FrontEnd2018

Antonio Natali

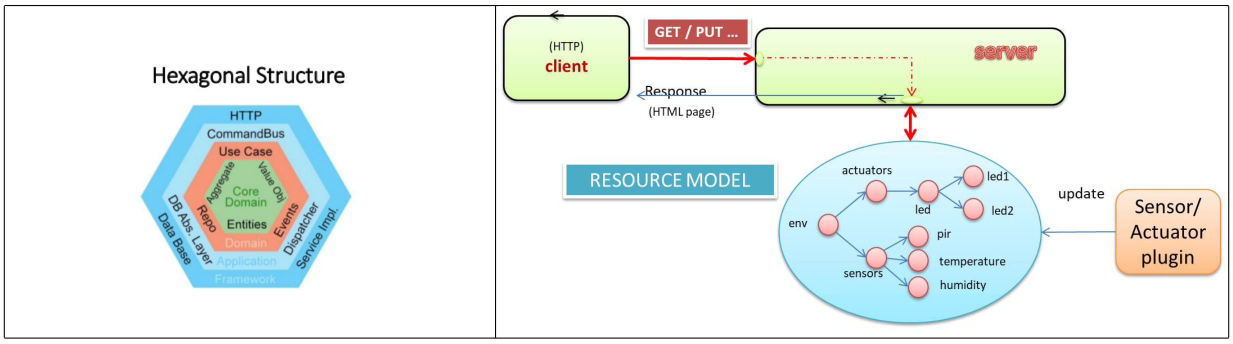
Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
antonio.natali@unibo.it

Table of Contents

1 A frontend server

In this work, we intend to develop a frontend server for applications involving a ddr-robot, by using Node.js and Express as our reference technology[1]. A good reference book for several topics we face in the work is: **Building the Web of Things** (ISBN: 9781617292682) of *D.D.Guinard and V.M. Trifa*, Manning 2016.

The goal is to build the prototype of a software system whose logic architecture is inspired to the HTTP/REST model adopted in the field of the *Web Of Things*. Informal logical reference architectures ar shown in the following picture (on the left the so called hexagonal (or port/adapter) architecture) :



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

The main aspects that qualify such an architecture are:

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

[1] An introduction to these technologies can be found in nodeExpressWeb.pdf.

1.1 A work-plan

The work-plan can be summarized as follows:

1. Set up a production environment based on Node.js and Express for the design and development of a frontend server for robot-based applications. The environment will be structured so to highlight an application structure based on the MVC pattern, by introducing a *model*, a *control* and one or more *views*.

2. Use Express to generate (Section 2) the code of the entry-point of our server according the Express pattern (Subsection 2.1). Afterwards, the structure of the generated code is slightly modified (Subsection 2.3) to explicitly introduce a directory for the *model* and a directory for the *controllers*.

3. Define the system entry-point as an HTTP server in which we load the middleware (Subsection 3.4) that defines the proper routes to the application logic for each external request pattern - an HTTP verb + URI (e.g. http://localhost:8080/applCommand/Cmd).

4. Introduce a resource model written in JSON (See Subsection 3.1) and make the server able to dynamically update (via socket.io) a representation of the model on a WebPage with information useful for the end-user, e.g. the state of a robot, the current values of sensors, etc. See Subsection 3.3.3 and Subsection 3.4.5.

5. Introduce a controller able to interact with external components (see Subsection 3.6.5) by using a MQTT broker. The external component can be written in any programming language, but the information exchanged will be represented according the QActor message format:

```
msg(MSGID,MSGTYPE,SENDER,RECEIVER,MSGCONTENT,NUM)
```

1.2 Usage

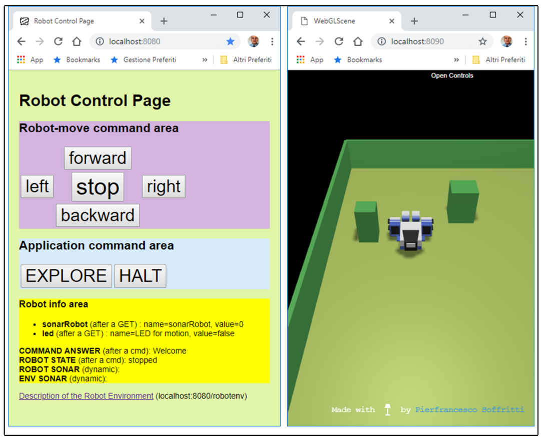
The code of the final system (see Section 3) is in the directory nodeCode/robotFrontend of the project it.unibo.mbot2018. To launch the system, the sequence of steps is:

1. Select a MQTT broker and set its address in the exported variable mqttbroker of the file robotFrontend/systemConfig.js[2].
2. Launch the virtual robot environment or activate a physical robot and set its address in the exported variable robotAddr of the file robotFrontend/systemConfig.js[3].
3. In nodeCode/frontend, execute node frontendServerRobot.js [ARG] with ARG= virtual | mbot | uniboRobot. If ARG is omitted, the system will use ARG=virtual. If ARG is specified, the system will load the corresponding robotPlugin (see Subsection 3.6.2).
4. Open a browser on localhost:8080.

On the screen we will see something like this:

[2] For example, if we launch a MQTT server on our machine by executing `docker run -ti -p 1883:1883 -p 9001:9001 eclipse-mosquitto`, then `mqttbroker=mqtt://localhost`.

[3] For example, if we launch the virtual robot environment on our machine by executing `node .../server/main 8999`, then `robotAddr={ip: "localhost", port: 8999}` .



Robot Control Page

Robot-move command area

Application command area

Robot info area

- **sonarRobot** (after a GET) : name=sonarRobot, value=0
- **led** (after a GET) : name=LED for motion, value=false

COMMAND ANSWER (after a cmd): Welcome
ROBOT STATE (after a cmd): stopped
ROBOT SONAR (dynamic):
ENV SONAR (dynamic):

Description of the Robot Environment (localhost:8080/robotenv)

A click on the buttons `EXPLORE` and `HALT` will produce effects only if we activate an external component able to handle messages sent by the frontend via `MQTT`. An example of such a component is given in Subsection 3.6.5.

1.2.1 `M2M` interaction .

To check the behavior of the system from the point of view of *Machine-to-Machine* (`M2M`) interaction, we might use a tool like `curl` or `postman`. For example the command[4]:

```
----------------------------------------------------------------------------------------------------------------
curl -X GET http://localhost:8080/root -H "Accept: application/json" -H "Content-Type: application/json"
----------------------------------------------------------------------------------------------------------------
```

returns a set of metadata (see Subsection 3.1.1):

```
----------------------------------------------------------------------------------------------------------------
{
    "id": "http://localhost:8080",
    "name": "Unibo Robot front-end",
    "description": "A front-end for controlling a drr robot.",
    "customFields": {
        "hostname": "localhost",
        "port": 8080,
        "secure": false,
        "dataArraySize": 40
    },
    "help": {
        "link": "http://infolab.ingce.unibo.it/iss2018/it.unibo.issMaterial/issdocs/Material/LectureCesena1819.html",
        "title": "Documentation"
    }
}
----------------------------------------------------------------------------------------------------------------
```

The command:

```
----------------------------------------------------------------------------------------------------------------
curl -H "Content-Type: application/json" -X POST http://localhost:8080/commands/w
----------------------------------------------------------------------------------------------------------------
```

moves the robot forwards (see Subsection 3.4.7).

For examples of programs that interacts with the frontend, see Subsection 4.

[4] If you use Windows, the usage of `"` instead of `'` is mandatory.

2 Starting

Execute [5] the following steps[6]:

```
npm install -g express-generator

Create a folder frontend and open a terminal in this folder

Execute: express
Execute: npm install
```

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

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

2.1 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 `nodeExpressWeb.pdf` section `7.8` that can be summarized as follows:

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

var app    = express();

app.use( ... ) ;

app.get( ... ) ;

module.exports = app;
```

– 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 (seeSubsection 3.4.2).

[5] The command to update `npm` is `npm install -g npm`.
[6] Read section `7.8` of `nodeExpressWeb.pdf`. The code can be found in the project `it.unibo.mbot2018/nodeCode/robotFrontend`.

2.1.1 The start-up .

The code in the generated file bin/www creates a httpServer and passes app as the handler:

```
var app = require('../app');
var http = require('http');

/**
 * Get port from environment and store in Express.
 */
var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

/**
 * Create HTTP server.
 */
var server = http.createServer(app);

/**
 * Listen on provided port, on all network interfaces.
 */
server.listen(port);
server.on('error', onError);
server.on('listening', onListening);

...
```

2.2 Provisioning

The process.env global variable is *injected* by Node.js at runtime; it represents the state of the system environment the application is in when it starts. For example, if the system has a PATH variable set, this will be made accessible through process.env.PATH which we can use to check where binaries are located and make external calls to them if required.

It is considered good practice to always treat service dependencies as attached resources and define these using the environment. The act of providing environment variables is referred to as provisioning. We can either set the environment through application level logic, or we can use a tool to provision an environment for us. For example, the application level tool dotenv allows us to load environment variables from a file named .env.

```
npm install dotenv --save
require('dotenv').config(); //In the application file
```

While this is convenient for development needs, it is considered bad practice to couple an environment with our application, so keep it out by adding .env to your .gitignore file.

At the infrastructure level, we can use deployment manager tools like PM2, Docker Compose and Kubernetes to specify the environment.

2.3 Refactoring according to the MVC pattern

Now, let us execute the steps 1-3 of nodeExpressWeb.pdf-section 7.9:

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:
```

```javascript
//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;
```

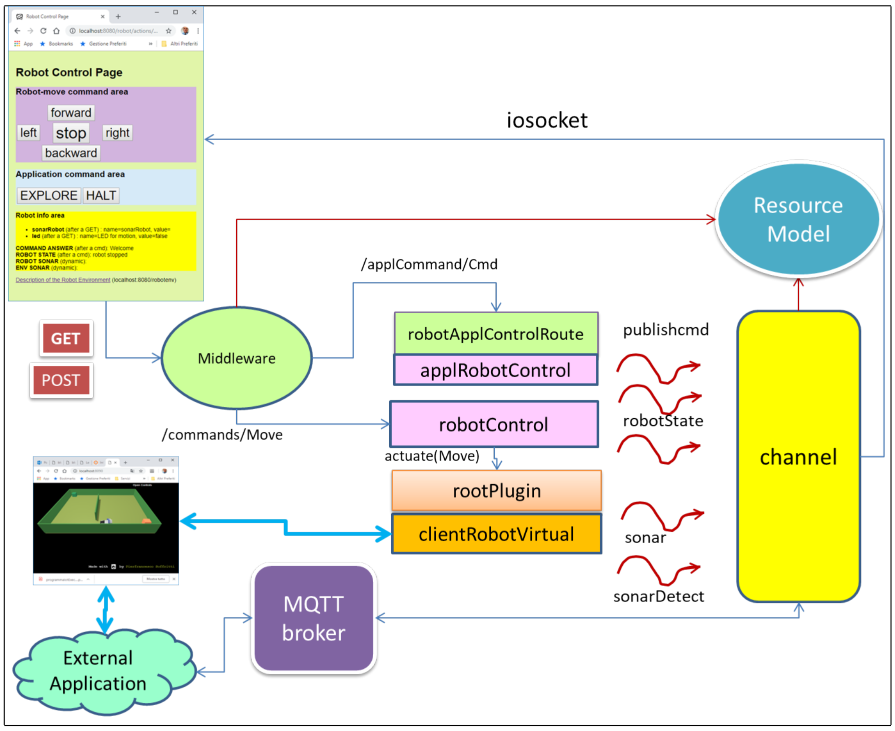
Listing 1.1. `app.js`

If we execute `node bin/www` and open a browser on http://localhost:3000/ all goes as before.

3 A model-centred prototype

In this section we introduce a new version of the server (see Subsection 3.3) that loads a `resource model` (see Subsection 3.1) and a new version of the application (see Subsection 3.4) that makes use of the `ejs` rendering engine, and does introduce a set of *routing rules* to the controllers.

Here is a picture of the systems we aim to build:



– At the centre of the new server system (see Subsection 3.3) there is a `resource model` (Subsection 3.1) that includes a model of the robot and of its environment.

– A `POST` request with `URL=/commands/Move (Move=w|s|a|d|h)` is managed by the *middleware* (a new version of `app.js`, see Subsection 3.4) that calls the utility `robotControl` (Subsection 3.6.1) to send (by using another utility (e.g.`clientRobotVirtual`, Subsection 3.6.2)) the `Move` command to the (virtual) robot.

– A `POST` request with `URL=/applCommand/Cmd (Cmd=explore|halt)` is *routed* by the *middleware* to the controller named `applRobotControl` (Subsection 3.6.3) that exploits an utility named `channel`[7] Subsection 3.7 to delegate (via `MQTT`) the execution of the request to an external application.

– Both `robotControl` and `applRobotControl` use `channel` to update the resource model and to update the set of connected Web pages (by using the `socket.io` library, Subsection 3.3.3) .

– The `channel` will also receive (via `MQTT`) data (e.g. the values of the sensors working on the robot) from the (virtual or physical) robot in order to update the set of connected Web pages with these data.

[7] The name of this utility should be changed to better capture the role of this component; for example a more appropriate name could be *observerUpdater*.

3.1 A resource model

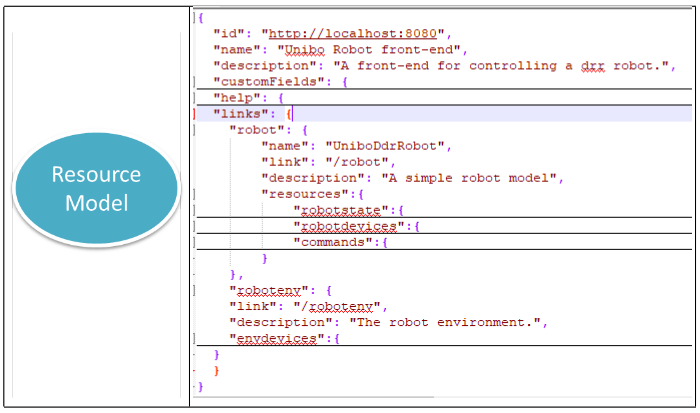
A Web Thing (WT) is a digital representation of a physical object, i.e. a Thing-accessible on the web. The cornerstone of the WT is to creating a model to describe physical Things with the right balance between expressiveness and usability.

To achieve the goal, every WT should be associated with:

– Model: metadata that defines various aspects of a WT, such as name, description, or configurations.
– Properties: represent the internal state of a WT. A property is a collection of data values that relate to some aspect of the WT. Properties can be modified through actions.
– Actions: functions offered by a WT (e.g. 'open','close','enable','move',...).

Actions represent the public interface of a web Thing while *properties* are the private parts. Limiting access to actions allows us to implement various control mechanisms for external requests such as access control, data validation, updating several properties atomically, and the like.

The standardization efforts [8] of the *Web of Things Interest Group* (http://www.w3.org/WoT/IG/) provide some suggestion for the definition of a resource model for our frontend system.



```
{
  "id": "http://localhost:8080",
  "name": "Unibo Robot front-end",
  "description": "A front-end for controlling a drr robot.",
  "customFields": {
  "help": {
  "links": {
    "robot": {
        "name": "UniboDdrRobot",
        "link": "/robot",
        "description": "A simple robot model",
        "resources":{
            "robotstate":{
            "robotdevices":{
            "commands":{
        }
    },
    "robotenv": {
    "link": "/robotenv",
    "description": "The robot environment.",
    "envdevices":{
  }
  }
}
```

Our resource model will be used as a formal representation for two main purposes:

– As a support to provide access to the main functionalities of our robot system.

– As a support to provide a description of our robot system in order to face a set of fundamental problems:
  1. How do we known where is our service, i.e. where to send request?
  2. How do we known what requests to send?
  3. How do we known the meaning of requests and of the responses we get?
  4. How do we can discover in automatic way our service by exploiting the HATEOAS principle (see Subsection 3.1.4) by exploiting web linking?

3.1.1 Basic information .

Let us start with a set of fields that could deliver useful data:

```
{
  "id": "http://localhost:8080",
  "name": "Unibo Robot front-end",
  "description": "A front-end for controlling a drr robot.",
  "customFields": {
    "hostname":"localhost",
    "port": 8080,
    "secure": false,
    "dataArraySize" : 40
```

[8] See for example the Web Thing Model in http://model.webofthings.io.

```
  },
 "help": {
   "link": "todo",
   "title": "Documentation"
 },
 "links": {
```

Listing 1.2. `appServer/models/robot.json`: useful data

These data will be part of the `metadata` we can send as a response to a `M2M` (machine to machine) `GET` request (see Subsection 1.2.1) or to a proper human request via a browser (see Subsection 3.5.2).

Afterwards, we can introduce a model for the robot and for the environment in which the robot will work.

3.1.2 A model for the robot .

In the model related to the robot we introduce fields that describe properties, structure and actions (commands to give to the robot).

```
 "links": {
   "robot": {
       "name": "UniboDdrRobot",
       "link": "/robot",
       "description": "A simple robot model",
       "resources":{
           "robotstate":{
               "link": "/robotstate",
               "state": "stopped"
           },
           "robotdevices":{
               "link": "/robotdevices",
               "resources":{
                   "sonarRobot": {
                       "name" : "sonarRobot",
                       "description": "The sonar on the front of the robot",
                       "value" : "0"
                    },
                   "led": {
                     "name": "LED for motion",
                     "description": "The LED on the robot ",
                     "value": false,
                     "gpio": 25
                   }
               }
           },
           "commands":{
               "link": "/commands",
               "description": "The set of robot moves",
               "resources":{
                   "moves": {
                       "w":{ "description": "Move the robot ahead" },
                       "s":{ "description": "Move the robot backward" },
                       "h":{ "description": "Stop the robot " },
                       "a":{ "description": "Move the robot left" },
                       "d":{ "description": "Move the robot right" }
                   }
               }
           }
       }
   },
```

Listing 1.3. `appServer/models/robot.json`: model for the robot

3.1.3 A model for the robot environment .

The model related to the robot environment describes a set of sensors (sonar, temprature) and actuators (a led):

```
   "robotenv": {
   "link": "/robotenv",
   "description": "The robot environment.",
   "envdevices":{
      "link": "/envdevices",
      "resources":{
         "sonar1": {
            "name" : "sonar1",
            "description": "The upper sonar.",
            "value": "0"
         },
         "sonar2": {
            "name" : "sonar2",
            "description": "The lower sonar.",
            "value": "0"
         },
         "temperature":{
            "name" : "temperature",
            "description": "An ambient temperature sensor.",
            "value": 0,
            "unit": "celsius",
            "gpio": 12
         },
        "led": {
          "name": "Hue Lamp",
          "description": "A REST lamp.",
          "value": false,
          "ip": "https://www2.meethue.com/it-it"
         }
      }
   }
  }
  }
}
```

Listing 1.4. `appServer/models/robot.json`: model for the environment

3.1.4 HATEOAS .

State changes within an application should happen by following `links`, which meets the `self-contained-messages` constraint.

A fully `self contained message` is a pure and complete representation of a specific event and can be published and archived as such. The message can - instantly and in future - be interpreted as the respective event without the need to rely on additional data stores that would need to be in time-sync with the event during message-processing.

Links enable clients to discover related resources, either by browsing in the case of a human user following links on pages, or by crawling in the case of a machine.

With `HATEOAS`, a client interacts with a network application whose application servers provide information dynamically through hypermedia.

Hypermedia As The Engine Of Application State (`HATEOAS`): in the `REST` (REpresentational State Transfer) application architecture, a client interacts with a network application whose application servers provide information dynamically through hypermedia (a medium of information that includes graphics, audio, video, plain text and `hyperlinks`). A `REST` client needs little to no prior knowledge about how to interact with an application or server beyond a generic understanding of hypermedia.

3.1.5 Using the model .

In the rest of this work, we will use the resource model in rather 'naive' way. In fact, its content is modified (and propagated to the Web page) by the *controllers* (Subsection 3.6) via the `channel` (Subsection 3.7). Moreover, part of the current state of the resource model is shown in the Web page by a function (`renderMainPage`, Subsection 3.4.6) put at the end of the middleware.

A better approach could be that of defining the resource model as an *observable* entity and define an observer that handles a change by activating a proper action, including (if it is the case) an action that updates the Web page via the iosocket.

3.2 Access to the model

An utility file can provide operations to support the access to the fields of the model. For example:

```
var resources = require('./robot.json');

exports.actionsLinkUrlStr = resources.links.robot.resources.commands.link;  // "/commands";
exports.stateLinkUrlStr = resources.links.robot.resources.robotstate.link;   // "/robotState";
exports.robotMoves = resources.links.robot.resources.commands.resources;

exports.getResourceModel = function(){ return resources; }

exports.getRobotState = function(){ return resources.links.robot.resources.robotstate.state; }

exports.extractFields = function(fields, object, target) {
      if(!target) var target = {};
      var arrayLength = fields.length;
      for (var i = 0; i < arrayLength; i++) {
        var field = fields[i];
        target[field] = object[field];
      }
      return target;
}
```

Listing 1.5. appServer/models/robot.js: access to the model

The frontend is able to return the model (see Subsection 3.5.2) to a human user or a machine, with the URL path: /showmodel:

```
---------------------------------------------------------------------------------------------------------------
http://localhost:8080/showmodel
curl -X GET http://localhost:8080/showmodel -H "Accept: application/json" -H "Content-Type: application/json"
---------------------------------------------------------------------------------------------------------------
```

3.3 The new server

The new version of the server is structured as the previous one (see Subsection 2.1). However, the code of the server is written in a user-defined file (named `frontendServerRobot.js`) and no more in `bin/www`. Moreover, we include the following additions:

- The declaration `require('dotenv').config()` for provisioning purposes.
- The usage of the `socketio` library.
- The introduction of a custom utility (`utils/channel.js`, see Subsection 3.7) to update the model and the Web page by emitting data over the `socket.io` and to publish data via `MQTT`.
- A reference to a new application file: `appFrontEndRobot.js`.
- The possibility to choose at the start-up to work with a physical or with a virtual robot.

3.3.1 The `socket.io` library .

> `socket.io` is a JavaScript library for realtime web applications. It enables realtime, bi-directional communication between web clients and servers. It has two parts: a client-side library that runs in the browser, and a server-side library for Node.js. Both components have a nearly identical `API`. Like Node.js, it is event-driven.

`socket.io` primarily uses the `WebSocket` protocol with polling as a fallback option, while providing the same interface. Although it can be used as simply a wrapper for `WebSocket`, it provides many more features, including broadcasting to multiple sockets, storing data associated with each client, and asynchronous `I/O`. `socket.io` requires using the library on both client and server side.

3.3.2 New server: the dependencies .

```
var app  = require('./appFrontEndRobot'); //the new application;
var debug = require('debug')('robotfrontend:server');
var http = require('http');

require('dotenv').config();

/**
 * Get port from environment and store in Express.
 */
var port = normalizePort( process.env.PORT || '3000' );
app.set('port', port);

/**
 * Create HTTP server.
 */
var server = http.createServer(app);

server.on('error', onError);
server.on('listening', onListening);
```

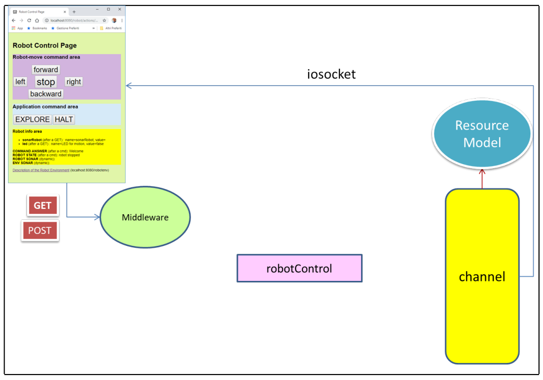
Listing 1.6. `frontendServerRobot.js`: configuration

3.3.3 New server: using socket.io .

A first extension upgrades our regular `HTTP` server into a `soketio`-server and does introduce our `channel` utility (see Subsection 3.7) and the `robotControl` (see Subsection 3.6.1):

```
const io           = require('socket.io').listen(server);
var echannel       = require("./appServer/utils/channel");
echannel.setIoSocket(io);
const robotControl = require('./appServer/controllers/robotControl');
```

Listing 1.7. `frontendServerRobot.js`



3.3.4 New server: start-up .

The next extension allows us to start the server with reference to a specific type of robot. Possible arguments are:

– `"virtual"`: for a virtual robot connected via TCP.
– `"mbot"`: for a robot connected with a `serial` line.
– `"unibo"`: for a physical robot connected via TCP.

If the user does not introduce any argument, the `"virtual"` option is set.

```
const systemConfig = require("./systemConfig");

const initMsg=
    "\n"+
    "-------------------------------------------------------\n"+
    "serverRobotCmd bound to port: "+ port + "\n" +
    "uses socket.io\n"+
    "USING THE ROBOT: " + systemConfig.getRobotType() + "\n"+
    "-------------------------------------------------------\n";
if( process.argv[2] ) systemConfig.setRobotType( process.argv[2] );
else systemConfig.setRobotType( "virtual" );
server.listen(port, function(){console.log(initMsg)});
```

Listing 1.8. `frontendServerRobot.js`: start up

3.3.5 Configuration file .

The configuration file provides the operation `setRobotType` that loads a proper `plugin` (see Subsection 3.6.2) to handle the specified type of robot.

```
const virtualRobot = "virtual";
const mbotlRobot = "mbot";
const uniboRobot = "uniboRobot";

var robotType    = virtualRobot;
var robotPlugin  = null;
var ledPlugin    = null;
```

```

exports.mqttbroker = "mqtt://localhost";
//exports.mqttbroker = "mqtt://iot.eclipse.org";

exports.robotAddr = {ip: "localhost", port: 8999};

exports.setRobotType = function( v ){
    if( v === virtualRobot || v === mbotlRobot || v === uniboRobot){
        robotType = v;
        setRobotPlugin();
        robotPlugin.start();
    }
}

exports.getRobotType = function(){ return robotType;  }
exports.getRobotPlugin = function(){ return robotPlugin; }

const isVirtual   = function(){ return robotType === virtualRobot ; }
const isMbot      = function(){ return robotType === mbotlRobot ; }
const isUnibo     = function(){ return robotType === uniboRobot ; }

const setRobotPlugin = function(){
    if( isVirtual() ) robotPlugin = require("./appServer/plugins/virtualRobot");
    if( isMbot() )  robotPlugin = require("./appServer/plugins/arduinioRobot");
    //console.log("\t robotConfig setRobotPlugin " + robotPlugin );
 }
const setLedPlugin = function(){
    ledPlugin = require("./appServer/plugins/ledPlugin");
}
```

Listing 1.9. `systemConfig.js`

3.3.6 New server: signal and failure handling .

The server defines also functions to handle `CRTL-C` signals and uncaught exceptions:

```
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;
  }
}

/**
 * Event listener for HTTP server "listening" event.
 */
function onListening() {
  var addr = server.address();
  var bind = typeof addr === 'string'
    ? 'pipe ' + addr
    : 'port ' + addr.port;
  debug('Listening on ' + bind);
}
/**
 * HANDLE User interruption commands.
 */
//Handle CRTL-C;
process.on('SIGINT', function () {
//  ledsPlugin.stop();
  console.log('serverRobot Bye, bye!');
  process.exit();
});
process.on('exit', function(code){
    console.log("Exiting code= " + code );
});
process.on('uncaughtException', function (err) {
    console.error('ERROR: serverRobot got uncaught exception:', err.message);
    process.exit(1);      //MANDATORY!!!;
});
```

Listing 1.10. `frontendServer.js`: handling signals and uncaught exceptions

3.4 The middleware

So far, the current application logic has been embedded in the file app.js. Now, let us introduce a new version of the application middleware in a file appFrontEndRobot.js (already in the the module dependencies - line 1 in Subsection 3.3.2):

3.4.1 Declarations .

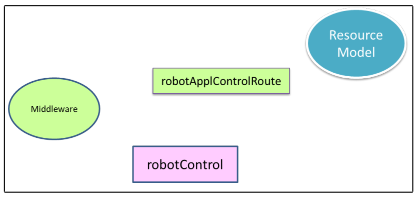
The first part of our new application middleware code sets the required resources:

```
var createError = require('http-errors');
var express     = require('express');
var path        = require('path');
var cookieParser = require('cookie-parser');
var logger      = require('morgan');
const cors      = require("cors");
const modelutils = require('./appServer/utils/modelUtils');
const routesGen = require('./appServer/routes/routesGenerator');
const app       = express();


//IMPORTANT POINT %%%%%%%%%%%%%%%%%%%%%
const modelInterface  = require('./appServer/models/robot');
const robotModel      = modelInterface.getResourceModel();
const robotControl    = require('./appServer/controllers/robotControl');

const routeInfo       = require('./appServer/routes/robotInfoRoute');
const robotApplRoute  = require('./appServer/routes/robotApplControlRoute');
//Other routes are generated by the routesGen.create (see later)
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
/*
 * Cross-origin resource sharing (CORS) is a mechanism that allows
 * restricted resources on a web page to be requested from another
 * domain outside the domain from which the first resource was served
 */
app.use( cors() );
```

Listing 1.11. appFrontEndRobot.js: starting



The application uses a resource model (named robotModel), a robot controller and some routes (see Subsection 3.5). Moreover, it uses the *cross-origin resource sharing* (cors) that allows requests to skip the "same origin policy" (to ensure that a site can't load any scripts from another domain) and access resources form remote hosts.

3.4.2 Static files .

```
app.use(express.static(path.join(__dirname, './appServer/public')));
```

Listing 1.12. appFrontEndRobot.js: static files

In the directory public, we include a specific logo, i.e. a favicon.ico.

3.4.3 View rendering engine .

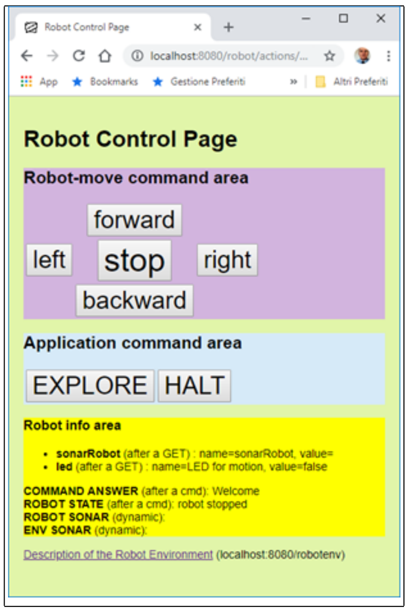
Afterwards, we introduce the usage of EJS[9] instead of Pug/Jade) as the rendering engine that combines data and a template to produce HTML.

[9] EJS is now replaced by donejs).

```
app.set('views', path.join(__dirname, './appServer', 'views'));
app.set("view engine", "ejs"); //npm install --save ejs
```

Listing 1.13. appFrontEndRobot.js: views

As an example, we suppose here to build a Web page that looks as follows:



Robot Control Page

Robot-move command area

forward
left stop right
backward

Application command area

EXPLORE HALT

Robot info area

- **sonarRobot** (after a GET) : name=sonarRobot, value=
- **led** (after a GET) : name=LED for motion, value=false

COMMAND ANSWER (after a cmd): Welcome
ROBOT STATE (after a cmd): robot stopped
ROBOT SONAR (dynamic):
ENV SONAR (dynamic):

Description of the Robot Environment (localhost:8080/robotenv)

In the Web page, we distinguish three main areas:

- An area (`Robot-move command area`) that provides buttons to move the robot.
- An area (`Application command area`) that provides buttons to start/stop an application.
- An area (`Robot info area`) that shows the current state of the robot, either after a `GET` request from the user or in a dynamic way from the application.

3.4.4 Showing the devices .

The last link in the Web page is generated by the following sentence in the `access.ejs` template:

```
-------------------------------------------------------------------------------------------
<a href="http://<%= refToEnv %>">Description  of the Robot Environment</a> (<%= refToEnv %>
-------------------------------------------------------------------------------------------
```

The variable `refToEnv` is set by a specific rule for the path `/envdevices`:

```
app.get('/envdevices', function (req, res) {
    req.myresult = JSON.stringify(
     modelInterface.envmodelToResources(robotModel.links.robotenv.envdevices.resources )
    );
    if (req.accepts('html')){ res.render('showdevs',
        {'title': 'Environment Devices', 'res': req.myresult, "where":"in the environment",
        'model':robotModel.links.robotenv.envdevices, 'host': req.headers.host }
    )} else if (req.accepts('application/json')) { res.send(req.myresult);}
     else res.send("please specify better");
});
```

Listing 1.14. appFrontEndRobot.js: rule for path /envdevices

This rule builds the answer according to the type of the caller. If the request specifies `application/json`, the answer is the part of the model that regards the environment. Otherwise, If the request specifies `html`, the answer is a Web page built by the `showdevs.ejs` template. Thus, the command:

```
-----------------------------------------------------------------------------------------------------------------
curl -X GET http://localhost:8080/robotdevices -H "Accept: application/json" -H "Content-Type: application/json"
-----------------------------------------------------------------------------------------------------------------
```

returns:

```
-----------------------------------------------------------------------------------------------------------------
[
    {
        "id": "sonar1",
        "name": "sonar1",
        "value": "0",
        "values": []
    },
    {
        "id": "sonar2",
        "name": "sonar2",
        "value": "",
        "values": []
    },
    {
        "id": "temperature",
        "name": "temperature",
        "value": 0,
        "values": []
    },
    {
        "id": "led",
        "name": "Hue Lamp",
        "value": false,
        "values": []
    }
]
-----------------------------------------------------------------------------------------------------------------
```

Note that the field named `values` is not part of the model definition introduced in Subsection 3.1.2. In fact, this field is dynamically introduced by a *route generator* (see Subsection 3.5.2) with the goal to store the `history of the values` taken by a device, while the field `value` stores its current value.

The history of the devices is updated by the `channel` introduced in Subsection 3.7.

A similar rule is introduced to get/show the devices on the robot:

```
app.get('/robotdevices', function (req, res) {
    req.myresult = JSON.stringify(
     modelInterface.envmodelToResources(robotModel.links.robot.resources.robotdevices.resources)
    );
    if (req.accepts('html')){ res.render('showdevs',
        {'title': 'Robot Devices', 'res': req.myresult, "where":"on the robot",
        'model': robotModel.links.robot.resources.robotdevices, 'host': req.headers.host }
    )} else if (req.accepts('application/json')) { res.send(req.myresult);}
     else res.send("please specify better");
});
```

Listing 1.15. `appFrontEndRobot.js`: rule for path `/robotdevices`

The command

```
-----------------------------------------------------------------------------------------------------------------
curl -X GET http://localhost:8080/robot -H "Accept: application/json" -H "Content-Type: application/json"
-----------------------------------------------------------------------------------------------------------------
```

could return

```
-----------------------------------------------------------------------------------------------------------------
[
    {
        "id": "sonarRobot",
        "name": "sonarRobot",
        "value": "collision:wallDown",
        "values": [
            "collision:wallDown",
            "collision:wallUp",
            "collision:wallUp",
            "collision:wallDown"
        ]
    },
    {
        "id": "led",
        "name": "LED for motion",
        "value": false,
        "values": []
    }
]
--------------------------------------------------------------------------------------------------------------------
```

In this case we can see the history of the vales taken by the `sonarRobot` on the virtual robot.

3.4.5 The template page .

Let us report here another snippet of the `access.ejs` template, related to the `Robot info` area:

```
<div style="background-color: #FFFF00">
<h3>Robot info area</h3>
<ul>
<% Object.keys(model.resources.robotdevices.resources).forEach( function( dev ) { %>
   <li><b><%= dev %></b> (after a GET) :
       name=<%= model.resources.robotdevices.resources[dev].name %>,
       value=<%= model.resources.robotdevices.resources[dev].value %></li>
 <% } );%>
</ul>
    <div>
        <b>COMMAND ANSWER</b> (after a cmd): <span id="displayCmd"><%= res %></span>
    </div>
    <div>
        <b>ROBOT STATE</b> (after a cmd): <span id="displayAnsw"><%= robotstate %></span>
    </div>
    <div>
        <b>ROBOT SONAR</b> (dynamic): <span id="displaySonar"></span>
    </div>
    <div>
        <b>ENV SONAR</b> (dynamic): <span id="envSonar"></span>
    </div>
 </div>
```

Listing 1.16. `robotFrontend/appServer/views/access.ejs`: a snippet

3.4.6 The renderMainPage .

The last action of the middleware chain calls a function that builds the response to the request:

```
app.use( function(req,res,next){
    console.log("last use - req.myresult=" + req.myresult );
    renderResponse( req, robotModel.links.robot.resources.robotstate.state, res );
} );
```

Listing 1.17. `appFrontEndRobot.js`: last action

The response rendering function builds the answer according to the content type of the request (`html`, `json`, etc.[10]):

```
var renderResponse = function(req, robotstate, res){
    if (req.accepts('html') &&
        checkContentType(req, 'application/x-www-form-urlencoded') ) { //from a web page button
         // Check if there's a custom renderer for this media type and resource;
        if (req.type) res.render(req.type, {'title': 'Resource Model', 'req': req});
        else{
            console.info('\t appFrontEndRobot renderResponse state=' + robotstate );
            res.render('access',
```

[10] `MessagePack` can be used to exchange information in efficient binary format, instead of strings.

```
                {'title': 'Robot Control Page', 'res': "", 'model': robotModel.links.robot,
                'robotstate': robotstate, 'refToEnv': req.headers.host+"/envdevices"});
        }
    }else if ( req.accepts('application/json') && checkContentType(req, 'application/json') ) {
        //console.log('\t renderResponse: JSON representation! ' );
        res.send(req.myresult);
    }else if (req.accepts('application/x-msgpack') && checkContentType(req,'application/x-msgpack')){
         res.send("Sorry, application/x-msgpack todo ... ");
    }else{
         console.info("Defaulting to Html representation! " );
    }
}

checkContentType = function ( req, ctype ){
    var contype = req.headers['content-type'];
    //console.info('\t appFrontEndRobot checkContentType req.type=' + req.type + " content=" + contype);
    if (contype == undefined ) return true ; //html is first
    return ( contype.indexOf(ctype) >= 0)
}
```

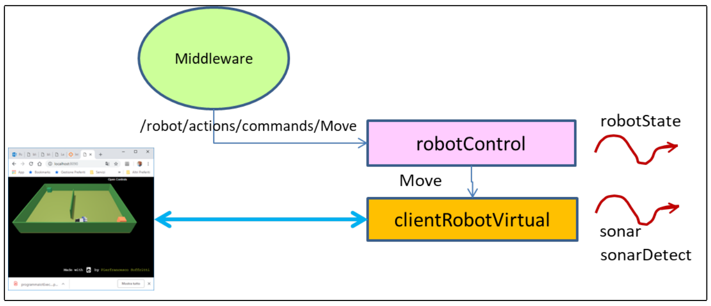
Listing 1.18. appFrontEndRobot.js: rendering

Note that, in case of a html request, the page is built by the access.ejs template, unless there is a value in the field req.type; in this case the page is built by a template with name req.type (the case is related to the dynamic creation of routing rules introduced in Subsection 3.5.2).

In the case of a application/json request, the answer is the value set in the req.myresult field. For an example, see Subsection 3.5.2.

3.4.7 Handling robot move-commands .

The commands sent by using the buttons in the Robot-move command area (URL= /commands/Move) are handled by a set of ad-hoc app.post rules:



```
app.post("/commands/w", function(req, res, next) {
    robotControl.actuate("w", req, res ); next();});
app.post("/commands/s", function(req, res, next) {
    robotControl.actuate("s", req, res ); next();});
app.post("/commands/h", function(req, res, next) {
    robotControl.actuate("h", req, res ); next(); });
app.post("/commands/d", function(req, res, next) {
    robotControl.actuate("d", req, res ); next(); });
app.post("/commands/a", function(req, res, next) {
    robotControl.actuate("a", req, res ); next(); });
```

Listing 1.19. appFrontEndRobot.js: robot move commands

The command:

```
------------------------------------------------------------------------------------------------------------
curl -H "Content-Type: application/json" -X POST http://localhost:8080/commands/w
------------------------------------------------------------------------------------------------------------
```

puts in execution the robotControl described in Subsection 3.6.1 that moves the robot forwards.

3.4.8 Promoting M2M interaction .

In a scenario of a M2M interaction, a program could ask to the server the set of robot moves that is is able to handle. To this end, w introduce a proper rule:

```
app.get( modelInterface.actionsLinkUrlStr, function( req, res ) {
   res.send( modelInterface.robotMoves );
});
```

Listing 1.20. `appFrontEndRobot.js`: promote M2M interaction

In this rule, we use the interface to the model introduce in Subsection 3.2.

3.5 Routing

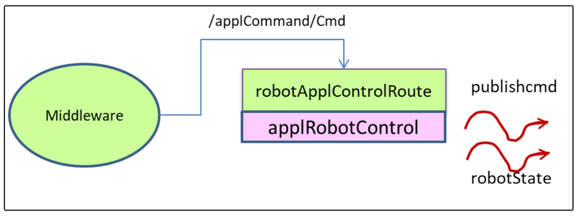
As an example of routing rules, we introduce first a rule related to the handling of application commands and next a generator of routing rules, based on the model of Subsection 3.1.

3.5.1 Handling application commands .

A URL starting with `/applCommand/Cmd` (`Cmd=explore|halt`) is routed to the `robotApplControlRoute` that in its turn calls the `actuate` action of the `applRobotControl` controller (Subsection 3.6.3)).

```
//Application command route
app.use('/applCommand', robotApplRoute);
```

Listing 1.21. `appFrontEndRobot.js`: routing rules



```
var express   = require('express'),
  router      = express.Router() ;

const robotControl = require('../controllers/applRobotControl');

router.post("/explore", function(req, res, next) {
    robotControl.actuate("explore", req, res );
    next();
});
router.post("/halt", function(req, res, next) {
    robotControl.actuate("halt", req, res );
    next();
});

module.exports = router;
```

Listing 1.22. `robotApplControlRoute.js`

The `robotControl` is described in Subsection 3.6.3.

3.5.2 Generation of routing rules .

In this section, we will follow the schema of routing rules generation shown in the book *Building the Web of Things*.

```
//Create Other Routes
app.use('/', routesGen.create(robotModel) );
```

Listing 1.23. `appFrontEndRobot.js`: routesGen

In order to give an example of a rule generator, let us generate routing rules for the following URLs:

- `/root`: operation *createRootRoute* that returns a list of metadata;
- `/showmodel`: operation *createModelRoutes* that returns the string representation of the JSON model.

```
var express = require('express'),
  router   = express.Router();
const modelutils = require('./../utils/modelUtils');

exports.create = function (model) {
//Extend the model with data
createDefaultData(model.links.robot.resources.robotdevices.resources);
createDefaultData(model.links.robotenv.envdevices.resources);

  // Let's create the routes
  createRootRoute(model);
  createModelRoutes(model);
//  createActionsRoutes(model);
  return router;
};

function createDefaultData(resources) {
    Object.keys(resources).forEach(function (resKey) {
        var resource = resources[resKey];
        resource.data = [];
    });
}

function createRootRoute(model) {
      router.route('/root').get(function (req, res, next) {
        req.model = model;
        req.type = 'robotInfo';

        var fields = ['id', 'name', 'description', 'customFields', 'help'];
        req.myresult = modelutils.extractFields(fields, model);

        //CREATE the Link header containing links to the other resources (HATEOAS);see the header
        res.links({
            robot: model.links.robot.link,
            commands: model.links.robot.resources.commands.link
        });
//      next();
        if (req.accepts('html')) {
            res.render(req.type,
                    {'title': 'Robot Metadata', 'res': "", 'model': model,'host': req.headers.host});
        }  else if (req.accepts('application/json')) res.send( req.myresult );
      });
};

function createModelRoutes(model) {
      // GET /showmodel
      router.route('/showmodel').get(function (req, res, next) {
        req.myresult = model;
        req.model    = model;
        req.modelStr = "\""+JSON.stringify(model, null, 2)+"\"";
        req.type  = 'showmodel';
        type = 'http://model.webofthings.io/';
        res.links({
          type: type
        });

        next();
      });
};
```

Listing 1.24. `routesGenerator.js`

3.5.3 Dynamic extensions of the model .

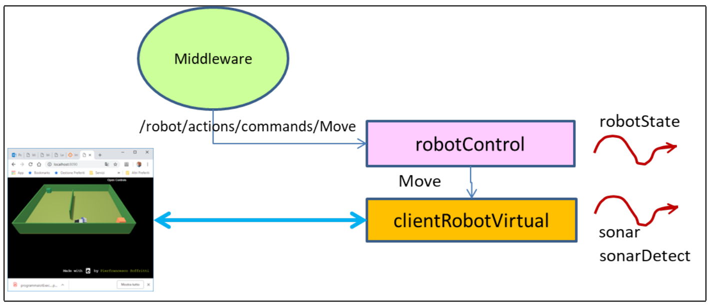
The lines `createDefaultData(model.links. ...)` in `routesGenerator.js` dynamically extend the model with new fields related to the state of devices.

New data entries can be added to the model each time a resource changes its state. In this way we can keep track of the history of changes. For an example, see Subsection 3.4.4.

3.6 The controllers

We introduce two controllers:

– `robotControl.js` (Subsection 3.6.1) : this controller performs basic robot moves. It is directly called by the middleware:

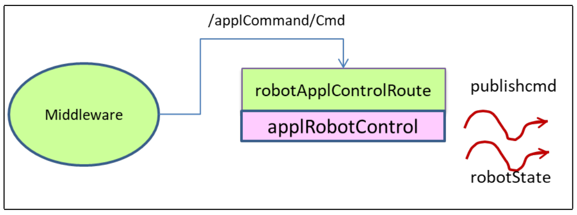


```
app.get( modelInterface.actionsLinkUrlStr, function( req, res ) {
    res.send( modelInterface.robotMoves );
});

/*
* ---------------------------------------------------------------
* 3b) HANDLE A POST REQUEST
* ---------------------------------------------------------------
*/
app.post("/commands/w", function(req, res, next) {
```

Listing 1.25. `appFrontEndRobot.js`: call `robotControl`

– `applRobotControl.js` (Subsection 3.6.3) : this controller delegates application actions at external services. It is called by the middleware via routing:



```
const robotApplRoute  = require('./appServer/routes/robotApplControlRoute');
...
app.use('/applCommand', robotApplRoute);
```

3.6.1 The robotControl .

The `robotControl` controller 'actuates' the command `/commands/Move` (`Move=w|s|a|d|h`) by calling a proper plugin according to the type of the robot.

```
const systemConfig = require("./../../systemConfig");
const echannel    = require("./../utils/channel");

exports.actuate = function( cmd, req, res ){
    console.log("\t robotControl actuate " + cmd );
    var newState    = "";
    if(      cmd === "w" ){ newState="robot moving forward"; }
    else if( cmd === "s" ){ newState="robot moving backward"; }
    else if( cmd === "h" ){ newState="robot stopped"; }
    else if( cmd === "a" ){ newState="robot moving left"; }
    else if( cmd === "d" ){ newState="robot moving right"; }
    else { console.log("\t robotControl actuate cmd unknown " ); return; } //cmd unknown
    var robotPlugin = systemConfig.getRobotPlugin();
    robotPlugin.actuate( cmd );
    echannel.emit("robotState", newState); //used to update on the Web page
    req.myresult = "move "+cmd+ " done"; //used by LAST ACTION in appFrontEndRobot
}
```

Listing 1.26. robotControl.js

3.6.2 The robot plugins .

For each type of robot, we introduce a plugin that defines and exports two operations:

- start(): this operations loads the proper support and executes possible initialization actions;
- actuate(cmd); this operation actuates a 'standard' robot-command move.

For example, the plugin for the virtual robot is:

```
var toVirtualRobot = require("./../utils/clientRobotVirtual");

exports.start = function( ){
    console.log("\t plugins/virtualRobot started with toVirtualRobot= " + toVirtualRobot );
}
exports.actuate = function( cmd ){
    var cmdToVirtual = "";
    //console.log("\t robotControl actuate " + cmd );
    if(     cmd === "w" ){ cmdToVirtual=`{ "type": "moveForward", "arg": -1 }`; }
    else if( cmd === "s" ){ cmdToVirtual=`{ "type": "moveBackward", "arg": -1 }` ; }
    else if( cmd === "h" ){ cmdToVirtual=`{ "type": "alarm", "arg": 1000 }` ; }
    else if( cmd === "a" ){ cmdToVirtual=`{ "type": "turnLeft", "arg": 1000 }` ; }
    else if( cmd === "d" ){ cmdToVirtual=`{ "type": "turnRight", "arg": 1000 }` ; }
    //console.log("\t plugins/virtualRobot actuateOnVirtual:" + cmdToVirtual );
    toVirtualRobot.send( cmdToVirtual );
}
```

Listing 1.27. appServer/plugins/virtualRobot.js

The file utils/clientRobotVirtual.js defines the support to interact with the virtual environment.

3.6.3 The applRobotControl .

The applRobotControl controller 'actuates' the command /applCommand/Cmd (Cmd=explore|halt) by delegating it to an external application:

```
const echannel  = require("./../utils/channel");

exports.actuate = function( cmd, req, res ){
    console.log("\t applRobotControl actuate " + cmd );
    if( cmd === "w" ){ delegate("w(low)", "moving forward", req,res); }
    else if( cmd === "s" ){ delegate("s(low)", "moving backward", req,res); }
    else if( cmd === "h" ){ delegate("h(low)", "stopped", req,res); }
    else if( cmd === "a" ){ delegate("w(low)", "moving left", req,res); }
    else if( cmd === "d" ){ delegate("w(low)", "moving right", req,res); }
    //Application
    else if( cmd === "explore" ){ delegate("explore", "robot working at application level", req,res); }
    else if( cmd === "halt" ){ delegate("halt", "robot halting the application level", req,res); }
}
```

Listing 1.28. applRobotControl.js: command handling

3.6.4 Delegating .

The 'delegation' of the execution of the command first updates the state of the robot in the resource model and then sends a message to an external entity. These action are performed by the utility channel (see Subsection 3.7), by raising proper events:

```
var delegate = function ( hlcmd, newState, req, res ){
   echannel.emit("robotState", newState);
   var eventstr = "msg(usercmd,event,js,none,usercmd( " + hlcmd + "),1)"
   //console.log("\t robotControl emits: "+ eventstr);
   echannel.emit("publishcmd", eventstr);
}
```

Listing 1.29. `applRobotControl.js`: delegate

The payload of the published message has the form:

```
msg(usercmd,event,js,none,usercmd( MOVE ) with MOVE=w|s|a|d|h
```

Thus, this message has the same internal representaion of a `QActor` event declared as:

```
Event usercmd : usercmd( MOVE )
```

3.6.5 An external component .

As an example of an external component able to handle the messages published by the frontend server, let us show a first 'prototype' of a possible application written in the `QActor` language:

```
/*
 * ==============================================================
 * demoRobotInWenv.qa
 * ==============================================================
 */
System demoRobotInWenv
Dispatch obstacle : obstacle(TARGET)
Dispatch robotCmd : cmd(X)               //X=w|a|s|d|h
Dispatch startAppl : startAppl(X)
Dispatch haltAppl : haltAppl(X)

Event sonarDetect : obstacle(X)  //From wenv-robotsonar-by clientTcpForVirtualRobot
Event sonar      : sonar(SONAR, TARGET, DISTANCE) //From wenv sonar
Event usercmd    : usercmd(X)   //from robot GUI; X=robotgui(CMD) CMD=s(low)
Dispatch userControl : cmd(X) //from console

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

Context ctxRobotInWenv  ip [ host="localhost" port=8028 ] //-httpserver

QActor cmdrobotconverter context ctxRobotInWenv -pubsub{
   State init normal [ ]
   transition stopAfter 600000
   whenEvent usercmd -> handleUserCmd
   finally repeatPlan

   State handleUserCmd resumeLastPlan[//MAPPING AFTER EXPERIMENTATION
       onEvent usercmd : usercmd( robotgui(h(X)) ) ->
              forward robotplayer -m robotCmd : cmd("h");
       onEvent usercmd : usercmd( robotgui(w(X)) ) ->
              forward robotplayer -m robotCmd : cmd("w");
       onEvent usercmd : usercmd( robotgui(a(X)) ) ->
              forward robotplayer -m robotCmd : cmd("a");
       onEvent usercmd : usercmd( robotgui(d(X)) ) ->
              forward robotplayer -m robotCmd : cmd("d");
       onEvent usercmd : usercmd( robotgui(s(X)) ) ->
              forward robotplayer -m robotCmd : cmd("s");

//Application     (see applRobotControl)
       onEvent usercmd : usercmd( explore ) ->
```

```
                forward mind -m startAppl : startAppl(ok);
        onEvent usercmd : usercmd( halt ) ->
                forward mind -m haltAppl : haltAppl(ok)
    ]
}

 QActor robotplayer context ctxRobotInWenv{
    State init normal [
        println("robotplayer STARTS") ;
        javaRun it.unibo.robotVirtual.basicRobotExecutor.setUp("localhost")
    ]
    switchTo doWork

    State doWork[
    ]
    transition stopAfter 600000
        whenMsg robotCmd   -> execMove
        finally repeatPlan

    State execMove resumeLastPlan[
        printCurrentMessage;
        onMsg robotCmd : cmd(MOVE) -> javaRun it.unibo.robotVirtual.basicRobotExecutor.doMove( MOVE )
    ]
 }

/*
 * APPLICATION level
 */
QActor mind context ctxRobotInWenv { //sensible to sonar events emitted by it.unibo.robotVirtual.basicRobotExecutor

    State init normal [
        println("mind WAITS")
    ]
    transition stopAfter 600000
        whenMsg startAppl -> startAppl

    State startAppl [
        println("mind START APPLICATION ") ;
        forward robotplayer -m robotCmd : cmd("a");
        delay 500;
        forward robotplayer -m robotCmd : cmd("d")
    ]
    switchTo doWork

    State doWork[
        //println("mind now just waits for sonars or for HALT ")
    ]
    transition stopAfter 600000
        whenMsg haltAppl     -> haltAppl,
        whenEvent sonar      -> handleSonar,
        whenEvent sonarDetect -> handleSonar
    finally repeatPlan

    State handleSonar resumeLastPlan [
        printCurrentEvent;
        onEvent sonarDetect : obstacle(X) ->
            publishEvent "unibo/qasys" -e sonarDetect : obstacle(X);

        onEvent sonar : sonar(SONAR, TARGET, DISTANCE) ->
            publishEvent "unibo/qasys" -e sonar : sonar(SONAR, TARGET, DISTANCE)
    ]

    State haltAppl resumeLastPlan[
        println("mind HALT APPLICATION")
    ]
    switchTo init

}
```

Listing 1.30. `demoRobotInWenv.qa`: external component

This system is composed of three actors:

- the `cmdrobotconverter` that transforms an event `usercmd:usercmd(MOVE)` into a dispatch `robotCmd` to the actor `robotplayer` or into a dispatch (`startAppl` or `haltAppl`) to the actor `mind`;
- the `robotplayer` that executes a given `robotCmd`;
- the `mind` that embeds the application logic.

This application has been built before the development of the frontend server. Thus, it introduces a `player` that directly interacts with the virtual robot. Since the control of the robot is 'subtracted' to the frontend, the `mind` publishes (line `102`) the event `sonar : sonar(SONAR, TARGET, DISTANCE)` on the `MQTT` broker in order to allow the server to update the Web page.

In a better approach[11], the player should send robot move messages to the frontend, so that any robot move (and consequent updating of the robot state on the Web page) is always performed by the frontend server.

The definition of this new version of the external component is left to the reader. We recall that the `QActor` support provides built-in actor methods (`sendRestPut` and `sendRestGet`) that implement `REST` requests to a given `HTTP` server, using a component like that introduced in Subsection 4.2.

[11] Logically better, but less efficient, since it increments the number of messages required to move the robot.

3.7 The channel

The utility named `channel`[12] is a Node `EventEmitter`.

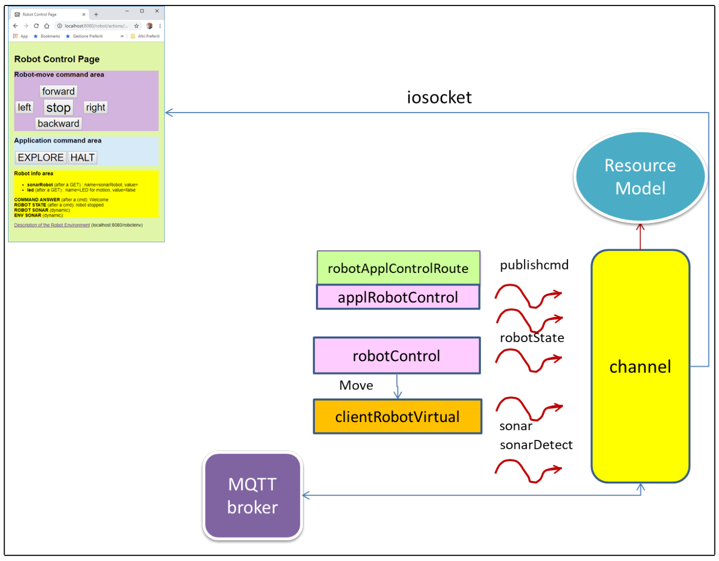
In Node, all objects that emit events are members of `EventEmitter` class. These objects expose an `eventEmitter.on()` function that allows one or more functions to be attached to named events emitted by the object.

- When the `EventEmitter` object emits an event, all of the functions attached to that specific event are called *synchronously* (in the order in which they are registered or attached to the event object).
- All values returned by the called listeners are ignored and will be discarded.
- The listener can be invoked only once using `eventEmitter.once()` method, while every time the event is emitted using `eventEmitter.on()` method.

If we want to break the synchronous flow and switch to *asynchronous* mode, then we can use `setImmediate()` or `process.nextTick()` methods.

The main goal of `channel` is to handle a set of events raised by the internal components of the system in order to update the Web page or to publish a command to the external world. More specifically, `channel` handles:

- the event named `'robotState'` raised by a controller when it change the motion state of the robot;
- the events named `'sonarDetect'` and `'sonar'` raised by the utilities that interact with the (physical or virtual) robot when they receive a new value from the sonar on the robot or from the robot environment;
- the event named `'publishcmd'` raised by the *applRobotControl* (see Subsection 3.6.3) to delegate the execution of a command to an external (non-JavaScript) component.



```
const events    = require('events');
const model     = require('./../models/robot.json');
const io        = require('socket.io');
const channel   = new events.EventEmitter();
const mqttUtils = null;
```

[12] The name of this utility should be changed to better capture the role of this component. For example a more appropriate name could be *observerUpdater*.

```

channel.setIoSocket = function( iosock ){
    console.log("\t CHANNEL setIoSocket=" + iosock );
    this.io = iosock;
}
channel.on('sonar', function(data) { //emitted by clientRobotVirtual or mqtt support;
    if( data.indexOf('undefined') >= 0 ) return;
    if( data.indexOf('msg') >= 0 ) return;
    console.log("\t CHANNEL sonar updates the model and the page with:" + data );
    model.links.robotenv.envdevices.resources.sonar2.value=data;
    //model.links.robotenv.envdevices.resources.sonar2.data.push(data); //history (quite long...)
    this.io.sockets.send( data );
});
channel.on('sonarDetect', function(data) { //emitted by clientRobotVirtual or mqtt support;
    console.log("\t CHANNEL sonarDetect updates the model and the page with:" + data );
    model.links.robot.resources.robotdevices.resources.sonarRobot.value=data;
    model.links.robot.resources.robotdevices.resources.sonarRobot.data.push(data); //history
    this.io.sockets.send( data );
});
channel.on('robotState', function(data) { //emitted by robotControl or applRobotControl;
    console.log("\t CHANNEL receives: " + data );
    model.links.robot.resources.robotstate.state=data; //shown in the page by app renderMainPage
    //not propagated via io.sockets since a robot state can change only after a user command (??)
    //CLEAR THE sonar
    model.links.robotenv.envdevices.resources.sonar2.value="";
    this.io.sockets.send( data );
    model.links.robot.resources.robotdevices.resources.sonarRobot.value="";
    this.io.sockets.send( data );
});
channel.on('publishcmd', function(data) { //emitted by robotControl;
    console.log("\t CHANNEL publishcmd: " + data + " on topic unibo/qasys" + " mqttUtils=" + mqttUtils);
//  if( mqttUtils == null ) mqttUtils = require('./../utils/mqttUtils');  //CIRCULAR!!!
    publish( data );  //topic = "unibo/qasys";
```

Listing 1.31. `channel.js`: event handling

3.7.1 Using `MQTT` .

Within the `channel` we include also utility code for interactions based on `MQTT`:

```
 * MQTT support
 * ------------------------------------------------
 */
const systemConfig = require("./../../systemConfig");
const mqtt   = require ('mqtt');
const topic  = "unibo/qasys";
var client   = mqtt.connect(systemConfig.mqttbroker);

client.on('connect', function () {
      client.subscribe( topic );
      console.log('\t MQTT client has subscribed successfully ');
});

//The message usually arrives as buffer, so we convert it to string data type;
client.on('message', function (topic, message){
    var msg = message.toString();
    console.log("\t MQTT RECEIVES:"+ msg); //if toString is not given, the message comes as buffer;
    if( msg.indexOf( "sonarDetect" ) > 0 ){
       channel.emit("sonarDetect", msg ); //to allow update of the WebPage
    }else if( msg.indexOf( "sonar" ) > 0 ){
       channel.emit("sonar", msg ); //to allow update of the WebPage
    }
});

publish = function( msg ){
    //console.log('\t MQTT publish ' + client);
    client.publish(topic, msg);
}
```

Listing 1.32. `channel.js`: `MQTT` support

4 Working with the server

As said in Subsection 1.2.1 the robot frontend server can be used both by humans and by machines. In this section we give examples of clients written in JavaScript and in Java.

4.1 A client written in JavaScript

A client written in JavaScript could be:

```
'use strict';
var request = require('request'); //npm install --save request;

var doGet = function(path){
    const optionsGetRoot = {
            method: 'GET',
            url: 'http://localhost:8080/'+path, //;
            headers: {
                'Content-Type': 'application/json',
                 'accept': 'application/json'
            }
        };
        request(optionsGetRoot, function(error, response, body){
            if (!error && response.statusCode == 200) {
                console.log(body);
            }
        } );
}

var doMove = function(move){
    var optionsMove = {
        method: 'POST',
        url: 'http://localhost:8080/commands/'+move, //;
        headers: { "Content-Type": "application/json" }
    };
    request(optionsMove, function(error, response, body){
        if (!error && response.statusCode == 200) {
            //console.log("done: " + move);
        }
    } );
}
//Get the metadata;
setTimeout( function(){ doGet("root") }, 100);
//Do some move;
setTimeout( function(){ doMove("w"); } , 400);
setTimeout( function(){ doMove("s"); } , 1000);
setTimeout( function(){ doMove("h"); } , 2000);
//Get the robotdevices;
setTimeout( function(){ doGet("robotdevices") }, 2300);
//Get the full model;
//setTimeout( function(){ doGet("showmodel") }, 3000);
```

Listing 1.33. robotFrontend/clients/aClient.js

This client first gets the metadata and then moves the robot. At the end, the client gets from the server the state of the robot devices, that could keep track of possible obstacles detected by the robot.
The output is:

```
{"id":"http://localhost:8080","name":"Unibo Robot front-end","description":...}
[{"id":"sonarRobot","name":"sonarRobot","value":"","values":["collision:wallDown"]},{"id":"led","name":"LED
for motion","value":false,"values":[]}]
```

4.2 A client written in Java

A client written in Java could be:

```java
public class ClientRestHttp {
private CloseableHttpClient httpclient;

    public ClientRestHttp( ) {
        httpclient = HttpClients.createDefault();
    }

    public String sendGet( String url ) {
      try {
//      System.out.println("\t ClientRestHttp sendGet url=" + url);
        HttpGet httpGet = new HttpGet(url);
        httpGet.addHeader("accept", "application/json");
        return sendRequest(httpGet);
      } catch ( Exception e) {
          e.printStackTrace();
          return "sendGet error";
     }
    }//sendGet

    public String sendPost( String data, String url ) {
        HttpPost request   = new HttpPost(url);
        if( data.length() > 0 ) {
            StringEntity params = new StringEntity(data,"UTF-8");
            params.setContentType("application/json");
            request.setEntity(params);
        }
        request.addHeader("Content-type", "application/json");
        request.addHeader("Accept", "*/*");
        request.addHeader("Accept-Encoding", "gzip,deflate,sdch");
        request.addHeader("Accept-Language", "en-US,en;q=0.8");
        return sendRequest( request );
    }//sendPost

    public String sendPut( String data, String url ) {
        HttpPut request   = new HttpPut(url);
        if( data.length() > 0 ) {
            StringEntity params = new StringEntity(data,"UTF-8");
            params.setContentType("application/json");
            request.setEntity(params);
        }
        request.addHeader("Content-type", "application/json");
        request.addHeader("Accept", "*/*");
        request.addHeader("Accept-Encoding", "gzip,deflate,sdch");
        request.addHeader("Accept-Language", "en-US,en;q=0.8");
        return sendRequest( request );
    }//sendPut

    protected String sendRequest( HttpRequestBase request ) {
        try {
            CloseableHttpResponse response = httpclient.execute(request);
            return handleResponse(response);
        } catch (ClientProtocolException e) { e.printStackTrace();
        } catch (IOException e) { e.printStackTrace(); }
        return "sendRequest errro";
    }//sendRequest

    protected String handleResponse(CloseableHttpResponse response) {
        int responseCode = response.getStatusLine().getStatusCode();
        String info = "handleResponse error responseCode=" + responseCode;
        try {
    //      System.out.println("\t ClientRestHttp responseCode " + responseCode);
            if (responseCode == 200 || responseCode == 204) {
                BufferedReader br = new BufferedReader(
                        new InputStreamReader((response.getEntity().getContent())));
                String data;
                info = "";
                while ((data = br.readLine()) != null) {
                    info = info + data;
                    //System.out.println(data);
                }
//         System.out.println(info);
        }
    }catch (Exception ex) {
 //        httpclient.close();
    }
    return info;
}//handleResponse


 public static void main (String args[]) throws Exception{
    System.out.println("============================================================");
    System.out.println("1) Activate a MQTT server on hostAddr:1883");
    System.out.println("2) Run node frontendServerRobot.js ");
    System.out.println("============================================================");
    String serverurl = "http://localhost:8080";
    ClientRestHttp client = new ClientRestHttp( );
    String response;

    System.out.println("----------- SECTION 1 ---------------");
    System.out.println("GET THE META DATA");
    response = client.sendGet(serverurl+"/root");
    System.out.println(response);
    System.out.println("GET THE SET OF ROBOT MOVES");
    response = client.sendGet(serverurl+"/commands");
    System.out.println(response);
    JSONObject reader = new JSONObject(response);

    System.out.println("----------- SECTION 2 ---------------");
    System.out.println("DISCOVER THE ROBOT MOVES");
    JSONObject moves = reader.getJSONObject("moves");
    System.out.println( moves.names() );
    //Consumer<String> consumer= v -> System.out.println(" Move:"+ v);
    moves.names().forEach( v -> System.out.println( v + ":" + moves.getJSONObject(""+v) ) );

    System.out.println("----------- SECTION 3 ---------------");
    System.out.println("MOVE THE ROBOT");
    response = client.sendPost("",serverurl+"/commands/w");
    System.out.println(response);
    Thread.sleep(1500);
    response = client.sendPost("",serverurl+"/commands/s");
    System.out.println(response);
    Thread.sleep(1500);
    response = client.sendPost("",serverurl+"/commands/h");
    System.out.println(response);
 }
}
```

Listing 1.34. `ClientRestHttp.java`

In this example:

– The `SECTION 1` executes `GET` requests that will accept `"application/json"` data in order to acquire information from the server, i.e. metadata and the set of moves that the robot can perform (see Subsection 3.4.8).
– The `SECTION 2` analyses the robot moves to get the meaning of each of them.
– The `SECTION 3` executes `POST` requests to move the robot.

The output is:

```
============================================================
1) Activate a MQTT server on  hostAddr:1883
2) Run node frontendServerRobot.js
============================================================
----------- SECTION 1 ---------------
GET THE META DATA
{"id":"http://localhost:8080","name":"Unibo Robot front-end",...}
GET THE SET OF ROBOT MOVES
{"moves":{"w":{"description":"Move the robot ahead"},...}}}
----------- SECTION 2 ---------------
DISCOVER THE ROBOT MOVES
["a","s","d","w","h"]
```

```
a:{"description":"Move the robot left"}
s:{"description":"Move the robot backward"}
d:{"description":"Move the robot right"}
w:{"description":"Move the robot ahead"}
h:{"description":"Stop the robot "}
----------- SECTION 3 ---------------
MOVE THE ROBOT
move  w done
move  s done
move  h done
```