nodeLab2018

Antonio Natali

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
antonio.natali@unibo.it

Table of Contents

nodeLab2018 1
Antonio Natali
1 Introduction 2
1.1 Object-oriented architectures 2
1.2 Start up 2
1.3 Decoupling from technological details 3
1.4 An architectural style 3
1.5 A frontend server 3
2 System models 5
2.1 Mock objects 5
2.2 Start up 5
2.2.1 The sensor. 5
2.2.2 The actuator. 6
2.2.3 The controller. 6
2.3 Sensor/Actuator models 7
2.3.1 The actuator. 7
2.3.2 The controller. 8
2.4 MVC 8
2.4.1 The Resource Model. 8
2.4.2 The system. 9
2.4.3 The controller. 9
2.4.4 An actuator: a Led Mock. 10
2.4.5 Another actuator: a Led on Arduino. 10
2.4.6 A Led on RaspberryPi. 11
3 A frontend server 12
3.1 Starting 13
3.2 Refactoring according to the MVC pattern 13
3.3 The Express use pattern 14
3.4 The server entry-point 15
3.5 applCode 16
3.5.1 Routing rules. 16
3.6 Routers: sensors 17
3.7 Routers: actuators 18
3.7.1 MQTT utils 19
3.8 Working with the server 19
3.9 From components to systems 22
3.10 A resource model in JSON 23
4 Node on RaspberryPi 25
4.1 Blink a Led with js 25
4.2 Led plugin 26
4.3 Temperature/Humidity plugin 27
4.4 An application 28
5 A new virtual robot 30
5.1 From the code to a model 32
5.2 Towards robot-based applications 34

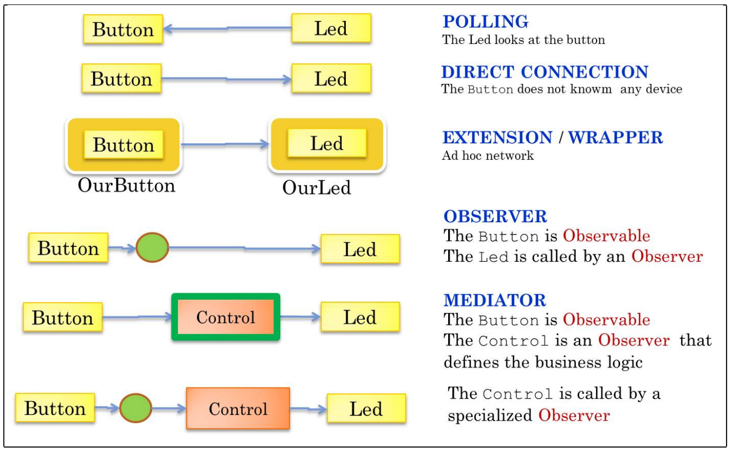
5.2.1 Step0: the system to build as a black-box . . . 35
5.2.2 Step1: the system as an actor . . . 35
5.2.3 Step2: the system with agent / adapter components . . . 35
5.2.4 Toward industrial standards . . . 36
6 A frontend server for the robots . . . 37
6.1 A model for the robot . . . 38
6.2 A robotFrontendServer . . . 38
6.2.1 The application code . . . 39
6.2.2 Feature configuration . . . 40
6.2.3 The user command page . . . 40
6.2.4 The interpretation of commands . . . 41
6.2.5 Internal execution of commands . . . 42
6.2.6 External execution of commands . . . 43
6.2.7 emitRobotCmd . . . 43
7 User authentication . . . 45
7.1 Setting up the authorization support . . . 45
7.1.1 Setting up the passport . . . 46
7.2 Handling authorization . . . 46
7.2.1 Check authorization . . . 47
7.3 A model for the user . . . 47
7.4 Handling (new) users . . . 49
7.5 View/Edit user's profile . . . 50
8 Final task 2018 . . . 51
8.1 Requirements . . . 51
55

1 Introduction

In this work we intend to build `IOT` applications as simple systems composed of sensors and actuators. For example, the sensor could be a Button or a Temperature sensor and the actuator could be a Led. Since our goal is to focus on the role of the architecture in software development, let us introduce first of all an overview of our logical workflow.

1.1 Object-oriented architectures

If our reference programming model is based on the traditional *object-oriented* paradigm in a *non-distributed* environment, a simple `ButtonLed` system can be designed and built by staring from one of the architecture informally introduced in the following picture:



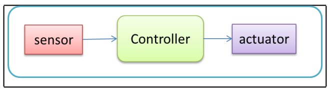
Since the application code cannot be responsibility neither of the Button nor of the Led, the schemes including an explicit `Control` component will be taken as our reference architectures.

Our goal now is to generalize the discussion by considering a set of possible sensors/actuators working in a distributed system with reference to some precise requirement; for example:

> `R0a`: When a Button is pressed, a Led must start blinking. When the Button is pressed again, the Led blinking stops.
> `R0b`: When the value of a Temperature sensor is higher than a prefixed value, a Led must be turned on; otherwise the Led is off.

1.2 Start up

Our first reference (distributed) architecture can be informally introduced as a `Control`-based architecture [1]



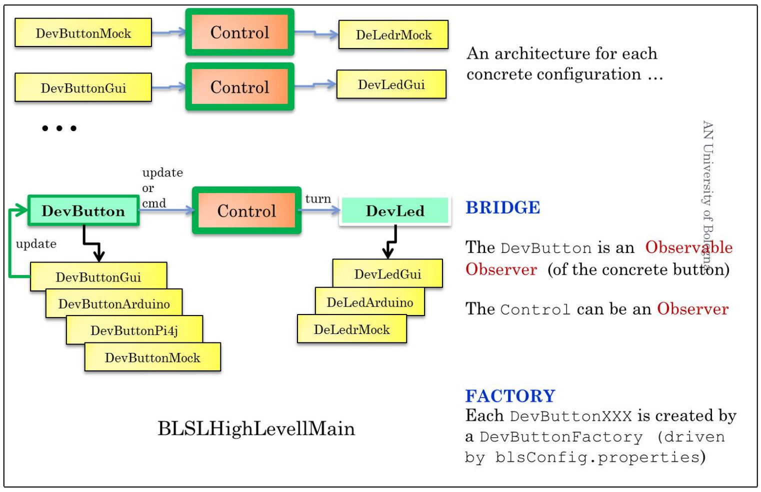
The basic idea is that each time the sensor change its state, the `Controller` performs some action on the actuator.

For an example, see Subsection 2.2.

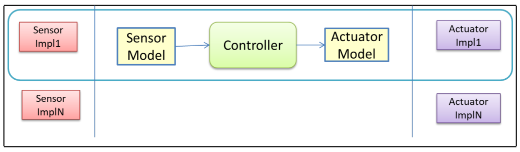
[1] The reader should decide whether this architecture scheme is the result of an analysis phase or a project phase.

1.3 Decoupling from technological details

Sensors and actuators can be of different types or can be of a specific type (e.g. a Temperature sensor, a Led) but with different possible implementations. An object-oriented approach can be based on appropriate design patterns:



More generally, our reference architecture could evolve by introducing `models` to decouple the controller (the business logic) form technological details:

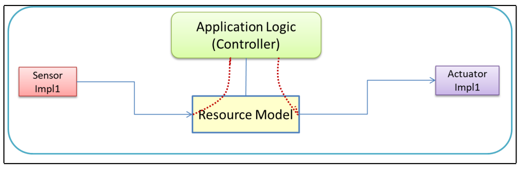


The idea is that each specific, technology-dependent sensor provides its own way to modify the sensor-model, while each modification in the model of the actuator-model should trigger an action in the technology-dependent actuator. The software designer can make reference to the `observer pattern` and/or to the `Model-View-Control (MVC)` architecture.

For an example, see Subsection 2.3.

1.4 An architectural style

The introduction of models for sensors and actuators lead us to propose a more general 'architectural style':

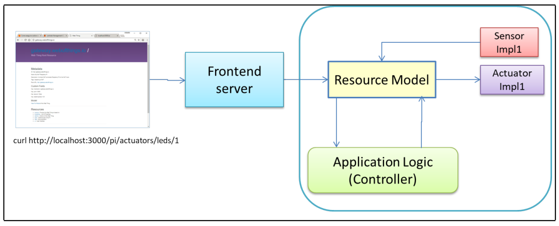


The idea is that the software designer should concentrate the attention on the most appropriate `Resource Model` in the application domain and delegate to the next step of 'architectural zooming' the details of the binding between the model and the concrete devices.

1.5 A frontend server

The last step could consist in introducing a `frontend` server so that:

R1: An human user or a machine can send command over the network to modify the state of an actuator (e.g. the Led) or to see the current state of a sensor (e.g. a Temperature sensor).



The idea is that the server should provide all the stuff required for (human) user interaction while reusing the system we have developed so far.

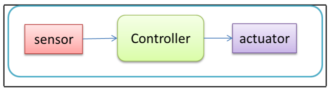
2 System models

Before entering in implementation details about sensors and actuators, let us capture in a formal way the different architectures introduced in Section 1.

2.1 Mock objects

In this section we will use a custom **Java** class (named `it.unibo.custom.guicustomBlsGui`[2]) that provides a Button Mock and a Led Mock as **GUI**-based components.

2.2 Start up



In this example, we will consider a ButtonLed system with the requirement `R0b` of Subsection 1.1.

Our formal specification starts with the definition of the events/messages used by the components to exchange information[3]:

```
System bls1

Event sensorEvent : sensorEvent( DATA ) //DATA : integer
Event ctrlEvent : ctrlEvent( CMD )  //CMD = on | off

Context bls1Ctx ip [ host="localhost" port=8019 ]
```

Listing 1.1. `bls1.qa`

The system is composed of 3 components, each modelled as an actor: a sensor, a controller and an actuator.

2.2.1 The sensor.

The sensor is modelled as an emitter of `sensorEvent`:

```
QActor qasensor context bls1Ctx {
    Plan init normal [
        println( qasensor(starts) );
        delay 1000;
        emit sensorEvent : sensorEvent( 20 ) ;
        delay 1000;
        emit sensorEvent : sensorEvent( 30 ) ;
        delay 1000;
        emit sensorEvent : sensorEvent( 28 ) ;
        delay 1000;
        emit sensorEvent : sensorEvent( 35 ) ;
        delay 1000
    ]
}
```

Listing 1.2. `bls1.qa`

At the moment we do not pay attention to any concrete device, since our goal is to capture the essence of the architecture.

[2] The class `guicustomBlsGui` is defined in the project `it.unibo.bls17.naive.qa`.

[3] At the moment we suppose to work within a single machine (*Context*), but we know that it will be easy to give to each component its own context.

2.2.2 The actuator.
The actuator is modelled as an actor that waits for a `ctrlEvent` event and than performs its job by using a Led Mock provided by the custom `Java` class `customBlsGui`[4].

```
QActor qaactuator context bls1Ctx{
    Plan init normal [
        println( qaactuator(starts) ) ;
        javaRun it.unibo.custom.gui.customBlsGui.createCustomLedGui()
    ]
    switchTo waitForCommand

    Plan waitForCommand[ ]
    transition stopAfter 100000
        whenEvent ctrlEvent -> handleCmd
    finally repeatPlan

    Plan handleCmd resumeLastPlan[
//      printCurrentEvent;
        onEvent ctrlEvent : ctrlEvent(on) -> javaRun it.unibo.custom.gui.customBlsGui.setLed("on");
        onEvent ctrlEvent : ctrlEvent(off) -> javaRun it.unibo.custom.gui.customBlsGui.setLed("off")
    ]
}
```

Listing 1.3. `bls1.qa`

2.2.3 The controller.
The controller is modelled as an actor that waits for a `sensorEvent` and then fulfils the requirement `R0b` of Subsection 1.1:

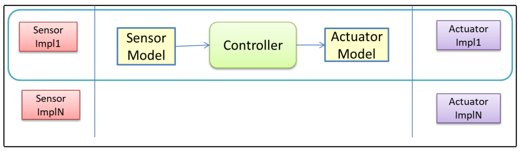
```
QActor qacontrol context bls1Ctx{
Rules{
    eval( ge, X, X ).
    eval( ge, X, V ):- eval( gt, X , V ) .
    evalTemperature( cold ) :-
        curTemperatureValue(V),
        //output(evalTemperature(V)),
        eval( lt, V , 30 ).
    evalTemperature( hot ) :-
        curTemperatureValue(V),
        //output(evalTemperature(V)),
        eval( ge, V , 30 ), !.
 }
    Plan init normal [
        println( qacontrol(starts) )
    ]
    switchTo waitForSensorEvent

    Plan waitForSensorEvent[ ]
    transition stopAfter 100000
        whenEvent sensorEvent -> handleSensorEvent
    finally repeatPlan

    Plan handleSensorEvent resumeLastPlan [
        printCurrentEvent;
        onEvent sensorEvent : sensorEvent( V ) ->
                   ReplaceRule curTemperatureValue(X) with curTemperatureValue(V);
        [ !? evalTemperature(hot) ] emit ctrlEvent : ctrlEvent(on) else emit ctrlEvent : ctrlEvent(off)
    ]
}
```

Listing 1.4. `bls1.qa`

Note that the business logic is captured in a declarative style by means of `Prolog` rules.

[4] The reader could use - within the sensor - the Button Mock provided by `customBlsGui` that works as an event generator.

2.3 Sensor/Actuator models



In this example, we will consider a ButtonLed system with the requirement `R0a` of Subsection 1.1.

Our formal specification starts with the definition of the events/messages used by the components to exchange information[5]:

```
System bls1m
Dispatch turn   : switch
Event local_click : clicked(N) //N : natural

Context bls1mCtx ip [ host="localhost" port=8049 ]
```

Listing 1.5. `bls1m.qa`

The system is composed of 2 components, each modelled as an actor: a controller and an actuator. In this formalization there is no explicit model for the sensor (Button). The sensor is now embedded as a Button Mock within the controller. since the Button provided by the class `customBlsGui` (xssmocks) is already modelled as a resource that emits an event when changes its state.

2.3.1 The actuator.

The Led (actuator) model is represented by the Prolog fact:

```
ledmodel( name(led1), value(off) ).
```

The Led knowledge-base provides also rule to modify the model:

```
QActor qaledm context bls1mCtx {
Rules{
   ledmodel( name(led1), value(off) ).
   switchLedValue(on) :-
      ledmodel( name(led1), value(off) ),
      replaceRule( ledmodel( NAME,value(off) ), ledmodel( NAME,value(on) ) ), !.
   switchLedValue(off) :-
      ledmodel( name(led1), value(on) ),
      replaceRule( ledmodel( NAME,value(on) ), ledmodel( NAME,value(off) ) ), !.
}
```

Listing 1.6. `bls1m.qa`: the led model

The Led is modelled as an actor that waits for a `turn` dispatch. Its task now is to execute the `switchLedValue` rule when a `turn` event is perceived. Since the rule binds a variable to the current state of the Led, it can be put in execution as a guard that allows us to execute a proper action on a concrete implementations (e.g. a Led Mock provided by the custom `Java` class `customBlsGui`):

```
   Plan init normal [
      javaRun it.unibo.custom.gui.customBlsGui.createCustomLedGui();
      delay 100;
      [ !? ledmodel( NAME, value(V) )] javaRun it.unibo.custom.gui.customBlsGui.setLed(V)
   ]
   switchTo waitForCmd

   Plan waitForCmd [ ]
   transition stopAfter 3000000
      whenMsg turn -> ledswitch
   finally repeatPlan

   //model-based behavior
   Plan ledswitch resumeLastPlan[
```

[5] At the moment we suppose to work within a single machine (*Context*), but we know that it will be easy to give to each component its own context.

```
        [ !? switchLedValue(V) ] javaRun it.unibo.custom.gui.customBlsGui.setLed(V)
    ]
}
```

Listing 1.7. `bls1m.qa`: the behaviour

2.3.2 The controller.

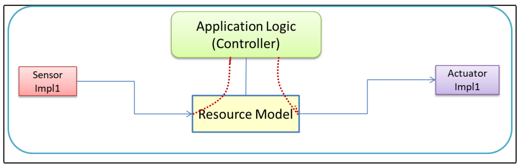
The controller is modelled as an actor that waits for a `local_click` event emitted by a Button Mock and then forwards a `turn` dispatch to the actuator:

```
QActor qacontrolm context bls1mCtx{
    Plan init normal [
        println( qacontrol(starts) ) ;
        javaRun it.unibo.custom.gui.customBlsGui.createCustomButtonGui()
    ]
    switchTo waitForClick

    Plan waitForClick[ ]
    transition stopAfter 100000
        whenEvent local_click : clicked(N) do forward qaledm -m turn : switch
    finally repeatPlan
}
```

Listing 1.8. `bls1m.qa`

2.4 MVC

We start the formalization of our MVC architecture with reference to the informal picture of Subsection 1.4:



In this section, we are making reference to the requirement `R0b` of Subsection 1.2.

2.4.1 The Resource Model.

In order to concentrate our attention on the most appropriate `Resource Model` in the application domain, let us introduce such a model as a `Prolog` Theory `resourceModel.pl`:

```
model( type(actuator, leds),   name(led1), value(off) ).
model( type(sensor, temperature), name(t1), value(25) ).
```

Listing 1.9. `resourceModel.pl`: resource model

Each resource must have a `type`, a `name` and a resource-specific `value`. Of course the `Prolog` syntax is not the only way to specify a resource model. The current trend is to use the `JSON` (*JavaScript Object Notation*) lightweight data-interchange format. However, an advantage of modelling resources in `Prolog` is the possibility to introduce declarative rules to get/modify the model:

```
getModelItem( TYPE, CATEG, NAME, VALUE ) :-
       model( type(TYPE, CATEG), name(NAME), value(VALUE) ).
changeModelItem( CATEG, NAME, VALUE ) :-
       replaceRule(
           model( type(TYPE, CATEG), name(NAME), value(_) ),
           model( type(TYPE, CATEG), name(NAME), value(VALUE) )
       ),!,
       %%output( changedModelAction(CATEG, NAME, VALUE) ),
       ( changedModelAction(CATEG, NAME, VALUE) %%to be defined by the appl designer
         ; true ).   %%to avoid the failure if no changedModelAction is defined
```

Listing 1.10. `resourceModel.pl`: model get/change rules

The `changemodelitem/3` rule ends by calling a `changedModelAction/3` to be written by the application designer in order to specify actions to be done after a model change. To facilitate the work of the application designer, let us introduce also some utility rules:

```
eval( ge, X, X ) :- !.
eval( ge, X, V ) :- eval( gt, X , V ) .

emitevent( EVID, EVCONTENT ) :-
    actorobj( Actor ),
    %%output( emit( Actor, EVID, EVCONTENT ) ),
    Actor <- emit( EVID, EVCONTENT ).
%%% initialize
initResourceTheory :- output("initializing the initResourceTheory ...").
:- initialization(initResourceTheory).
```

Listing 1.11. `resourceModel.pl`: utility rules

The rule `emitevent/2` can be used to emit events with reference to the current working actor, given by the fact `actorobj/1`.

2.4.2 The system.

Our formal specification of the system starts with the definition of the events/messages used by the components to exchange information:

```
System blsMvc
Event sensorEvent : sensorEvent( NAME, DATA )
Event changeModel : changeModelItem( TYPE, CATEG, NAME, VALUE )
Event ctrlEvent : ctrlEvent( CATEG, NAME,CMD )  //CMD depends on CATEG/NAME

Event inputCtrlEvent : inputEvent( CATEG, NAME, VALUE )
Event outputCtrlEvent : outputEvent( DATA ) //DATA : integer

pubSubServer "tcp://localhost:1883"
//pubSubServer "tcp://192.168.43.229:1883"
//pubSubServer "tcp://m2m.eclipse.org:1883"
//pubSubServer "tcp://test.mosquitto.org:1883"

Context blsMvcCtx ip [ host="localhost" port=8019 ] -httpserver
EventHandler evadapter for sensorEvent { //maps a sensorEvent from t1 into a inputCtrlEvent
emit inputCtrlEvent fromContent sensorEvent( t1, DATA ) to inputEvent( temperature, t1, DATA )
};
```

Listing 1.12. `blsMVC.qa`

Note that each `sensorEvent` emitted by the temperature device named `t1` is now mapped into a `inputCtrlEvent`.

2.4.3 The controller.

Our controller now:

- reacts to `inputCtrlEvent` events emitted after a change in the sensor model;
- performs its task by changing the model of some resource, by using the `changemodelitem/3` rule;
- specify `changedModelAction/3` rules that will be executed after the change of the model resource;
- exploits (within `changedModelAction/3`) the `emitevent/2` action to propagate actuator-change information (the `ctrl-event`) to other actors that can perform concrete actions with reference to real o mock devices.

```
QActor mvccontroller context blsMvcCtx -pubsub{
Rules{ //The model is in the theory resourceModel.
       //Here we write the actions to be performs when the model changes.
       //The change of the temperature t1 could modify a Led
    limitTemperatureValue( 25 ).
    changedModelAction( temperature, t1, V ):-
           limitTemperatureValue( MAX ),
           eval( ge, V , MAX ), !,
           changeModelItem( leds, led1, on).
    changedModelAction( temperature, t1, V ):-
           changeModelItem( leds, led1, off).
```

```

    //The change of a Led model must activate an actuator (working as an event listener)
    changedModelAction( leds, led1, V ):-
            emitevent( ctrlEvent, ctrlEvent( leds, led1, V) ).
 }
    Plan init normal [
        demo consult("./resourceModel.pl"); //contains the models and related rules
        println( qacontrol(starts) )
    ]
    switchTo waitForInputEvent

    Plan waitForInputEvent[ ]
    transition stopAfter 6000000
        whenEvent inputCtrlEvent -> handleInputEvent
    finally repeatPlan

    Plan handleInputEvent resumeLastPlan [
//      demo a;
        printCurrentEvent;
        onEvent inputCtrlEvent : inputEvent( CATEG, NAME, VALUE ) -> //change the model
            demo changeModelItem( CATEG, NAME, VALUE )
    ]
}
```

Listing 1.13. `blsMVC.qa`: the controller

The technology details related to the usage of a specific Led can be embedded in a actor that waits for a `ctrl-event` and then exploit its own technology.

2.4.4 An actuator: a Led Mock.

A Led Mock as a GUI can be introduced as follows:

```
QActor ledmockgui context blsMvcCtx -pubsub{
    Plan init normal [
        println( ledmockgui(starts) ) ;
        javaRun it.unibo.custom.gui.customBlsGui.createCustomLedGui()
    ]
    switchTo waitForCommand

    Plan waitForCommand[ ]
    transition stopAfter 100000
        whenEvent ctrlEvent -> handleCmd
    finally repeatPlan

    Plan handleCmd resumeLastPlan[
//          printCurrentEvent;
        onEvent ctrlEvent : ctrlEvent(leds, led1, on) -> javaRun it.unibo.custom.gui.customBlsGui.setLed("on");
        onEvent ctrlEvent : ctrlEvent(leds, led1, off) -> javaRun it.unibo.custom.gui.customBlsGui.setLed("off")
    ]
}
```

Listing 1.14. `blsMVC.qa`: a Led Mock

2.4.5 Another actuator: a Led on Arduino.

A Led working on Arduino can be introduced as follows:

```
QActor ledarduino context blsMvcCtx {
    Plan init normal [
        println( ledarduino(starts) );
        javaRun it.unibo.utils.arduino.connArduino.initPc("COM9", "9600")
    ]
    switchTo waitForCommand

    Plan waitForCommand[ ]
    transition stopAfter 6000000
        whenEvent ctrlEvent -> handleCmd
    finally repeatPlan

    Plan handleCmd resumeLastPlan[
//          printCurrentEvent;
        onEvent ctrlEvent : ctrlEvent(leds, led1, on) ->
```

```
        javaRun it.unibo.utils.arduino.connArduino.sendToArduino("1");
      onEvent ctrlEvent : ctrlEvent(leds, led1, off) ->
        javaRun it.unibo.utils.arduino.connArduino.sendToArduino("0")
   ]
}
```

Listing 1.15. `blsMVC.qa`: a Led on Arduino

2.4.6 A Led on RaspberryPi.

A Led working on RaspberryPi is defined in a project named `it.unibo.bls17.ledrasp` (read also `lowLevelZooming.pdf`):

```
System ledOnRasp
Event ctrlEvent : ctrlEvent( CATEG, NAME,CMD )  //CMD depends on CATEG/NAME

pubSubServer "tcp://192.168.43.229:1883"
//pubSubServer "tcp://192.168.137.1"    //does not work (perhaps public?)
//pubSubServer "tcp://m2m.eclipse.org:1883"
//pubSubServer "tcp://test.mosquitto.org:1883"

Context ctxLedOnRasp ip [ host="192.168.43.18" port=8079 ]
//Context blsMvcCtx ip  [ host="192.168.43.229" port=8019 ] -standalone

QActor ledrasp context ctxLedOnRasp -pubsub{ //-pubsub required since it must be connected
   Plan init normal [
      println( ledraspmqtt(started) )
   ]
   switchTo doBlinckAtStart

   Plan doBlinckAtStart[
      javaOp "customExecute(\"sudo bash led25GpioTurnOn.sh\")";
      delay 300;
      javaOp "customExecute(\"sudo bash led25GpioTurnOff.sh\")";
      delay 300;
      javaOp "customExecute(\"sudo bash led25GpioTurnOn.sh\")";
      delay 300;
      javaOp "customExecute(\"sudo bash led25GpioTurnOff.sh\")"
   ]
   switchTo waitForCommand

   Plan waitForCommand[ ]
   transition stopAfter 6000000
      whenEvent ctrlEvent -> handleCmd
   finally repeatPlan

   Plan handleCmd resumeLastPlan[
      printCurrentEvent;
      onEvent ctrlEvent : ctrlEvent(leds, led1, on) ->
          javaOp "customExecute(\"sudo bash led25GpioTurnOn.sh\")";
      onEvent ctrlEvent : ctrlEvent(leds, led1, off) ->
          javaOp "customExecute(\"sudo bash led25GpioTurnOff.sh\")"
   ]
}
```

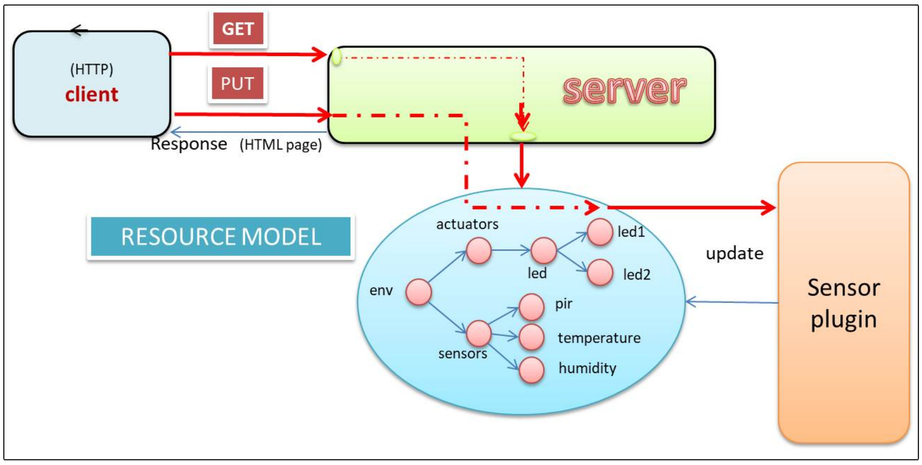
Listing 1.16. `ledOnRasp.qa`: a Led on RaspberryPi

3 A frontend server

In this section we focus our attention on the `frontend` server by using `Node.js` and `Express` as our reference technology. An introduction to these technologies can be found in `nodeExpressWeb.pdf`.

The goal is to build the prototype of a software system whose logic architecture is inspired to the `HTTP/REST` model adopted in the field of the *Web Of Things* (`WoT`) (see `IntroductionQa2017.pdf`, section 9).

The informal logical reference architecture is shown in the following picture:



In the `WoT`, devices and their services are fully integrated in the web because they use the same standards and techniques as traditional websites. We can write applications that interact with embedded devices in exactly the same way as we would interact with any other web service that uses web APIs, in particular, `REST`ful architectures.

The main aspects that qualify such an architecture are:

- `Integration patterns`. Things must be integrated to the Internet and the `WEB` in several ways: using `REST` (*Representational State Transfer*) on device, by means of Applications `Gateways` (via specific `IoT` protocols, like the `UDP`-based `CoAP` (*Constrained Application Protocol*)), or by means of remote servers using the Cloud (via publish-subscribe protocols like `MQTT`).
- `Resource (model) design`. Each Thing should provide some functionality or service that must be modelled and organized into an hierarchy. Usually, physical resources are currently mapped into REST resources by means of description files written in `JSON`.
- `Representation design`. Each resource must be associated with some representation, e.g. `JSON`, `HTML`, `MessagePack`, ect.
- `Interface design`. Each service can be used by means of a set of commands that must be properly designed. In the `REST` model, commands are expressed by means of `HTTP` verbs (`GET, PUT, POST`, etc.) and often associated with *publish-subscribe* interaction via `WebSockets`.
- `Resource linking design`. The different resources must be discovered over the network are often logically linked to each other, for example according to the `HATEHOAS` (*Hyepermedia as Enigine of Application State*) principle, based on the Web-linking mechanisms: the `HTTP` header of a response contains the links to related resources.

The work-plan can be summarized as follows:

1. We start (Subsection 3.1) by setting up a production environment based on `Node.js` and `Express` for the design and development of our frontend server. The environment will be structured (Subsection 3.2) so to highlight an application structure based on a *model*, a *control* and one or more *views*.
2. The next step is to define the code of the entry-point (Subsection 3.4) of our server according the `Express` pattern (Subsection 3.3). The entry-point is an `HTTP` server in which we load the application logic (Subsection 3.5) that defines the proper `routes` for each external request pattern - an `HTTP` verb + `URI` like `http://localhost:3000/pi/sensors/temperature` or :

```
curl -H "Content-Type: application/json" -X PUT -d "{\"value\": \"true\" }" http://localhost:3000/pi/actuators/leds/1
curl http://localhost:3000/pi/actuators/leds/1
```

The entry-point performs also the installation of a set of sensor/actuator `plugins`. Each plugin can act:
 - as a bridge between the logical resource model and some concrete implementation of the resource;
 - as a simulator of a resource;
 - as a generator of (`MQTT`) events towards some external component.

3. Afterwards, we define the application code (Subsection 3.5) according to the `Express` pattern. The most relevant part of this code performs the routing (Subsection 3.5.1) of external requests to specialized parts of code defined in two main files: `routes/sensors.js` and `routes/actuators.js`. Each file maps an `HTTP` verb + `URI` to a request hander.
4. Our final step can follow two different strategies:
   (a) Introduction of a resource model written in `JSON`. Thus is the 'conventional way' of *Web of Things* (`WOT`) applications. In this case we introduce also sensor/actuator `plugins` that simulate the resources, in order to make testing easier. See Subsection 3.10.
   (b) Use the server as a frontend for the application of Subsection 2.4. In this case (see Subsection 3.8) the sensor/actuator `plugins` will publish on the topic `unibo/qasys` `MQTT` messages of the form:

```
msg(ctrlEvent,event,js,none,ctrlEvent(leds, led1, VAL,1)      //for actuators (Led)
msg(inputCtrlEvent,event,js,none,inputEvent(temperature, t1, VAL,1) //for sensors (Temperature)
```

3.1 Starting

Read section `7.8` of `nodeExpressWeb.pdf` and execute the following steps:

```
npm install -g express-generator

Create a new project it.unibo.frontend
Create the folder nodeCode/frontend and open a terminal in this folder

Execute express
Execute npm install
Move node_module under nodeCode (to share it with other projects)
```

Now, execute `node bin/www` and open a browser on `http://localhost:3000/`. In order to understand the work of the server during the rendering phase, read sections `7.5, 7.6, 7.7` of `nodeExpressWeb.pdf`. Here we can recall that:

- `Middleware`. In contrast to vanilla Node, where your requests flow through only one function, Express has a middleware stack, which is effectively an array of functions, called a *middleware stack*. Express middleware is completely compatible with `connect` middleware.
- `Routing`. Routing is a lot like middleware, but the functions are called only when you visit a specific `URL` with a specific `HTTP` method.
- `Extensions` to request and response objects. Express extends the request and response objects with extra methods and properties for developer convenience.
- `Views`. Views allow you to dynamically render `HTML`. This both allows you to change the `HTML` on the fly and to write the `HTML` in other languages.

3.2 Refactoring according to the MVC pattern

Read section `7.9` of `nodeExpressWeb.pdf` and execute the steps `1-3`:

1. Create a new folder called `appServer`.
2. In `appServer` create two new folders, called `models` and `controllers`.
3. Move the `views` and `routes` folders from the `root` of the application into the `appServer` folder.

Now modify the `app.js` to keep into account the modifications:

```
var express    = require('express');
var path       = require('path');
var favicon    = require('serve-favicon');
var logger     = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var index = require('./appServer/routes/index');      //modified as 7.9:
//var users = require('./routes/users');

var app = express();

// view engine setup;
app.set('views', path.join(__dirname, 'appServer', 'views')); //modified as 7.9;
app.set('view engine', 'jade');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', index);
//app.use('/users', users);

// catch 404 and forward to error handler;
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development;
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page;
  res.status(err.status || 500);
  res.render('error');
});

module.exports = app;
```

Listing 1.17. `app.js`

If we open a browser on `http://localhost:3000/` all goes as before. Since this code simply shows a web page, we will upgrade it later (Subsection 3.5).

3.3 The Express use pattern

The file `app.js` defines the application logic of the server and is structured according to the Express pattern introduced in section `7.8` that can be summarized as follows:

```
var express = require("express");
var http   = require("http");

var app    = express();

app.use( ... ) ;

app.get( ... ) ;

http.createServer(app).listen(3000);
```

– The `express()` function starts a new Express application and returns a request handler function.

- `app.use(...)` is intended for *binding* middleware to your application. It means *"Run this on ALL requests"* regardless of `HTTP` verb used (`GET, POST, PUT` ...)
- `app.get(...)` is part of Express' application routing. It means *"Run this on a GET request, for the given URL"*. There is also be `app.post`, which respond to `POST` requests, or `app.put`, or any of the `HTTP` verbs. They work just like middleware; it's a matter of when they're called.

When a request comes in, it will always go through the *middleware* functions, in the same order in which you use them. Express's static middleware (`express.static`) allows us to show files out of a given directory.

3.4 The server entry-point

The generated file `node bin/www` contains the code of a server that simply starts the application code according the scheme of Subsection 3.3. Let us introduce now a new version of the server that works like the previous one, by adding a function that loads a `model` (see Subsection 3.10) and one or more resource `plug-in` (see Subsection 4.2):

```
/*
 * frontend/frontendServer.js
 */
var appl          = require('./applCode'); //previously was app;
var resourceModel = require('./appServer/models/model');
var http          = require('http');

var createServer = function (port ) {
  console.log("process.env.PORT=" + process.env.PORT + " port=" + port);
  if (process.env.PORT) port = process.env.PORT;
  else if (port === undefined) port = resourceModel.customFields.port;

  initPlugins();

  server = http.createServer(appl);
  server.on('listening', onListening);
  server.on('error', onError);
  server.listen( port );
};

function initPlugins() {
//  ledsPlugin = require('./plugins/internal/ledsPlugin'); //global variable;
//  ledsPlugin.start( {'simulate': true, 'frequency': 5000} );
//
//   dhtPlugin = require('./plugins/internal/DHT22SensorPlugin'); //global variable;
//  dhtPlugin.start({'simulate': true, 'frequency': 2000});
}

createServer(3000);
```

Listing 1.18. `frontendServer.js`

The new application logic is embedded in the `applCode.js` file (see Subsection 3.5).
The server defines also functions to handle events and uncaught exceptions:

```
function onListening() {
      var addr = server.address();
      var bind = typeof addr === 'string'
        ? 'pipe ' + addr
        : 'port ' + addr.port;
      console.log('Listening on ' + bind);
}
function onError(error) {
    if (error.syscall !== 'listen') {
        throw error;
    }
     var bind = typeof port === 'string'
           ? 'Pipe ' + port
           : 'Port ' + port;
          // handle specific listen errors with friendly messages;
          switch (error.code) {
            case 'EACCES':
              console.error(bind + ' requires elevated privileges');
```

```
            process.exit(1);
            break;
          case 'EADDRINUSE':
            console.error(bind + ' is already in use');
            process.exit(1);
            break;
          default:
            throw error;
        }
}
//Handle CRTL-C;
process.on('SIGINT', function () {
//  ledsPlugin.stop();
//  dhtPlugin.stop();
  console.log('frontendServer Bye, bye!');
  process.exit();
});
process.on('exit', function(code){
    console.log("Exiting code= " + code );
});
process.on('uncaughtException', function (err) {
    console.error('mqtt got uncaught exception:', err.message);
    process.exit(1);      //MANDATORY!!!;
});
```

Listing 1.19. `frontendServer.js`

3.5 applCode

The new application code continues to be structured according to the Express pattern introduced in section `7.8` of `nodeExpressWeb.pdf`. The first part is quite 'standard':

```
var express       = require('express');
var path          = require('path');
var favicon       = require('serve-favicon');
var logger        = require('morgan');  //see 10.1 of nodeExpressWeb.pdf;
var cookieParser  = require('cookie-parser');
var bodyParser    = require('body-parser');
var fs            = require('fs');
var index         = require('./appServer/routes/index');
var actuatorsRoutes = require('./appServer/routes/actuators');
var sensorsRoutes = require('./appServer/routes/sensors');

var app = express();

// view engine setup;
app.set('views', path.join(__dirname, 'appServer', 'views'));
app.set('view engine', 'jade');

//create a write stream (in append mode) ;
var accessLogStream = fs.createWriteStream(path.join(__dirname, 'morganLog.log'), {flags: 'a'})
app.use(logger("short", {stream: accessLogStream}));

//Creates a default route. Overloads app.use('/', index);
//app.get("/", function(req,res){ res.send("Welcome to frontend Server"); } );

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));          //shows commands, e.g. GET /pi 304 23.123 ms - -;
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

Listing 1.20. `applCode.js`: starting

3.5.1 Routing rules.

The most relevant part of the application code deals with request routing (see section `7.4` of `nodeExpressWeb.pdf`)

```
//DEFINE THE ROUTES ;
app.use('/', index);
app.use('/pi/actuators', actuatorsRoutes);
app.use('/pi/sensors', sensorsRoutes);

//Creates a default route for /pi;
app.get('/pi', function (req, res) {
  //for( i in req.body ){ console.info('req body field %s ', i ); };
  //console.info(' get /pi req URL = %s ', req.url );
  res.send('This is the frontend-Pi!')
});

//REPRESENTATION;
app.use( function(req,res){
    console.info("SEND THE ANSWER ...");
    res.send(req.result); }
);
```

Listing 1.21. `applCode.js`: routing

The last part deals with errors:

```

// catch 404 and forward to error handler;
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handler;
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page;
  res.status(err.status || 500);
  res.render('error');
});

module.exports = app;
```

Listing 1.22. `applCode.js`: error handling

3.6 Routers: sensors

The router action for a sensor simply gets the value of the model.

```
/*
 * appServer/routes/sensors.js
 */
var express    = require('express'),
  router       = express.Router(),
  resourceModel = require('../models/model');

router.route('/').get(function (req, res, next) {
    req.type = "defaultView" ;
    req.result = resourceModel.pi.sensors;
    next();
});

router.route('/pir').get(function (req, res, next) {
  req.result = resourceModel.pi.sensors.pir;
  next();
});

router.route('/temperature').get(function (req, res, next) {
      console.log( "..............." );
      console.log( req.result );
```

```
    req.result = resourceModel.pi.sensors.temperature;
    console.log( req.result );
  console.log( "..............." );
  next();
});

router.route('/temperatureProlog').get(function (req, res, next) {
  var tval = resourceModel.pi.sensors.temperature.value ;
  console.log(tval);
  req.result = "msg( sensor, event, temperatureDev, none, "+ tval+", 0 )";
  next();
});

router.route('/humidity').get(function (req, res, next) {
  req.result = resourceModel.pi.sensors.humidity;
  next();
});

module.exports = router;
```

Listing 1.23. `appServer/routes/sensors.js`

3.7 Routers: actuators

The router action for an actuator must also deal with `PUT/POST` verbs that change a model.

```
/*
 * appServer/routes/actuators.js
 *
 * WE SHOULD CHECK THE LOGIN
 */
var express    = require('express'),
  router       = express.Router(),
  resourceModel = require('../models/model');

router.route('/').get(function (req, res, next) {
      req.result = resourceModel.pi.actuators;
      next();
});

router.route('/leds').get(function (req, res, next) {
  req.result = resourceModel.pi.actuators.leds;
  next();
});

router.route('/leds/:id').get(function (req, res, next) {
//(curl) http://localhost:3000/pi/actuators/leds/1;
  req.result = resourceModel.pi.actuators.leds[req.params.id];
  next();
})
.put(function(req, res, next) {
//curl -H "Content-Type: application/json" -X PUT -d "{\"value\": \"true\" }" http://localhost:3000/pi/actuators/leds/1;
  var selectedLed = resourceModel.pi.actuators.leds[req.params.id];
  selectedLed.value = req.body.value;   //CHANGE THE MODEL;
  console.info('route LED Changed LED %s value to %s', req.params.id, selectedLed.value);
  req.result = "LED " + req.params.id + "= " + selectedLed.value;
  emitLedInfo(selectedLed.value);   //EMIT STATE CHANGE EVENT;
  next();
```

Listing 1.24. `appServer/routes/actuators.js`

The `emitinfo` operation performs the step 4b of Section 3 in order to propagate the information that a led value has been changed.

```

/*
 * Emit the new led value according to the blsMVC model
 */
var mqttUtils   = require('./../../uniboSupports/mqttUtils');

```

```
var emitLedInfo = function( ledValue ){
    var val = "off";
    if( ledValue === "true" ) val = "on";
    var eventstr = "msg(ctrlEvent,event,js,none,ctrlEvent(leds, led1, " +val + "),1)"
        console.log(" actuators LED emits> "+ eventstr);
        mqttUtils.publish( eventstr );
 }

module.exports = router;
```

Listing 1.25. `appServer/routes/actuators.js`

3.7.1 MQTT utils .

```
/*
* =====================================
* uniboSupports/mqttUtils.js
* =====================================
*/
const mqtt  = require ('mqtt');
const topic = "unibo/qasys";
//var client = mqtt.connect('mqtt://iot.eclipse.org');
var client  = mqtt.connect('mqtt://localhost');
//var client  = mqtt.connect('tcp://192.168.43.229:1883');

console.log("mqtt client= " + client );

client.on('connect', function () {
      client.subscribe( topic );
      console.log('client has subscribed successfully ');
});

//The message usually arrives as buffer, so I had to convert it to string data type;
client.on('message', function (topic, message){
  console.log("mqtt RECEIVES:"+ message.toString()); //if toString is not given, the message comes as buffer
});

exports.publish = function( msg ){
    //console.log('mqtt publish ' + client);
    client.publish(topic, msg);
}
```

Listing 1.26. `frontend/uniboSupports/mqttUtils.js`

3.8 Working with the server

At his point we have a server that emits via `MQTT` information about the change of the state of an actuator. More specifically, the state of our Led can be changed by means of a command like:

```
curl -H "Content-Type: application/json" -X PUT -d "{\"value\": \"false\" }"
      http://192.168.43.229:3000/pi/actuators/leds/1
curl http://192.168.43.229:3000/pi/actuators/leds/1 (to read)
```

When the server handles the `PUT` command above, it emits the event:

```
ctrlEvent : ctrlEvent(leds, led1, V) //V = on | off
```

Since the specific Led of Subsection 2.4.6 or the general Led of Subsection 2.4 are able to react to this kind of events, we can control in a remote way a Led (connected to the `RaspberryPi`). Moreover, we can activate the Led via a machine-to-machine interaction. For example, a client written in `Javascript` could be:

```
 /*
* =====================================
* it.unibo.frontend/nodeCode/frontend/clientRest.js
* =====================================
```

```
*/

//see https://www.npmjs.com/package/node-rest-client;
'use strict';
var RestClient = require('node-rest-client').Client;
var client = new RestClient();

//var urlLed1 = 'http://localhost:3000/pi/actuators/leds/1';
//var urlLed1 = 'http://192.168.43.229:3000/pi/actuators/leds/1';
var urlLed1 = 'http://192.168.43.18:3000/pi/actuators/leds/1';

var doGet = function(){
    client.get(urlLed1, function (data, response) {
        // parsed response body as js object;
        console.log(data);
        // raw response;
    //   console.log(response);
    });
}

var doPut = function(newvalue){
//set content-type header and data as json in args parameter
    var args = {
        data: { value: newvalue },
        headers: { "Content-Type": "application/json" }
    };

    client.put(urlLed1, args, function (data, response) {
        // parsed response body as js object;
        console.log("PUT done");
//      console.log(data);
        // raw response;
    //   console.log(response);
});
}

setTimeout(doGet, 100);
setTimeout( function(){ doPut("true"); } , 400);
setTimeout(doGet, 700);
setTimeout(function(){ doPut("false"); } , 1000);
```

Listing 1.27. `frontend/clientRest.js`

A client written in **Java** could be:

```
/*
 * it.unibo.frontend/src/it/unibo/frontend/RestClientHttp.java
 */
package it.unibo.frontend;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import org.apache.http.client.entity.UrlEncodedFormEntity;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.methods.HttpPut;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import java.util.ArrayList;
import java.util.List;
import org.apache.http.message.BasicNameValuePair;
import org.apache.http.util.EntityUtils;
import org.apache.http.HttpEntity;
import org.apache.http.NameValuePair;

public class restClientHttp {
//private static String hostAddr = "localhost";
//private static String hostAddr = "192.168.43.18";
private static String hostAddr = "192.168.137.2";



    public static int sendPut(String data, String url) {
```

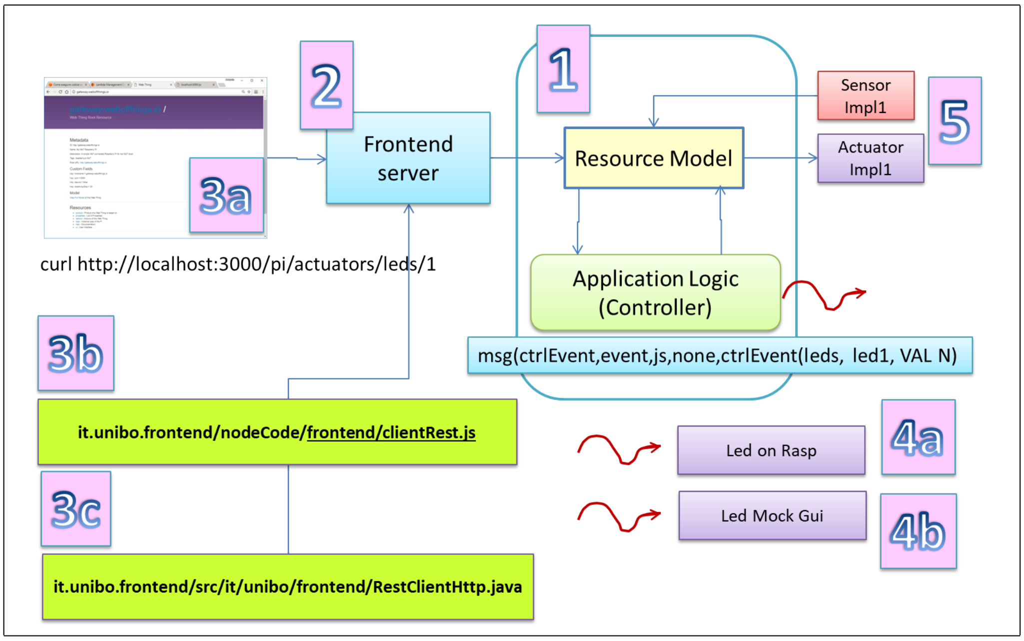
```java
        int responseCode = -1;
        CloseableHttpClient httpclient = HttpClients.createDefault();
        try {
            HttpPut request = new HttpPut(url);
            StringEntity params =new StringEntity(data,"UTF-8");
            params.setContentType("application/json");
            request.addHeader("content-type", "application/json");
            request.addHeader("Accept", "*/*");
            request.addHeader("Accept-Encoding", "gzip,deflate,sdch");
            request.addHeader("Accept-Language", "en-US,en;q=0.8");
            request.setEntity(params);
            CloseableHttpResponse response = httpclient.execute(request);
            responseCode = response.getStatusLine().getStatusCode();
            if (response.getStatusLine().getStatusCode() == 200 || response.getStatusLine().getStatusCode() == 204) {
                BufferedReader br = new BufferedReader(new InputStreamReader((response.getEntity().getContent())));
                String output;
                String info = "";
                while ((output = br.readLine()) != null) {
                    info = info + output;
                }
                System.out.println(info);
            }
            else{ throw new RuntimeException("Failed : HTTP error code : "
                        + response.getStatusLine().getStatusCode());
            }

        }catch (Exception ex) {
        } finally {//         httpclient.close();
        }
        return responseCode;
    }

    public static void connectPost(){
        CloseableHttpClient httpclient = HttpClients.createDefault();
        HttpPost httpPost = new HttpPost("http://"+hostAddr+":3000");
        List <NameValuePair> nvps = new ArrayList <NameValuePair>();
        nvps.add(new BasicNameValuePair("username", "vip"));
        nvps.add(new BasicNameValuePair("password", "secret"));
        try {
            httpPost.setEntity(new UrlEncodedFormEntity(nvps));
            CloseableHttpResponse response2 = httpclient.execute(httpPost);
            HttpEntity entity2 = response2.getEntity();
            // do something useful with the response body and ensure it is fully consumed
            EntityUtils.consume(entity2);
          } catch ( Exception e) {
                e.printStackTrace();
        }
    }
    public static void connectGet(String url){
     try {
        CloseableHttpClient httpclient = HttpClients.createDefault();
        HttpGet httpGet = new HttpGet(url);
        CloseableHttpResponse response = httpclient.execute(httpGet);
        if (response.getStatusLine().getStatusCode() != 200) {
            throw new RuntimeException("Failed : HTTP error code : "
               + response.getStatusLine().getStatusCode());
        }
        BufferedReader br = new BufferedReader(
                        new InputStreamReader((response.getEntity().getContent())));
        String output;
        String info = "";
        while ((output = br.readLine()) != null) {
            info = info + output;
        }
        System.out.println(info);
      } catch ( Exception e) { e.printStackTrace(); }
    }
    public static void work() throws InterruptedException {
        for( int i=1; i<=3; i++) {
            //curl http://192.168.43.229:3000/pi/actuators/leds/1;
            connectGet("http://"+hostAddr+":3000/pi/actuators/leds/1");
            sendPut("{\"value\": \"true\" }", "http://"+hostAddr+":3000/pi/actuators/leds/1");
            Thread.sleep(700);
            sendPut("{\"value\": \"false\" }", "http://"+hostAddr+":3000/pi/actuators/leds/1");
```

```java
        //curl -H "Content-Type: application/json" -X PUT -d "{\"value\": \"false\" }"
            http://localhost:3000/pi/actuators/leds/1;
        connectGet("http://"+hostAddr+":3000/pi/actuators/leds/1");
        Thread.sleep(700);
      }
    }
    public static void main (String args[]) throws Exception{
      System.out.println("==================================================================================");
      System.out.println("1) Activate a MQTT server on hostAddr:1883");
      System.out.println("2) Run node frontendServer.js in it.unibo.frontend/nodeCode/frontend");
      System.out.println("3) Activate
          it.unibo.bls17.ledMockGui.qa/src-gen/it/unibo/ctxLedMockGui/MainCtxLedMockGui.java");
      System.out.println("4) Activate it.unibo.bls17.ledrasp.qa/src-gen/it/unibo/ctxLedOnRasp/MainCtxLedOnRasp.java");
      System.out.println("==================================================================================");
      work();
    }
}
```

Listing 1.28. `restClientHttp.java`

3.9 From components to systems

Here is a picture of the systems we are able to build with the components so far introduced.



1. At the centre of our design there is the `resource model` and the application logic written with reference to that model. In Subsection 2.4 the model is written in `Prolog`, while in Subsection 3.10 the model shall be written in `JSON`.
2. A `frontend` (`REST`) server is introduced to provide actions both for reading and for updating the resource model.
3. Model access can be performed by `humans` via web pages (3a) or via tools like `curl` (3b) or by `machines` by using `REST-HTTP` clients written in `JavaScript` (3c) `Java` (3c) or in some other language.
4. Physical resources can be (dynamically) added by means of mock objects (4b) or by concrete, smart objects (4a) handled by micro-controllers (`Arduino`) or low-cost computers (`RaspberryPi`).
5. In some case, when the server and the model are working near the edge, physical resources can be (dynamically) added by means of plugins loaded and executed at the server site (5). See Subsection 4.2 and Subsection 4.3.

For an example:

```
1) Activate a MQTT server on localhost:1883 (docker run -ti -p 1883:1883 -p 9001:9001 eclipse-mosquitto)
2) Run node frontendServer.js in it.unibo.frontend/nodeCode/frontend
3) Activate it.unibo.bls17.ledMockGui.qa/src-gen/it/unibo/ctxLedMockGui/MainCtxLedMockGui.java
4) Execute it.unibo.frontend/src/it/unibo/frontend/restClientHttp.java
```

The server can be used also as a front-end for the system of Subsection 2.4 according to the strategy `4b` of Section 3. Any change of temperature could lead to a change of the Led. Both these changes can be reflected into a resource-monitoring web page provided by the server by exploiting the `Socket.IO` library. An example can be found in Section 9 of `nodeExpressWeb.pdf` and in `it.unibo.frontend/nodeCode/frontend/applCodeSocket.js`.

3.10 A resource model in JSON

In this section we will follow the strategy `4a` of Section 3 by introducing in `JSON` a simple model of a set of sensor/actuators resources: a `passive infrared` (`PIR`) sensor, a `temperature/humidity` sensor and a `LED`.

```
{
"@context": "http://schema.org/Product",
"@id": "http://localhost:8484",
"name": "My WoT Raspberry PI",
"description": "A simple WoT-connected Raspberry PI for the WoT book.",
"productID" : "asin:B00T2U7R7I",
"manufacturer" : "Raspberry Pi",
"model" : "100437",
"image" : "http://devices.webofthings.io:9090/snapshot.cgi?user=snapshots&pwd=4MXfTSr0gH",
"tags": [
"raspberry",
"pi",
"WoT"
],
  "pi": {
    "name": "WoT Pi",
    "description": "A simple WoT-connected Raspberry PI for the WoT book.",
    "port": 8484,
    "sensors": {
      "temperature": {
        "name": "Temperature Sensor",
        "description": "An ambient temperature sensor.",
        "unit": "celsius",
        "value": 0,
        "gpio": 12
      },
      "humidity": {
        "name": "Humidity Sensor",
        "description": "An ambient humidity sensor.",
        "unit": "%",
        "value": 0,
        "gpio": 12
      },
      "pir": {
        "name": "Passive Infrared",
        "description": "A passive infrared sensor. When 'true' someone is present.",
        "value": true,
        "gpio": 17
      }
    },
    "actuators": {
      "leds": {
        "1": {
          "name": "LED 1",
          "value": false,
          "gpio": 25
        },
        "2": {
          "name": "LED 2",
          "value": false,
          "gpio": 9
        }
      }
```

```
      }
    }
  }
```

Listing 1.29. `appServer/models/resources.json`

The following `model.js` file loads the `JSON` model from the `resources.json` file; the `exports` makes this object available as a node module we can use in our applications.

```
var resources = require('./resources.json');
//var resources = require('./piJsonLd.json');
module.exports = resources;
```

Listing 1.30. `appServer/models/model.js`

4 Node on RaspberryPi

From www.npmjs.com/package/pi-gpio we read: Raspbian has node installed, but it's quite old. To get to a more recent version:

```
===============================================================================================
activate/share an Internet connection

This is important to keep your RasPi system synchronized with security updates, etc.
sudo apt-get update //download the latest package lists from the software repositories
sudo apt full-upgrade -y
These two commands should be issued together and run periodically.

sudo nano /etc/resolv.conf  //insert nameserver 8.8.8.8
ping google.com

uname -m                     //armv7l
Result starts with ?armv6?: you are running a Raspberry Pi based on the older
ARMv6 chipset and the next Node.js installation step will not work

-------------------------------------------------
UPDATE wiringPi
download wiringPi-xxx.tar.gz
tar xfs wiringPi-8d188fa.tar.gz
cd wiringPi-8d188fa/
./build

-------------------------------------------------
UPDATE node
    pi@raspberrypi:~ $  node -v
    v0.10.29
    pi@raspberrypi:~ $  sudo su -
    root@raspberrypi:~ # apt-get remove nodered -y
    root@raspberrypi:~ # apt-get purge nodejs npm
    root@raspberrypi:~ # apt-get remove npm  -y # if you installed npm
    root@raspberrypi:~ # curl -sL https://deb.nodesource.com/setup_9.x | sudo bash -
    root@raspberrypi:~ # apt-get install nodejs -y
    root@raspberrypi:~ # node -v
    v8.9.1
    root@raspberrypi:~ # npm -v
    6.0.1
    exit
------------ As an alternative ------------------
This will download the node package, extract it
and move it into your PATH, making it accessible anywhere.

wget https://nodejs.org/dist/v8.9.1/node-v8.9.1-linux-armv7l.tar.xz
tar -xvf node-v8.9.1-linux-armv7l.tar.xz
cd node-v8.9.1-linux-armv7l
sudo cp -R * /usr/local/
===============================================================================================
```

4.1 Blink a Led with js

From http://webofthings.org/2016/10/23/node-gpio-and-the-raspberry-pi/ we read: You can find a dozen Node.js `GPIO` libraries for the Pi, offering different abstraction layers and functionality. We decided to use one called `onoff`.

```
npm install onoff --save
```

The 'Hello World' equivalent of the IOT is to make a real LED blink:

```javascript
/*
 * =====================================
 * frontend/uniboSupports/ledGpio.js
 * =====================================
 */
var onoff = require('onoff'); //#A;

```

```
var Gpio = onoff.Gpio,
  led = new Gpio(25, 'out'), //#B;
  interval;

interval = setInterval(function () { //#C;
      var value = (led.readSync() + 1) % 2; //#D;
      led.write(value, function() { //#E;
      console.log("Changed LED state to: " + value);
  });
}, 2000);

process.on('SIGINT', function () { //#F;
  clearInterval(interval);
  led.writeSync(0); //#G;
  led.unexport();
  console.log('Bye, bye!');
  process.exit();
});

// #A Import the onoff library
// #B Initialize pin 4 to be an output pin
// #C This interval will be called every 2 seconds
// #D Synchronously read the value of pin 4 and transform 1 to 0 or 0 to 1
// #E Asynchronously write the new value to pin 4
// #F Listen to the event triggered on CTRL+C
// #G Cleanly close the GPIO pin before exiting
```

Listing 1.31. Execution of `ledGpio.js`

4.2 Led plugin

The `ledsPlugin` plugin can be started by the frontend server (see Subsection 3.4, lines 21-27). This plugin can be used during the testing phase to simulate changes of the Led state. Alternatively, if the server is working on a `RaspberryPi`, the plugin can be used to manage a concrete Led whose `anode` is connected to a `GPIO` pin (as defined in the resource model).

In the following version, the `ledsPlugin` works also as an `observer` that could emit the `ctrlEvent/5` event, as done by the actuator of Subsection 3.7.

```
/*
 * frontend/plugins/internal/ledsPlugin.js
 */
var resourceModel = require('./../../appServer/models/model');
var observable  = require('./../../uniboSupports/observableFactory');
var mqttUtils   = require('./../../uniboSupports/mqttUtils');

var actuator, interval;
var ledModel    = resourceModel.pi.actuators.leds['1'];
var pluginName  = ledModel.name;
var localParams = {'simulate': false, 'frequency': 2000};
var counter     = 1;

exports.start = function (params) {
  localParams = params;
  observe(ledModel);  //work as an observer;

  if (localParams.simulate) {
//    simulate();
  } else {
    connectHardware();
  }
};

exports.stop = function () {
  if (localParams.simulate) {
    clearInterval(interval);
  } else {
    actuator.unexport();
  }
  console.info('%s plugin stopped!', pluginName);
```

```
};

function observe(what) {
    console.info('plugin observe: ' + localParams.frequency + " CHANGE MDOEL INTO OBSERVABLE");
    console.info( what );
//Change the ledModel into an observable;
const whatObservable = new observable(what);
observable          = whatObservable.data;
whatObservable.observe('value', () => {
    var val = "off";
    if( observable.value === "true" ) val = "on";
    var eventstr = "msg(ctrlEvent,event,js,none,ctrlEvent(leds, led1, " +val + ")," + counter++ + ")"
        console.log(" ledPlugin LED emits> "+ eventstr);
//      mqttUtils.publish( eventstr );
    });
};

function switchOnOff(value) {
  if (!localParams.simulate) {
    actuator.write(value === true ? 1 : 0, function () {
      console.info('Changed value of %s to %s', pluginName, value);
    });
  }
};

function connectHardware() {
  var Gpio = require('onoff').Gpio;
  actuator = new Gpio(ledModel.gpio, 'out');
  console.info('Hardware %s actuator started!', pluginName);
};

function simulate() {
  interval = setInterval(function () {
    // Switch value on a regular basis;
    if (ledModel.value) {
      ledModel.value = false;
    } else {
      ledModel.value = true;
    }
//  console.log("LED=" + ledModel.value);
  }, localParams.frequency);
  console.info('Simulated %s actuator started!', pluginName);
};
```

Listing 1.32. `frontend/plugins/internal/ledsPlugin.js`

4.3 Temperature/Humidity plugin

The `DHT22sensorPlugin` can be used either to simulate changes of a temperature/humidity sensor or to manage a concrete `DHT22` device connected to a `GPIO` pin (as defined in the resource model).

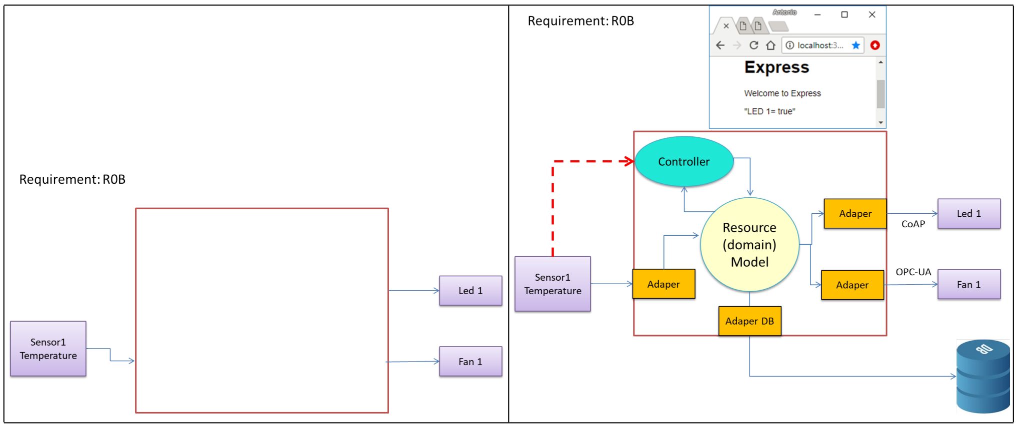
```
/*
 * frontend/plugins/internal/DHT22sensorPlugin.js
 */
var
  resources = require('./../../appServer/models/model'),
  utils     = require('./../../utils.js');
var interval, sensor;
var model      = resources.pi.sensors;
var pluginName = 'Temperature & Humidity';
var localParams = {'simulate': true, 'frequency': 5000};

exports.start = function (params) {
  localParams = params;
  if (params.simulate) {
    simulate();
  } else {
    connectHardware();
  }
};
exports.stop = function () {
```

```
  if (localParams.simulate) {
    clearInterval(interval);
  } else {
    sensor.unexport();
  }
  console.info('%s plugin stopped!', pluginName);
};

function connectHardware() {
 var sensorDriver = require('node-dht-sensor');
  var sensor = {
    initialize: function () {
      return sensorDriver.initialize(22, model.temperature.gpio);
    },
    read: function () {
      var readout = sensorDriver.read();
      model.temperature.value = parseFloat(readout.temperature.toFixed(2));
      model.humidity.value  = parseFloat(readout.humidity.toFixed(2));
      showValue();
      setTimeout(function () {
        sensor.read(); //#D
      }, localParams.frequency);
    }
  };
  if (sensor.initialize()) {
    console.info('Hardware %s sensor started!', pluginName);
    sensor.read();
  } else {   console.warn('Failed to initialize sensor!'); }
};

function simulate() {
  interval = setInterval(function () {
    model.temperature.value = utils.randomInt(0, 40);
    model.humidity.value  = utils.randomInt(0, 100);
    showValue();
  }, localParams.frequency);
  console.info('Simulated %s sensor started!', pluginName);
};

function showValue() {
  console.info('Temperature: %s C, humidity %s \%',
  model.temperature.value, model.humidity.value);
  emitInfo(model.temperature.value);
};

/*
 * Emit the new led value according to the blsMVC model
 */
var mqttUtils   = require('./../../uniboSupports/mqttUtils');

var emitInfo = function( value ){
    var eventstr = "msg(inputCtrlEvent,event,js,none,inputEvent(temperature, t1, " +value + "),1)"
        console.log(" DHT22Plugin emits> "+ eventstr);
        mqttUtils.publish( eventstr );
}
```

Listing 1.33. `frontend/plugins/internal/DHT22sensorPlugin.js`

4.4 An application

Now we can re-design the system related to the requirement ROB of Section 1:



See project `it.unibo.frontend`.

5 A new virtual robot

The site https://github.com/PierfrancescoSoffritti/ConfigurableThreejsApp includes a configurable 3D web-based application built with Three.js. The webpage (available at `http://localhost:8080/` or better at `http://localhost:8088/`) is served from an internal Node.js server working on port 8999.

The scene configuration file contains a JavaScript object saved into a variable:

```
const config = {
    floor: {
        size: { x: 40, y: 40 }
    },
    player: {
        position: { x: 0.5, y: 0.8 },
        speed: 0.2
    },
    sonars: [
        {
            name: "sonar-1",
            position: { x: 0.8, y: 0.8 },
            senseAxis: { x: true, y: false }
        },
        {
            name: "sonar-2",
            position: { x: 0.2, y: 0.2 },
            senseAxis: { x: false, y: true }
        }
    ],
    movingObstacles: [
        {
            name: "moving-obstacle-1",
            position: { x: .5, y: .4 },
            directionAxis: { x: true, y: false },
            speed: 1,
            range: 4
        },
        {
            name: "moving-obstacle-2",
            position: { x: .5, y: .2 },
            directionAxis: { x: true, y: true },
            speed: 2,
            range: 2
        }
    ],
    staticObstacles: [
```

```
        {
            name: "static-obstacle-1",
            centerPosition: { x: 0.2, y: 0.5},
            size: { x: 0.4, y: 0.01}
        },
        {
            name: "static-obstacle-2",
            centerPosition: { x: 0.8, y: 0.5},
            size: { x: 0.4, y: 0.01}
        }
    ]
}
export default config;
```

It's possible to send and receive messages from the server with a TCP connection. An example of usage:

```
/*
 * it.unibo.frontend/nodeCode/frontend/jsCode/clientRobotVirtual.js
 */
const net      = require('net')
const SEPARATOR = ";"
const client   = new Client({ip: "localhost", port: 8999})

function Client({ port, ip }) {
    const self = this
    let clientSocket
    const outQueue = []
    connectTo(port, ip)
    function flushOutQueue() {
        while(outQueue.length !== 0) {
            const data = outQueue.shift()
            self.send(data)
        }
    }
    function connectTo(port, ip) {
            const client = new net.Socket()
            clientSocket = client
            client.connect({ port, ip }, () => console.log(`\t clientRobotVirtual Connecting...`) )
        client.on('connect', () => {
            console.log(`\t clientRobotVirtual Connected`)
            flushOutQueue()
        })
        client.on('data', message => {
            String(message)
                    .split(SEPARATOR)
                    .map( string => string.trim() )
                    .filter( string => string.length !== 0 )
                    .map( JSON.parse )
                    .forEach( message => console.log(message) )
        })
        client.on('close', () => console.log(`\t clientRobotVirtual Connection closed`) )
        client.on('error', () => console.log(`\t clientRobotVirtual Connection error`) )
    }
    this.send = function(message) {
        if(!clientSocket.connecting)
            clientSocket.write(SEPARATOR +message +SEPARATOR)
        else {
            console.log(`\tSocket not created, message added to queue`)
            outQueue.push(message)
        }
    }
    this.finish = function() {
        if(clientSocket.connecting)
            clientSocket.on('connect', clientSocket.end )
        else
            clientSocket.end()
    }
}
//TEST
function test(){
    var msg = "{\"type\": \"moveForward\", \"arg\": 800 }";
    console.log("sending " + msg + " to " + client);
    client.send(msg);
}
```

```
//test();

module.exports=client;
```

Listing 1.34. `it.unibo.frontend/nodeCode/frontend/jsCode/clientRobotVirtual.js`

5.1 From the code to a model

The Node.js server working on port 8999 (from now on `Server8999`) provides a service whose functionalities are described by means of the `ReadMe` notes on the site https://github.com/PierfrancescoSoffritti/ConfigurableThreejsApp. These notes allow us to use the service, but do not describe a formal model of the service itself. To provide a formal model for a façade-service for `Server8999`, let us introduce the following `QActor` specification:

```
1  /*
2   * ====================================================================================
3   * webGuiExecutor.qa
4   * ====================================================================================
5   */
6  System webGuiExecutor
7  Event usercmd     : usercmd(CMD)                //from web gui
8  Event sonar       : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
9  Event sonarDetect : sonarDetect(X)              //From (virtual robot) sonar
10 
11 Dispatch moveRobot : usercmd(CMD)
12 Dispatch startAppl : startAppl(CMD)
13 
14 pubSubServer "tcp://localhost:1883"
15 
16 Context ctxWebGuiExecutor ip [ host="localhost" port=8032 ] -httpserver
17 EventHandler evh for usercmd {  //event-driven ; no Mqtt support yet
18     forwardEvent player -m moveRobot  //from event to message
19 };
20 /*
21  * ----------------------------------------------------
22  * The player is a an interpreter of moveRobot dispatch
23  * ----------------------------------------------------
24  */
25 QActor player context ctxWebGuiExecutor -pubsub {  //The name MUST be player
```

Listing 1.35. `it.unibo.mbot.virtual/src/webGuiExecutor.qa`

The initial part of the `QActor` specification declares that:

- The service handles a dispatch (`moveRobot`) and a set of events (lines 7-9).
- The service maps (`EventHandler` at lines 14-16) an event named `usercmd` into a `moveRobot` dispatch.
- The service is an interpreter of `usercmd` dispatches that works as an actor (`player`).

The main task of the `player` interpreter is to translate a `usercmd` dispatch into a `TCP` call to the `Server8999`:

```
QActor player context ctxWebGuiExecutor -pubsub {  //The name MUST be player
    Plan init normal [
        javaRun it.unibo.utils.clientTcp.initClientConn("localhost","8999");
        delay 1000;
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': 800 }");
        delay 1000;
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveBackward', 'arg': 800 }");
        println("player START")
    ]
    switchTo waitForCmd

    Plan waitForCmd[ ]
    transition stopAfter 3600000  //1h
        whenMsg moveRobot -> execMove
```

```
   finally repeatPlan

   Plan execMove resumeLastPlan[
      printCurrentMessage;
      onMsg moveRobot : usercmd( robotgui(h(X)) ) ->
         javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'alarm' }");
      onMsg moveRobot : usercmd( robotgui(w(X)) ) ->
         javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': -1 }");
      onMsg moveRobot : usercmd( robotgui(s(X)) ) ->
         javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveBackward', 'arg': -1 }");
      onMsg moveRobot : usercmd( robotgui(a(X)) ) ->
         javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'turnLeft', 'arg': 800 }");
         onMsg moveRobot : usercmd( robotgui(d(X)) ) ->
         javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'turnRight', 'arg': 800 }");
      onMsg moveRobot : usercmd( robotgui(x(X)) ) -> forward robotpfrs -m startAppl : startAppl(go)
   ]
}
```

Listing 1.36. `it.unibo.mbot.virtual/src/webGuiExecutor.qa`

Note that:

- a `moveRobot` dispatch can put in execution actions that do not terminate (when `arg': -1`);
- the `moveRobot` dispatch with content `usercmd(robotgui(x(X)))` sends a dispatch to another actor (named `robotpfrs`) that will perform some `application-specific` work.

The server provides also an actor that handles events emitted by the `Server8999`:

```
QActor sonarguidetector context ctxWebGuiExecutor {
Rules{
   checkEnd( D ) :-
      eval( lt(D,5) ),
      output( checkEnddddddddddddddddddddddddddddddddddddddddddddddd( D ) ),
      assert( endreached ).
 }
   Plan init normal [ println("sonarguidetector STARTS ") ]
   switchTo waitForEvents

   Plan waitForEvents[
      [ ?? endreached ] forward robotpfrs -m moveRobot : usercmd( robotgui(h(low)) )
      ]
   transition stopAfter 3600000  //1h
      whenEvent sonar : sonar(sonar2, TARGET, DISTANCE ) do demo checkEnd(DISTANCE), // println( sonar(NAME, TARGET,
          DISTANCE ) ),
      whenEvent sonarDetect : sonarDetect( TARGET ) do println( sonarguidetector( TARGET ) )
   finally repeatPlan
}
```

Listing 1.37. `it.unibo.mbot.virtual/src/webGuiExecutor.qa`

To perform its job, the server makes use of an utility class defined by the application designer to interact with a specific (virtual) robot:

```
public class clientTcp {
private static String hostName = "localhost";
private static int port     = 8999;
private static String sep   = ";";
protected static Socket clientSocket ;
protected static PrintWriter outToServer;
protected static BufferedReader inFromServer;

public static void initClientConn(QActor qa ) throws Exception {
      initClientConn(qa, hostName, ""+port);
}
   public static void initClientConn(QActor qa, String hostNameStr, String portStr) throws Exception {
       hostName = hostNameStr;
       port    = Integer.parseInt(portStr);
       clientSocket = new Socket(hostName, port);
       //outToServer = new DataOutputStream(clientSocket.getOutputStream()); //DOES NOT WORK!!!!;
       inFromServer = new BufferedReader( new InputStreamReader(clientSocket.getInputStream()) );
       outToServer = new PrintWriter(clientSocket.getOutputStream());
```

```java
        startTheReader( qa );
    }
    public static void sendMsg(QActor qa, String jsonString) throws Exception {
        JSONObject jsonObject = new JSONObject(jsonString);
        String msg = sep+jsonObject.toString()+sep;
        outToServer.println(msg);
        outToServer.flush();
    }
```

Listing 1.38. `it.unibo.mbot.virtual/src/it/unibo/utils/clientTcp.java`

The utility class activates also a task that waits for information coming from `Server8999` and generates a `sonar` or a `sonarDetect` event:

```java
    protected static void startTheReader(final QActor qa) {
        new Thread() {
            public void run() {
                while( true ) {
                    try {
                        String inpuStr = inFromServer.readLine();
                        //System.out.println( "reads: " + inpuStr);
                        String jsonMsgStr = inpuStr.split(";")[1];
                        //System.out.println( "reads: " + jsonMsgStr + " qa=" + qa.getName() );
                        JSONObject jsonObject = new JSONObject(jsonMsgStr);
                        //System.out.println( "type: " + jsonObject.getString("type"));
                        switch (jsonObject.getString("type") ) {
                        case "webpage-ready" : System.out.println( "webpage-ready " );break;
                        case "sonar-activated" : {
                            //wSystem.out.println( "sonar-activated " );
                            JSONObject jsonArg = jsonObject.getJSONObject("arg");
                            String sonarName = jsonArg.getString("sonarName");
                            int distance     = jsonArg.getInt( "distance" );
                            //System.out.println( "sonarName=" + sonarName + " distance=" + distance);
                            qa.emit("sonar",
                                "sonar(NAME, player, DISTANCE)".replace("NAME", sonarName.replace("-",
                                    "")).replace("DISTANCE", (""+distance) ));
                            break;
                        }
                        case "collision" : {
                            //System.out.println( "collision" );
                            JSONObject jsonArg = jsonObject.getJSONObject("arg");
                            String objectName = jsonArg.getString("objectName");
                            //System.out.println( "collision objectName=" + objectName );
                            qa.emit("sonarDetect",
                                    "sonarDetect(TARGET)".replace("TARGET", objectName.replace("-", "")));
                            break;
                        }
                        };
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();
    }
```

Listing 1.39. `it.unibo.mbot.virtual/src/it/unibo/utils/clientTcp.java`

Now, if we:

1. activate `Server8999`;
2. run `MainCtxWebGuiExecutor.java`;
3. open a browser on `localhost:8080`;

then, we can use the `QActor` built-in web interface to move the virtual robot on `Server8999`.

5.2 Towards robot-based applications

The component `webGuiExecutor` can be used as a building-block for more complex applications. Our general problem can be introduced as follows:

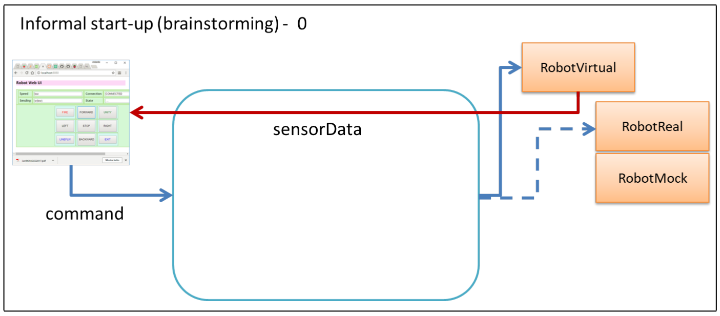
RobotSys: Thinking of a robot as a '*smart thing*', build a software system that allows us to send commands to the robot and to receive sensor data from the robot (or from the robot environment).

Before starting any coding activity, our software-development team observes that :

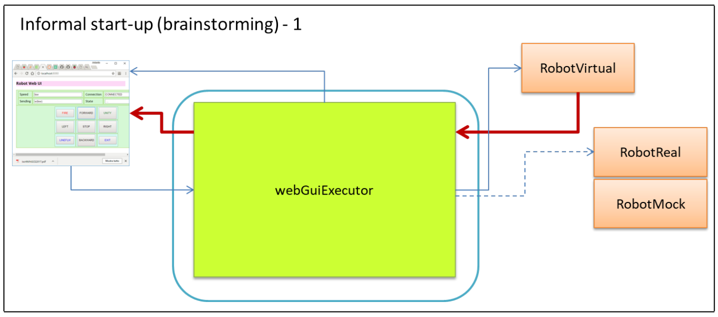
– A robot logically works as an *interpreter* of commands and as an *emitter* of (sensor) information.
– A robot is a physical or a logical 'thing' that runs software provided by the customer. It can be viewed as an example of on-premises software. Usually the robot can be modelled as a POJO that provides an application programming interface (API) as a set of procedures. An example of this kind of robot is the real robot.
– A robot in our system should be modelled as an entity that provides remote APIs that allow developers to manipulate the robot through protocols, regardless of their implementation language. An example of this kind of robot is the virtual robot of Section 5.
– The system should include a support allowing bot human and machine interaction with the robot.

These considerations promote the introduction of an initial informal picture of the logical architecture of the system:

5.2.1 Step0: the system to build as a black-box .



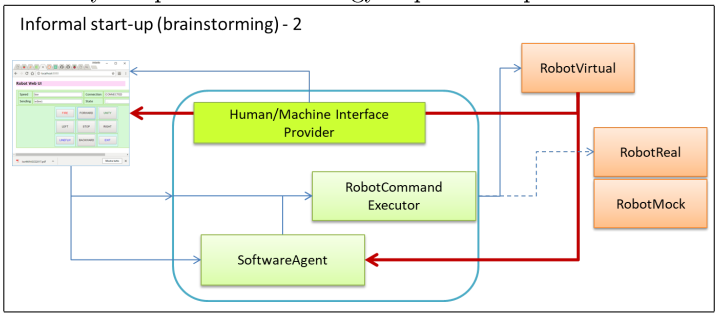
5.2.2 Step1: the system as an actor .



This informal architecture can be formalized by introducing a `QActor` model in which the Robot is modelled as an actor working on its own computational node. The webGuiExecutor.qa of Subsection 5.1 allows the team to define a working prototype in a very short time, in order to fix the requirements with the customer and the end-users.

5.2.3 Step2: the system with agent / adapter components .

The system solution provides a 'technology-independent' part devoted to the application logic that should be easily adapted to 'technology-dependent' parts.



In this case, the `webGuiExecutor.qa` of Subsection 5.1 can be used as an `adapter` (the `RobotCommandExecutor`) from the 'high level' application agent to the virtual robot. Moreover, the `Human/MachineInterface Provider` is introduced to inject into the system a more advanced component than that provided by the `QActor` infrastructure. An example of such a component is given in Section 6.

5.2.4 Toward industrial standards .

In modern *Industrial Internet of Things* (`IIOT`) a robot can be viewed as an example of a `smart machine`, i.e. an intelligent device that uses machine-to-machine technology and other cognitive computing systems that are able to make decisions and solve problems without human intervention.

In other words, our robot could be viewed as a machine that provides a set of `services` (a `Device As A Service` or `DaaS`machine), built according to standard architectures/platforms. e.g. the `OPC-UA` architecture, that incorporates `SOA` features.

The architecture of Subsection 5.2.3 can lead a software designer to the usage, for the input from the robot, of `MTConnect`, a manufacturing technical standard to retrieve process information from numerically controlled machine tools. `MTConnect` is referred to as a *read-only standard*, meaning that it only defines the extraction (reading) of data from control devices, not the writing of data to a control device. Data from shop floor devices is presented in `XML` format, and is retrieved from information providers, called *Agents*, using `HTTP` as the underlying transport protocol. `MTConnect` provides a `RESTful` interface, which means the interface is stateless. No session must be established to retrieve data from an `MTConnect` Agent, and no `logon` or `logoff` sequence is required (unless overlying security protocols are added which do). *Lightweight Directory Access Protocol* (`LDAP`) is recommended for discovery services.

6 A frontend server for the robots

Let us introduce now the following requirement:

RobotRest: Thinking of a robot as a '*smart thing*' (in the domain of IOT), provide a RESTful interface to send commands to the robot.

Our workflow could be stated as follows:

1. Define a model for the robot (see Subsection 6.1).
2. Define a command client for the virtual robot (frontend/jsCode/clientRobotVirtual.js) and test it (see Subsection 6.2.5).
3. Define in Node/Express a front-end server for virtual robot control:
   - frontend/robotFrontendServer.js, see Subsection 6.2.5;
   - frontend/applCodeRobot.js, see Subsection 6.2;
   - frontend/appServer/viewRobot/access.ejs, see Subsection 6.2.1.
4. Add a user-authentication part, see Subsection ??.

The template for the frontend human-interaction page is written in EJS[6] that combines data and a template to produce HTML.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>ResourceAccess</title>
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstrap.min.css">
<script type="text/javascript" src="/socket.io/socket.io.js"></script>
<script>
       var socket = io.connect();
       console.log("CONNECTED VIA SOCKET " );
       socket.on('message', function(v){ document.getElementById('display').innerHTML=v; });
</script>
<script type="text/javascript" src="QActorWebUI.js"></script>
</head>

<body>
<div class="container">
<h1>Robot Control</h1>

Welcome, <b><%= currentUser.displayName || "" %></b>

<br/><br/>
<center>
           <table style="background-color: #d9f7f3">
                  <tr>
                      <td align="center">
                          <form action="/robot/actions/commands/appl" method="post">
                              <input type="submit" value="appl">
                          </form>
                      </td>
                      <td align="center">
                          <form action="/robot/actions/commands/w" method="post">
                              <input type="submit" value="forward" class="btn btn-primary btn-block">
                          </form>
                      </td>
                      <td align="center"> </td>
                  </tr>
                  <tr>
                      <td align="center">
                          <form action="/robot/actions/commands/a" method="post">
                              <input type="submit" value="Left" class="btn btn-primary btn-block">
                          </form>
                      </td>
                      <td align="center">
                          <form action="/robot/actions/commands/h" method="post">
```

[6] EJS is now replaced by donejs).

```
                                <input type="submit" value="Stop" class="btn btn-primary btn-block">
                            </form>
                        </td>
                        <td align="center">
                            <form action="/robot/actions/commands/d" method="post">
                                <input type="submit" value="Right" class="btn btn-primary btn-block">
                            </form>
                        </td>
                    </tr>
                    <tr>
                        <td align="center"> </td>
                        <td align="center">
                            <form action="/robot/actions/commands/s" method="post">
                                <input type="submit" value="Backward" class="btn btn-primary btn-block">
                            </form>
                        </td>
                        <td align="center"> </td>
                    </tr>
                </table>
</center>
</div>
<div>
    <span id="display">Display area</span>
</div>
</body>
</html>
```

Listing 1.40. `appServer/viewRobot/access.ejs`

6.1 A model for the robot

```
{
  "robot": {
        "name": "WoT Robot",
        "description": "A simple robot model",
        "port": 8484,
        "state": "stopped",
        "sonar1": {
            "name" : "sonar1",
            "value": 0
        },
        "sonar2": {
            "name" : "sonar1",
            "value": 0
        },
        "sonarRobot": {
            "name" : "sonarRobot",
            "value" : 0
        }
    }
}
```

Listing 1.41. `it.unibo.frontend/nodeCode/frontend/appServer/models/robot.json`

6.2 A robotFrontendServer

The `frontend` server for the virtual robot is an http server that supports publish-subscribe interaction by using `Socket.IO` (see section 9.2 of `nodeExpressWeb.pdf`.

```
/*
 * frontend/robotFrontendServer.js
 */
var appl          = require('./applCodeRobot'); //previously was applCode;
var resourceModel = require('./appServer/models/robot');
var http          = require('http');
var io            ;  //Upgrade ro socketIo;
```

```

var createServer = function ( port ) {
    console.log("createServer " + port);
  initPlugins();
  server = http.createServer( appl );
  io     = require('socket.io').listen(server); //Upgrade fro socketio;
  server.on('listening', onListening);
  server.on('error', onError);
  setInterval( showResourceState, 1000 ); //show the robot state;
  server.listen( port );
};

function showResourceState(){
    var now = new Date() ;
    var info = "ROBOT state="+resourceModel.robot.state+"\n"+
//      resourceModel.robot.sonar1.name+"="+resourceModel.robot.sonar1.value+"\n"+
//      resourceModel.robot.sonar2.name+"="+resourceModel.robot.sonar2.value+"\n"+
//      resourceModel.robot.sonarRobot.name+"="+resourceModel.robot.sonarRobot.value+"\n"+ //;
        "time=" + now.getHours() + ":" + now.getMinutes() + ":" + now.getSeconds();
    io.sockets.send( info );
}

function initPlugins() {}
```

Listing 1.42. `frontend/robotFrontendServer.js`

The publish-subscribe interaction is used to show the current state of the robot (and its environment) at a given rate (e.g. `1/sec`).

6.2.1 The application code .

The application code:

- is based on the Express framework and makes use of the EJS templating language (instead of Pug) to generate HTML pages with plain JavaScript.

```
/*
 * it.unibo.frontend/nodeCode/frontend/applCodeRobot.js
 */
var express       = require('express');
var path          = require('path');
var favicon       = require('serve-favicon');
var logger        = require('morgan');  //see 10.1 of nodeExpressWeb.pdf;
var cookieParser  = require('cookie-parser');
var bodyParser    = require('body-parser');
var fs            = require('fs');
var toRobot       = require("./jsCode/clientRobotVirtual");
var serverWithSocket= require('./robotFrontendServer');
var cors          = require('cors');
var robotModel    = require('./appServer/models/robot');
var User          = require("./appServer/models/user");
var mqttUtils     ;   //to be set later;
var session       ;   //to be set later for AUTH;
var passport      ;   //to be set later for AUTH;
var setUpPassport ;   //to be set later for AUTH;
var mongoose      ;   //to be set later for AUTH;
var flash         ;   //to be set later for AUTH;

var app            = express();

// view engine setup;
app.set('views', path.join(__dirname, 'appServer', 'viewRobot'));
app.set("view engine", "ejs");

//create a write stream (in append mode) ;
var accessLogStream = fs.createWriteStream(path.join(__dirname, 'morganLog.log'), {flags: 'a'})
app.use(logger("short", {stream: accessLogStream}));

app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));          //shows commands, e.g. GET /pi 304 23.123 ms - -;
app.use(bodyParser.json());
```

```
app.use( cors() ); //npm install cors --save ;
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());

//app.use(express.static(path.join(__dirname, 'jsCode')))
```

Listing 1.43. `frontend/applCodeRobot.js: starting`

6.2.2 Feature configuration .

The server can be further 'configured' by means of two internal boolean variables:

- `externalActuator`: when `true`, it means that the server will delegate requested actions to some external application, by publising a `MQTT` message;
- `withAuth`: when `true`, it means that the server permits access to a set of registered users. In this case, the server exploits `session` and `flash`: see Section 7.

```
var externalActuator = true; //when true, the application logic is external to the server;
var withAuth         = false   ;

if( externalActuator ) mqttUtils = require('./uniboSupports/mqttUtils');
if( withAuth ){
    session         = require("express-session");
    passport        = require("passport");
    setUpPassport   = require("./setuppassport");
    mongoose        = require("mongoose");
    flash           = require("connect-flash");

    setUpAuth();
}
```

Listing 1.44. `frontend/applCodeRobot.js: feature configuration`

The `setUpAuth` operation will be described in Subsection 7.1.

6.2.3 The user command page .

The `ejs` pages are defined in the directory `appServer/viewRobot`:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>ResourceAccess</title>
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstrap.min.css">
<script type="text/javascript" src="/socket.io/socket.io.js"></script>
<script>
      var socket = io.connect();
      console.log("CONNECTED VIA SOCKET " );
      socket.on('message', function(v){ document.getElementById('display').innerHTML=v; });
</script>
<script type="text/javascript" src="QActorWebUI.js"></script>
</head>

<body>
<div class="container">
<h1>Robot Control</h1>

Welcome, <b><%= currentUser.displayName || "" %></b>

<br/><br/>
<center>
          <table style="background-color: #d9f7f3">
                 <tr>
                     <td align="center">
                         <form action="/robot/actions/commands/appl" method="post">
                             <input type="submit" value="appl">
```

```
                        </form>
                    </td>
                    <td align="center">
                        <form action="/robot/actions/commands/w" method="post">
                            <input type="submit" value="forward" class="btn btn-primary btn-block">
                        </form>
                    </td>
                    <td align="center"> </td>
                </tr>
                <tr>
                    <td align="center">
                        <form action="/robot/actions/commands/a" method="post">
                            <input type="submit" value="Left" class="btn btn-primary btn-block">
                        </form>
                    </td>
                    <td align="center">
                        <form action="/robot/actions/commands/h" method="post">
                            <input type="submit" value="Stop" class="btn btn-primary btn-block">
                        </form>
                    </td>
                    <td align="center">
                        <form action="/robot/actions/commands/d" method="post">
                            <input type="submit" value="Right" class="btn btn-primary btn-block">
                        </form>
                    </td>
                </tr>
                <tr>
                    <td align="center"> </td>
                    <td align="center">
                        <form action="/robot/actions/commands/s" method="post">
                            <input type="submit" value="Backward" class="btn btn-primary btn-block">
                        </form>
                    </td>
                    <td align="center"> </td>
                </tr>
            </table>
</center>
</div>
<div>
    <span id="display">Display area</span>
</div>
</body>
</html>
```

Listing 1.45. `appServer/viewRobot/access.ejs`

- uses the *cross-origin resource sharing* (`cors`) that allows requests to skip the "same origin policy" (to ensure that a site can't load any scripts from another domain) and access resources form remote hosts;
- makes us of a specific logo (`favicon.ico`);
- exploits express static middleware to make available application-specific Javascript code in the directory `isCode`,

.

6.2.4 The interpretation of commands .

Commands are '*action resources*' represented as

```
/robot/actions/commands/CMD, with
    CMD = appl | w | s | a | d | h
```

Actions could be explicitly represented in the model introduced in Subsection 6.1 in order to describe the set of commands supported by the robot (our web Thing). In this case, *actions* will represent the `public interface` of a web Thing, while other *properties* are the `private parts`.

The server accepts commands via the `POST` verb. The response should be:

– `204 NON CONTENT` if the action is executed immediately;
– `202 ACCEPTED` if the action will be executed at a later time.

```
    });
  });

/*
 * ====================== COMMANDS ================
 */
  app.post("/robot/actions/commands/appl", function(req, res) {
      console.info("START THE APPLICATION " );
      if( externalActuator ) delegate( "x(low)", "application", req, res);
  });

  app.post("/robot/actions/commands/w", function(req, res) {
      if( externalActuator ) delegate( "w(low)", "moving forward", req, res);
      else actuate( `{ "type": "moveForward", "arg": -1 }`, "server moving forward", req, res);

  });
  app.post("/robot/actions/commands/s", function(req, res) {
      if( externalActuator ) delegate( "s(low)", "moving backward", req, res );
      else actuate( `{ "type": "moveBackward", "arg": -1 }`, "server moving backward", req, res);
  });
  app.post("/robot/actions/commands/a", function(req, res) {
      if( externalActuator ) delegate( "a(low)", "moving left", req, res );
      else actuate( `{ "type": "turnLeft", "arg": 1000 }`, "server moving left", req, res);
  });
  app.post("/robot/actions/commands/d", function(req, res) {
      if( externalActuator ) delegate( "d(low)", "moving right", req, res );
      else actuate( `{ "type": "turnRight", "arg": 1000 }`, "server moving right", req, res);
  });
  app.post("/robot/actions/commands/h", function(req, res) {
```

Listing 1.46. `frontend/applCodeRobot.js: commands`

The server executes directly and immediately the action if `externalActuator` is false. Otherwise, it delegates the execution of the action to some external entity.

6.2.5 Internal execution of commands .

The immediate execution of an action is a 'technology-dependent' action that exploits the `clientRobotVirtual.js` to send the command to the virtual robot.

```
function actuate(cmd, newState, req, res ){
    toRobot.send( cmd );
    robotModel.robot.state = newState;
    if( ! passport ) res.locals.currentUser={name:"Guest", displayName:"UserAnonymous"};
    res.render("access");
}
```

Listing 1.47. `frontend/applCodeRobot.js: actuate`

The `clientRobotVirtual.js` is also able to receive data from the robot:

```
/*
 * it.unibo.frontend/nodeCode/frontend/jsCode/clientRobotVirtual.js
 */
const net     = require('net')
const SEPARATOR = ";"
const client  = new Client({ip: "localhost", port: 8999})

function Client({ port, ip }) {
    const self = this
    let clientSocket
    const outQueue = []
    connectTo(port, ip)
    function flushOutQueue() {
        while(outQueue.length !== 0) {
```

```
            const data = outQueue.shift()
            self.send(data)
        }
    }
    function connectTo(port, ip) {
            const client = new net.Socket()
            clientSocket = client
            client.connect({ port, ip }, () => console.log(`\t clientRobotVirtual Connecting...`) )
        client.on('connect', () => {
            console.log(`\t clientRobotVirtual Connected`)
            flushOutQueue()
        })
        client.on('data', message => {
            String(message)
                    .split(SEPARATOR)
                    .map( string => string.trim() )
                    .filter( string => string.length !== 0 )
                    .map( JSON.parse )
                    .forEach( message => console.log(message) )
        })
        client.on('close', () => console.log(`\t clientRobotVirtual Connection closed`) )
        client.on('error', () => console.log(`\t clientRobotVirtual Connection error`) )
    }
    this.send = function(message) {
        if(!clientSocket.connecting)
            clientSocket.write(SEPARATOR +message +SEPARATOR)
        else {
            console.log(`\tSocket not created, message added to queue`)
            outQueue.push(message)
        }
    }
    this.finish = function() {
        if(clientSocket.connecting)
            clientSocket.on('connect', clientSocket.end )
        else
            clientSocket.end()
    }
}
//TEST
function test(){
    var msg = "{\"type\": \"moveForward\", \"arg\": 800 }";
    console.log("sending " + msg + " to " + client);
    client.send(msg);
}
//test();

module.exports=client;
```

Listing 1.48. `ffrontend/jsCode/clientRobotVirtual.js`

6.2.6 External execution of commands .

```
function delegate( hlcmd, newState, req, res ){
    robotModel.robot.state = newState;
    emitRobotCmd(hlcmd);
    if( ! passport ) res.locals.currentUser={name:"Guest", displayName:"UserAnonymous"};
    res.render("access");
}
```

Listing 1.49. `frontend/applCodeRobot.js: delegate`

6.2.7 emitRobotCmd .

```
var emitRobotCmd = function( cmd ){ //called by delegate;
    var eventstr = "msg(usercmd,event,js,none,usercmd(robotgui( " +cmd + ")),1)"
```

```
        console.log("emits> "+ eventstr);
        mqttUtils.publish( eventstr ); //topic = "unibo/qasys";
}
```

Listing 1.50. `frontend/applCodeRobot.js: emitRobotCmd`

7 User authentication

HTTP is stateless; in order to associate a request to any other request, we need a way to store user data between HTTP requests. Cookies and URL parameters are both suitable ways to transport data between the client and the server. But they are both readable and on the client side.

The express-session module allows us to assign the client an ID and it makes all further requests using that ID. Information associated with the client is stored on the server linked to this ID. Whenever we make a request from the same client again, we will have their session information stored with us (given that the server was not restarted).

flash is a special area of the session used for storing messages. Messages are written to the flash and cleared after being displayed to the user. The flash is typically used in combination with redirects, ensuring that the message is available to the next page that is to be rendered.

Passport.js is a simple, unobtrusive authentication middleware for Node.js which we can use for session management. Passport doesn't dictate how you authenticate your users; it's there only to provide helpful boilerplate code. In this section, we'll use Passport to authenticate users stored in a Mongo database, but Passport also supports authentication with providers like Facebook, Google, Twitter, and over 100 more. The control flow is:

1. User enters username and password. The credentials used to authenticate a user will only be transmitted during the `login` request.
2. The application checks if they are matching.
3. If they are matching, a session will be established and it sends a `Set-Cookie` header that will be used to authenticate further pages. Each subsequent request will not contain credentials, but rather the unique cookie that identifies the session.
4. When the user visits pages from the same domain, the previously set cookie will be added to all the requests. In order to support login sessions, Passport will serialize and deserialize user instances to and from the session. Moreover, Passport will populate `req.user`.

7.1 Setting up the authorization support

Let us show here the `setUpAuth` operation introduced in Subsection 6.2.2 (we use the variables defined in Subsection 6.2.1).

```
function setUpAuth(){
   try{
      console.log("\tWORKING WITH AUTH ... " ) ;
      mongoose.connect("mongodb://localhost:27017/test");
      setUpPassport();
      app.use(session({
           secret: "LUp$Dg?,I#i&owP3=9su+OB%‘JgL4muLF5YJ~{;t",
           resave: true,
           saveUninitialized: true
      }));
      app.use(flash());
      app.use(passport.initialize());
      app.use(passport.session());
      app.use(function(req, res, next) {
           res.locals.currentUser = req.user;
           res.locals.errors    = req.flash("error");
           res.locals.infos     = req.flash("info");
           next();
         });
   }catch( e ){
      console.log("SORRY, ERROR ... " + e) ;
   }
}
```

Listing 1.51. `frontend/applCodeRobot.js: setUpAuth`

There are three options we pass to express-session:

- `secret`: allows each session to be encrypted from the clients. This deters hackers from hacking into users' cookies.
- `resave`: option required by the middleware. When true, the session will be updated even when it hasn't been modified.
- `saveUninitialized`: another required option. This resets sessions that are uninitialized.

Passport will populate `req.user` and `connect-flash` will populate some flash values. Moreover (line 14-19), every view will now have access to `currentUser` (see the form in `access.ejs` from Subsection 6.2.1), which pulls from `req.user`, which is populated by Passport.

7.1.1 Setting up the passport .

There are two main parts here:

- Tell Passport how to serialize and deserialize users. This is a short amount of code that effectively translates a user's session into an actual user object.
- Tell Passport how to authenticate user by setting up a strategy. Some strategies include authentication with sites like `Facebook` or `Google`; the strategy we use here is a `local strategy`; that means we have to write a little `Mongoose` code (line *14*, see also Subsection 7.3).

```
1  var passport     = require("passport");
2  var LocalStrategy = require("passport-local").Strategy;
3  var User         = require("./appServer/models/user");
4  module.exports   = function() {
5    passport.serializeUser(function(user, done) {
6      done(null, user._id);
7    });
8    passport.deserializeUser(function(id, done) {
9      User.findById(id, function(err, user) {
10       done(err, user);
11     });
12   });
13   passport.use("login", new LocalStrategy(function(username, password, done) {
14     User.findOne({ username: username }, function(err, user) {
15       if (err) { return done(err); }
16       if (!user) {
17         return done(null, false, { message: "Sorry, user not allowed!" });
18       }
19       user.checkPassword(password, function(err, isMatch) {
20         if (err) { return done(err); }
21         if (isMatch) {
22           return done(null, user);
23         } else {
24           return done(null, false, { message: "Invalid password." });
25         }
26       });
27     });
28   }));
29 };
```

Listing 1.52. `frontend/setuppassport.js`

At the end, our local strategy calls the `done` callback that returns the user object if it's found and `false` otherwise.

7.2 Handling authorization

If the variable `setAuth` of Subsection 6.2.2 is set to `true`, any access to the main page will be redirected (via `POST`) to the `login` page. `passport.authenticate` returns a *request handler function* that you pass instead one you write yourself.

```
app.get('/', function(req, res) {
    if( withAuth ) res.render("login");
    else{
        res.locals.currentUser={name:"Guest", displayName:"UserAnonymous"};
        res.render("access");
    }
});

app.get("/login", function(req, res) {
     res.render("login");
});
if( passport )
app.post("/login", passport.authenticate("login", {
      successRedirect: "/access",
      failureRedirect: "/login",
      failureFlash: true
}));
app.get("/access", ensureAuthenticated, function(req, res, next) {
    if( ! passport ) res.locals.currentUser={name:"Guest", displayName:"UserAnonymous"};
    res.render("access");
});
app.get("/logout", function(req, res) {
  req.logout(); //a new function added by Passport;
  res.redirect("/");
});
```

Listing 1.53. `frontend/applCodeRobot.js: handle auth`

7.2.1 Check authorization .

```
function ensureAuthenticated(req, res, next) {
      if( ! passport ) next(); //always authenticated
      if (req.isAuthenticated()) {
           next();
      } else {
           req.flash("info", "You must be logged in to see this page.");
           res.redirect("/login");
      }
}//
```

Listing 1.54. `frontend/applCodeRobot.js: ensureAuthenticated`

7.3 A model for the user

The user is an abstraction that can be modelled as a 'traditional' object associated with a set of properties:

- `Username`. A unique name. This will be required.
- `Password`. This will also be required.
- `Time joined`. A record of when the user joined the site.
- `Display name`. A name that is displayed instead of the username. This will be optional.
- `Biography`. An optional bunch of text that is displayed on the user's profile page.

The set of properties related to our user objects should be stored in permanent way on a database, e.g. the `MongoDB`. Let us recall some points from `NodeExpressWeb.pdf`), section 4:

1. `MongoDB` is a document (non relational, non-transactionsl) database that stores documents as `BSON` (Binary `JSON`). Let us recall here some common commands:

```
docker pull mongo %%load a mongo image
docker run --name database -d -p 27017:27017 mongo --noauth --bind_ip=0.0.0.0 %%start a container
docker exec -it c07b136da935 /bin/bash    %%Run a command in a running container
------------------------------------------------------------------------------------------------------------
mongo
show dbs %%should include test
use <one of the dbs> (set db) %% e.g. use test
show collections %%should diplay users
db.<acollection>.find() %% e.g. db.users.find()
------------------------------------------------------------------------------------------------------------
%% insert a new document
db.<acollection>.insertOne( {...} )
%% delete a (matching)  document
db.<acollection>.deleteOne( {...} )

%%% EXAMPLES
db.users.insertOne({ username: "admin", password: "123", createdAt: ISODate("2018-05-27T07:34:33.007Z"),
"bio" : "human", "displayName" : "zorro"  })

db.users.deleteOne( { username: "admin" } )
```

2. while `MongoDB` is useful to store and handle documents, `data modelling` is supported by `Mongoose` that provides schema-based solutions by including built-in type casting, validation, query building, business logic hooks and more. With `Mongoose`, all document creation and retrieval from the database is handled by models.

The mongoose `Schema` is what is used to define attributes for our users as `MongoDB` documents (see section `4.3` of `NodeExpressWeb.pdf`).

```
var bcrypt = require("bcrypt-nodejs");
var mongoose = require("mongoose");

var userSchema = mongoose.Schema({
  username: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  createdAt: { type: Date, default: Date.now },
  displayName: String,
  bio: String
});
```

Listing 1.55. `frontend/appServer/models/user.js`: the schema

Besides the schema with the properties, we can define also methods. For example, in order to never store the real password, we'll apply a one-way hash to it using the `bcrypt` algorithm by defining a `pre-save` action. We also need an operation to compare the real password to a password guess.

```
var SALT_FACTOR = 10;
var noop = function() {};

userSchema.pre("save", function(done) {
  var user = this;

  if (!user.isModified("password")) {
    return done();
  }

  bcrypt.genSalt(SALT_FACTOR, function(err, salt) {
    if (err) { return done(err); }
    bcrypt.hash(user.password, salt, noop, function(err, hashedPassword) {
      if (err) { return done(err); }
      user.password = hashedPassword;
      done();
    });
  });
});

userSchema.methods.checkPassword = function(guess, done) {
  bcrypt.compare(guess, this.password, function(err, isMatch) {
    done(err, isMatch);
  });
};

```

```
userSchema.methods.name = function() {
  return this.displayName || this.username;
};

var User = mongoose.model("User", userSchema);

module.exports = User;
```

Listing 1.56. `frontend/appServer/models/user.js`: operations

Note that, the `checkPassword` operation does use `bcrypt.compare` instead of a simple equality check (e.g. `===`). This is for security reasons; it helps keep us safe from a complicated hacker trick called a timing attack.

7.4 Handling (new) users

Now add more routes to our page: one for the `sign-up` page and one to do the actual `signing up`. The `body-parser` middleware is required here to parse form data.

```
    app.get("/signup", function(req, res) {
      res.render("signup");
    });
if( passport )
    app.post("/signup", function(req, res, next) {
      var username = req.body.username;
      var password = req.body.password;
      User.findOne({ username: username }, function(err, user) {
        if (err) { return next(err); }
        if (user) {
          req.flash("error", "User already exists");
          return res.redirect("/signup");
        }
        var newUser = new User({
          username: username,
          password: password
        });
        newUser.save(next);
      });
    }, passport.authenticate("login", {
      successRedirect: "/",
      failureRedirect: "/signup",
      failureFlash: true
    }));
```

Listing 1.57. `frontend/applCodeRobot.js`: signup

The page that allows an user to register can be defined as follows:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>ResourceAccess</title>
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstrap.min.css">
</head>
<body>
<div class="navbar navbar-default navbar-static-top" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <a class="navbar-brand" href="/">UniboRobot access</a>
    </div>
  </div>
</div>
<h1>Sign up</h1>

<form action="/signup" method="post">
  <input name="username" type="text" class="form-control" placeholder="Username" required autofocus>
  <input name="password" type="password" class="form-control" placeholder="Password" required>
  <input type="submit" value="Sign up" class="btn btn-primary btn-block">
</form>

```

```
</body>
</html>
```

Listing 1.58. `frontend/appServer/viewRobot/signup.ejs`

7.5 View/Edit user's profile

Another feature could be the viewing/editing of user profiles:

```
    app.get("/users/:username", function(req, res, next) {
      User.findOne({ username: req.params.username }, function(err, user) {
        if (err) { return next(err); }
        if (!user) { return next(404); }
        res.render("profile", { user: user });
      });
    });
    app.get("/edit", ensureAuthenticated, function(req, res) {
      res.render("edit");
    });

    app.post("/edit", ensureAuthenticated, function(req, res, next) {
      req.user.displayName = req.body.displayname;
      req.user.bio = req.body.bio;
      req.user.save(function(err) {
        if (err) {
          next(err);
          return;
        }
        req.flash("info", "Profile updated!");
        res.redirect("/edit");
      });
    });
```

Listing 1.59. `frontend/applCodeRobot.js`: user profile

The page that shows the user profile can be defined as follows:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>ResourceAccess</title>
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstrap.min.css">
</head>
<body>
<div class="navbar navbar-default navbar-static-top" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <a class="navbar-brand" href="/">UniboRobot access</a>
    </div>
  </div>
</div>

<h1>Profile</h1>
<% if ((currentUser) && (currentUser.id === user.id)) { %>
  <a href="/edit" class="pull-right">Edit your profile</a>
<% } %>

<h1><%= user.name() %></h1>
<h2>Joined on <%= user.createdAt %></h2>

<% if (user.bio) { %>
  <p><%= user.bio %></p>
<% } %>

</body>
</html>
```

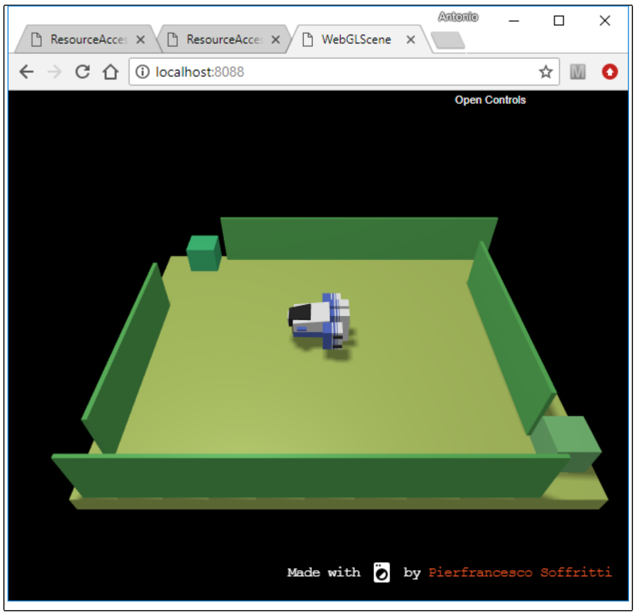
Listing 1.60. `frontend/appServer/viewRobot/profile.ejs`

8 Final task 2018

8.1 Requirements

In a home of a given city (e.g. Bologna), a `ddr` robot is used to clean the floor of a room (`R-FloorClean`).

The floor in the room is a flat floor of solid material and is equipped with two *sonars*, named `sonar1` and `sonar2` as shown in the picture (`sonar1` is that at the top). The initial position (`start-point`) of the robot is detected by `sonar1`, while the final position (`end-point`) is detected by `sonar2`.



The robot works under the following conditions:

1. `R-Start`: an `authorized user` has sent a `START` command by using a human `GUI` interface (`console`) running on a conventional `PC` or on a smart device (`Android`).
2. `R-TempOk`: the value temperature of the city is not higher than a prefixed value (e.g. `25` degrees Celsius).
3. `R-TimeOk`: the current clock time is within a given interval (e.g. between `7` a.m and `10` a.m)

While the robot is working:

– it must blink a `Led` put on it, if the robot is a `real` robot (`R-BlinkLed`).
– it must blink a Led Hue Lamp available in the house, if the robot is a `virtual` robot (`R-BlinkHue`).
– it must avoid fixed obstacles (e.g. furniture) present in the room (`R-AvoidFix`) and/or mobile obstacles like balls, cats, etc. (`R-AvoidMobile`).

Moreover, the robot must stop its activity when one of the following conditions apply:

1. `R-Stop`: an `authorized user` has sent a `STOP` command by using the `console`.
2. `R-TempKo`: the value temperature of the city becomes higher than the prefixed value.
3. `R-TimeKo`: the current clock time is beyond the given interval.
4. `R-Obstacle`: the robot has found an obstacle that it is unable to avoid.
5. `R-End`: the robot has finished its work.

During its work, the robot can optionally:

– `R-Map`: build a map of the room floor with the position of the fixed obstacles. Once built, this map can be used to define a plan for an (optimal) path form the `start-point` to the `end-point`.

Other requirements:

1. The work can be done by a team composed of `NT` people, with `1<=NT<=4`.
2. If `NT>1`, the team must explicitly indicate the work done by each component.
3. If `NT==4`, the requirement `R-Map` is mandatory.