

nodeLab2018

Antonio Natali

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

Table of Contents

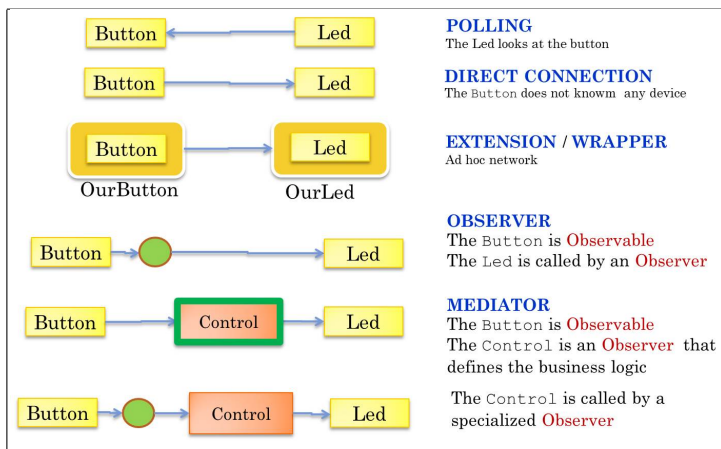
nodeLab2018	1
<i>Antonio Natali</i>	
1 Introduction.....	2
1.1 Object-oriented architectures	2
1.2 Start up.....	2
1.3 Decoupling from technological details	3
1.4 An architectural style	3
1.5 A frontend server.....	3
2 System models	5
2.1 Mock objects	5
2.2 Start up.....	5
2.2.1 The sensor.	5
2.2.2 The 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.....	12
3.2 Refactoring according to the MVC pattern	13
3.3 The Express use pattern	13
3.4 The server entry-point	14
3.5 applCode.....	15
3.5.1 Routing rules.	16
3.6 Routers: sensors	16
3.7 Routers: actuators	17
3.7.1 MQTT utils.....	18
3.8 Introduce a model.....	18
3.9 Led plugin	19
3.10 Temperature/Humidity plugin	20
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 starting 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



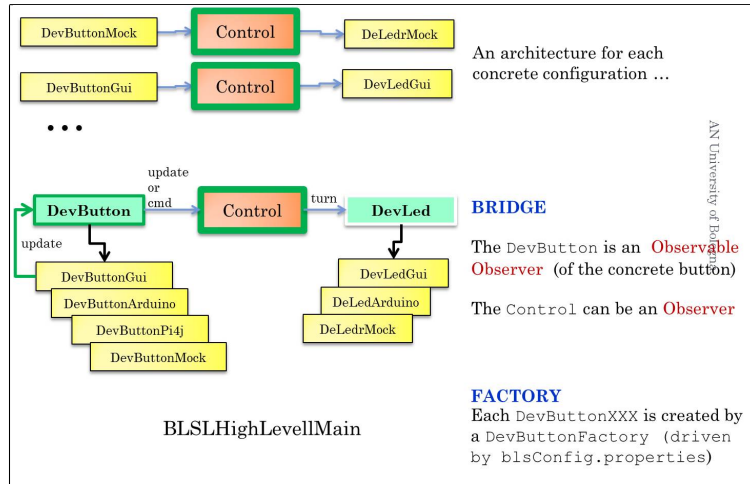
The basic idea is that each time the sensor change its state, the **Controller** performs some action on the actuator.

For an example, see Subsection 2.2.

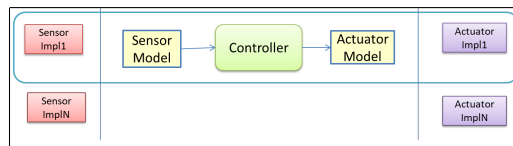
¹ The reader should decide whether this architecture scheme is the result of an analysis phase or a project phase.

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) from technological details:

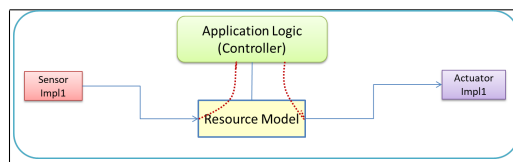


The idea is that each specific, technology-dependent sensor provides its own way to modify the sensor-model, while each modification in the model of the actuator-model should trigger an action in the technology-dependent actuator. The software designer can make reference to the **observer pattern** and/or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

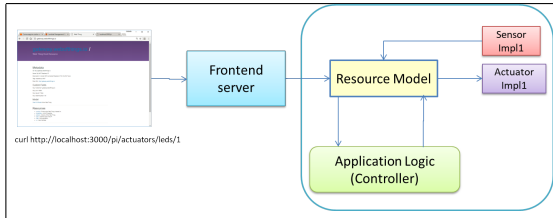


The idea is that the software designer should concentrate the attention on the most appropriate **Resource Model** in the application domain and delegate to the next step of 'architectural zooming' the details of the binding between the model and the concrete devices.

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.

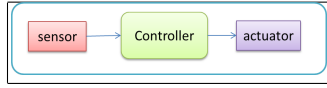
2 System models

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 **ROb** of Subsection 1.1.

Our formal specification starts with the definition of the events/messages used by the components to exchange information³:

```
1 System bls1
2
3 Event sensorEvent : sensorEvent( DATA ) //DATA : integer
4 Event ctrlEvent : ctrlEvent( CMD ) //CMD = on | off
5
6 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`:

```
1 QActor qasensor context bls1Ctx {
2   Plan init normal [
3     println( qasensor(starts) );
4     delay 1000;
5     emit sensorEvent : sensorEvent( 20 ) ;
6     delay 1000;
7     emit sensorEvent : sensorEvent( 30 ) ;
8     delay 1000;
9     emit sensorEvent : sensorEvent( 28 ) ;
10    delay 1000;
11    emit sensorEvent : sensorEvent( 35 ) ;
12    delay 1000
13  ]
14 }
```

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 then performs its job by using a Led Mock provided by the custom Java class `customBlsGui`⁴.

```

1 QActor qaactuator context bls1Ctx{
2   Plan init normal [
3     println( qaactuator(starts) ) ;
4     javaRun it.unibo.custom.gui.customBlsGui.createCustomLedGui()
5   ]
6   switchTo waitForCommand
7
8   Plan waitForCommand[ ]
9   transition stopAfter 100000
10    whenEvent ctrlEvent -> handleCmd
11    finally repeatPlan
12
13   Plan handleCmd resumeLastPlan[
14 //    printCurrentEvent;
15    onEvent ctrlEvent : ctrlEvent(on) -> javaRun it.unibo.custom.gui.customBlsGui.setLed("on");
16    onEvent ctrlEvent : ctrlEvent(off) -> javaRun it.unibo.custom.gui.customBlsGui.setLed("off")
17  ]
18 }

```

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:

```

1 QActor qacontrol context bls1Ctx{
2 Rules{
3   eval( ge, X, X ).
4   eval( ge, X, V ):- eval( gt, X, V ) .
5   evalTemperature( cold ) :-
6     curTemperatureValue(V),
7     //output(evalTemperature(V)),
8     eval( lt, V, 30 ).
9   evalTemperature( hot ) :-
10    curTemperatureValue(V),
11    //output(evalTemperature(V)),
12    eval( ge, V, 30 ), !.
13 }
14 Plan init normal [
15   println( qacontrol(starts) )
16 ]
17 switchTo waitForSensorEvent
18
19 Plan waitForSensorEvent[ ]
20 transition stopAfter 100000
21 whenEvent sensorEvent -> handleSensorEvent
22 finally repeatPlan
23
24 Plan handleSensorEvent resumeLastPlan [
25   printCurrentEvent;
26   onEvent sensorEvent : sensorEvent( V ) ->
27     ReplaceRule curTemperatureValue(X) with curTemperatureValue(V);
28   [ !? evalTemperature(hot) ] emit ctrlEvent : ctrlEvent(on) else emit ctrlEvent : ctrlEvent(off)
29 ]
30 }

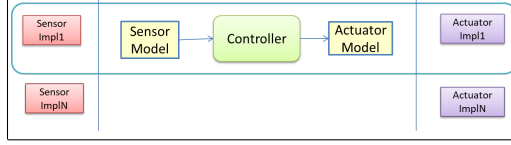
```

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 information⁵:

```

1 System blsim
2 Dispatch turn : switch
3 Event local_click : clicked(N) //N : natural
4
5 Context blsimCtx ip [ host="localhost" port=8049 ]

```

Listing 1.5. blsim.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:

```

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

```

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

```

1 QActor qaledm context blsimCtx {
2 Rules{
3   ledmodel( name(led1), value(off) ).
4   switchLedValue(on) :-
5     ledmodel( name(led1), value(off) ),
6     replaceRule( ledmodel( NAME,value(off) ), ledmodel( NAME,value(on) ) ), !.
7   switchLedValue(off) :-
8     ledmodel( name(led1), value(on) ),
9     replaceRule( ledmodel( NAME,value(on) ), ledmodel( NAME,value(off) ) ), !.
10 }

```

Listing 1.6. blsim.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`):

```

1 Plan init normal [
2   javaRun it.unibo.custom.gui.customBlsGui.createCustomLedGui();
3   delay 100;
4   [ !? ledmodel( NAME, value(V) ) ] javaRun it.unibo.custom.gui.customBlsGui.setLed(V)
5 ]
6 switchTo waitForCmd
7
8 Plan waitForCmd [ ]
9 transition stopAfter 3000000
10 whenMsg turn -> ledswitch
11 finally repeatPlan
12
13 //model-based behavior
14 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.


```

15 [ !? switchLedValue(V) ] javaRun it.unibo.custom.gui.customBlsGui.setLed(V)
16 ]
17 }

```

Listing 1.7. blsim.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:

```

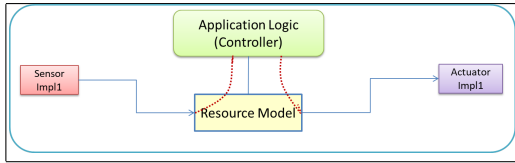
1 QActor qacontrolm context blsimCtx{
2   Plan init normal [
3     println( qacontrol(starts) ) ;
4     javaRun it.unibo.custom.gui.customBlsGui.createCustomButtonGui()
5   ]
6   switchTo waitForClick
7
8   Plan waitForClick[ ]
9   transition stopAfter 100000
10    whenEvent local_click : clicked(N) do forward qaledm -m turn : switch
11    finally repeatPlan
12 }

```

Listing 1.8. blsim.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`:

```

1 model( type(actuator, leds), name(led1), value(off) ).
2 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:

```

1 getModelItem( TYPE, CATEG, NAME, VALUE ) :-
2   model( type(TYPE, CATEG), name(NAME), value(VALUE) ).
3 changeModelItem( CATEG, NAME, VALUE ) :-
4   replaceRule(
5     model( type(TYPE, CATEG), name(NAME), value(_) ),
6     model( type(TYPE, CATEG), name(NAME), value(VALUE) )
7   ),!,
8   %%output( changedModelAction(CATEG, NAME, VALUE) ),
9   ( changedModelAction(CATEG, NAME, VALUE) %%to be defined by the appl designer
10    ; 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:

```

1 eval( ge, X, X ) :- !.
2 eval( ge, X, V ) :- eval( gt, X, V ) .
3
4 emitEvent( EVID, EVCONTENT ) :-
5     actorobj( Actor ),
6     %%output( emit( Actor, EVID, EVCONTENT ) ),
7     Actor <- emit( EVID, EVCONTENT ).
8 %% initialize
9 initResourceTheory :- output("initializing the initResourceTheory ...").
10 :- 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:

```

1 System blsMvc
2 Event sensorEvent : sensorEvent( NAME, DATA )
3 Event changeModel : changeModelItem( TYPE, CATEG, NAME, VALUE )
4 Event ctrlEvent : ctrlEvent( CATEG, NAME,CMD ) //CMD depends on CATEG/NAME
5
6 Event inputCtrlEvent : inputEvent( CATEG, NAME, VALUE )
7 Event outputCtrlEvent : outputEvent( DATA ) //DATA : integer
8
9 pubSubServer "tcp://192.168.137.1:1883"
10 //pubSubServer "tcp://192.168.43.229:1883"
11 //pubSubServer "tcp://m2m.eclipse.org:1883"
12 //pubSubServer "tcp://test.mosquitto.org:1883"
13
14 Context blsMvcCtx ip [ host="192.168.137.1" port=8019 ]

```

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 or mock devices.

```

1 * performs ACTIONS that change the model of some sensor/actuator resource
2 * and activates changedModelAction/3 that performs an action for the world
3 */
4 QActor mvccontroller context blsMvcCtx -pubsub{
5 Rules{ //The model is in the theory resourceModel.
6     //Here we write the actions to be performed when the model changes.
7     //The change of the temperature t1 could modify a Led
8     limitTemperatureValue( 25 ).
9     changedModelAction( temperature, t1, V ):-
10         limitTemperatureValue( MAX ),
11         eval( ge, V, MAX ), !,
12         changeModelItem( leds, led1, on).
13     changedModelAction( temperature, t1, V ):-
14         changeModelItem( leds, led1, off).

```

```

15
16 //The change of a Led model must activate an actuator (working as an event listener)
17 changedModelAction( leds, led1, V ):-
18     emitEvent( ctrlEvent, ctrlEvent( leds, led1, V ) ).
19 }
20 Plan init normal [
21     demo consult("./resourceModel.pl"); //contains the models and related rules
22     println( qacontrol(starts) )
23 ]
24 switchTo waitForInputEvent
25
26 Plan waitForInputEvent[ ]
27 transition stopAfter 6000000
28     whenEvent inputCtrlEvent -> handleInputEvent
29 finally repeatPlan
30
31 Plan handleInputEvent resumeLastPlan [
32     demo a;
33     printCurrentEvent;
34     onEvent inputCtrlEvent : inputEvent( CATEG, NAME, VALUE ) -> //change the model

```

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:

```

1  * ACTUATOR
2  * manage a Led working with a SPECIFIC Led implementation
3  */
4  QActor ledmockgui context blsMvcCtx{
5      Plan init normal [
6          println( ledmockgui(starts) );
7          javaRun it.unibo.custom.gui.customBlsGui.createCustomLedGui()
8      ]
9      switchTo waitForCommand
10
11      Plan waitForCommand[ ]
12      transition stopAfter 100000
13          whenEvent ctrlEvent -> handleCmd
14      finally repeatPlan
15
16      Plan handleCmd resumeLastPlan[
17          printCurrentEvent;
18          onEvent ctrlEvent : ctrlEvent(leds, led1, on) -> javaRun it.unibo.custom.gui.customBlsGui.setLed("on");

```

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:

```

1  ]
2  }
3
4  QActor ledarduino context blsMvcCtx {
5      Plan init normal [
6          println( ledarduino(starts) );
7          javaRun it.unibo.utils.arduino.connArduino.initPc("COM9", "9600")
8      ]
9      switchTo waitForCommand
10
11      Plan waitForCommand[ ]
12      transition stopAfter 6000000
13          whenEvent ctrlEvent -> handleCmd
14      finally repeatPlan
15
16      Plan handleCmd resumeLastPlan[
17          printCurrentEvent;
18          onEvent ctrlEvent : ctrlEvent(leds, led1, on) -> javaRun it.unibo.utils.arduino.connArduino.sendToArduino("1");

```

Listing 1.15. blsMVC.qa: a Led on Arduino

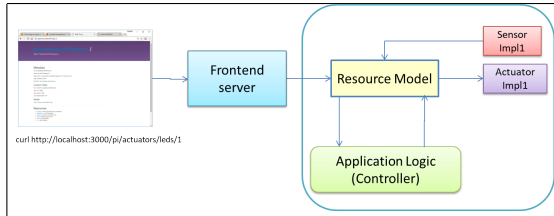
2.4.6 A Led on RaspberryPi. A Led working on RaspberryPi can be introduced as follows (read also [lowLevelZooming.pdf](#)):

```
1 System ledOnRasp
2 Event ctrlEvent : ctrlEvent( CATEG, NAME,CMD ) //CMD depends on CATEG/NAME
3
4 pubSubServer "tcp://192.168.43.229:1883"
5 //pubSubServer "tcp://192.168.137.1" //does not work (perhaps public?)
6 //pubSubServer "tcp://m2m.eclipse.org:1883"
7 //pubSubServer "tcp://test.mosquitto.org:1883"
8
9 Context ctxLedOnRasp ip [ host="192.168.43.18" port=8079 ]
10 //Context blsMvcCtx ip [ host="192.168.43.229" port=8019 ] -standalone
11
12 QActor ledrasp context ctxLedOnRasp -pubsub{ //-pubsub required since it must be connected
13   Plan init normal [
14     println( ledraspmqtt(starts) )
15   ]
16   switchTo waitForCommand
17
18   Plan waitForCommand[ ]
19   transition stopAfter 6000000
20   whenEvent ctrlEvent -> handleCmd
21   finally repeatPlan
22
23   Plan handleCmd resumeLastPlan[
24     printCurrentEvent;
25     onEvent ctrlEvent : ctrlEvent(leds, led1, on) ->
26       javaOp "customExecute(\"sudo bash led25GpioTurnOn.sh\")";
27     onEvent ctrlEvent : ctrlEvent(leds, led1, off) ->
28       javaOp "customExecute(\"sudo bash led25GpioTurnOff.sh\")"
29   ]
30 }
```

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 :

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

```
1 msg(ctrlEvent,event,js,none,ctrlEvent(leds, led1, VAL,1) //for actuators (Led)
2 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:

- ```
1 Create a new project it.unibo.frontend
2 Create the folder nodeCode/frontend and open a terminal in this folder
3 Execute express
4 Execute npm install
5 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:

```
1 var express = require('express');
2 var path = require('path');
3 var favicon = require('serve-favicon');
4 var logger = require('morgan');
5 var cookieParser = require('cookie-parser');
6 var bodyParser = require('body-parser');
7
8 var index = require('./appServer/routes/index'); //modified as 7.9:
9 //var users = require('./routes/users');
10
11 var app = express();
12
13 // view engine setup;
14 app.set('views', path.join(__dirname, 'appServer', 'views')); //modified as 7.9;
15 app.set('view engine', 'jade');
16
17 // uncomment after placing your favicon in /public
18 //app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
19 app.use(logger('dev'));
20 app.use(bodyParser.json());
21 app.use(bodyParser.urlencoded({ extended: false }));
22 app.use(cookieParser());
23 app.use(express.static(path.join(__dirname, 'public')));
24
25 app.use('/', index);
26 //app.use('/users', users);
27
28 // catch 404 and forward to error handler;
29 app.use(function(req, res, next) {
30 var err = new Error('Not Found');
31 err.status = 404;
32 next(err);
33 });
34
35 // error handler
36 app.use(function(err, req, res, next) {
37 // set locals, only providing error in development;
38 res.locals.message = err.message;
39 res.locals.error = req.app.get('env') === 'development' ? err : {};
40
41 // render the error page;
42 res.status(err.status || 500);
43 res.render('error');
44 });
45
46 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 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:

```

1 var express = require("express");
2 var http = require("http");
3
4 var app = express();
5
6 app.use(...);
7
8 app.get(...);
9
10 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 `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):

```

1 /*
2 * frontend/frontendServer.js
3 */
4 var appl = require('./applCode'); //previously was app;
5 var resourceModel = require('./appServer/models/model');
6 var http = require('http');
7
8 var createServer = function (port) {
9 console.log("process.env.PORT=" + process.env.PORT + " port=" + port);
10 if (process.env.PORT) port = process.env.PORT;
11 else if (port === undefined) port = resourceModel.customFields.port;
12
13 initPlugins();
14
15 server = http.createServer(appl);
16 server.on('listening', onListening);
17 server.on('error', onError);
18 server.listen(port);
19 };
20
21 function initPlugins() {
22 ledsPlugin = require('./plugins/internal/ledsPlugin'); //global variable;
23 ledsPlugin.start({ 'simulate': true, 'frequency': 5000 });
24
25 dhtPlugin = require('./plugins/internal/DHT22SensorPlugin'); //global variable;
26 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:

```

1 }
2
3 createServer(3000);
4
5 function onListening() {
6 var addr = server.address();

```

```

7 var bind = typeof addr === 'string'
8 ? 'pipe ' + addr
9 : 'port ' + addr.port;
10 console.log('Listening on ' + bind);
11 }
12 function onError(error) {
13 if (error.syscall !== 'listen') {
14 throw error;
15 }
16 var bind = typeof port === 'string'
17 ? 'Pipe ' + port
18 : 'Port ' + port;
19 // handle specific listen errors with friendly messages;
20 switch (error.code) {
21 case 'EACCES':
22 console.error(bind + ' requires elevated privileges');
23 process.exit(1);
24 break;
25 case 'EADDRINUSE':
26 console.error(bind + ' is already in use');
27 process.exit(1);
28 break;
29 default:
30 throw error;
31 }
32 }
33 //Handle CTRL-C;
34 process.on('SIGINT', function () {
35 ledsPlugin.stop();
36 dhtPlugin.stop();
37 console.log('frontendServer Bye, bye!');
38 process.exit();
39 });
40 process.on('exit', function(code){
41 console.log("Exiting code= " + code);
42 });
43 process.on('uncaughtException', function (err) {
44 console.error('mqtt got uncaught exception:', err.message);
45 process.exit(1); //MANDATORY!!!
46 });

```

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':

```

1 var express = require('express');
2 var path = require('path');
3 var favicon = require('serve-favicon');
4 var logger = require('morgan'); //see 10.1 of nodeExpressWeb.pdf;
5 var cookieParser = require('cookie-parser');
6 var bodyParser = require('body-parser');
7 var fs = require('fs');
8 var index = require('./appServer/routes/index');
9 var actuatorsRoutes = require('./appServer/routes/actuators');
10 var sensorsRoutes = require('./appServer/routes/sensors');
11
12 var app = express();
13
14 // view engine setup;
15 app.set('views', path.join(__dirname, 'appServer', 'views'));
16 app.set('view engine', 'jade');
17
18 //create a write stream (in append mode) ;
19 var accessLogStream = fs.createWriteStream(path.join(__dirname, 'morganLog.log'), {flags: 'a'})
20 app.use(logger("short", {stream: accessLogStream}));
21
22 //Creates a default route. Overloads app.use('/', index);
23 //app.get("/", function(req,res){ res.send("Welcome to frontend Server"); });

```



```

24
25 // uncomment after placing your favicon in /public
26 //app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
27 app.use(logger('dev')); //shows commands, e.g. GET /pi 304 23.123 ms - -;
28 app.use(bodyParser.json());
29 app.use(bodyParser.urlencoded({ extended: false }));
30 app.use(cookieParser());
31 app.use(express.static(path.join(__dirname, 'public')));

```

Listing 1.20. `appCode.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](#))

```

1 //DEFINE THE ROUTES ;
2 app.use('/', index);
3 app.use('/pi/actuators', actuatorsRoutes);
4 app.use('/pi/sensors', sensorsRoutes);
5
6 //Creates a default route for /pi;
7 app.get('/pi', function (req, res) {
8 //for(i in req.body){ console.info('req body field %s ', i); };
9 //console.info(' get /pi req URL = %s ', req.url);
10 res.send('This is the frontend-Pi!')
11 });
12
13 //REPRESENTATION;
14 app.use(function(req,res){
15 res.send(req.result); }
16);
17 //app.use(converter());

```

Listing 1.21. `appCode.js`: routing

The last part deals with errors:

```

1 // catch 404 and forward to error handler;
2 app.use(function(req, res, next) {
3 var err = new Error('Not Found');
4 err.status = 404;
5 next(err);
6 });
7
8 // error handler;
9 app.use(function(err, req, res, next) {
10 // set locals, only providing error in development
11 res.locals.message = err.message;
12 res.locals.error = req.app.get('env') === 'development' ? err : {};
13
14 // render the error page;
15 res.status(err.status || 500);
16 res.render('error');
17 });
18
19 module.exports = app;

```

Listing 1.22. `appCode.js`: error handling

### 3.6 Routers: sensors

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

```

1 /*
2 * appServer/routes/sensors.js
3 */
4 var express = require('express'),
5 router = express.Router(),

```

```

6 resourceModel = require('../models/model');
7
8 router.route('/').get(function (req, res, next) {
9 req.type = "defaultView" ;
10 req.result = resourceModel.pi.sensors;
11 next();
12 });
13
14 router.route('/pir').get(function (req, res, next) {
15 req.result = resourceModel.pi.sensors.pir;
16 next();
17 });
18
19 router.route('/temperature').get(function (req, res, next) {
20 console.log(".....");
21 console.log(req.result);
22 req.result = resourceModel.pi.sensors.temperature;
23 console.log(req.result);
24 console.log(".....");
25 next();
26 });
27
28 router.route('/temperatureProlog').get(function (req, res, next) {
29 var tval = resourceModel.pi.sensors.temperature.value ;
30 console.log(tval);
31 req.result = "msg(sensor, event, temperatureDev, none, "+ tval+", 0)";
32 next();
33 });
34
35 router.route('/humidity').get(function (req, res, next) {
36 req.result = resourceModel.pi.sensors.humidity;
37 next();
38 });
39
40 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.

```

1 /*
2 * appServer/routes/actuators.js
3 */
4 var express = require('express'),
5 router = express.Router(),
6 resourceModel = require('../models/model');
7
8 router.route('/').get(function (req, res, next) {
9 //console.info(resourceModel.pi.actuators);
10 req.result = resourceModel.pi.actuators;
11 next();
12 });
13
14 router.route('/leds').get(function (req, res, next) {
15 req.result = resourceModel.pi.actuators.leds;
16 next();
17 });
18
19 router.route('/leds/:id').get(function (req, res, next) {
20 //(curl) http://localhost:3000/pi/actuators/leds/1;
21 req.result = resourceModel.pi.actuators.leds[req.params.id];
22 next();
23 })
24 .put(function(req, res, next) {
25 //(curl -H "Content-Type: application/json" -X PUT -d '{"value": |"true|" }' http://localhost:3000/pi/actuators/leds/1;
26 var selectedLed = resourceModel.pi.actuators.leds[req.params.id];
27 selectedLed.value = req.body.value;
28 console.info('route LED Changed LED %s value to %s', req.params.id, selectedLed.value);
29 req.result = selectedLed;

```

```

30 emitInfo(selectedLed.value);
31 next();
32 });

```

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.

```

1 /*
2 * Emit the new led value according to the blsMVC model
3 */
4 var mqttUtils = require('.././../uniboSupports/mqttUtils');
5
6 var emitInfo = function(ledValue){
7 var val = "off";
8 if(ledValue === "true") val = "on";
9 var eventstr = "msg(ctrlEvent,event,js,none,ctrlEvent(leds, led1, " + val + "),1)"
10 console.log(" ledPlugin LED emits> " + eventstr);
11 mqttUtils.publish(eventstr);
12 }
13
14 module.exports = router;

```

Listing 1.25. appServer/models/actuators.js

### 3.7.1 MQTT utils .

```

1 /*
2 * =====
3 * uniboSupports/mqttUtils.js
4 * =====
5 */
6 const mqtt = require ('mqtt');
7 const topic = "unibo/qasys";
8 //var client = mqtt.connect('mqtt://iot.eclipse.org');
9 var client = mqtt.connect('mqtt://localhost');
10
11 console.log("mqtt client= " + client);
12
13 client.on('connect', function () {
14 client.subscribe(topic);
15 console.log('client has subscribed successfully ');
16 });
17
18 //The message usually arrives as buffer, so I had to convert it to string data type.
19 client.on('message', function (topic, message){
20 console.log("mqtt RECEIVES:" + message.toString()); //if toString is not given, the message comes as buffer
21 });
22
23 exports.publish = function(msg){
24 //console.log('mqtt publish ' + client);
25 client.publish(topic, msg);
26 }

```

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`.

```

1 {
2 "pi": {
3 "name": "WoT Pi",

```

```

4 "description": "A simple WoT-connected Raspberry PI for the WoT book.",
5 "port": 8484,
6 "sensors": {
7 "temperature": {
8 "name": "Temperature Sensor",
9 "description": "An ambient temperature sensor.",
10 "unit": "celsius",
11 "value": 0,
12 "gpio": 12
13 },
14 "humidity": {
15 "name": "Humidity Sensor",
16 "description": "An ambient humidity sensor.",
17 "unit": "%",
18 "value": 0,
19 "gpio": 12
20 },
21 "pir": {
22 "name": "Passive Infrared",
23 "description": "A passive infrared sensor. When 'true' someone is present.",
24 "value": true,
25 "gpio": 17
26 }
27 },
28 "actuators": {
29 "leds": {
30 "1": {
31 "name": "LED 1",
32 "value": false,
33 "gpio": 4
34 },
35 "2": {
36 "name": "LED 2",
37 "value": false,
38 "gpio": 9
39 }
40 }
41 }
42 }
43 }

```

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.

```

1 var resources = require('./resources.json');
2 module.exports = resources;

```

Listing 1.28. appServer/models/model.js

### 3.9 Led plugin

```

1 /*
2 * frontend/plugins/internal/ledsPlugin.js
3 */
4 var resourceModel = require('../../appServer/models/model');
5 var observable = require('../../uniboSupports/observableFactory');
6 var mqttUtils = require('../../uniboSupports/mqttUtils');
7
8 var actuator, interval;
9 var ledModel = resourceModel.pi.actuators.leds['1'];
10 var pluginName = ledModel.name;
11 var localParams = {'simulate': false, 'frequency': 2000};
12
13 exports.start = function (params) {
14 localParams = params;
15 observe(ledModel); ///#A
16 }

```

```

17 if (localParams.simulate) {
18 // simulate();
19 } else {
20 connectHardware();
21 }
22 };
23
24 exports.stop = function () {
25 if (localParams.simulate) {
26 clearInterval(interval);
27 } else {
28 actuator.unexport();
29 }
30 console.info('%s plugin stopped!', pluginName);
31 };
32
33 function observe(what) {
34 console.info('plugin observe: ' + localParams.frequency + " CHANGE MDOEL INTO OBSERVABLE");
35 console.info(what);
36 //Change the ledModel into an observable;
37 const whatObservable = new observable(what);
38 observable = whatObservable.data;
39 whatObservable.observe('value', () => {
40 var val = "off";
41 if(observable.value === "true") val = "on";
42 var eventstr = "msg(ctrlEvent,event,js,none,ctrlEvent(leds, led1, " + val + "),1)"
43 console.log(" ledPlugin LED observed> "+ observable.value);
44 // console.log(" ledPlugin LED emits> "+ eventstr);
45 // mqttUtils.publish(eventstr);
46 // mqttUtils.publish("LED1 value=" + observable.value);
47 // sendMsg("msg(jsdata,event,jsSource,none,jsdata(led1, " + observable.value + "),1)");
48 });
49 };
50
51 function switchOnOff(value) {
52 if (!localParams.simulate) {
53 actuator.write(value === true ? 1 : 0, function () {
54 console.info('Changed value of %s to %s', pluginName, value);
55 });
56 }
57 };
58
59 function connectHardware() {
60 var Gpio = require('onoff').Gpio;
61 actuator = new Gpio(ledModel.gpio, 'out');
62 console.info('Hardware %s actuator started!', pluginName);
63 };
64
65 function simulate() {
66 interval = setInterval(function () {
67 // Switch value on a regular basis;
68 if (ledModel.value) {
69 ledModel.value = false;
70 } else {
71 ledModel.value = true;
72 }
73 // console.log("LED=" + ledModel.value);
74 }, localParams.frequency);
75 console.info('Simulated %s actuator started!', pluginName);
76 };

```

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

### 3.10 Temperature/Humidity plugin

```

1 /*
2 * frontend/plugins/internal/DHT22sensorPlugin.js
3 */
4 var
5 resources = require('../../appServer/models/model'),

```

```

6 utils = require('.././../utils.js');
7 var interval, sensor;
8 var model = resources.pi.sensors;
9 var pluginName = 'Temperature & Humidity';
10 var localParams = {'simulate': true, 'frequency': 5000};
11
12 exports.start = function (params) {
13 localParams = params;
14 if (params.simulate) {
15 simulate();
16 } else {
17 connectHardware();
18 }
19 };
20 exports.stop = function () {
21 if (localParams.simulate) {
22 clearInterval(interval);
23 } else {
24 sensor.unexport();
25 }
26 console.info('%s plugin stopped!', pluginName);
27 };
28
29 function connectHardware() {
30 var sensorDriver = require('node-dht-sensor');
31 var sensor = {
32 initialize: function () {
33 return sensorDriver.initialize(22, model.temperature.gpio);
34 },
35 read: function () {
36 var readout = sensorDriver.read();
37 model.temperature.value = parseFloat(readout.temperature.toFixed(2));
38 model.humidity.value = parseFloat(readout.humidity.toFixed(2));
39 showValue();
40 setTimeout(function () {
41 sensor.read(); //##D
42 }, localParams.frequency);
43 }
44 };
45 if (sensor.initialize()) {
46 console.info('Hardware %s sensor started!', pluginName);
47 sensor.read();
48 } else { console.warn('Failed to initialize sensor!'); }
49 };
50
51 function simulate() {
52 interval = setInterval(function () {
53 model.temperature.value = utils.randomInt(0, 40);
54 model.humidity.value = utils.randomInt(0, 100);
55 showValue();
56 }, localParams.frequency);
57 console.info('Simulated %s sensor started!', pluginName);
58 };
59
60 function showValue() {
61 console.info('Temperature: %s C, humidity %s %%',
62 model.temperature.value, model.humidity.value);
63 emitInfo(model.temperature.value);
64 };
65
66 /*
67 * Emit the new led value according to the blsHVC model
68 */
69 var mqttUtils = require('.././../uniboSupports/mqttUtils');
70
71 var emitInfo = function (value){
72 var eventstr = "msg(inputCtrlEvent,event,js,none,inputEvent(temperature, t1, " +value + "),1)"
73 console.log(" DHT22Plugin emits> "+ eventstr);
74 mqttUtils.publish(eventstr);
75 }

```

Listing 1.30. frontend/plugins/internal/DHT22sensorPlugin.js