus: object
  state: "off"
  intensity: 0
  lightLevel: 2000
  presenceInRoom: true
  lightLevelWithLamp: 2000

19/1/2022, 17:41:43  node: 984f763f85c28181
msg.payload : string[26]
"New state of the light: on"

19/1/2022, 17:41:44  node: 984f763f85c28181
lampStatus : msg.payload : Object
▼ object
▼ lampStatus: object
  state: "on"
  intensity: 100
  lightLevel: 2000
  presenceInRoom: true
  lightLevelWithLamp: 3000
```

Figure 9: Example of log for the static mashup

2.4 Agents mashup

An agent is a first level abstract that allows to model situated software. It can be seen as a reactive component, as an actor, but also active and proactive. An actor reacts only to events or external requests, while an agent, with his active/proactive component, tries to reach one or more goals.

An agent continuously observes the environment in which it's located and, starting from a series of goal, tries to reach them. The tasks, or goals, that can be submit to an agent can be divided into two main categories, achieved and maintained.

Achieved task the agent tries to accomplish the goal by doing some operations, in order to bring the environment into a state where the task is satisfied. When the agent reaches a state where the task is satisfied, it stops.

Maintained task the agent continuously observe the environment in which it is situated and tries to maintain the state of the environment stable over time. So in this type of task the state is maintained stable over time, on the other hand.

2.4.1 Model

The system designed is made up three principal agents which try to satisfy the three main tasks required by the specification document: don't waste energy, proper light level and listen user.

The way in which I specify to an agents what to do is the traditional one, that is the way in which the specific code to satisfy a task is passed to the agent and it executes the code depending on the situation. It's also necessary to specify how the agent observes the environment and interacts with it.

The task are of type maintained, so the agents are started and try to maintain the environment in a stable situation.

Specifically, the agents developed are the following:

- **DontWasteEnergyAgent:** the agent observes the presence detector thing's event to understand when switch on the light (there's a presence in the room) and, on the other hand, keep the light off when no one is in the room.
- **ProperLightAgent:** the agents observes the light sensor thing and the light thing's event, in order to understand if inside the room there is the right light level and, if not, it acts on the intensity of the light thing.
- **ListenUserAgent:** the agent listens for any user vocal commands and tries to accomplish them. This agent communicates with other agents in order to signal when their behaviors must not be active because the user has bypassed them.
- **DebugAgent:** this agent is an observer of the environment. It takes care of reporting the data to the end user (the developer) in a correct and legible manner. In a real system, this agent would not exist because the interactions of the agents

with the environment would be visible in the environment itself (turning on a real light, listening to voice commands).

2.4.2 Implementation

For the implementation of the thing, through node-wot, see section 2.3.

The system was implemented as an event-oriented multi-agent system. Java was used as a language, while Vertx as an event framework. Vertx allows to make asynchronous calls and to work with promises, to which it is possible to associate handlers that manage the result of asynchronous calls.

API The first part, that of API, sees the creation of four different APIs for the four different things, in which are defined the operations that can be performed on a specific thing. The structure of light thing is shown below.

```
public interface LightThingAPI {

    /** PROPERTIES */
    Future<Boolean> isOn();

    Future<Boolean> isOff();

    Future<Integer> getIntensity();

    Future<String> getStatus();

    /** ACTIONS */
    Future<String> increase();

    Future<String> increase(Integer step);

    Future<String> decrease();

    Future<String> decrease(Integer step);

    Future<String> switchOn();

    Future<String> switchOff();

    /** EVENTS */
    void subscribeToChangeState(Handler<String> handler);

    void subscribeToChangeIntensity(Handler<Integer> handler);
}
```

Proxy Through a **proxy**, the operation on the thing was then decoupled from the protocol with which it is done, in this specific case, http. The proxy implements the API, making the correct calls on the thing.

An example of implementation is the following, in which the reading of the property **isOn** on light thing is implemented. Through Vertx, an asynchronous http call is made and a future is returned, which in Vertx is similar to a javascript promise to which a handler can be associated.

```
@Override
public Future<Boolean> isOn() {
    Promise<Boolean> promise = Promise.promise();
    client.get(thingPort, thingHost, PROPERTY_ISON)
        .send()
        .onSuccess(response -> {
            promise.complete(response.bodyAsString());
        })
        .onFailure(err -> {
            promise.fail("Can't retrieve light level from: " + thingId);
        });
    return promise.future();
}
```

Agents Agents on the other hand are implemented as **Verticle**, which are based on event-loops. Agents in this way are active-reactive components, which communicate with other agents through messages on the event bus, and wait for events, which can be events concerning a thing, or events concerning messages on the event bus.

Below is an example, for DontWasteEnergyAgent, which waits for the presence detector thing and performs the operation to turn on the light.

```
presenceDetector.subscribeToDetectPresence(presence -> {
    this.presence = true;
    if (lightLevel < highLightThreshold) {
        if (userCommand.isEmpty()){
            if (!lightIsOn){
                log("Presence detected, going to switch on the light");
                lightThing.switchOn();
            } else
                log("Presence detected,
                    but nothing to do because the light is already on");
        } else {
            log("Presence detected,
                but nothing to do because the user set a command");
        }
    }
});
```

Beliefs Each agent has local variables, as in the reported case **lightLevel**, **lightIsOn**. These represent the agent’s knowledge of the environment and come from subscribing to events on the various things. These variables can be seen as **beliefs**, what the agent thinks to know about the environment, which however may not be true. In fact, it could happen that at the same time that the agent has a beliefs, the environment has been changed, making the knowledge of the agent obsolete.

Gui The developed mashup is not meant for a graphical interface. In fact, interacting with the system would mean entering a room or giving voice commands, as well as the actions performed by the agents would be visible on the environment, such as switching on a light or changing its intensity.

This aspect was simulated using a simple user interface, shown below.

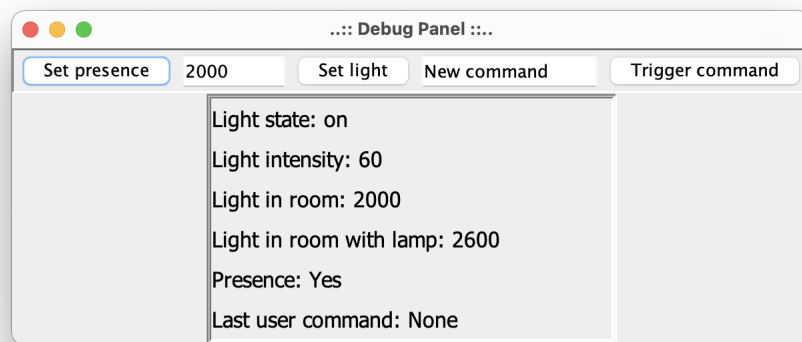


Figure 10: User interface for room agents mashup

3 Exercise 3: About Industrial IOT and Digital Twin

3.1 The problem

Let's consider the smart coffee machine case of exercise 1. In this case we consider the maintainer point of view.

A maintainer is interested in monitoring the state and behaviour of a smart coffee machine, in order to provide a better service and optimize resources. In particular a maintainer is interested to:

- track the quantity of the consumed/remaining resources and items (e.g. coffee, tea, sugar, glasses,...), in order to organize the refill
- track the machine usage, to generate alerts if some specific pattern occurs
- monitor the state of the machine (working, not available, out of service), in order to promptly react to problems

A single maintainer can have a large dynamic set of smart-cm to manage, distributed in the territory.

3.2 Goals

Given this scenario:

- Think about a cloud-based solution to manage the set of smart-cm, to be modelled using the digital twin architecture, eventually integrated with the solution designed in exercise 1
- Implement a simplified version adopting any available IIoT / DT platform (e.g. Ditto, Azure) or a custom ad hoc implementation, integrated with the prototype developed in exercise 1, in particular including a simple dashboard (as a web app) that makes it possible to track and monitor the smart-cms.

3.3 Model

Initially, the basic idea was to use a DT platform such as Ditto or Azure. The first approach with these platforms however, led to a preference for a completely custom ad hoc solution, due to the fact that to become familiar with Ditto or Azure, it would take many days.

The solution designed is composed by two parts, an **API** and a **web app**.

The API acts as a kind of hub and manager for digital twins. It is possible through it to register new coffee machines and keep them updated to the current state through **shadowing**.

The web app part, on the other hand, uses this API and the data it makes available, to allow the maintainer to view the list of the coffee machines, their status, properties and receive events from them.

3.3.1 API

The API acts as a kind of hub and manager for digital twins. It is possible through it to register new coffee machines and keep them updated to the current state through shadowing. The API also provides methods to have a list of all the coffee machines with the main information and expand them to have a detail of a specific machine, if required.

The API handles events part via web socket. A web socket server has been implemented, when digital twins produce an event, this is send via websocket to all the subscribers.

Digital twin implementation In the solution designed, each coffee machine is represented with its own digital twin, that is the digital representation of the physical object. It was found difficult to think and create the digital twin, probably because the coffee machine developed with node wot in exercise 1 was already the digital representation of the physical object, being simulated.

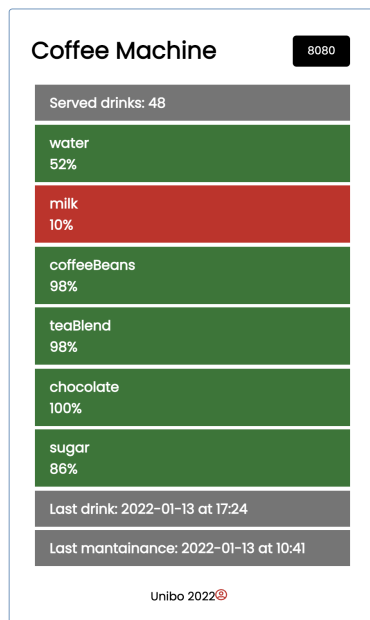
The solution designed for the digital twin is therefore very similar to the one seen in the lesson and similar to the proxy object created for exercise 2.

Hub and data collection The API offers the possibility to a maintainer of registering his coffee machines and keeping them updated at their current state, displaying useful data, so that the maintainer can be warned in case of limited/finished resources, or malfunctions, and plan a maintenance.

3.3.2 Web App

The web app part uses the API and the data it makes available, to allow the user to view the coffee machines, view their status, properties and receive events from them. It was developed, as in the case of the app in exercise 1, through the React framework.

Using the following screen, the user can add a coffee machine to his list, and keep it monitored. In red are shown the coffee machines with exhausted resources or maintenance problems, while in green are shown the correctly functioning coffee machines. These coffee machines are retrieved from the API, where they are represented as Digital Twins.



The 'Maintainer' interface allows users to add new machines and view existing ones. It includes a form with fields for 'Machine host' and 'Machine port', and a button 'Add new machine'. Below the form, a list of machines is shown, with 'localhost:8081' in green and 'localhost:8080' in red.

By clicking on one of the machines, it's possible to view the **detail** of the specific machine, like the available resources and other statistics.

The **events**, on the other hand, were represented as popups, even if the original idea, much more efficient, was to create a real notification center. Either way, notifications look like this. By clicking on one of the notifications, it's possible to view the problem in detail.

