nodeLab2018

Antonio Natali

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

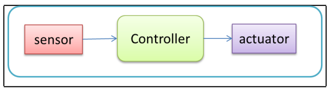
Table of Contents

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 Start up

Our first reference architecture can be informally introduced as follows [1]

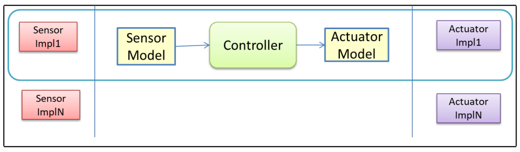


The basic idea is that each time the sensor chage its state, the controller performs some action on the actuator. For example, we could have the following requirements:

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 Decoupling from technological details

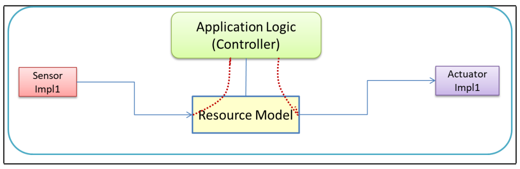
Sensors and actuators can be of different types or can be of a specific type (e.g. a Temperature sensor, a Led) but with different possible implementations. Our reference architecture could evolve by introducing models to decouple the controller form technological details:



The idea is that each specific, technology-dependent sensor provides its own way to modify its 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.

1.3 An architectural style

This step can lead us to propose a more general 'architectural style':



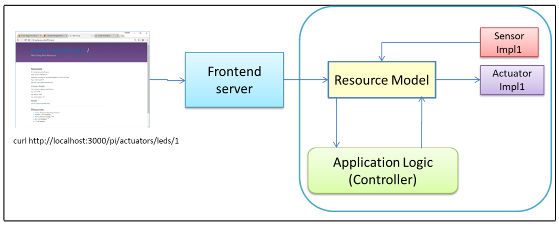
The idea is that the software designer should concentrate the attention on the most appropriate Resource Model in the application domain and delegate to the next step of 'architectural zooming' the details of the binding between the model and the concrete devices.

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

1.4 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 A core application

1. Define a custom `Java` class (named `customBlsGui`) to provide a Gui for a Button and for a Led.
2. Define a model for the Led. For example:

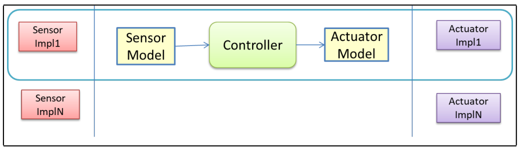
```
   ledmodel( name(led1), value(off) ).
   changeledValue(V) :- replaceRule( ledmodel( NAME,VALUE ), ledmodel( NAME,value(V) ) ).
```

3. Define a model for the architecture of a distributed system. For example:

```
System bls1
Dispatch turn   : switch
Event local_click : clicked(N) //N : natural
Event usercmd   : usercmd(N) //N : natural

Event ledcmd : ledcmd( CMD )  //CMD = on | off

Context bls1Ctx ip [ host="localhost" port=8029 ]

/*
 * LED subsystem
 */
QActor ledimpl context bls1Ctx {
Plan init normal [
        javaRun it.unibo.custom.gui.customBlsGui.createCustomLedGui()
    ]
    switchTo waitForCmd

    Plan waitForCmd [ ]
    transition stopAfter 3000000
        whenMsg ledcmd -> handleLedCmd
    finally repeatPlan

    Plan handleLedCmd resumeLastPlan[
        onEvent ledcmd : ledcmd( on ) -> javaRun it.unibo.custom.gui.customBlsGui.setLed("on");
        onEvent ledcmd : ledcmd( off ) -> javaRun it.unibo.custom.gui.customBlsGui.setLed("off")
    ]
}
QActor ledmodel context bls1Ctx {
Rules{
    ledmodel( name(led1), value(off) ).
    changeledValue( V ) :-
        replaceRule( ledmodel( NAME,VALUE ), ledmodel( NAME,value(V) ) ).
}
    Plan init normal [
        delay 500;  //give to the implementation the time to start
        [ !? ledmodel( NAME, value(V) )] emit ledcmd : ledcmd( V ) //RLed1
    ]
    switchTo waitForCmd

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

    //model-based behavior
    Plan ledswitch resumeLastPlan[
        [ !? ledmodel( NAME, value(off) )] addRule newLedValue( on );
        [ !? ledmodel( NAME, value(on) )] addRule newLedValue( off );
        [ ?? newLedValue(V)] demo changeledValue( V ) ;
        [ !? ledmodel( NAME, value(V) )] emit ledcmd : ledcmd( V ) //RLed2
     ]
}
/*
 * BUTTON subsystem
 */
QActor buttonimpl context bls1Ctx{
    Plan init normal [
        println( buttonimpl(starts) ) ;
        javaRun it.unibo.custom.gui.customBlsGui.createCustomButtonGui()
    ]
}
```

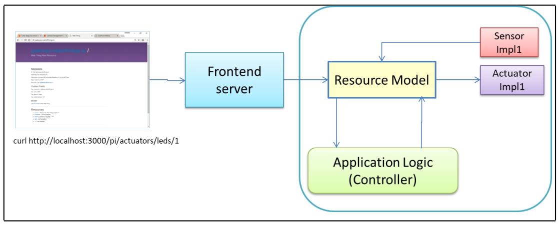
```
QActor control context bls1Ctx{
   Plan init normal [ ]
   switchTo waitForClick

   Plan waitForClick[ ]
   transition stopAfter 100000
      whenEvent local_click : clicked(N) do forward ledmodel -m turn : switch //RAppl1
   finally repeatPlan
}
```

Listing 1.1. `bls1.qa`

The reference architecture is: 

– The Controller is the `control` actor, that sends messages to the Led-Model.
– The Controller handles the events generated by the `buttonimpl` actor that uses a Button implemented in `Java`.
– The Led-Model is defined in Prolog by the `ledmodel` actor.
– The Led implementation is the the `ledimpl` actor that uses a mock led as a GUI implemented in `Java`.

Our goal now could be that of introducing a `frontend` server to provide a `REST`ful command interface with reference to the following architecture:



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`.

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 1.2. 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 Introduce a server

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 3.7):

```
/*
 * 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} );
}

createServer(3000);
```

Listing 1.3. `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();
  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 1.4. `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 1.5. `applCode.js`: starting

The 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 1.6. `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 1.7. `applCode.js`: error handling

3.6 Introduce a model

Let us introduce now a simple model of a set of sensor/actuators resources: a passive infrared (PIR) sensor, a temperature/humidity sensor and an 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 1.8. 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 1.9. appServer/models/model.js

3.7 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', () => {
        console.log(" ledPlugin LED observed> "+ observable.value);
        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 1.10. `frontend/plugins/internal/ledsPlugin.js`

```
/*
* =====================================
* uniboSupports/mqttUtils.js
* =====================================
*/
const mqtt  = require ('mqtt');
const topic = "unibo/actuators/led/1";
```

```
//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 1.11. `frontend/uniboSupports/mqttUtils.js`