nodeLab2018

Antonio Natali

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

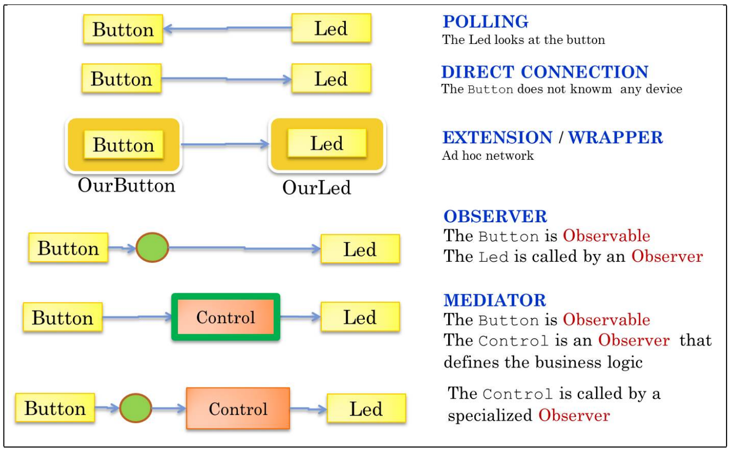
Table of Contents

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:



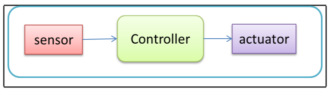
Since the application code cannot be responsibility neither of the Button nor of the Led, the schemes including an explicit `Control` component will be taken as our reference architectures.

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]



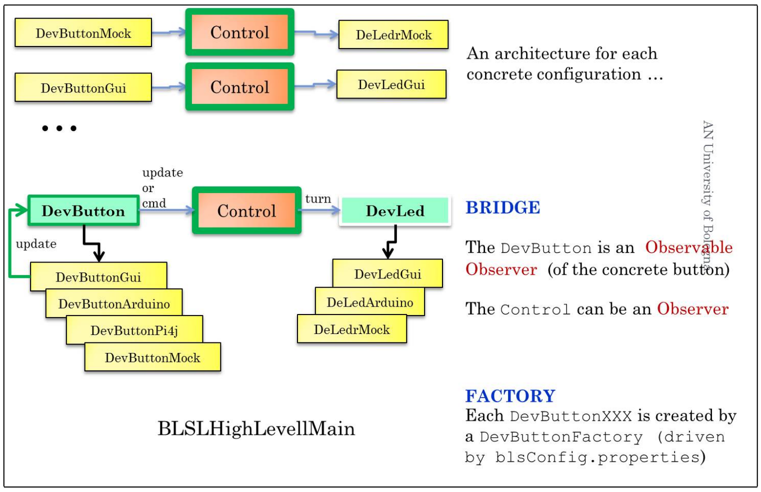
The basic idea is that each time the sensor change its state, the `Controller` performs some action on the actuator.

For an example, see Subsection 2.2.

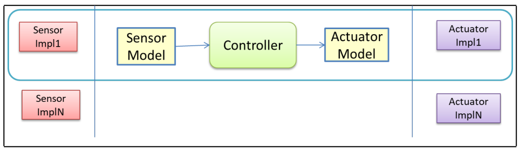
[1] The reader should decide whether this architecture scheme is the result of an analysis phase or a project phase.

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:

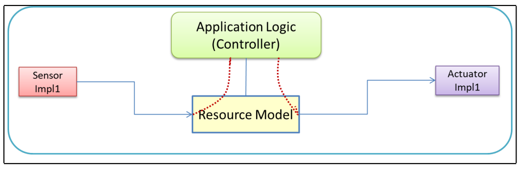


The idea is that each specific, technology-dependent sensor provides its own way to modify the sensor-model, while each modification in the model of the actuator-model should trigger an action in the technology-dependent actuator. The software designer can make reference to the observer pattern and/or 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':

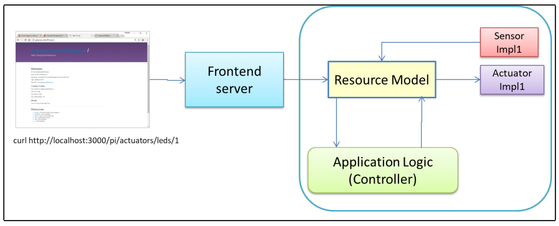


The idea is that the software designer should concentrate the attention on the most appropriate Resource Model in the application domain and delegate to the next step of 'architectural zooming' the details of the binding between the model and the concrete devices.

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.

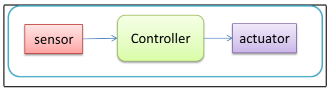
2 System models

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:

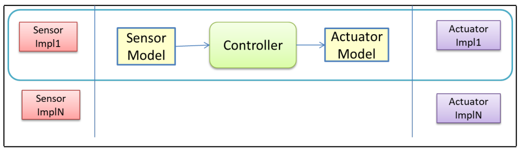
```
QActor qacontrol context bls1Ctx{
Rules{
    eval( ge, X, X ).
    eval( ge, X, V ):- eval( gt, X , V ) .
    evalTemperature( cold ) :-
        curTemperatureValue(V),
        //output(evalTemperature(V)),
        eval( lt, V , 30 ).
    evalTemperature( hot ) :-
        curTemperatureValue(V),
        //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

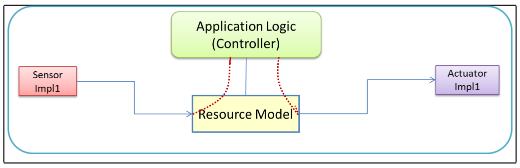
2.3.2 The controller.
The controller is modelled as an actor that waits for a `local_click` event emitted by a Button Mock and then forwards a `turn` dispatch to the actuator:

```
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://192.168.137.1:1883"
//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 ]
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 {
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{
    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.

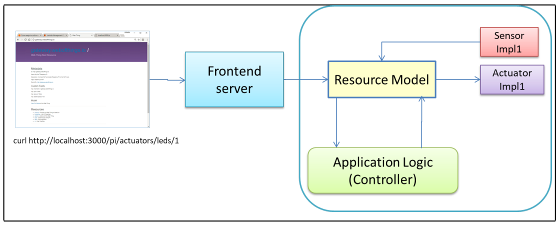
A Led working on RaspberryPi is defined in a project named `it.unibo.bls17.ledrasp` (read also `lowLevelZooming.pdf`):

```
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(starts) )
    ]
    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 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.
   (b) Use the server as a frontend for the application of Subsection 2.4. In this case 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:

```
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.

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 simple shows 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 one or more resource plug-in (see Subsection 3.9):

```
/*
 * 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});
```

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:

```
}

createServer(3000);

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){
    res.send(req.result); }
);
//app.use(converter());
```

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/models/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
 */
var express    = require('express'),
  router       = express.Router(),
  resourceModel = require('../models/model');

router.route('/').get(function (req, res, next) {
      //console.info( resourceModel.pi.actuators );
      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;
  console.info('route LED Changed LED %s value to %s', req.params.id, selectedLed.value);
  req.result = selectedLed;
```

```
  emitInfo(selectedLed.value);
  next();
});
```

Listing 1.24. `appServer/models/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 emitInfo = function( ledValue ){
    var val = "off";
    if( ledValue === "true" ) val = "on";
    var eventstr = "msg(ctrlEvent,event,js,none,ctrlEvent(leds, led1, " +val + "),1)"
        console.log(" ledPlugin LED emits> "+ eventstr);
        mqttUtils.publish( eventstr );
}

module.exports = router;
```

Listing 1.25. `appServer/models/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');

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 Introduce a model

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`.

```
{
  "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": 4
        },
        "2": {
          "name": "LED 2",
          "value": false,
          "gpio": 9
        }
      }
    }
  }
}
```

Listing 1.27. `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');
module.exports = resources;
```

Listing 1.28. `appServer/models/model.js`

3.9 Led plugin

```
/*
 * 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};

exports.start = function (params) {
  localParams = params;
  observe(ledModel); //#A

```

```
  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 + "),1)"
        console.log(" ledPlugin LED observed> "+ observable.value);
//      console.log(" ledPlugin LED emits> "+ eventstr);
//      mqttUtils.publish( eventstr );
//      mqttUtils.publish("LED1 value=" + observable.value );
//      sendMsg("msg(jsdata,event,jsSource,none,jsdata(led1, " + observable.value + "),1)");
    });
};

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 1.29. `frontend/plugins/internal/ledsPlugin.js`

3.10 Temperature/Humidity plugin

```
/*
 * 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 1.30. frontend/plugins/internal/DHT22sensorPlugin.js