NodeExpressWeb

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 explore different ways to design and build servers (and applications) with Node.js, JavaScript technology. We will follow an incremental approach, starting from very simple and 'naive' solutions to more evolute systems, involving the MVC patterns, the *Express* frameworks, data persistence (with MongoDB), RESTful interfaces and *microservices*. The logical flow can be summarized as follows:

1. A **naive data server** (Subsection 2.1): a Node HTTP server that allows us to write and read (volatile) application data with poor attention to software structuring aspects. The basic server is extended in Subsection 2.1 to return HTML pages that embed a POST request.
2. A **server that reads/writes a file** (Section 3): the server handles GET, PUT and POST verbs, reads JSON data on a file (users.txt) in synchronous way and writes on it asynchronously. Data can be sent in JSON form or as strings sent by a browser. Subsection 3.3 shows how to interact with the server by using a program that sends data in both the forms.
3. Towards a **server that reads/writes on a database** (Section 4). Since application data are usually stored in a database rather than in files, we introduce basic notions of the MongoDB database in Subsection 4.1, while section Subsection 4.2 is devoted to introduce the Mongoose framework that provides a schema-based solution to model persistent application data as discussed in Subsection 4.3. Subsection 4.5 show how - with Mongoose - all document creation and retrieval from the database can be handled by models. This gives us the opportunity to structure our software system with reference to the MVC pattern.
4. The MVC **pattern**: in Section 5) we introduce the main concepts of MVC and its 'variants' (MVVM, MVVMC, MVCVM, inSubsection 5.1). A new version of the server that handles requests according to the MVC pattern (by delegating the work to proper read/write controllers) is shown in Subsection 5.4
5. A RESTful **interface to data** (Section 6): define a REST API to interact with our database through HTTP calls and perform the common CRUD functions. In Subsection 6.3 we introduce a server that reacts to the requests path /api/user by giving CRUD semantics to the HTTP verbs POST, GET,PUT,DELETE. This server accepts data in JSON format only. A tester is shown in Subsection 6.5.
6. A RESTful **server** (Subsection 2.1): a CRUD server that uses the REST API to the database in order to avoid any reference to concrete data representation.
7. **The** Express **framework** (Section 7): introduction to the main features of express (middleware, routing,extensions and views) that will promote the further refactoring of our server. This section shows also how to start from a generated working environment (Subsection 7.8) to a working environment with an explicit folder for the controllers (Subsection 7.9).
8. An **express-based** CRUD **server** (Section 8): redesign the CRUD server of Subsection 6.6 by using express in the MVC style of Subsection 7.9.
9. **Publish/Subscribe** with socket.io (Section 9): besides the *request-response* pattern, we need a publish/subscribe (pub/sub) model to allow further decoupling between data consumers (*subscribers*) and data producers (*publishers*). The socket.io library is introduced in Subsection 9.2, together with some examples of its usage.
10. **Microservices**: the technical stuff introduced so far can be the basic support for the design and development of microservices, introduced in Section 11 together with a preliminary example (Subsection 11.4)

2 Node Http

Let us introduce a Node HTTP server that allows us to write and read (volatile) application data.

2.1 A naive data-server

The server is written here in a 'naive way', with a elementary attention to structuring aspects:

```
/*
* ====================================
* nodejs/HttpServerRequest.js
* ====================================
*/
var utils = require('../utils');
var http = require('http');

var applData = []; //Application data;

var handleRequest = function (request, response) { //request has many info;
    console.log('HttpServerNodeRequest REQUEST METHID=', request.method);
    console.log('HttpServerNodeRequest REQUEST URL=', request.url);
    if( request.url==="/" ) {
        var outS = "Hello from HttpServerRequest. Current data:\n";
        outS = outS + utils.readData(applData);
        response.end( outS );
        return;
    }
    if( request.url==="/read" ) {
        if( applData.length == 0 ) response.end( "Sorry, no data" );
        else response.end( utils.readData(applData) );
        return;
    }
    if( request.url==="/add" ) {
        utils.addData(applData);
        response.end( utils.readData(applData) );
        return;
    }
    response.end( "Sorry, I don't undederstand " );
};


function main(){
//configure the system;
    console.log('HttpServerRequest cretae a servere and register handleRequest');
    var server = http.createServer();
    server.on( 'request' , handleRequest);
//start;
    console.log('HttpServerNodeRequest running on 3000/');
    server.listen(3000, function() {
        console.log('bound to port 3000');
    });
}

main();
```

Listing 1.1. *HttpServerRequest.js*

The `listen` method tells the `HTTP` server object to begin listening for connections on the given port. Node **doesn't block** waiting for the connection to be established; rather, it will 'react' to connections. In fact, when a connection is established, a `'request'` event is emitted, and the callback function `handleRequest` is called.[1]

The user can use this system by opening a browser and heading over

[1] When processing a web request from a browser, that browser may send more than one request. For instance, a browser may also send a second request, looking for a `favicon.ico`.

– to localhost://3000/read to read application data;
– to localhost://3000/add to add application data.

In both cases, we use here the HTTP GET verb, while a POST or a PUT should be more appropriate when adding data. We will return on this point later (see Subsection 3.1 and Section 6).

2.2 The http module

Node servers created by the http.createServer method are long-running processes that serve many requests throughout their lifetimes.

For every HTTP request, the request callback function will be invoked with new request and response objects. Prior to the callback being triggered, Node will parse the request up through the HTTP headers and provide them as part of the request. But Node doesn't start parsing the body of the request until the callback has been fired.

Node will not automatically write any response back to the client; it's our responsibility to end the response using the response() method. In this way we can run any asynchronous logic during the lifetime of the request before ending the response. If the server fail to end the response, the request will hang until the client times out or it will just remain open.

2.3 Handling application data

In the naive server of Subsection 2.1, application data are represented by the array appData, initially empty. The application data elements are 'users' represented as JSON objects. For example:

```
{name: 'name1', age: '11', password: 'pwd1'}
```

Application data are handled by functions defined into an utility file:

```
exports.readData = function(dataArray){
    var s = "";
    dataArray.forEach( function(el,index){
        s = s + JSON.stringify(el) +"\n";
    });
    return s;
}
exports.addData = function(dataArray){
    dataArray.push( buildNewData(dataArray.length + 1) );
}
buildNewData = function(n){
    var item = { user:"name"+n, age:n+10, pswd:"pwd"+n };
    return item;
}
```

Listing 1.2. *utils.js*: application data handling

The utility file includes also code that read/write data from/on files.

```
var fs = require('fs');

exports.readSynchArrayDataFromFile = function( fName ){
    var lines = fs.readFileSync(fName, 'utf-8')
    .split('\n')
    .filter(Boolean); //lines is ana arry of stringds;
    var data = [] ; //an array of json objects;
    lines.forEach( function(el,index) {
        data.push( JSON.parse(el) );
    });
    return data;
}
```

```
exports.readeDataFromFile = function(fName, cb){
    fs.readFile(fName, 'utf8',
        function(err, data) {
            if (err){
                return 'Could not find or open file for reading';
            }else{
                cb( data) ;
            }
    });
};
exports.writeArrayDataOnFile = function(fName, data, cb ){
    var s = "";
    data.forEach( function(el,index){
        s = s + JSON.stringify(el) +"\n";
    });
    fs.writeFile(fName, s, 'utf8',
            function(err) {
                if (err) throw err;
                else cb( "Now the file has " + data.length + " elements " ) ;
            });
};
```

Listing 1.3. *utils.js*: data on files

Note that data are always written in a file in asynchronous way, while they can be read also by using a synchronous method.

Finally, we include also code that 'reacts' to uncaught exceptions.

```
process.on('uncaughtException', function (err) {
    console.error('ERROR: got uncaught exception:', err.message);
    process.exit(1);      //MANDATORY!!!
});
process.on('exit', function(code){
    console.log("Exiting code= " + code );
});
```

Listing 1.4. *utils.js*: exception handling

2.4 A conventional server then returns pages

Many web applications usually serve static files, e.g. `html`, `css`, `js`, `jpg`, etc.

A basic `HTTP` server for serving static assets can be defined as follows:

```
/*
* ====================================
* nodejs/HttpBasicServer.js
* ====================================
*/
var http   = require("http");
var fs     = require('fs');
var parse  = require('url').parse;
var join   = require('path').join;
var srvUtil = require("./ServerUtils");

var root = __dirname; //set by Node to the directory path to the file;

var handleRequest = function (request, response) {
    var url = parse(request.url);
    console.log("HttpBasicServer request.method=" + request.method + " url.pathname=" + url.pathname);
    //console.log('HttpBasicServer root=', root);
    if( url.pathname === "/" ) url.pathname = "/index.html";
    var path = join(root, url.pathname);
    console.log('HttpBasicServer path=', path);
    srvUtil.renderStaticFile(path,response);
 };

 function main(){
    //configure the system;
        console.log('HttpServerRequest cretae a servere and register handleRequest');
```

```
27          var server = http.createServer();
28          server.on( 'request' , handleRequest);
29      //start;
30          console.log('HttpServerNodeRequest running on 3000/');
31          server.listen(3000, function() {
32              console.log('bound to port 3000');
33          });
34  };
35  main();
```

Listing 1.5. *HttpBasicServer.js*

The callback `handleRequest` now *i)* extracts relevant information from the request and *ii)* checks for the existence of the requested file. Afterwords, it delegates the work to other functions, defined in the utility file `ServerUtils.js` (see Subsection 2.5).

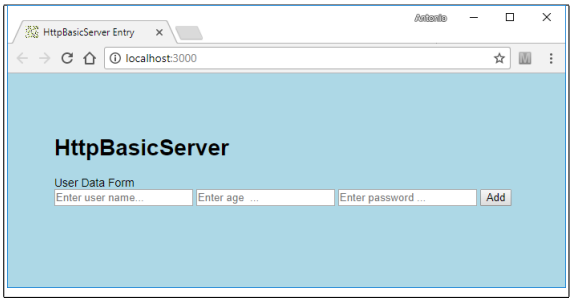
If no file is requested, the server looks at the default file `index.html` (line 18) that can be defined as follows:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset="utf-8">
5           <title>HttpBasicServer Entry</title>
6           <link rel="stylesheet" href="style.css">
7       </head>
8
9       <body>
10          <h1>HttpBasicServer</h1>
11          <form method="post" action="/adduser">
12              <label>User Data Form</label><br>
13              <input type="text" name="name" placeholder="Enter user name..." required>
14              <input type="text" name="age" placeholder="Enter age ..." required>
15              <input type="text" name="password" placeholder="Enter password ..." required>
16              <input type="submit" value="Add">
17          </form>
18      </body>
19  </html>
```

Listing 1.6. *index.html*

This page includes a form that allows users to send (via `POST`) Strings formatted as in the example hereunder:

```
1   name=Bob&age=33&password=pswdBob
```



The user can use this system by opening a browser and heading over:

- to `localhost://3000/` for the main page `index.html`;
- to `localhost://3000/users.txt` for the current list of users.

At the moment, if we fill the form and click on the `Add` button, we will receive the answer:

```
1   Sorry, file Not Found C:\aLab2016\lab2014Bo\it.unibo.nodejs.web.intro\code\nodejs\adduser
```

In fact, there is no explicit handling of the `POST` request sent by the browser. We will face this aspect in Section 3.

2.5 ServerUtils

In the utility file `ServerUtils.js` we include:

– the function `renderStaticFile`, to render a static file:

```
/*
* ====================================
* nodejs/ServerUtils.js
* ====================================
*/
var fs = require('fs');

exports.renderStaticFile = function(path,response){
    try{
        fs.stat(path, function(err, stat){ //stat provides information about the file;
            if (err) {
                renderFileError(err, path, response);
            } else {
                if( stat.size == 0 ){
                    response.statusCode = 500;
                    response.end('Attempt to acces to an empty or non-existing file ' );
                }else{
                    //readStaticFile(path,response);
                    readStaticFilePiping(path,response);
                }

            }
        });
    }catch(exception){
        response.statusCode = 400;
        response.end('ERROR ' + exception);
    }
};
```

Listing 1.7. *ServerUtils.js*: renderStaticFile

– the function `renderFileError`, in the case that a file does not exists:

```
var renderFileError = function(err, path, response){
    if ('ENOENT' == err.code) {
        response.statusCode = 404;
        response.end('Sorry, file Not Found ' + path );
    } else {
        response.statusCode = 500;
        response.end('Internal Server Error ' + err);
    }
}
```

Listing 1.8. *ServerUtils.js*: renderFileError

– the function `readStaticFile`, to read a file and return its content:

```
var readStaticFile = function(path, response){
    var stream = fs.createReadStream(path);
    //console.log('ServerUtils renderStaticFile path=', path);
    stream.on('data', function(chunk){
        response.write(chunk);
    });
    stream.on('error', function(err){
        response.statusCode = 500;
        response.end('Internal Server Error ' + err);
        /*
         * If we have no index.html file and omit test at line 14, we have:
         * Internal Server Error Error: EISDIR: illegal operation on a directory, read
         */
    });
    stream.on('end', function(){
        response.end();
    });
}
```

Listing 1.9. *ServerUtils.js*: readStaticFile

2.6 Streams and Piping

The function `readStaticFilePiping` performs file reading by using the higher-level mechanism of `Stream#pipe`:

```
var readStaticFilePiping = function(path, response){
    var stream = fs.createReadStream(path);
    //console.log('ServerUtils renderStaticFilePiping path=', path);
    stream.pipe(response);
    stream.on('error', function(err){
        response.statusCode = 500;
        response.end('Internal Server Error ' + err);
    });
}
```

Listing 1.10. *ServerUtils.js*: readStaticFilePiping

A `Stream` is an EventEmitter that implements some specials methods. Depending on the methods implemented, a Stream becomes *Readable*, *Writable*, or *Duplex*. *Readable* streams let you read data from a source while *writable* streams let you write data to a destination. The `request` of the Node `HTTP` module is a readable stream and `response` is a writable stream.

`Piping` allows us to read data from the source and write to destination without managing the flow. Since the method `pipe()` returns the destination stream, we can easily utilize this to chain multiple streams together. For example:

```
var fs   = require('fs');
var zlib = require('zlib');

fs.createReadStream('input.txt.gz')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('output.txt'));
```

First, we create a simple readable stream from the file `input.txt.gz`. Next, we pipe this stream into another stream `zlib.createGunzip()` to un-gzip the content. Lastly, as streams can be chained, we add a writable stream in order to write the un-gzipped content to the file.

3 A Server that manages data on a file

The code of an HTTP server can be structured in many ways. Let us show a compact version for a server that allows us to to read stored data and to store new data ('users' as in Subsection 2.3) given by JSON strings or by data sent via an HTML form.

```
1  /*
2  * ====================================
3  * ndejs/HttpServerFile.js
4  * ====================================
5  */
6
7  var utils  = require('../utils');
8  var http   = require('http');
9  var fs     = require('fs');
10 var parse  = require('url').parse;
11 var join   = require('path').join;
12 var srvUtil = require("./ServerUtils");
13
14 var root = __dirname; //set by Node to the directory path to the file;
15
16 http.createServer(function (request, response) {
17     var method = request.method;
18     var url   = parse(request.url);
19     var path  = url.pathname;
20     console.log('HttpServerFile method=', method);
21     console.log('HttpServerFile path =', path);
22     if( path === "/" ) path = "/index.html";
23     var fpath = join(root, path);
24     if(method === 'GET' && path === '/users' ){
25         srvUtil.renderStaticFile( join(root, "/users.txt"),response);
26         return;
27     };
28     if(method === 'GET' ){ //return a file;
29         srvUtil.renderStaticFile(fpath,response);
30         return;
31     };
32     if(method === 'POST' && path === '/adduser' ) {
33         getDataToAdd(request, response, addDataInFile);
34         return;
35     }
36     if(method === 'PUT' && path === '/adduser' ) {
37         getDataToAdd( request, response, addDataInFile );
38         return;
39     }
40     response.end( "Sorry, I don't understand" );
```

Listing 1.11. *HttpServerFile.js*

Note that we are reusing the utility functions introduced in Subsection 2.5. The data to be added must be acquired by handling the `event 'data'`. This task has been delegated (line 33 and 37) to the function `getDataToAdd`:

```
1
2  /*
3   * Data are acquired by handling the event 'data'
4   * When data are all available, the given callback is called
5   */
6  var getDataToAdd = function( request, response, callback ){
7      var inData = '';
8      request.on('data', function (data) { //data is of type Buffer;
9          inData += data;
10         // Too much data, kill the connection!;
11         if (inData.length > 1e6) request.connection.destroy();
12     });
13     request.on('end', function () { //data are all available;
14         var jsonData = cvtDataToJson(inData);
15         callback( JSON.stringify(jsonData), response );
16       });
```

Listing 1.12. *HttpServerFile.js*: getDataToAdd

When data are all available, the given callback is called, to update the file. The data that represent a new user are string with two possible formats:

- JSON representations, e.g. `'"name": "Ann", "age": "35", "password": "pswdAnn"'`
- strings sent by a browser, e.g. `"name=Alice&age=25&password=pswdAlice"`

The task to handle these different possible representations is delegated to the function cvtDataToJson:

```
1
2  function cvtDataToJson(inData){
3     //console.log( "cvtDataToJson "+ inData );
4     var jsonData;
5     try{ //assume data already in JSON form;
6         jsonData = JSON.parse( inData );
7     }catch(exception){ //assume data from browser;
8         jsonData = cvtPostStringToJson(inData);
9     }
10    //console.log( jsonData );
11    return jsonData;
12 }
13
14 function cvtPostStringToJson(inData){
15    //console.log( "cvtPostStringToJson:" + inData );
16    var jsonData     = {};
17    var rawdata      = inData.split('&');
18    jsonData.name    = rawdata[0].split('=')[1] ;
19    jsonData.age     = rawdata[1].split('=')[1] ;
20    jsonData.password = rawdata[2].split('=')[1] ;
21    return jsonData;
```

Listing 1.13. *HttpServerFile.js*: cvtDataToJson

The callback for getDataToAdd that stores data in a file can be defined as follows:

```
1
2  /*
3   * A callback for getDataToAdd. Read old data is sycnh, write is asynch.
4   */
5  var addDataInFile = function(jsonDataStr, response){
6     //console.log( "addDataInFile " + jsonDataStr );
7     var curData = fs.readFileSync('./users.txt', 'utf-8');
8     if( curData.length > 0 ){
9         curData = curData + "\n";
10    }
11    curData = curData + jsonDataStr;
12    fs.writeFile('users.txt', curData, 'utf8',
13        function(err) {
14            if (err) throw err;
15            srvUtil.renderStaticFile( join(root, "/users.txt"),response);
16            //response.end("data added");
17        } );
```

Listing 1.14. *HttpServerFile.js*: addDataInFile

The current content of the file is read in synchronous way, while the new data are written, at the end of reading, in asynchronous way. When the write is terminated, the current content of the file is rendered as answer.

Note that, in order to insert new data, the user must use either the PUT or the POST verbs.

3.1 About PUT, PATCH and POST

The HTTP methods POST and PUT aren't the HTTP equivalent of the CRUD's *create* and *update*.

Use PUT when you want to modify a singular resource which is already a part of resources collection. PUT replaces the resource in its entirety. PUT is idempotent, so you can cache the response.

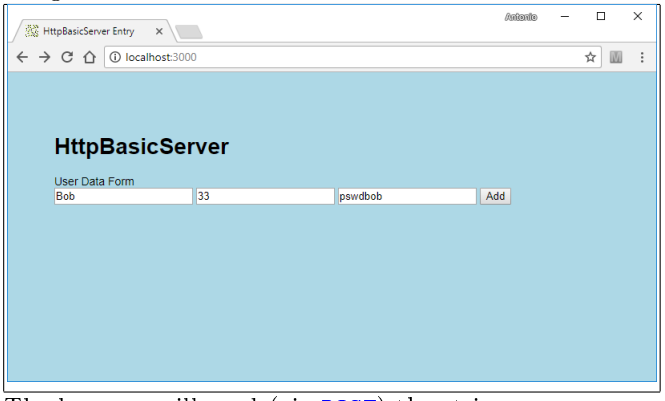
Use PATCH if request updates part of a resource.

Use POST when you want to add a child resource under resources collection. POST is NOT idempotent. So if you retry the request N times, you will end up having N resources with N different URIs created on server.

3.2 Testing PUT/POST

To test the system, we can

1. Use a browser. Since the file `index.html` is the same of Subsection 2.4, let us fill now the given form with some data and click on the `Add` button.



The browser will send (via POST) the string:

```
name=Bob&age=33&password=pswdBob
```

2. Use a program that sends JSON Strings or browser-like strings:

```
/*
 * nodejs/userAddRequest.js
 */

var request = require("request");
require('../utils');

var urlAdd = 'http://localhost:3000/add';
var options = { method: 'PUT', url: urlAdd,
  headers:
   { 'Cache-Control': 'no-cache',
     'Content-Type': 'application/json' },
  body: { name: 'Jack', age: '57', password: 'pswdJack' },
  json: true
};

request(options, function (error, response, body) {
  if (error) throw new Error(error);
  console.log("ANSWER from server:"+body);
});

console.log("userAddRequest START");
```

Listing 1.15. *userAddRequest.js*

3. Work with tools like `curl` or `postman` sending JSON Strings or browser-like strings

```
curl -X PUT -d "{\"name\": \"Alice\",\"age\": \"25\",\"password\": \"pp\"}"
     http://localhost:3000/adduser

curl -X POST -d "name=Alice&age=25&password=pswdAlice" http://localhost:3000/adduser

curl -X PUT ^
-d  "{\"name\": \"Alice\",\"age\": \"25\",\"password\": \"pswdAlice\"}" ^
http://localhost:3000/adduser
```

3.3 Machine to machine

To facilitate the interaction of programs with the server, let us introduce two functions to perform `read/write` requests:

```
/*
 * nodejs/serverUsage.js
 */
require('../utils');
var request = require("request");

var urlReadAdd = 'http://localhost:3000/users';
readData = function(cbflow){
    var options = { method: 'GET', url: urlReadAdd };
    request(options, function (error, response, body) {
      if (error) throw new Error(error);
      console.log(" --- serverUsage readData:" );
      console.log(body);
      cbflow( null, 'read' );
    });
};

var urlAdd = 'http://localhost:3000/adduser';
addJsonData = function(jsonData, cbflow){
    var options = { method: 'PUT', url: urlAdd,
           body: jsonData,
           json: true};
    request(options, function (error, response, body) {
        if (error) throw new Error(error);
        console.log(" --- serverUsage addJsonData:" );
        console.log( body);
        cbflow( null, 'writeJson' );
    });
};

addPostLikeData = function(stringData, cbflow){
    var options = { method: 'POST', url: urlAdd,
           body: stringData,
           json: false};
    request(options, function (error, response, body) {
        if (error) throw new Error(error);
        console.log(" --- serverUsage addPostLikeData:" );
        console.log( body);
        cbflow( null, 'writePostLike' );
    });
};
```

Listing 1.16. *serverUsage.js*

Each action is performer in asynchronous way. The (callback) argument `cbflow` is introduced to force serial flow control among these actions by means of the `async` library:

```
var async  = require('async');
//sequential flow;
var sequenceCall = function(){
    async.series([
           function(cbflow){ readData( cbflow ); },
           function(cbflow){ addJsonData({name: 'Alice', age: '22', password: 'pswdAlice' }, cbflow );},
           function(cbflow){ addPostLikeData( "name=Dave&age=34&password=pswdDave", cbflow );},
           function(cbflow){ addJsonData({name: 'Bob', age: '25', password: 'pswdBob' }, cbflow );},
```

```
9             function(cbflow){ readData( cbflow ); }
10        ],
11        // optional callback;
12        function(err, results) {
13            console.log("result=" + results);
14        });
15  };
16
17  sequenceCall();
```

Listing 1.17. *serverUsage.js*

TODO: the sane program in Java

4 Using databases

It is quite common that application data are stored in a database rather than in files.

4.1 Mongo

MongoDB is a document (non relational) database that stores documents as BSON (Binary JSON. i.e. *Binary JavaScript Serialized Object Notation*). MongoDB isn't a transactional database, and shouldn't be used as such.

While MongoDB is useful to store and handle documents, *data modelling* is supported by Mongoose, a popular ODM (*Object data modeller*) for MongoDB in Node.js, that provides a straight-forward, schema-based solutions. It includes built-in type casting, validation, query building, business logic hooks and more (see Subsection 4.3).

For our experiments, we will use a MongoDB server via a docker image. Some common commands are summarized hereunder:

```
docker run --name natmongo -d -p 27017:27017 mongo --noauth --bind_ip=0.0.0.0
docker start natmongo (if not ruuning in ps)
docker exec -it natmongo bin/bash
mongo
show dbs
use ...
db.Users.find()
db.dropDatabase()
```

Let us introduce here an example of a MongoDB client at work. First of all let us define an operation to connect our application to a db named "mongoNaive" and to execute the application code when the connection is set:

```
var MongoClient = require('mongodb').MongoClient;
require('../utils.js');

var dbase ;
var client ;
var dbUrl = "mongodb://localhost:27017";

var init = function( application){
    console.log("init " );
    //Connect to the db from version 3.0;
    MongoClient.connect(dbUrl, function (err, curclient) {
        if (err) throw err;
        console.log("connected " );
        client = curclient;
        dbase  = client.db('mongoNaive'); //the db used;
        application();
    });
}
```

Listing 1.18. *mongoclient.js*: init

The main task of the init phase is to set the variables dbase and client. Now we define a function that writes data on our db :

```
var itemsNum = 0;
var writeData = function( ){
    dbase.collection('users', function (err, coll) {
        if( err ) throw err;
        itemsNum = itemsNum + 1;
        var name = 'Name' + itemsNum;
        var age = itemsNum*2;
        var pswd = 'pwd' + itemsNum;
        coll.insert( {  name: name, age: age, password: pswd } );
        console.log('writeData done name=' + name + " age:" + age+ " pswd=" + pswd);
    });
```

```
12 }//writeData;
```

Listing 1.19. *mongoclient.js*: write data

A function that reads data from our `db` can be introduced as follows:

```
1  var curData;
2  var readData = function( ){
3      dbase.collection( 'users' , function (err, data) {
4          if (err) throw err;
5          curData = data.find( {} ); //The result for the query is actually a cursor object;
6       });
7  };//readData;
8
9  var showCurData = function( ){
10     curData.each( function(err, item) {
11         if(err)       throw err;
12         if(item==null) return;
13         console.log( item.name + " age:" + item.age + " pswd=" + item.password );
14     });
15 }
```

Listing 1.20. *mongoclient.js*: read data

Finally, we define our application code, that simply reads/writes data:

```
1  var applicationJob = function( ){
2      console.log("application starts " );
3      dbase.collection( 'users' ).count() //a promise;
4      .then( function(value) {
5          itemsNum = value;
6          console.log("itemsNum=" + itemsNum);
7      })
8      .then( function(){ console.log("initially we have " + itemsNum + " users"); } )
9      .then( writeData )
10     .then( writeData )
11     .then( function(){ console.log("at the end we have " + itemsNum + " users"); } )
12     .then( readData )
13     .then( showCurData )
14     .then( function(){
15         console.log("==================== ")
16         client.close();
17     });
18 }
19
20 init( applicationJob );
```

Listing 1.21. *mongoclient.js*: application

If we work with an empty data base, the output is (remember that `readData` is executed in asynchronous way):

```
1  init
2  connected
3  application starts
4  itemsNum=0
5  initially we have 0 users
6  writeData done name=Name1 age:2 pswd=pwd1
7  writeData done name=Name2 age:4 pswd=pwd2
8  at the end we have 2 users
9  ====================
10 Name1 age:2 pswd=pwd1
11 Name2 age:4 pswd=pwd2
12 Exiting code= 0
```

4.2 Mongoose

An *Object Data Model* ("ODM") represents data as JavaScript objects, which are then mapped to the underlying database. The benefit of using an ODM is that programmers can continue to think in terms of objects rather than database semantics. Thus, using ODM often results in lower costs for software development and maintenance.

From the documentation we read that *"Mongoose provides a straight-forward, schema-based solution to modelling your application data and includes built-in type casting, validation, query building, business logic hooks and more, out of the box"*.

Mongoose[2] opens a pool of five[3] reusable connections when it connects to a MongoDB database. This pool of connections is shared between all requests. Mongoose will publish events, based on the status of the connection. Thus, let us introduce an utility file:

```
var mongoose = require( 'mongoose' );
mongoose.Promise = global.Promise;

var dbURI = 'mongodb://localhost:27017/mongoNaive';
if (process.env.NODE_ENV === 'production') {dbURI = process.env.MONGOLAB_URI;}

mongoose.connect( dbURI,{useMongoClient: true}); //option required from 4.11.0;
```

Listing 1.22. *utilsMongoose.js*:connection

When used in a applications, this file opens a connection to our specific database. Opening and closing connections to databases can take time, especially if our database is on a separate server or service. The best practice is to open the connection when your application starts up, and to leave it open until your application restarts or shuts down.

The Node environment variable NODE_ENV is used to allow running in two environments, each of which should use a different database: the development database and the final application live database. For example, Heroku should set NODE_ENV to "production" so that the application will run in production mode on their server.

Our next step is to define the event-handling operations:

```
mongoose.connection.on('connected', function () {
    console.log('Mongoose connected to ' + dbURI);
});
mongoose.connection.on('error',function (err) {
    console.log('Mongoose connection error: ' + err);
});
mongoose.connection.on('disconnected', function () {
    console.log('Mongoose disconnected');
});
```

Listing 1.23. *utilsMongoose.js*: event handling

Then we define functions that 'reacts' to termination events:

```
var gracefulShutdown = function(msg, callback) {
    mongoose.connection.close(function() {
        console.log('Mongoose disconnected through ' + msg);
        callback();
    });
};
// For nodemon restarts;
process.once('SIGUSR2', function () {
    console.log("*** SIGUSR2");
    gracefulShutdown('nodemon restart', function () {
        process.kill(process.pid, 'SIGUSR2');
    });
});
```

[2] To install Mongoose execute: npm install mongoose -save.

[3] Five is just the default number and can be increased or decreased in the connection options.

```
// For app termination;
process.on('SIGINT', function() {
    console.log("*** SIGINT");
    gracefulShutdown('app termination', function () {
        process.exit(0);
    });
});
// For Heroku app termination;
process.on('SIGTERM', function() {
    console.log("*** SIGTERM");
    gracefulShutdown('Heroku app shutdown', function () {
        process.exit(0);
    });
});
```

Listing 1.24. *utilsMongoose.js*: closing connections

Let us introduce also a function (`requestUtil`) that extracts the relevant information from the `URL` associated with a request and then calls a given callback:

```
/*
 * Finding url parts
 */
var url = require('url'),
   qs   = require('querystring');

var requestUtil = function(req,res,cb){
var urlParts   = url.parse(req.url, true),
    urlParams  = urlParts.query,
    urlPathname = urlParts.pathname,
    body = '',
    reqInfo = {};

req.on('data', function (data) {
 body += data;
});
req.on('end', function () {
    reqInfo.method    = req.method;
    reqInfo.urlPathname = urlPathname;
    reqInfo.urlParams = urlParams;
    reqInfo.body      = body;
    reqInfo.urlParts  = urlParts;
    console.log(reqInfo);
    cb(reqInfo,res);
 });
}
module.exports=requestUtil;
```

Listing 1.25. *utilsMongoose.js*:requestUtil

Finally, we introduce operations related to handling of *uncaught Exceptions*:

```
/*
 * uncaughtException handling
 * See https://coderwall.com/p/4yis4w/node-js-uncaught-exceptions
 */
process.on('uncaughtException', function (err) {
    console.error('ERROR: got uncaught exception:', err.message);
    process.exit(1);      //MANDATORY!!!
});
process.on('exit', function(code){
    console.log("Exiting code= " + code );
});
```

Listing 1.26. *utilsMongoose.js*:event handling

4.3 Data models

The mongoose Schema is what is used to define attributes for our documents. The allowed Schema Types are: String, Number, Date, Buffer, Boolean. Mixed. ObjectId, Array.

Mongoose provides built-in and custom validators, and synchronous and asynchronous validators. It allows you to specify both the acceptable range or values and the error message for validation failure in all cases.

In our example of 'users' with *name*, *age* and *password*, we can define:

```
var dataSchema = new mongoose.Schema({
    name: { type: String, required: true },
    age: { type: Number, required: true },
    password: { type: String, required: true }
});
```

Mongoose Methods can also be defined on a mongoose Schema.

```
dataSchema.methods.addPrefix = function() {
    this.name = "user"+this.name;
}
```

Models are constructors compiled from our Schema definitions. Instances of these models represent documents which can be saved and retrieved from our database.

```
var User = mongoose.model('User', dataSchema); //collection users;
```

The first argument is the singular name of the collection your model is for. Mongoose automatically looks for the plural version of your model name. Thus, for the example above, the model User is for the users collection in the database.

Documents are instances of our model.

```
var alice = new User({
     name: 'Alice',
     age: 22,
     password: 'pswdAlice'
});
```

All document creation and retrieval from the database is handled by these models.

```
//Saving the document in the database
alice.save(function(err) {
    if (err) throw err;
    console.log('User saved successfully!');
});
```

4.4 Mongoose Query

Mongoose documents represent a one-to-one mapping to documents as stored in MongoDB. Each document is an instance of its Model.

Mongoose models have several methods available to them to help with querying the database. Among them:

find	General search based on a supplied query object
findById	Look for a specific ID
findOne	Get the first document to match the supplied query
geoNear	Find places geographically close to the provided latitude and longitude
geoSearch	Add query functionality to a geoNear operation

The methods can be chained as follows:

```
Loc
    .findById(locationid)          //apply method;
    .exec(function(err, location) { //execute query
        console.log("findById complete");
    });
```

The database interaction is asynchronous.
Queries can be executed with or without the specification of a callback. More precisely:

– when executing a query with a callback function, you specify your query as a JSON document.

```
var query = User.findOne({ 'name': 'Alice' }, function (err, user) {
  if (err) throw err;
  console.log('%s %s %s.', user.name, user.age, user.password);
})
```

The JSON document's syntax is the same as the MongoDB shell. The query was executed immediately and the results passed to the callback.

– when a callback is not passed, an instance of Query is returned.

```
var query = User.findOne( { 'name': 'Alice' } ); //find
query.select('name password'); //select some field
query.exec( function (err, user) { //exec query at a later time
  if (err) throw err;
    console.log('%s %s %s.', user.name, user.age, user.password);
});
```

A Query provides a special query builder interface that enables you to build up a query using chaining syntax, rather than specifying a JSON object. In mongoose 4, a Query has a .then() function, and thus can be used as a promise.

4.5 Mongoose at work: an example

With Mongoose, all document creation and retrieval from the database is handled by models. Let us introduce an example step by step:

1. First of all we define the dbURI and the data Schema:

```
/*
 * mongoNaive/useMongoose.js
 */
var mongoose     = require( 'mongoose' );
var utilsMongoose = require('./utilsMongoose');
mongoose.Promise = global.Promise; //to avoid deprecated promise;
var dbURI = 'mongodb://localhost:27017/mongoNaive';

var dataSchema = new mongoose.Schema({
    name:    { type: String, required: true },
    age:     { type: Number, required: true },
    password: { type: String, required: true }
});
//custom methods;
dataSchema.methods.setName = function(newame) {
    this.name = newame;
    console.log("name= " + this.name);
    return this.name;
}
dataSchema.methods.incage = function() {
    this.age = this.age+10;
    console.log("age= " + this.age);
    return this.age;
}
//At every save, hash the password;
dataSchema.pre('save', function(next){
```

```
    var user      = this;
    user.password = user.password+"1";
    next();
});
```

Listing 1.27. *useMongoose.js*: dataSchema

2. Now we create a data model (compile the the schema into a model) and define a function to find documents:

```

//Create a data model using the schema;
var User = mongoose.model('user', dataSchema); //collection users;
//Query;
var findUser = function( name, agemin, agemax ){
     User.
         findOne( {} ).
         select('name age password').
         where('name').equals(name).
         where('age').gt(agemin).lt(agemax).
         exec( showUser );
}
//Function that can be used as a callback;
var showUser = function (err, user) {
     if (err) throw err;
     if( user == null ) console.log("no data");
     else console.log('user: %s age=%s pswd=%s.', user.name, user.age, user.password)
};
```

Listing 1.28. *useMongoose.js*: model and find

3. Now we create some document (instances of the model):

```
//Create a new User;
var alice = new User({
     name: 'Alice',
     age: 22,
     password: 'pswdAlice'
});
var bob = new User({
     name: 'Bob',
     age: 33,
     password: 'pswdBob'
});
```

Listing 1.29. *useMongoose.js*: documents

4. Now we introduce functions to store documents:

```
//Store in sequence;
var storeSomeDataCallback = function(){
    alice.save(function(err) {
        if (err) throw err;
        console.log('Alice saved successfully!');
        bob.save(function(err) {
            if (err) throw err;
            console.log('Bob saved successfully!');
            //At the end we close the connection;
            mongoose.connection.close();
        });
    });
};

//Store asynch;
var storeSomeData = function(){
    alice.save(function(err) {
        if (err) throw err;
        console.log('Alice saved successfully!');
        findUser("Alice", 0,40);
    });
    bob.save(function(err) {
        if (err) throw err;
```

```
24        console.log('Bob saved successfully!');
25        findUser("Bob", 0,40);
```

Listing 1.30. *useMongoose.js*: documents

5. Finally we define our application:

```
1  };
2
3  var main = function(){
4      console.log("CONNECT");
5      storeSomeDataCallback();
6  // storeSomeDataCallback();
7  //  storeSomeData();  //WARNING: asynch!!;
8      console.log("----------------------");
9      var query = User.findOne({ 'name': 'Alice' } );
10     query.select('name age password');  //select fields;
11     query.exec( showUser );
12     console.log("----------------------");
13 }
14
15 main( );
```

Listing 1.31. *useMongoose.js*: documents

If we work with an empty data base, the output is:

```
1  CONNECT
2  ----------------------
3  ----------------------
4  Mongoose connected to mongodb://localhost:27017/mongoNaive
5  no data
6  Alice saved successfully!
7  Bob saved successfully!
8  Mongoose disconnected
9  Exiting code= 0
```

5 MVC

A better way to structure our application code is to make reference to the *Model-View-Controller* (MVC) architecture, that separates out the data (model), the display (view), and the application logic (controller). This separation aims to remove any tight coupling between the application components, making code more maintainable and reusable.

MVC focusses on the following main different concerns that we encounter while building an application:

- Data structure and persistence
- Business logic
- Presentation logic

The model is the representation of a specific category of data, or entity, within the application. A Model should not know about the rest of the application.The View should contain logic for rendering structured data. The View should not have knowledge of the rest of the application. The Controller uses the Models and Views to accept and handle inputs and commands from the user. It may be a good idea to extract the business logic out to a service class and treat the controller as a method of accepting and returning (http) requests. Extraction can also be done by creating commands or events.

5.1 MVVM, MVCVM

MVC starts to loose some of it's usefulness when you start building a structure that has components that are not always available. If the View has a lot of user interaction, it is better to use the *Model-View-ViewModel* (MVVM). The ViewModel transforms the model data into something consumable by the view. It handles presentation logic and thus simplifies the view by extracting the presentation logic from the DOM structure: the data is stored in your javascript and the ViewModel handles updates to and from the DOM. In the *Model-View-Viewmodel-Controller* (MVVMC), the ViewConttroller is a specific kind of controller that is coupled and scoped to one View.

MVCVM (or MVVCVM) is a combination of the MVC and MVVM patterns. This is a client side implementation. Models and Controllers are global, while Views, ViewModels and ViewControllers are local. Messaging between components takes place via events. The basic MVCVM mantra:

- ViewModels are just dumb containers that hold a certain shape of data: they do not know where the data comes from or where it is displayed.
- Views display a certain shape of data (via bindings to a view model): They do not know where the data comes from, only how to display it.
- Models supply real data: They do not know where it is consumed.
- Controllers implement application logic. They listen and broadcast messages/events. Controllers populate VMs from Model and listen for VM changes and update the Model.

Basically Controllers become the glue that hold complete independent components together and turn building blocks into applications.

5.2 Views

A view engine can be defined as a *module that does the actual rendering of views*. View engines like EJS[4] and Jade[5] (now called Pug for legal reasons) can be used to generate

[4] https://github.com/tj/ejs

[5] See http://jade-lang.com/reference/ and https://naltatis.github.io/jade-syntax-docs/

HTML pages. In the following we will use the Pug[6] engine that can be installed with the command npm install pug -save. The command npm install pug-cli -g will install a user interface to interact with plug via a console.

Here is an example of a program that generates a HTML file form a Pug template:

```
/*
 * mvcBasic/pugIntro.js
 */
var pug = require('pug');
var fs = require('fs');

const compiledFunction = pug.compileFile('template.pug');
var html = compiledFunction({
    name: 'Bob'
});

fs.writeFile('genFromTemplate.html', html, 'utf8',
       function(err) { if (err) throw err;
           console.log("HTML written on file genFromTemplate.html" );
       }
);

console.log("html=" + html );
```

Listing 1.32. *pugIntro.js*

The operation pug.compile() will compile the Pug source code into a JavaScript function that takes a data object (called "locals)" as an argument; call that resultant function with your data, and it will return a string of HTML rendered with your data.

The Pug template is:

```
html(lang="en")
  head
    title= template2Pug
    script(type='text/javascript').
      //alert("hello")
  body
    h1 Names
    #container.col
      p #{name}
```

Listing 1.33. template.plug

The output is:

```
<html lang="en"><head><title></title><script
     type="text/javascript">//alert("hello")</script></head><body><h1>Names</h1><div class="col"
     id="container"><p>Bob </p></div></body></html>
```

Listing 1.34. genFromTemplate.html

Pug also provides the pug.render() family of functions that combine compiling and rendering into one step. However, the template function will be re-compiled every time render is called, which might impact performance. Alternatively, you can use the cache option with render, which will automatically store the compiled function into an internal cache.

5.3 Model

Our model for the 'users' can be defined as follows:

[6] See https://pugjs.org/api/getting-started.html and https://pugjs.org/api/reference.html

```
/*
* mvcBasic/dataModel.js
*/
var mongoose = require( 'mongoose' );

var dataSchema = new mongoose.Schema({
    name:    { type: String, required: true },
    age:     { type: Number, required: true },
    password: { type: String, required: true }
});
//Create a data model using the schema;
var User = mongoose.model('user', dataSchema); //collection users;
module.exports = User;
```

Listing 1.35. *dataModel.js*

5.4 A MVC Server

Our MVC server now handles requests by delegating the work to proper controllers. The operation `requestUtil` exported from the utility file `utilsMongoose.js` of Subsection 4.2 is used to extract relevant information from the request.

```
/*
* ====================================
* mvcBasic/ServerMVCNaive.js
* ====================================
*/
var http       = require('http');
var fs         = require('fs');
var requestUtil = require('./utilsMongoose.js'); //sets connections;
var ctrlOut    = require('./ctrlShowUsers');
var ctrlAdd    = require('./ctrlAddUser');
var utils      = require('../utils.js');

var readFile  = utils.readeDataFromFile;

http.createServer( function (request, response) {//request is an IncomingMessage;
     requestUtil( request, response, handleTheRequest );
}).listen(3000, function() { console.log('bound to port 3000'); });
```

Listing 1.36. `ServerMVCNaive.js`

The work of the server is defined within the callback `handleRequest` given to `requestUtil`:

```
var handleTheRequest = function(reqInfo,response){
    var request = reqInfo.method;
    var url     = reqInfo.urlPathname;
    console.log("Server request=" +request + " url=" + url );
    if (request === 'GET' && url==='/') {
        ctrlOut.showUsers( response );
        return;
    }
    if (request === 'GET' && url==='/add') {
        readFile('./index.html', function(data){ response.end(data); });
        return;
    }
    if (request === 'GET' ){ //style, favicon;
        readFile("."+ url, function(data){ response.end(data); });
        return;
    }
    if (request === 'POST' && url==='/adduser') {
        ctrlAdd.addUser( reqInfo.body , response);
        return;
    }
    if (request === 'PUT' && url==='/adduser') {
        ctrlAdd.addUser( reqInfo.body , response);
        return;
```

```
    }
    response.end( "Sorry, I don't understand" );
};
```

Listing 1.37. *ServerMVCNaive.js*: work

5.5 Controllers: the reader

Our first controller has the task to build a page with the data read from the data base, by using `Mongoose` and `Pug`.

```
/*
* mvcBasic/ctrlShowUsers.js
*/
var dmodel   = require('./dataModel');
var pug      = require('pug');

const compiledFunction = pug.compileFile('dataView.pug');

exports.showUsers = function(response){
    var query = dmodel.find( function (err, users) {
          var html ;
          if (err) throw err;
          if( users == null ) html = compiledFunction({ items: [] });
          else html = compiledFunction({ items: users });
          response.statusCode=200;
          response.write(html);
          response.end();
    } );
};
```

Listing 1.38. `ctrlShowUsers.js`

In this version, the controller sets the `response` given as argument, while we could limit the controller to define the 'business logic only' (for an example, see Section 6).

5.6 Controllers: the writer

The task to add (via `POST`) or `PUT`) new data to the data base is given to another controller:

```
/*
* mvcBasic/ctrlAddUser.js
*/
var User     = require('./dataModel');
var ctrlOut  = require('./ctrlShowUsers');

module.exports.addUser = function(indata, response){
    var data = cvtDataToJson(indata);
    var newData = new User( {
          name:    data.name,
          age:     data.age,
          password: data.password
    } );
    console.log("ctrlAddUser SAVING " + JSON.stringify( newData) );
    newData.save( function(err) {
        if (err) throw err;
        ctrlOut.showUsers( response );
    });
}
```

Listing 1.39. *ctrlAddUser.js*

This controller handles input data and stores a new `JSON` object in the database. Afterwards it calls the `ctrlShowUsers` controller to show the new set of data.

Input data can take two forms:

1. JSON data, e.g. `"name": "Ann", "age": "35", "password": "pswdAnn"`
2. String sent by a browser, e.g. `"name=Alice&age=25&password=pswdAlice"`

The function `cvtDataToJson` called by the writer (line 8) handles these two cases:

```
1  function cvtDataToJson(inData){
2     //console.log( "cvtDataToJson "+ inData );
3     var jsonData;
4     try{ //assume data already in JSON form;
5        jsonData = JSON.parse( inData );
6     }catch(exception){ //assume data from browser;
7        jsonData = cvtPostStringToJson(inData);
8     }
9     //console.log( jsonData );
10    return jsonData;
11 }
12
13 function cvtPostStringToJson(inData){
14    //console.log( "cvtPostStringToJson:" + inData );
15    var jsonData     = {};
16    var rawdata      = inData.split('&');
17    jsonData.name    = rawdata[0].split('=')[1] ;
18    jsonData.age     = rawdata[1].split('=')[1] ;
19    jsonData.password = rawdata[2].split('=')[1] ;
20    return jsonData;
21 }
```

Listing 1.40. `ctrlAddUser.js`: cvtDataToJson

5.7 Usage

```
1 http://localhost:3000        ; to show the current set of users in the db
2 http://localhost:3000/add    ; to add an user
3
4 curl -X POST -d "name=Alice&age=25&password=pswdAlice" http://localhost:3000/adduser
5
6 curl -H "Content-Type: application/json" -X POST -d "{\"name\":
    \"Dave\",\"age\":\"32\",\"password\":\"pswdDave\"}" http://localhost:3000/adduser
7
8 curl -X DELETE -d "name=Alice&age=25&password=pswdAlice" http://localhost:3000/api/user
```

6 A RESTful interface to data

In Section 5, we built answers to HTTP requests, by sending to the browser a HTML file built from a template. However, with an API[7] we instead want to send a status code and some (JSON) data.

Our goal here is to build a REST API[8] so that we can interact with our database through HTTP calls and perform the common CRUD functions: *create, read, update*, and *delete*. In this section, we will reuse the `utilsMongoose.js` file introduced in Subsection 4.2.

The controllers introduced in Subsection 5.5 and in Subsection 5.6 will be reused from a logical point of view. However, we will redefine the controllers so that:

- a controller focusses on the 'business logic only';
- at the end of its work, a controller calls a given callback, without setting the response;
- a controller receives in input the `response` argument to overcome scoping problems.

Moreover we will introduce also controllers to eliminate data (DELETE verb) and modify data (PUT verb).

6.1 Responses and status codes

Status code	Name	Use case
200	OK	A successful GET or PUT request
201	Created	A successful POST request
204	No content	A successful DELETE request
400	Bad request	GET, POST, or PUT request with invalid content
401	Unauthorized	Requesting a restricted URL with incorrect credentials
403	Forbidden	Making a request that isn't allowed
404	Not found	Unsuccessful request due to an incorrect parameter in the URL
405	Method not allowed	Request method not allowed for the given URL
409	Conflict	POST request when object already exists with the same data
500	Internal server error	Problem with your server or the database server

6.2 The API

Of course, HTML views are now out of the scope of the project. The main design task now is to define a plan for our API. For example:

Action	Verb	URL	Parameters	example
Create user	POST	/api/user	a	a
List of users	GET	/api/user	-	-
Specific user	GET	/api/user	a	a
Update user	PUT	/api/user	a	a
Delete user	DELETE	/api/user	a	a

This 'plan' exploits the different HTTP verbs to give different semantics to a unique path `/api/user`.

[7] API is an abbreviation for Application Program Interface, which enables applications to talk to each other.

[8] REST stands for REpresentational State Transfer, which is an architectural style.

6.3 The server

The server has the same structure of Subsection 5.4. It can be re-defined now as follows:

```
/*
* ====================================
* mvcBasic/ServerMVCRest.js
* ====================================
*/
var ctrlGet    = require('./ctrlGetUsersRest');
var ctrlAdd    = require('./ctrlAddUserRest');
var ctrlDel    = require('./ctrlDeleteUserRest');
var ctrlChng   = require('./ctrlChangeUserRest');
var fs         = require('fs');
var http       = require('http');
var requestUtil = require('./utilsMongoose.js'); //sets connections;

http.createServer( function (request, response) { //request is an IncomingMessage;
   requestUtil( request, response, handleRequest );
 }).listen(3000, function() { console.log('bound to port 3000'); });

var handleRequest = function(reqInfo,response){
   var request = reqInfo.method;
   var url     = reqInfo.urlPathname;

   console.log("Server request=" +request + " url=" + url);
   switch ( request ){
       case 'GET' :
           if( url === '/api/user' ){
               ctrlGet.getUsers( response, doAnswerStr );
           }else{ response.statusCode=400; response.end("ERROR on GET"); }
           break;
       case 'POST' :
           if( url==='/api/user' ){
               ctrlAdd.addUser( reqInfo.body, response, doAnswerStr );
           }else{ response.statusCode=400; response.end("ERROR on POST"); }
           break;
       case 'PUT':
           if( url === '/api/user' ){ //accepts only JSON format;
               var jsonBody = JSON.parse( reqInfo.body );
               console.log("oldUser=" + jsonBody.old);
               console.log("chngUser=" + jsonBody.new);
               ctrlChng.changeUser(jsonBody.old, jsonBody.new, response, doAnswerStr );
           }else{ response.statusCode=400; response.end("ERROR on PUT"); }
           break;
       case 'DELETE':
           if( url === '/api/user' ){ // for JSON only;
               ctrlDel.deleteUser( JSON.parse( reqInfo.body ), response, doAnswerStr);
           }else{ response.statusCode=400; response.end("ERROR on DLEETE"); }
           break;
       default:{
           response.writeHead(405, {'Content-type':'application/json'});
           response.end( "METHOD ERROR" );
       }
   }//switch;
}
```

Listing 1.41. *ServerMVCRest.js*

For each request, the server calls the proper controller, by giving to it the following callback function, that expects a String as its `msg` argument:

```
var doAnswerStr = function(err, response, msg){
   if( err ){ response.statusCode=500; response.end(msg); }
   else{ response.statusCode=200; response.end(msg); }
}
```

Listing 1.42. `ServerMVCRest.js`: doAnswerStr

Thus, each controller will terminate its work by a call like the following one:

```
callback( err, response, answer );
```

where `err` is related to the result of its activity and `response` is the value received in input from the server.

6.4 The CRUD Controllers

Our controllers can be now defined as follows:

6.4.1 List of the users .

```
/*
* mvcBasic/ctrlGetUsersRest.js
*/
var dmodel   = require('./dataModel');

module.exports.getUsers = function( response, cb ) {
    var query = dmodel.find( function (err, users) { //users=array of JSON objects;
          var s = "";
          if (err) s = "ERROR " + err; //throw err;
          if( users.length == 0) s = "[]";
          else{
              users.forEach( function(user,i){
                  s = s  + JSON.stringify(user) + "\n";
              });
          }
          cb( err, response, s ); //standard;
    } );
};
```

Listing 1.43. *ctrlGetUsersRest.js*

6.4.2 Create and add an user .

```
/*
* mvcBasic/ctrlAddUserRest.js
*/
var User = require('./dataModel');

module.exports.addUser = function(indata, response, cb){
    var data  = cvtDataToJson(indata);
    var newData = new User( {
          name:    data.name,
          age:     data.age,
          password: data.password
    } );
    console.log("ctrlAddUserRest SAVING " + JSON.stringify( newData) );
    newData.save( function(err) {
        var s = "";
        if( err ) s="ERROR in adding user";
        else s="Data saved successfully";
        cb( err, response, s ); //standard;
    });
}

function cvtDataToJson(inData){
    //console.log( "cvtDataToJson "+ inData );
    var jsonData;
    try{ //assume data already in JSON form;
        jsonData = JSON.parse( inData );
    }catch(exception){ //assume data from browser;
        jsonData = cvtPostStringToJson(inData);
    }
    //console.log( jsonData );
    return jsonData;
}
```

```
33
34 function cvtPostStringToJson(inData){
35     //console.log( "cvtPostStringToJson:" + inData );
36     var jsonData     = {};
37     var rawdata      = inData.split('&');
38     jsonData.name    = rawdata[0].split('=')[1] ;
39     jsonData.age     = rawdata[1].split('=')[1] ;
40     jsonData.password = rawdata[2].split('=')[1] ;
41     return jsonData;
42 }
```

Listing 1.44. *ctrlAddUserRest.js*

6.4.3 Modify an user .

First we find the old data:

```
1  /*
2  * mvcBasic/ctrlChangeUserRest.js
3  */
4  var User = require('./dataModel');
5
6  module.exports.changeUser = function(data, newdata, response, cb){ //data newdata are JSON obj;
7      console.log("ctrlDeleteUserRest Changing " + JSON.stringify( data ) );
8      User
9          .find(data)
10         .exec(
11             function (err, user) { //user is an array;
12                 if (err ){
13                     cb( err, response, "ERROR in changing user");
14                 }else{
15                     console.log("CHANGING " + JSON.stringify( user ) + " " + user[0]._id);
16                     // Do here something with the document to change;
17                     changeUserData( user[0], newdata, response, cb );
18                 }
19             }
20         );
21 }
```

Listing 1.45. *ctrlChangeUserRest.js*: find

Than we modify and save the data:

```
1  var changeUserData = function(user, newdata, response, cb){
2      user.name     = newdata.name;
3      user.age      = newdata.age;
4      user.password = newdata.password;
5      user.save( function(err) {
6          var s ="";
7          if (err) s = "ERROR in changing user";
8          else s= "Data changed successfully";
9          cb( err, response, s );
10     });
11 }
```

Listing 1.46. *ctrlChangeUserRest.js*: change

6.4.4 Delete an user .

First we find the identifier of the data to delete:

```
1  /*
2  * mvcBasic/ctrlDeleteUserRest.js
3  */
4  var User = require('./dataModel');
5
6  module.exports.deleteUser = function(data, response, cb){ //data is a JSON obj;
7      console.log("ctrlDeleteUserRest DELETING " + JSON.stringify( data ) );
8      User
```

```
        .findOne(data)
        .exec(
           function (err, user) { //user is a single obj;
               if (err) cb( err, response, "ERROR in deleting user");
               else{
                   console.log("REMOVING " + JSON.stringify( user ) + " " + user._id);
                   // Do here something with the document to remove;
                   removeUserById( user._id, response,cb );
               }
           }
        );
}
```

Listing 1.47. *ctrlDeleteUserRest.js*: find

Than we remove the element with the given identifier:

```
    User.findByIdAndRemove(userId, function(err) {
        var s ="";
        if (err) s = "ERROR in deleting user";
         else s = "Data removed successfully";
         cb( err, response, s ); //standard;
    });
}
```

Listing 1.48. *ctrlDeleteUserRest.js*: remove

6.5 Using the REST-API

In order to test our new REST API, let us define a set of CRUD operations:

```
/*
 * mvcBasic/applRestful.js
 */
var request = require("request");
require('../utils');

var dbUrl = "http://localhost:3000/api/user";

//Create a new user Alice ==================================;
var optionsCreateAlice = {
   method: 'POST',
   url: dbUrl,
   body: "name=Alice&age=25&password=pswdAlice",
 };
var createUserAlice = function(){
    request( optionsCreateAlice, function (error, response, body) {
      if (error) throw new Error(error);
      console.log("ANSWER createUserAlice from server:"+body);
    });
};

//Create a new user Bob ====================================;
var optionsCreate = { method: 'POST', url: dbUrl,
   headers:
   { 'Cache-Control': 'no-cache',
     'Content-Type': 'application/json' },
  body: { name: 'Bob', age: '28', password: 'pswdBob' },
  json: true
};
var createUserBob = function(){
    request( optionsCreate, function (error, response, body) {
      if (error) throw new Error(error);
      console.log("ANSWER createUserBob from server:"+body);
    });
};

//Read all the users ======================================;
var optionsRead = { method: 'GET', url: dbUrl,
    headers:
```

```
    { 'Cache-Control': 'no-cache',
      'Content-Type': 'application/json' },
      json: true
};
var readUsers = function(){
    request(optionsRead, function (error, response, body) {
      if (error) throw new Error(error);
      console.log( body );
    });
};

//Change an user ====================================;
var optionsChangeBob = { method: 'PUT', url: dbUrl,
    headers:
    {   'Cache-Control': 'no-cache',
            'Content-Type': 'application/json' },
        body: { "old": {"name": "Bob", "age": "28", "password": "pswdBob"},
                "new":{"name": "Bob", "age": "28", "password": "newPswdBob"} },
        json: true
    };
var updateUser = function(){
    request(optionsChangeBob, function (error, response, body) {
      if (error) throw new Error(error);
      console.log("ANSWER updateUser from server:"+body);
    });
};

//Delete an user ====================================;
var optionsDeleteBob = { method: 'DELETE', url: dbUrl,
    headers:
    {   'Cache-Control': 'no-cache',
        'Content-Type': 'application/json' },
         body: { name: 'Bob', age: '28', password: 'newPswdBob' },
         json: true
};
var deleteUser = function(){
    request(optionsDeleteBob, function (error, response, body) {
      if (error) throw new Error(error);
      console.log("ANSWER deleteUser from server:"+body);
    });
};
```

Listing 1.49. *applRestful.js*

A simple test can be defined as follows:

```
//Application ====================================;
var application = function(){
    readUsers();
    setTimeout( createUserAlice, 200 );
    setTimeout( readUsers,    400  );
    setTimeout( createUserBob, 600 );
    setTimeout( readUsers,    800  );
    setTimeout( updateUser,   1000 );
    setTimeout( readUsers,    1200 );
    setTimeout( deleteUser,   1400 );
    setTimeout( readUsers,    1500 );
}
application( );
```

Listing 1.50. *applRestful.js*

If we work with an empty data base, the output is:

```
[]
ANSWER createUserAlice from server:Data saved successfully
{ _id: '5a671648fba1ce40882086a3',
  name: 'Alice',
  age: 25,
  password: 'pswdAlice',
  __v: 0 }
ANSWER createUserBob from server:Data saved successfully
{"_id":"5a671648fba1ce40882086a3","name":"Alice","age":25,"password":"pswdAlice","__v":0}
```

```
{"_id":"5a671648fba1ce40882086a4","name":"Bob","age":28,"password":"pswdBob","__v":0}

ANSWER updateUser from server:Data changed successfully
{"_id":"5a671648fba1ce40882086a3","name":"Alice","age":25,"password":"pswdAlice","__v":0}
{"_id":"5a671648fba1ce40882086a4","name":"Bob","age":28,"password":"newPswdBob","__v":0}

ANSWER deleteUser from server:Data removed successfully
{ _id: '5a671648fba1ce40882086a3',
  name: 'Alice',
  age: 25,
  password: 'pswdAlice',
  __v: 0 }
Exiting code= 0
```

6.6 A CRUD Server

Thank to our REST API, we can now redefine the server of Subsection 5.4 by extending the server of Subsection 6.3 so that:

- the server works (machine-to-machine) according to the REST API;
- the server accepts data both in JSON and in browser form;
- the server interacts with a browser by delegating the work to controllers that exploit the REST API to read/write data. In this way the controllers do not depend any more on any data representation.

```
/*
* ====================================
* mvcBasic/CrudRestServer.js
* ====================================
*/
var ctrlGet    = require('./ctrlGetUsersRest'); //returns a string of JSON string lines;
var ctrlAdd    = require('./ctrlAddUserRest');
var ctrlDel    = require('./ctrlDeleteUserRest');
var ctrlChng   = require('./ctrlChangeUserRest');
//var ctrlIn     = require('./ctrlAddUser');
var ctrlOut    = require('./ctrlShowUsersRestApi'); //returns a generated HTML string;
var fs         = require('fs');
var http       = require('http');
var requestUtil = require('./utilsMongoose.js'); //sets connections;
var utils      = require('../utils.js');

var readFile  = utils.readeDataFromFile;

http.createServer( function (request, response) { //request is an IncomingMessage;
    requestUtil( request, response, handleRequest );
 }).listen(3000, function() { console.log('bound to port 3000'); });

var handleRequest = function(reqInfo,response){
    var request = reqInfo.method;
    var url     = reqInfo.urlPathname;
    console.log("Server request=" +request + " url=" + url );

    switch ( request ){
        case 'GET' :
            if( url === '/' ){
                ctrlOut.showUsers( response, doAnswerStr );
            }else if( url === '/add' ){ //open the file index.html with the add form;
                readFile('./index.html', function(data){ response.end(data); });
            }else if( url === '/api/user' ){
                ctrlGet.getUsers( response, doAnswerStr );
            }else{
                readFile("."+ url, function(data){ response.end(data); });
            }
            break;
        case 'POST' :
            if( url === "/adduser"){ //sent by the browser (index.html);
                //ctrlIn.addUser( reqInfo.body , response);
```

```
43              ctrlAdd.addUser( reqInfo.body, response, doAnswerStr );
44          }else if( url==='/api/user' ){
45              ctrlAdd.addUser( reqInfo.body, response, doAnswerStr );
46          }else{ response.statusCode=400; response.end("ERROR on POST"); }
47          break;
48      case 'PUT':
49          if( url === '/api/user' ){ //sent via curl or POSTMAN;
50              var jsonBody = JSON.parse( reqInfo.body );
51              console.log("oldUser=" + jsonBody.old + " chngUser=" + jsonBody.new);
52              ctrlChng.changeUser(jsonBody.old, jsonBody.new, response, doAnswerStr);
53          }else{ response.statusCode=400; response.end("ERROR on PUT"); }
54          break;
55      case 'DELETE':
56          if( url === '/api/user' ){ //sent via curl or POSTMAN;
57              ctrlDel.deleteUser( JSON.parse( reqInfo.body ), response, doAnswerStr);
58          }else{ response.statusCode=400; response.end("ERROR on DLEETE"); }
59          break;
60      default:{
61          response.writeHead(405, {'Content-type':'application/json'});
62          response.end( "METHOD ERROR" );
63      }
64  }//switch;
65  }
66
67  var doAnswerStr = function(err, response, answer){ //answer: a string ;
68      if( err ){ response.statusCode=500; response.end(answer); }
69      else { response.statusCode=200; response.end(answer); }
```

Listing 1.51. *CrudRestServer.js*

The controller that builds the list of users without any reference to the data base is now:

```
1   /*
2   * mvcBasic/ctrlShowUsersRestApi.js
3   */
4   var request = require("request");
5   var pug    = require('pug');
6
7   const compiledFunction = pug.compileFile('dataView.pug');
8
9   exports.showUsers = function(response, doAnswer){
10
11      var dbUrl = "http://localhost:3000/api/user";
12      var optionsRead = { method: 'GET', url: dbUrl,
13              headers:
14              { 'Cache-Control': 'no-cache',
15                'Content-Type': 'application/json' },
16                json: false
17          };
18      var readUsers = function(response, doAnswer){      //get data using the REST API;
19           request(optionsRead, function (error, resp, data) { //data is a list of JSON strings
20                if (error) throw error;
21                var html = compiledFunction({ items: getJSONArray(data) });
22                doAnswer(error,response,html); //standard;
23              });
24      };
25      readUsers(response, doAnswer);
26  };
27
28  function getJSONArray(jsonStr){
29      var jsonData = [];
30      if (jsonStr==="[]" ) return jsonData;
31      var rawdata = jsonStr.split("\n");
32      rawdata.forEach( function( s,i ){
33          try{
34              if( s.length>0) //jsd=JSON.parse(s);
35              jsonData.push( JSON.parse(s) );
36          }catch( e ){
37              //console.log("ERROR " + e + " for " + s );
38          }
39      });
40      return jsonData;
41  }
```

Listing 1.52. *ctrlShowUsersRestApi.js*

7 Express

Express[9] is an abstraction layer on top of the Node `HTTP` server that provides four major features:

- `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 (see Section 10).
- `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.

The use pattern of Express can be summarized as follows:

```
1  var express = require("express");
2  var http    = require("http");
3
4  var app     = express();
5
6  app.use( ... ) ;
7
8  app.get( ... ) ;
9
10 http.createServer(app).listen(3000);
```

- The `express()` function starts a new Express application and returns a request handler function.
- `app.use(...)` is intended for *binding* middleware to your application. It means *"Run this on ALL requests"* regardless of `HTTP` verb used (`GET, POST, PUT` ...)
- `app.get(...)` is part of Express' application routing. It means *"Run this on a GET request, for the given URL"*. There is also 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.

An application might have three main middleware functions:

- `logging`. The logging middleware will log every request and continue on to the next middleware.
- `authentication`. The authentication middleware will continue only if the user is authorized.
- `response`. The final middleware won't continue on because nothing follows it.

It's common to find that the functionality we want is already available in somebody else's middleware.

[9] The API docs are at: http://expressjs.com/api.html

7.1 Extensions

Express augments the request and response objects that you're passed in every request handler, as documented in http://expressjs.com/4x/api.html.

```
req = {
    _startTime    :   Date,
    app           :   function(req,res){},
    body          :   {},
    client        :   Socket,
    complete      :   Boolean,
    connection    :   Socket,
    cookies       :    {},
    files         :    {},
    headers       :   {},
    httpVersion   :   String,
    httpVersionMajor  :   Number,
    httpVersionMinor  :    Number,
    method        :   String, // e.g. GET POST PUT DELETE
    next          :   function next(err){},
    originalUrl   :   String,    /* e.g. /erer?param1=23ŭm2=45 */
    params        :   [],
    query         :   {},
    readable      :   Boolean,
    res           :   ServerResponse,
    route         :   Route,
    signedCookies :   {},
    socket        :   Socket,
    url           :   String /*e.g. /erer?param1=23ŭm2=45 */
}
```

Notable extensions for `response` are the methods:

- `response.redirect(..)`, to redirect to another site
- `response.sendfile(..)`, to deliver `HTML` files

```
res = {
    app           :   function(req, res) {},
    chunkedEncoding:  Boolean,
    connection    :    Socket,
    finished      :   Boolean,
    output        :   [],
    outputEncodings:  [],
    req           :   IncomingMessage,
    sendDate      :   Boolean,
    shouldkeepAlive  : Boolean,
    socket        :    Socket,
    useChunkedEncdoingByDefault :   Boolean,
    viewCallbacks :   [],
    writable      :    Boolean
}
```

Notable extensions for `request` are the methods:

- `request.ip`, to get the `IP` address
- `request.get`, to get incoming `HTTP` headers

7.2 Middleware

The following example shows a middleware that affects the response (but it doesn't have to).

```
/*
* ====================================
* expressBasic/app.js
* ====================================
```

```
*/
var express = require("express");
var http    = require("http");
var logger = require("morgan");
/*
 * Calls the express function to start a new Express application
 */
var app = express();      //returns a requestHandler function;

app.use( logger("short") ) ; //logger("short") return a middleware function written by Morgan;

app.use(function(request, response, next) {
    console.log("Request =" + request.url);
    response.write("middleware0-");
    next();
});

app.use(function(request, response, next) {
    console.log("middleware1-" + response.statusCode);
    response.write("middleware1-");
    next();
});

app.use(function(request, response, next) {
    console.log("middleware2=" + response);
    response.write("middleware2-");
    response.end("Hello, world from expressBasic/app.js");
});

http.createServer(app).listen(3000, function(){
    console.log('bound to port 3000');
});
```

Listing 1.53. *app.js*

If we write in a browser, `http://localhost:3000/`, we will see the following output:

```
In the browser:
middleware0-middleware1-middleware2-Hello, world from expressBasic/app.js

On the server console:
bound to port 3000
Request =/
middleware1-200
middleware2=[object Object]
::1 - GET / HTTP/1.1 200 - - 4.181 ms
```

The last row on the console is produced by the `morgan` logger.

7.3 Static files

Express's static middleware (`express.static`) allows us to show files out of a given directory. For example, to show any file in the directory named `public`, we can write:

```
/*
* ====================================
* expressBasic/appStaticFiles.js
* ====================================
*/
var express = require("express");
var http    = require("http");
var path    = require("path");

var app = express();

//Sets up the public path, using NodeâĂŹs path module;
var publicPath = path.resolve(__dirname, "public");
app.use(express.static(publicPath));

app.use(function(request, response) {
```

```
    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("Static file not found.");
});

http.createServer(app).listen(3000, function(){
    console.log('bound to port 3000');
});
```

Listing 1.54. *appStaticFiles.js*

Let us include in the `public` directory the following file:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>PublicFile</title>
        <link rel="stylesheet" href="stylesheets/style.css">
    </head>

    <body>
        <h1>Hello from public/index.html</h1>
    </body>
</html>
```

Listing 1.55. *public/index.html*

If we write in a browser: `http://localhost:3000`, the server will show the content of the file, by using the style defined in `public/stylesheets/style.css`.

7.4 Routing

`app.get()` is part of Express' application routing and is intended for matching and handling a specific route when requested with the `GET` verb. The first argument is a path, like `/about` or simply `/` (the site's root). The second argument is a request handler function that work just like middleware; it's a matter of when they're called.

```
/*
* ====================================
* expressBasic/appRouting.js
* ====================================
*/
var express = require("express");
var http    = require("http");

var app = express();

app.use(function(request, response, next) {
    console.log("Request method=" + request.method + " request.url=" + request.url);
    response.write("middlew0-");
    next();
});

//no next => terminate;
app.get("/", function(request, response) {
    console.log("get / Request =" + request.url);
    response.write("get middlew1:");
    response.end("Welcome to my homepage!");
});

app.get("/about", function(request, response,next) {
    console.log("about page " );
    response.write("get middlew1:Welcome to the about page!-");
    next();
});

app.post("/", function(request, response,next) {
    console.log("post page " );
    response.write("post middlew1-");
```

```
33      next();
34  });
35
36  //no next => terminate;
37  app.use(function( request, response ) {
38      console.log("middlew2");
39      response.end("middlewEND");
40  });
41
42  //main
43  http.createServer(app).listen(3000, function(){
44      console.log('bound to port 3000');
45  });
```

Listing 1.56. *appRouting.js*

http://localhost:3000	middlew0-get middlew1-Welcome to my homepage!
http://localhost:3000/about	middlew0-get middlew1:Welcome to the about page!-middlewEND
curl -X POST -d "..." http://localhost:3000/	middlew0-post middlew1-middlewEND
curl -X PUT -d "..." http://localhost:3000/	middlew0-middlewEND

7.5 Views

In the traditional approach, an HTTP server dynamically generates HTML pages[10]; this approach is also known as one-way data binding model: the hard work is done on the server, leaving the browser to just render HTML and run any JavaScript interactivity.

The Express function render can be used for compiling view templates, like those introduced in Subsection 5.2. We will continue to use here the Pug engine, by exploiting its extension facility. For example:

```
1  extends layout.jade
2  block header
3      h1= title + ":an example"
4  block content
5      p Welcome to #{title}
```

Listing 1.57. *index.jade*

At the top of the file is a statement declaring that this file is an extension of the layout.jade file. The next statement defines a block of code that belongs to a specific area of the layout file, an area called content in this instance. Finally, there's the minimal content that is displayed on the Express index page.

The layout file can be defined as follows:

```
1  doctype html
2  html
3      head
4          meta(charset="utf-8")
5          title= title
6          link(rel="stylesheet" href="style.css")
7      block header
8  body
9      hr
10     block content
11 footer
12     hr
13     small AN-DISI-Unibo 2018
```

Listing 1.58. *layout.jade*

[10] However, a modern alternative are single-page applications (SPA) built by using frameworks like AngularJS.

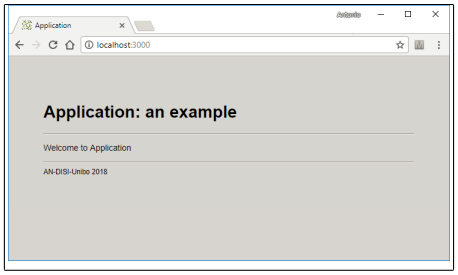
There are a `head`, a `body` and a `footer` sections. There are also lines (`block header` and `block content`) with nothing inside it. This named blocks can be referenced by other Jade templates, such as the previous `index.jade` file.

Let us consider the following express application:

```
/*
* ====================================
* expressBasic/appViewJade.js
* ====================================
*/
var express = require("express");
var http   = require("http");
var logger = require("morgan");
var path   = require("path");
var app    = express();

app.use( logger("short") ) ;

app.use(express.static(path.join(__dirname, 'views'))); //for style.css;

app.set("view engine", "jade");
app.set("views", path.resolve(__dirname, "views"));

app.get("/", function(request, response) {
       response.render("example" ,{
           title: "Application"
       });
});

http.createServer(app).listen(3000, function(){
    console.log('bound to port 3000');
});
```

Listing 1.59. *appViewJade.js*

The output is:



7.6 Returning Json data

When buildini an API (like that of Section **??**) we want to anwer to a request by sending a status code and some (`JSON`) data. Express makes this really easy with the following commands:

```
res.status(status)    //Send response status code, e.g. 200
res.json(content)     //Send response data, e.g. {?status? : ?success?}
```

Thus, let us introduce in the file `utils.js` the following utility operation:

```
exports.doAnswerApi = function(status, response, jsonData){
    if( err ){ response.status(status); response.json(jsonData); }
    else { response.status(status); response.json(jsonData); }
}
```

7.7 The *render* function

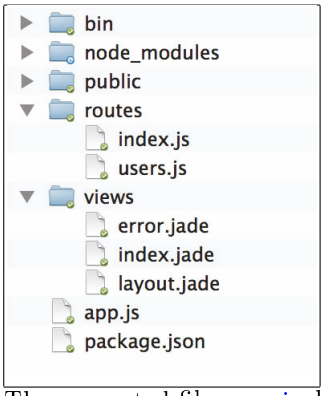
`render` is the Express function for compiling a view template to send as the `HTML` response. It takes the name of the view template and a JavaScript data object. For example, with reference to the file index.jade of Subsection 7.5, we can build a `HTML` page by writing:

```
res.render(?index?, {title:?Express render?});
```

The template file doesn't need to have the file extension suffix. You also don't need to specify the path to the view folder if you've already done this in the Express setup related to the view engine (see Subsection 7.8 and Subsection 7.9).

7.8 Express set up

If we run the command `express` in a new directory, this command will create a bunch of folders and files that will form the basis of our Express/Node application.



The generated file `app.js` has the following content:

```
/*
 * expresssetup/app.js
 */
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('./routes/index');
var users = require('./routes/users'); ;
var app  = express();
// view engine setup;
app.set('views', path.join(__dirname, 'views'));
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 1.60. `app.js`

The file `package.json` that contain various metadata about the project, including the packages that it depends on to run, is:

```
{
  "name": "xxx",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.18.2",
    "cookie-parser": "~1.4.3",
    "debug": "~2.6.9",
    "express": "~4.15.5",
    "jade": "~1.11.0",
    "morgan": "~1.9.0",
    "serve-favicon": "~2.4.5"
  }
}
```

Listing 1.61. `package.json`

If we run `npm install`, we will see a new directory `node_modules`, populated with the required modules. To exclude the saving of these files in the `GIT` repository, include in the main directory a `.gitignore` file, like[11]:

```
node_modules/
```

Listing 1.62. `.gitignore`

7.8.1 body-parser .

The package `body-parser` is important, since it facilitates the handling of `POST/PUT` requests according to the following schema:

```
app.put("/", function(req,res,next){
    var path = url.parse(req.url).pathname;
    console.log( "PUT path=" + path + " PUT args=" + req.body.value);
...
    res.send( ... );
});
```

If we send the following request:

```
curl -H "Content-Type: application/json" -X PUT -d "{\"value\": \"50\" }" http://localhost:8080
```

the outpute will be:

[11] To show the file, deselect: `Package Explorer : Arrow -> Filters -> *.resources`

```
PUT path=/ PUT args==50
```

For a more complete example, see Subsection ??.

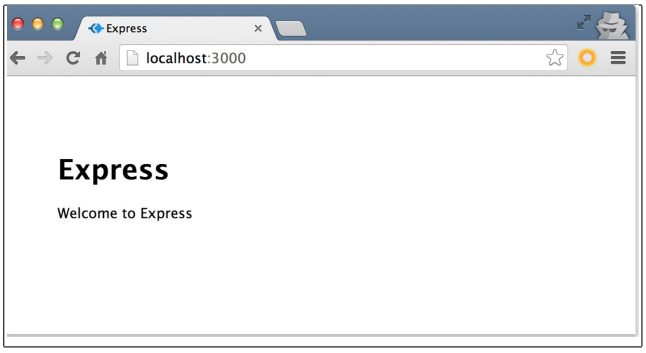
7.8.2 Start-up .

The file `package.json` contains also a start script (line 6) that will be consulted when you will start the application with the command `npm start`. In our case the script `./bin/www` starts with the following code:

```
#!/usr/bin/env node
/**
 * Module dependencies.
 */
var app = require('../app');
var debug = require('debug')('xxx:server');
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);
```

Listing 1.63. The built-in start script `./bin/www`

If now we open a browser with `localhost://3300`, we will see the following output:



7.8.3 routes/index.js .

Note that the generated application workspace includes a folder named `routes` with the file `index.js` defining the routing rules.

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

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
```

```
8
9  module.exports = router;
```

Listing 1.64. `routes/index.js`

This file is required by the `app.js` code (line 8) and used for the home page (line 25).

7.8.4 views/index.jade .

The generated application workspace includes also a folder named `views` that includes the files that define the view templates.

```
1  extends layout
2
3  block content
4    h1= title
5    p Welcome to #{title}
```

Listing 1.65. `/views/index.jade`

where `views/layout.jade` is:

```
1  doctype html
2  html
3    head
4      title= title
5      link(rel='stylesheet', href='/stylesheets/style.css')
6    body
7      block content
```

Listing 1.66. `/views/layout.jade`

This folder is set by the `app.js` code (line 14) together with the jade view-engine (line 15).

7.9 Express MVC

The file structure built by the command `express` includes a `views` folder, but no explicit folders for the *models* and the *controllers*. Thus we rearrange the working space as follows:

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.
4. Modify the file `app.js` according to the changes done:

```
1  var routes = require('./appServer/routes/index'); // line 8-9
2
3  // view engine setup
4  app.set('views', path.join(__dirname, 'appServer', 'views')); // line 14
5
6  app.use('/', routes); // line 25-26
```

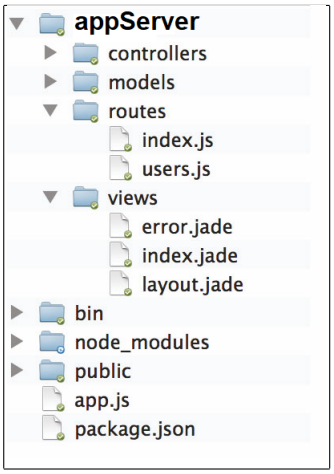
The application works as the original one.

Until now, controllers are considered as part of the routes. However, routing should just map `URL` requests to controllers embedding the application logic. To achieve this goal, let us introduce the following further modifications of the workspace:

– Create within the folder fnameappServer/controllers a set of *controller files* implementing the application logic[12].

[12] Controller files can be introduced by looking at logical collections of data. These collections can be initially determined by collating separate application screens related to user stories.

The final result is:



8 Refactoring the CRUD server

In this section we will redesign the CRUD server of Subsection 6.6 by using express in the MVC style of Subsection 7.9.

Let us start by customizing the generated app.js according to the directory structure introduced in Subsection 7.9:

```
/*
 * expressMvcCrud|app.js
 */
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 requestUtil = require('./utilsMongoose.js'); //sets connections;

var routes  = require('./appServer/routes/index'); // *** CUSTOMIZED;
var app     = express();
console.log("STARTING at:" + __dirname);       // *** CUSTOMIZED;

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

Listing 1.67. expressMvcCrud/app.js

8.1 Routes

Now, let us define the routes. :

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

var readFile    = utils.readeDataFromFile;
//REST API;
var ctrlGet    = require('../controllers/ctrlGetUsersRest');
var ctrlAdd    = require('../controllers/ctrlAddUserRest');
var ctrlDel    = require('../controllers/ctrlDeleteUserRest');
var ctrlChng   = require('../controllers/ctrlChangeUserRest');
//HTML;
var ctrlOut    = require('../controllers/ctrlShowUsersRestApi');

router.get('/',         ctrlOut.showUsers );
router.get('/add',      showAddForm       );
//router.post("/adduser", ctrlAdd.addUser  );  //from FORM (legacy)

router.get(  "/api/user", ctrlGet.getUsers );
router.post( "/api/user", ctrlAdd.addUser );
router.delete("/api/user", ctrlDel.deleteUser );
router.put(  "/api/user", ctrlChng.changeUser );

module.exports = router;

function showAddForm(request,response){
   readFile('./public/addUser.html', function(data){ response.end(data); });
}
```

Listing 1.68. appServer/routes/index.js

The file public/addUser.html provides the input form to load a new user. It wil genrate a POST request with path /api/user:

```
<!DOCTYPE html>
<html>
```

```
3     <head>
4         <meta charset="utf-8">
5         <title>ServerCRUDExpress</title>
6         <link rel="stylesheet" href="stylesheets/style.css">
7     </head>
8
9     <body>
10        <h1>Server CRUD MVC-Express</h1>
11        <form method="post" action="/api/user">
12            <label>User Data Form</label><br>
13            <input type="text" name="name" placeholder="Enter user name..." required>
14            <input type="text" name="age" placeholder="Enter age ..." required>
15            <input type="text" name="password" placeholder="Enter password ..." required>
16            <input type="submit" value="Add">
17        </form>
18    </body>
19 </html>
```

Listing 1.69. `public/addUser.html`

In this phase, we plan the set of controllers, each providing an entry point with signature `function( request, response)`:

The following controllers implement the `REST API` introduced in Section 6:

- a controller that returns the list of users: Subsection 8.2
- a controller that adds a new user: Subsection 8.3
- a controller that delete an user: Subsection 8.4
- a controller that changes an user: Subsection 8.5

Moreover, we introduce a controller that shows the list of the current users, by building an `HTML` file (see Subsection 8.6).

8.2 Controller: List of users

The controller finds the users in data base with Mongoose and then sends as answer the obtained array of `JSON` objects:

```
1  /*
2  * appServer/controllers/ctrlGetUsersRest.js
3  */
4  var User = require('../models/dataModel');
5  var utils = require('../../utils.js');
6
7  module.exports.getUsers = function( request, response ) {
8      var query = User.find( function (err, users) { //users=array of JSON objects;
9            if (err) throw err;
10           utils.doAnswerApi(200,response,users);
11     } );
12 };
```

Listing 1.70. `appServer/controllers/ctrlGetUsersRest.js`

`USAGE:`

```
1  curl -X  GET  http://localhost:3000/api/user
2  ANSWER: ... (lst of users)
```

8.3 Controller: Adding an user

```
/*
* appServer/controllers/ctrlAddUserRest.js
*/
var User = require('../models/dataModel');
//var utils = require('../../utils.js');

module.exports.addUser = function(request, response){
    var data = request.body;
    var newData = new User( {
         name:    data.name,
         age:     data.age,
         password: data.password
    } );
    console.log("ctrlAddUserRest SAVING " + JSON.stringify( newData) );
    newData.save( function(err) {
        if( err ) throw err;
        else response.redirect("/")
    });
}
```

Listing 1.71. appServer/controllers/ctrlAddUserRest.js

USAGE:

```
curl -H "Content-Type: application/json" -X POST -d "{\"name\":
      \"Dave\",\"age\":\"32\",\"password\":\"pswdDave\"}" http://localhost:3000/api/user
ANSWER: Found. Redirecting to /
```

8.4 Controller: Removing an user

```
/*
* appServer/controllers/ctrlDeleteUserRest.js
*/
var User  = require('../models/dataModel');
var utils = require('../../utils.js');
module.exports.deleteUser = function(request, response ){
    var data = request.body;
    console.log("ctrlDeleteUserRest DELETING " + JSON.stringify( data ) );
    User
        .findOne(data)
        .exec(
            function (err, user) { //user is a single obj;
                if (err) utils.doAnswerStr( err, response, "ERROR in deleting user" );
                else{
                    // Do here something with the document to remove;
                    removeUserById( user._id, response );
                }
            }
        );
 }

var removeUserById = function(userId, response ) {
    User.findByIdAndRemove(userId, function(err) {
        var s ="";
        if (err) s = "ERROR in deleting user";
         else s = "Data removed successfully";
         utils.doAnswerStr( err, response, s );
    });
}
```

Listing 1.72. appServer/controllers/ctrlDeleteUserRest.js

USAGE:

```
curl  -X DELETE -d "name=Dave&age=32&password=pswdDave" http://localhost:3000/api/user
ANSWER: Data removed successfully
```

8.5 Controller: Changing an user

```
/*
* appServer/controllers/ctrlChangeUserRest.js
*/
var User = require('../models/dataModel');
var utils = require('../../utils.js');

module.exports.changeUser = function(request, response ){
    var oldUser = request.body.old ;
    var chngUser = request.body.new ;
    console.log( request.body );
    console.log( oldUser );
    console.log("oldUser=" + oldUser + " chngUser=" + chngUser);
    User
        .find(oldUser)
        .exec(
            function (err, user) { //user is an array;
                if (err ){
                    utils.doAnswerStr( err, response, "ERROR in changing user" );
                }else{
                    // Do here something with the document to change;
                    changeUserData( user[0], chngUser, response );
                }
            }
        );
 };

var changeUserData = function(user, newdata, response ){
    if( ! user ){
        console.log("ERROR in changing : no user");
        utils.doAnswerStr( null, response, "ERROR: no user" );
        return;
    }
    user.name    = newdata.name;
    user.age     = newdata.age;
    user.password = newdata.password;
    user.save( function(err) {
        var s ="";
        if (err) s = "ERROR in changing user";
        else s= "Data changed successfully";
        utils.doAnswerStr( err, response, s );
    });
}
```

Listing 1.73. appServer/controllers/ctrlChangeUserRest.js

USAGE:

```
curl -H "Content-Type: application/json" -X PUT -d "{ \"old\": {\"name\": \"Dave\", \"age\": \"32\",
     \"password\": \"pswdDave\"}, \"new\":{\"name\": \"Dave\", \"age\": \"33\", \"password\":
     \"newPswdDave\"} }" http://localhost:3000/api/user
ANSWER: Data changed successfully
```

8.6 Controller: Building a view with the list of users

The controller finds the users in data base with Mongoose and then renders the answer in a generated HTML file:

```
/*
* appServer/controllers/ctrlShowUsersRestApi.js
*/
var request = require("request");

exports.showUsers = function(req,response){
    //get data using the REST API;
    var dbUrl = "http://localhost:3000/api/user";
    var optionsRead = { method: 'GET', url: dbUrl,
            headers:
```

```
11            { 'Cache-Control': 'no-cache',
12              'Content-Type': 'application/json' },
13              json: true
14         };
15     request(optionsRead, function (error, resp, jsonData) { //array of JSON data;
16          if (error) throw error;
17          response.render( "dataView", { users: jsonData } );
18         });
19 };
```

Listing 1.74. appServer/controllers/ctrlShowUsersRestApi.js

USAGE:

```
1 http://localhost:3000/
2
3 curl -X GET http://localhost:3000
```

9 Publish/Subscribe

Using a request-response pattern with REST over HTTP isn't efficient because we have to constantly poll the resources. *Interactive* and *reactive* applications require mechanisms to send events or receive notifications. Besides the *request-response* pattern, we need a publish/subscribe (pub/sub) model to allow further decoupling between data consumers (*subscribers*) and data producers (*publishers*).

In the pub/sub model, publishers send messages to a central server (a *broker*), that handles the routing and distribution of the messages to the various subscribers, depending on the type or content of messages. The notion of *server push* means that a server can provide content to clients without having to wait for them to send a request.

9.1 Web Sockets

The WebSocket protocol is part of the HTML5 specification and is a good candidate to implement pub/sub support. WebSocket enables a full-duplex communication channel over a single TCP connection over port 80; thus they aren't blocked by firewalls and can traverse proxies. A WebSocket permanent connection takes place as follows:

1. send an HTTP call to the server with a special header asking for the protocol to be upgraded to WebSockets;
2. the server replies with a 101 Switching Protocols status code, acknowledging the opening of a full-duplex TCP socket;
3. the client and the server eventually sends a control frame to signal that the communication is over and can be closed.

The HTTP/2 specification:

- allows multiplexing responses, i.w., sending responses in parallel;
- introduces compressed headers using an efficient and low-memory compression format;
- introduces the notion of server push

In the long run, widespread adoption of s HTTP/2 might even remove the need for an additional protocol for push like WebSocket.

9.2 SocketIO

At present, the problem with the WebSocket protocol is that it's not yet finalized, and although some browsers have begun shipping with WebSocket, there are still a lot of older versions out there.

Socket.IO solves this problem by utilizing WebSocket when it's available in the browser, and falling back to other browser-specific tricks to simulate the behavior that WebSocket provides, even in older browsers.

Let us show here the structure of a minimal Socket.IO application[13]

[13] we have to run `npm install socket.io -save`.

9.2.1 The server .

```
/*
 * timeServer/server.js
 */
var app  = require('http').createServer(handler);
var io   = require('socket.io').listen(app);  //Upgrade;
var fs   = require('fs');
var html = fs.readFileSync('index.html', 'utf8');

function handler (req, res) {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html, 'utf8'));
  res.end(html);
}

function tick () {
  var now = new Date().toUTCString();
  io.sockets.send(now);
}

setInterval(tick, 1000);

app.listen(8080, function(){console.log("bound to port 8080")});
```

Listing 1.75. `timeServer/server.js`

The line `5` performs the upgrade of the regular `HTTP` server to `Socket.IO` server; the serve will automatically serve the client file to via `http://localhost:<port>/socket.io/socket.io.js`.

Our server always serves the `index.html` file and invokes a function (`tick`) once per second to notify all the connected clients of the server's time.

9.2.2 A client as a `HTML` page .

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="/socket.io/socket.io.js">
    </script>
    <script type="text/javascript">
      var socket = io.connect();
      socket.on('message', function (time) {
        document.getElementById('time').innerHTML = time;
      });
    </script>
  </head>
  <body>Current server time is: <b><span id="time"></span></b>
  </body>
</html>
```

Listing 1.76. The client: `timeServer/index.html`

Note the line `5`:

```
<script src="/socket.io/socket.io.js"></script>
```

You might be wondering where the `/socket.io/socket.io.js` file comes from, since we neither add it and nor does it exist on the filesystem. This is part of the magic done by `io.listen` on the server. It creates a handler on the server to serve the `socket.io.js` script file.

`USAGE`: open in two browsers to see the changes in synch

```
http://localhost:8080/
```

9.3 Watchers for File Changes

Node.js provides APIs for watching files:

- `fs.watchFile()`: reliable, cross-platform but rather expensive;
- `fs.watch()`: optimized, but with behavioral differences on certain platforms;
- `node-watch`: A wrapper and enhancements for `fs.watch` that generates events named `update` or `remove`.

A very simple example:

```
/*
 * watchFileServer/fileChange.js
*/
var watch = require('node-watch'); //returns a fs.FSWatcher;
  console.log("fileChange STARTS");

  watch('aFile.txt', function(event, filename) {
      console.log("**** " + event + " " + filename);
  });
```

Listing 1.77. `watchFileServer/fileChange.js`

USAGE

```
node fileChange
update the file aFile.txt. The result is:
**** update aFile.txt
```

Let us define now a server that automatically notifies its clients when a file is updated:

9.3.1 A server that notifies file changes .

```
/*
 * watchFileServer/fileWatchServer.js
 */
var fs     = require('fs');
var url    = require('url');
var express = require('express');
var http   = require('http');
var path   = require('path');
var watch  = require('node-watch');

var app    = express();
var server = http.createServer(app);
var io     = require('socket.io').listen(server); //Wrap;

var root   = __dirname;

app.get("/", function(req,res,next){
   var path = url.parse(req.url).pathname;
   createWatcher('./index.html','reload' );
   res.sendFile(root + '/index.html');
});

app.use(express.static(root));

var watchers = {}; //List of being watched files;

function createWatcher (file, mode) {
  var absolute = path.join(root, file);
  console.log("createWatcher: " + absolute );
  if (watchers[absolute]) { return; } //already created;
  watch(absolute, function (event, file) {
      console.log("**** " + event + " " + file + " mode="+mode);
      io.sockets.emit(mode, file);
```

```
34    });
35    watchers[absolute] = true; //Mark file as being watched;
36  }
37
38  server.listen(8080, function(){console.log("bound to port 8080")});
```

Listing 1.78. `watchFileServer/fileWatchServer.js`

The `Socket.io` Server listens properly (line 13) to our `HTTP` server. Thus, it will automatically serve the client file to via `http://localhost:<port>/socket.io/socket.io.js`.

9.3.2 A client as a `HTML` page .

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Socket.IO dynamically reloading CSS stylesheets</title>
5      <link rel="stylesheet" type="text/css" href="styles.css" />
6      <script src="/socket.io/socket.io.js"></script>
7      </head>
8       <script type="text/javascript">
9          window.onload = function () {
10             var socket = io.connect();
11             socket.on('reload', function () {
12                 window.location.reload();
13           });
14        }
15     </script>
16   <body>
17     <h2>Changing page</h2>
18     <div id="body">
19       If this file is edited, the server will send the message <code>'reload'</code>
20       to the browser using <code>Socket.IO</code>.
21     </div>
22   </body>
23 </html>
```

Listing 1.79. The client: `watchFileServer/index.html`

9.4 A Watcher for a sensor

In order to handle real data, let us introduce here a `sensor plugin` that simulates values of temperature and humidity, with reference to the physical sensor `DHT22`.

```
1  /*
2   * plugins/DHT22SensorPlugin.js
3   */
4  var resources = require('../appServer/models/model');
5  var  utils    = require('../utils.js');
6
7  var interval;
8  var pluginName = 'Temperature & Humidity';
9  var localParams = {'simulate': false, 'frequency': 5000};
10 var modelTemperature, modelHumidity;
11
12 var Dht22Plugin = exports.Dht22Plugin = function (params) {
13       modelTemperature = utils.findProperty("temperature");
14       modelHumidity   = utils.findProperty("humidity");
15       this.addValue([0, 0]); //temperature humidity;
16       this.showValues();
17 };
18
19 Dht22Plugin.prototype.addValue = function(values) {
20     utils.cappedPush(modelTemperature.data, {"t": values[0], "timestamp": utils.isoTimestamp()});
21     utils.cappedPush(modelHumidity.data, {"h": values[1], "timestamp": utils.isoTimestamp()});
22 };
```

```
23
24 Dht22Plugin.prototype.showValues = function () {
25       console.info('Temperature: %s C, Humidity: %s \%',
26              modelTemperature.data[modelTemperature.data.length-1].t,
27              modelHumidity.data[modelHumidity.data.length-1].h);
28 };
29 Dht22Plugin.prototype.start = function () {
30   console.log("DHT22SensorPlugin STARTED simulate every (msec):" + localParams.frequency);
31   this.simulate();
32 };
33
34 Dht22Plugin.prototype.stop = function () {
35    clearInterval(interval);
36    console.info('%s plugin stopped!', pluginName);
37 };
38
39 Dht22Plugin.prototype.simulate = function() {
40      var self = this;
41      this.interval = setInterval(function () {
42          self.addValue([utils.randomInt(0, 40), utils.randomInt(20, 100)]);
43          self.showValues();
44      }, localParams.frequency);
45 }
```

Listing 1.80. plugins/DHT22SensorPlugin.js

10 Beyond Connect

Connect is a framework that uses modular components called middleware[14] to implement web application logic in a reusable manner. In Connect, a middleware component is a function that intercepts the request and response objects provided by the HTTP server, executes logic, and then either ends the response or passes it to the next middleware component.

In Connect, a middleware component is a JavaScript function that by convention accepts three arguments: a request object, a response object, and an argument commonly named next, which is a callback function indicating that the component is done and the next middleware component can be executed.

Connect includes the concept of mounting, a simple yet powerful organizational tool that allows you to define a path prefix for middleware or entire applications. Mounting allows you to write middleware as if you were at the root level (the /) and use it on any path prefix without altering the code.

Core web application functions like logging, sessions, and virtual hosting are all provided by Connect out of the box.

10.1 Logging: Winston and Morgan

Winston is an async logging library for Node.js, designed to be a simple and universal logging library with support for multiple storage devices (transports). Logging levels in Winston conform to the severity ordering specified by RFC5424: severity of all levels is assumed to be numerically ascending from most important to least important.

Morgan is a HTTP request logger middleware for Node.js. It standardizes and automatically creates request logs. It saves developers time because they don't have to manually create common logs. Morgan can operate standalone, but commonly it's used in combination with Winston. Winston is able to transport logs to an external location, or query them when analysing a problem.

In the following example we use both Winston and Morgan:

```
1  /*
2   * logging/singleValueServer.js
3   * npm i winston@next --save
4   */
5  var url       = require('url');
6  var express   = require('express');
7  var http      = require('http');
8  var path      = require('path');
9  var bodyParser = require('body-parser');
10 var fs        = require('fs');
11 var winston   = require('winston');
12 var morgan    = require('morgan');
13
14 const logger = winston.createLogger({
15      level: 'info',      //provides a convenience method
16      format: winston.format.json(),
17      transports: [
18        // - Write to all logs with level `info` and below to `winstonLog.log` ;
19        // - Write all logs error (and below) to `error.log`. ;
20        new winston.transports.File({ filename: 'error.log', level: 'error' }),
21        new winston.transports.File({ filename: 'winstonLog.log' })
22      ]
23    });
24
25    /*
26    If we're not in production then log to the `console` with the format:
27    `${info.level}: ${info.message} JSON.stringify({ ...rest }) `
28    */
```

[14] The concept of middleware was initially inspired by Ruby?s Rack framework.

```
29      if (process.env.NODE_ENV !== 'production') {
30        logger.add(new winston.transports.Console({
31          format: winston.format.simple()
32        }));
33      }
34
35  var app        = express();
36  var server     = http.createServer(app);
37  var io         = require('socket.io').listen(server); //Wrap;
38
39  var value  = 0;
40
41  //create a write stream (in append mode) ;
42  var accessLogStream = fs.createWriteStream(path.join(__dirname, 'morganLog.log'), {flags: 'a'})
43  app.use(morgan("short", {stream: accessLogStream}));
44
45  app.use( bodyParser.json( ) );
46
47  server.listen(3000, function(){ console.log("bound to port 3000. NODE_ENV=" + process.env.NODE_ENV ) });
48
49  app.get("/", function(req,res,next){
50      var path = url.parse(req.url).pathname;
51      console.log( "GET path=" + path );
52      res.send( "Server answer tp GET:"+ value );
53      logger.info(""+ path);
54  });
55
56  app.put("/", function(req,res,next){
57      var path = url.parse(req.url).pathname;
58      console.log( "PUT path=" + path + " req.body.value=" + req.body.value);
59      value = req.body.value;
60      io.sockets.emit('message', value);  //Push ;
61      res.send( "Server answer to PUT:"+ value );
62  });
```

Listing 1.81. *singleValueServer.js*: init

```
1  curl -X GET http://localhost:3000
2  curl -H "Content-Type: application/json" -X PUT -d "{\"value\": \"50\" }" http://localhost:3000
3  curl -X GET http://localhost:3000
```

File `winstonLog.log` (logs only the `GET` verb):

```
1  {"message":"/","level":"info"}
2  {"message":"/","level":"info"}
```

File `morganLog.log`:

```
1  ::1 - GET / HTTP/1.1 200 22 - 7.403 ms
2  ::1 - PUT / HTTP/1.1 200 23 - 78.614 ms
3  ::1 - GET / HTTP/1.1 200 23 - 3.490 ms
```

10.2 User authentication

Authentication is for identifying users and provide different access rights and content depending on their id. In most cases the application provides a login form with certain credentials to verify a user.

Let us introduce a basic express starter setup, which simply creates a webserver and serves the static files from the template on the home route.

We start by creating a `MongoDB` schema to describe the fields that we will have in our form and specify the data it can expect:

```
1  var mongoose = require('mongoose');
2  var bcrypt  = require('bcrypt');
3
4  var UserSchema = new mongoose.Schema({
```

```
  email: {
    type: String,
    unique: true,
    required: true,
    trim: true
  },
  username: {
    type: String,
    unique: true,
    required: true,
    trim: true
  },
  password: {
    type: String,
    required: true,
  },
  passwordConf: {
    type: String,
    required: true,
  }
});
```

Listing 1.82. `user.js`

```
var User = mongoose.model('User', UserSchema);
module.exports = User;
```

Listing 1.83. `user.js`: end of module

As second step, we add a prehook to our mongoose schema using the bcrypt[15] password hashing function:

```
//hashing a password before saving it to the database;
UserSchema.pre('save', function (next) {
  var user = this;
  bcrypt.hash(user.password, 10, function (err, hash) {
    if (err) {
      return next(err);
    }
    user.password = hash;
    next();
  })
});
```

Listing 1.84. `user.js`: prehook

Now we introduce a method to authenticate input against database:

```
//authenticate input against database;
UserSchema.statics.authenticate = function (email, password, callback) {
  User.findOne({ email: email })
    .exec(function (err, user) {
      if (err) {
        return callback(err)
      } else if (!user) {
        var err = new Error('User not found.');
        err.status = 401;
        return callback(err);
      }
      bcrypt.compare(password, user.password, function (err, result) {
        if (result === true) {
          return callback(null, user);
        } else {
          return callback();
```

[15] bcrypt is based on the Blowfish cipher. Besides incorporating a salt - random data included with the input for the hash function - to protect against rainbow table attacks, bcrypt is an adaptive function: over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power.

```
      }
    })
  });
}
```

Listing 1.85. `user.js`: authenticate

Our application starts by connection to `MongoDB`:

```
/*
 * authentication/app.js
 *
 * npm install bcrypt --save
 * npm install express-session --save
 * npm install connect-mongo --save
 */
var express   = require('express');
var app       = express();
var bodyParser = require('body-parser');
var mongoose  = require('mongoose');
var session   = require('express-session');
var MongoStore = require('connect-mongo')(session);

//connect to MongoDB ;
mongoose.connect('mongodb://localhost/testForAuth');
var db = mongoose.connection;

//handle mongo error ;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function () {
  //connected! ;
});
```

Listing 1.86. `app.js`

Since `HTTP` is a stateless protocol, web servers don't keep track of who is visiting a page. Displaying specific content to logged-in users require this tracking. Therefore we must add a session middleware, by using the `express-session` package for tracking logins:

```
//use sessions for tracking logins ;
app.use(session({
  secret: 'work hard',
  resave: true,
  saveUninitialized: false,
  store: new MongoStore({
    mongooseConnection: db
  })
}));
```

Listing 1.87. `app.js`

Cookies are key/value pairs managed by browsers that correspond with the sessions of the server.

The rest of our server:

```
// serve static files from template ;
app.use(express.static(__dirname + '/templateLogReg'));

// include routes ;
var routes = require('./routes/router');
app.use('/', routes);

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

// error handler as the last app.use callback ;
```

```
app.use(function (err, req, res, next) {
  res.status(err.status || 500);
  res.send(err.message);
});


// listen on port 3000 ;
app.listen(3000, function () {
  console.log('Express app listening on port 3000');
});
```

Listing 1.88. app.js

The router:

```
/*
 * authentication/routes/router.js
 */
var express = require('express');
var router = express.Router();
var User = require('../models/user');


// GET route for reading data ;
router.get('/', function (req, res, next) {
  return res.sendFile(path.join(__dirname + '/templateLogReg/index.html'));
});


//POST route for updating data ;
router.post('/', function (req, res, next) {
  // confirm that user typed same password twice ;
  if (req.body.password !== req.body.passwordConf) {
    var err = new Error('Passwords do not match.');
    err.status = 400;
    res.send("passwords dont match");
    return next(err);
  }

  if (req.body.email &&
    req.body.username &&
    req.body.password &&
    req.body.passwordConf) {

    var userData = {
      email: req.body.email,
      username: req.body.username,
      password: req.body.password,
      passwordConf: req.body.passwordConf,
    }

    User.create(userData, function (error, user) {
      if (error) {
        return next(error);
      } else {
        req.session.userId = user._id;
        return res.redirect('/profile');
      }
    });

  } else if (req.body.logemail && req.body.logpassword) {
    User.authenticate(req.body.logemail, req.body.logpassword, function (error, user) {
      if (error || !user) {
        var err = new Error('Wrong email or password.');
        err.status = 401;
        return next(err);
      } else {
        req.session.userId = user._id;
        return res.redirect('/profile');
      }
    });
  } else {
    var err = new Error('All fields required.');
    err.status = 400;
```

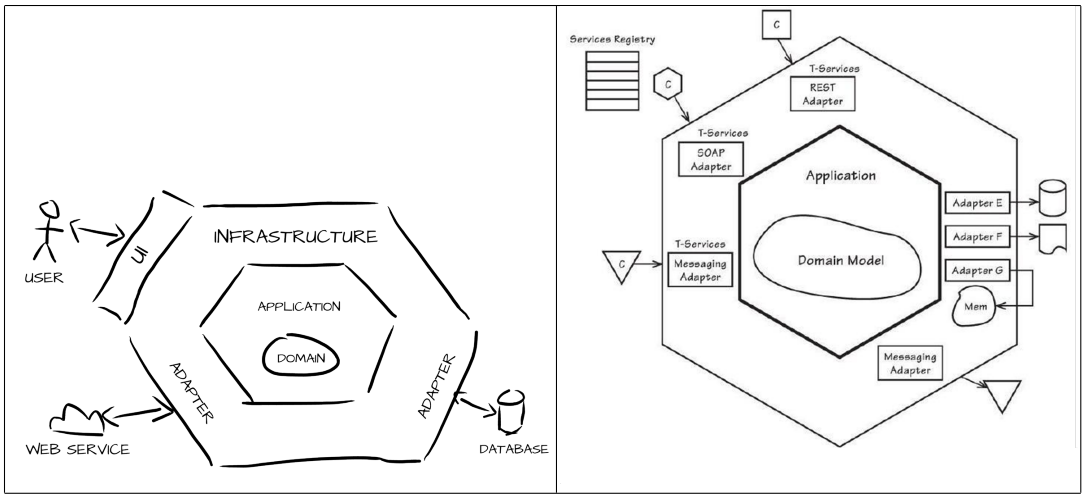
```
60      return next(err);
61    }
62  })
63
64  // GET route after registering ;
65  router.get('/profile', function (req, res, next) {
66    User.findById(req.session.userId)
67      .exec(function (error, user) {
68        if (error) {
69          return next(error);
70        } else {
71          if (user === null) {
72            var err = new Error('Not authorized! Go back!');
73            err.status = 400;
74            return next(err);
75          } else {
76            return res.send('<h1>Name: </h1>' + user.username + '<h2>Mail: </h2>' + user.email + '<br><a
                  type="button" href="/logout">Logout</a>')
77          }
78        }
79      });
80  });
81
82  // GET for logout logout ;
83  router.get('/logout', function (req, res, next) {
84    if (req.session) {
85      // delete session object
86      req.session.destroy(function (err) {
87        if (err) {
88          return next(err);
89        } else {
90          return res.redirect('/');
91        }
92      });
93    }
94  });
95
96  module.exports = router;
```

Listing 1.89. `router.js`

Software-defined networking (SDN) technology is a novel approach to cloud computing that facilitates network management and enables programmatically efficient network configuration in order to improve network performance and monitoring.

11 Towards micro-services

At the core of any modern application is the business logic, which is implemented by modules that define services, domain objects, and events. Surrounding the core are adapters that interface with the external world, according to the modular hexagonal architecture model:



Despite having a logically modular architecture, several applications are packaged and deployed as a monolith, e.g. as `JAR` or `WAR` files. `Node.js` applications are packaged as a directory hierarchy.

Monolithic applications:

- are simple to develop since our `IDE`s and other tools are focused on building a single application;
- simple to test, e.g. by launching the application and testing the UI with Selenium);
- simple to deploy. e.g. by copying the packaged application to a server;
- simple to scale, e.g. by running multiple copies behind a load balancer.

(Un)fortunately, applications usually grow over time and after some time, they become a large, complex monolith, too large for any single developer to fully understand. As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming. What's more, this tends to be a downwards spiral. If the codebase is difficult to understand, then changes won't be made correctly. You will end up with a monstrous, incomprehensible *big ball of mud*. Moreover, complex monolithic application are an obstacle to continuous deployment and difficult to scale when different modules have conflicting resource requirements. Another problem with monolithic applications is reliability: a bug in any module, such as a memory leak, can potentially bring down the entire process. Last but not least, monolithic applications make it extremely difficult to adopt new frameworks and languages

Many organizations, such as Amazon, eBay, and Netflix, have solved this problem by adopting the Microservices Architecture pattern.

11.1 The microservice architectural style

The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating

with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

One way to think about the *Microservices Architecture* pattern is that it's service-oriented architecture (SOA) without the *Web service Specifications* (WS-*) and an *Enterprise Service Bus* (ESB). ESB approaches often include sophisticated facilities for message routing, choreography, transformation, and applying business rules. Microservice-based applications favor simpler, lightweight protocols such as REST, rather than WS-*; they implement ESB-like functionality in the microservices themselves.

The key characteristics of the microservice architecture can be summarized as follows:

- Services. They are the building blocks of modern distributed software systems; they are small, highly decoupled and focus on doing a small task. Moreover, they are software components independently replaceable, upgradeable, and deployable. Service interfaces put restrictions on introducing undesirable tight coupling between components and leaking of functionality from one component into another.
- Messaging. The biggest issue in changing a monolith into microservices lies in changing the *communication pattern*. A naive conversion from in-memory method calls to RPC leads to chatty communications which don't perform well. Instead of using in-memory function calls, microservices interact by exchanging messages by using lightweight communication protocols.

  In the recent years, a number of standards for the physical, network, and transport layers, as well as security mechanisms tailored to resource-constrained IoT devices have been introduced. The recently standardized CoAP and MQTT protocols together with HTTP finalize the protocol stack by building an application layer. Several industrial alliances and research projects are working on integrating these and new standards in different application domains, enabling interoperability among hardware and software vendors, and defining best practices for building large-scale (IoT) platforms and applications. In a messaging based system, both the input and output from services are defined as either *commands* or *events*. Each service subscribes to the events that it is interested in consuming, and then receives these events reliably via a mechanism such as a messaging queue/broker, when the events are placed on the queue by other services.
- Smart endpoints and dump pipes. Microservices use the communication medium to barely exchange messages[16]; whether it is HTTP request response or a lightweight protocols for asynchronous communication with routing, the business logic in microservice architecture always remains in the endpoints - the services. Applications built from microservices aim to be as decoupled and as cohesive as possible their own domain logic and act more as filters in the classical Unix sense receiving a request, applying logic as appropriate and producing a response. These are choreographed using simple REST protocols rather than complex protocols such as WSChoreography or BPEL or orchestration by a central tool.
- Organization around Business Capabilities. Microservice architecture motivates organization around business capabilities instead of the traditional way of building teams based on the technology layers. This results in cross-functional teams, where each team

[16] dumb means that the infrastructure acts as a message router only.

has the full range of skills required for a specific business area and prevents the "logic everywhere" siloed architectures[17].

- Decentralized Governance. Each service in a system built with microservice architecture can use its own technology that is most suitable for the job. This flexibility in the choice of implementation technology provides the benefits of choosing the best tools and platforms considering their trade-offs, as well as allows to gradually adopt new technologies. The centralized governance of standards and technology platforms can be relaxed.
- Decentralized Data Management. Microservice architecture enables decentralized data management, implying decentralization in both the conceptual models and the storage backends used by services. The decentralization in the models means that different components (services) have different conceptual models of the world, e.g., by operating with different attributes of the same entities[18]. The decentralization in the storage backend means that every service has its own, independent, storage subsystem that is isolated from other services.
- Evolutionary Design. Services decomposition is a driving force to enable frequent and controlled changes in a system. On the one hand, limited functionality of small tasks limits the efforts required to introduce changes in individual services. On the other hand, independently deployable and replaceable components together with decentralized governance allow the services to be re-implemented from scratch, possibly using another technology, without affecting the rest of the system.

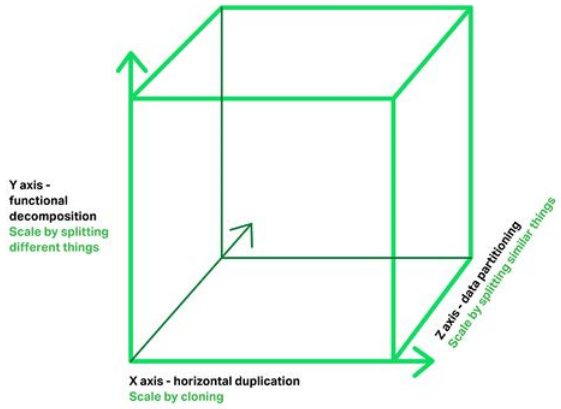
11.2 The benefits

Software systems built with microservice architecture are typically associated with the following benefits:

- Technology heterogeneity. The decentralized governance and data management allows coexistence of different technologies used by different components in the system, leading to *polyglot programming and persistence*.
- Resilience. Components with clear boundaries allow to isolate failures and gradually degrade the system functionality, as well as update and deploy individual services independently.
- Scaling. with reference to the three axis of the scaling cube:

[17] In business management and information technology (IT) a silo describes any management system that is unable to operate with any other system, meaning it's closed off from other systems. Silos create an environment of individual and disparate systems within an organization.

[18] With the *Command Query Responsibility Segregation* (CQRS) you can use a different model to update information than the model you use to read information. For some situations, this separation can be valuable, but for most systems CQRS adds risky complexity.



in addition to the typical scaling by horizontal duplication (`X-axis`) and data partitioning (`Z-axis`), microservices also enable scaling by functional decomposition (`Y-axis`)

- `Organizational alignment`. Organization around business capabilities motivates smaller focused teams working on components with smaller code-bases.
- `Composability`. New system capabilities can be created by composing and re-using existing services.

Each functional area of a (huge) application can be implemented by its own microservice. Thus, each *microservice is a mini-application* that has its own hexagonal architecture consisting of business logic along with various adapters. Some microservices would expose an `API` that's consumed by other microservices or by the application's clients. Other microservices might implement a web `UI`. This makes it easier to deploy `distinct experiences` for specific users, devices, or specialized use cases.

Finally, at runtime, each instance is often a cloud `VM` or a *Docker* container.

Thanks to its scalability, this architectural method is considered particularly ideal when you have to enable support for a range of platforms and devices-spanning web, mobile, Internet of Things, and wearables - or simply when you're not sure what kind of devices you'll need to support in an increasingly cloudy future.

Two of the most important things to consider when designing a microservices based system are: *i)* the ability to deal with change such as adding, removing or modifying services without affecting the operation or code of other services, *ii)* gracefully handling service failure.

The reason why REST based microservices examples are most popular is more than likely due to their simplicity; services communicate directly and synchronously with each other over HTTP, without the need for any additional infrastructure.

11.3 The challenges

- The `term microservice`. The term *microservice* places excessive emphasis on service size. The goal of microservices is to sufficiently decompose the application in order to facilitate agile application development and deployment.
- `Partitioned database` architecture.
  The Microservices Architecture pattern significantly impacts the relationship between the application and the database. Rather than sharing a single database schema with other services, each service has its own database schema. On the one hand, this approach

is at odds with the idea of an enterprise-wide data model. Also, it often results in duplication of some data. However, having a database schema per service is essential if you want to benefit from microservices, because it ensures loose coupling.
Thus, in a microservices-based application,we need to update multiple databases owned by different services. Using distributed transactions is usually not an option, and not only because of the CAP theorem. They simply are not supported by many of today's (highly scalable NoSQL) databases and messaging brokers. You end up having to use an eventual consistency based approach, which is more challenging for developers.

– A microservices application is a distributed system. Software design and coding is much more complex than in a monolithic applications where modules invoke one another via language-level method/procedure calls. Some key-point are:
  - Designers must introduce code to handle partial failure since the destination of a request might be slow or unavailable.
  - Cohesion must b obtained via a shared knowledge of queue names, and a consistent and well known command/event format; an event or command fired by one service should be able to be consumed by the subscriber services.
  - A major challenge with the Microservices Architecture pattern is implementing changes that span multiple services.
  - Due to the dispersed and autonomous nature of messaging based microservices, it can be difficult to fully get a clear view of the flow of messages within the system.
  - Testing a microservices application is much more complex.
  - When designing software systems with microservices, deployment and operational aspects of the resulting systems need to be considered carefully. The growing popularity of DevOps culture and the infrastructure automation tools address many of these challenges.

11.4 A first example of a microservice

As an introductory example to the microservice world, let us define a service that has a very simple functional goal: to allow users to read/write some data. The serivce is implemented here as a Node server with sockets that implements the HTTP verbs GET and PUT and that notifies each PUT to subscribed clients. A client can be a program written in any language or a browser showing to end user an HTML page provided by the server.

The server can be introduced as a typical express application (see Section ??) that handles (for the sake of simplicity) a single integer value:

```
1  /*
2   * singleValueAppl/singleValueServer.js
3   * npm install body-parser --save
4   */
5  var url        = require('url');
6  var express    = require('express');
7  var http       = require('http');
8  var path       = require('path');
9  var bodyParser = require('body-parser');
10
11 var app        = express();
12 var server     = http.createServer(app);
13 var io         = require('socket.io').listen(server); //Wrap;
14
15 var value  = 0;
16
17 app.use( bodyParser.json( ) );
18
19 server.listen(8080, function(){console.log("bound to port 8080")});
```

Listing 1.90. singleValueAppl/singleValueServer.js

A GET request is handled by returning to the caller the current value:

```
app.get("/", function(req,res,next){
    var path = url.parse(req.url).pathname;
    console.log( "GET path=" + path );
    res.send( "Server answer tp GET:"+ value );
});
```

Listing 1.91. singleValueAppl/singleValueServer.js: GET

A PUT request is handled by returning to the caller some answer and by emitting (pushing) a messages to the subscribers:

```
app.put("/", function(req,res,next){
    var path = url.parse(req.url).pathname;
    console.log( "PUT path=" + path + " req.body.value=" + req.body.value);
    value = req.body.value;
    io.sockets.emit('message', value);  //Push ;
    res.send( "Server answer to PUT:"+ value );
});
```

Listing 1.92. singleValueAppl/singleValueServer.js: GET

11.4.1 Testing .

```
curl -H "Content-Type: application/json" -X PUT -d "{\"value\": \"50\" }" http://localhost:8080
ANSWER: Server answer to PUT:50

curl -X GET http://localhost:8080
ANSWER: Server answer to GET:50
```

11.4.2 A client in Node .

```
/*
 * singleValueAppl/singleValueClient.js
 * npm install socket.io-client --save
*/
var request     = require("request");
var urlServerAddr = 'http://localhost:8080';
var io          = require('socket.io-client');

readTheValue = function(){
    //console.log(" --- singleValueClient readTheValue from " + urlServerAddr );
    var options = { method: 'GET',
              url: urlServerAddr,
              json: false };
            request(options, function (error, response, body) {
              if (error) throw new Error(error);
              console.log("ANSWER="+body);
            });
};


setTheValue = function(v, jsonData){
    var outdata;
    if ( jsonData ) outdata = v;
    else outdata = { 'value': v };
    var options = { method: 'PUT', url: urlServerAddr,
           body: outdata,
           json: true
       };
    request(options, function (error, response, body) {
      if (error) throw new Error(error);
      console.log("ANSWER to PUT from server:" + body);
```

```
    });
};


init = function(){
    var conn =io.connect(urlServerAddr);
    conn.on('connect', function(){ console.log("singleValueClient connected to socket"); } );
    conn.on('message', function(v){ console.log("   Notified of current value:"+ v); } );
}


init();
setTimeout( readTheValue, 1000) ;
setTimeout( function(){setTheValue({value: 10}, true); }, 1500) ;
setTimeout( readTheValue, 2000) ;
setTimeout( function(){setTheValue(20, false); }, 2500) ;
setTimeout( readTheValue, 3000) ;
```

Listing 1.93. singleValueAppl/singleValueClient.js

The output is:

```
singleValueClient connected to socket
ANSWER=Server answer tp GET:50
       Notified of current value:10
ANSWER to PUT from server:Server answer to PUT:10
ANSWER=Server answer tp GET:10
       Notified of current value:20
ANSWER to PUT from server:Server answer to PUT:20
ANSWER=Server answer tp GET:20
```

11.4.3 A client in Java .

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

public class RestClientHttp {
    public static int sendPut(String data, String url) {
        int responseCode = -1;
        CloseableHttpClient httpclient = HttpClients.createDefault();
        try {
            HttpPut request = new HttpPut(url);
            StringEntity params =new StringEntity(data,"UTF-8");
            params.setContentType("application/json");
            request.addHeader("content-type", "application/json");
            request.addHeader("Accept", "*/*");
            request.addHeader("Accept-Encoding", "gzip,deflate,sdch");
            request.addHeader("Accept-Language", "en-US,en;q=0.8");
            request.setEntity(params);
            CloseableHttpResponse response = httpclient.execute(request);
            responseCode = response.getStatusLine().getStatusCode();
```

```
37            if (response.getStatusLine().getStatusCode() == 200 || response.getStatusLine().getStatusCode()
                  == 204) {
38                BufferedReader br = new BufferedReader(new
                      InputStreamReader((response.getEntity().getContent())));
39                String output;
40                String info = "";
41                while ((output = br.readLine()) != null) {
42                    info = info + output;
43                }
44                System.out.println(info);
45            }
46            else{ throw new RuntimeException("Failed : HTTP error code : "
47                        + response.getStatusLine().getStatusCode());
48            }
49
50        }catch (Exception ex) {
51        } finally {//        httpclient.close();
52        }
53        return responseCode;
54    }
55
56    public static void connectPost(){
57        CloseableHttpClient httpclient = HttpClients.createDefault();
58        HttpPost httpPost = new HttpPost("http://localhost:8080");
59        List <NameValuePair> nvps = new ArrayList <NameValuePair>();
60        nvps.add(new BasicNameValuePair("username", "vip"));
61        nvps.add(new BasicNameValuePair("password", "secret"));
62        try {
63            httpPost.setEntity(new UrlEncodedFormEntity(nvps));
64            CloseableHttpResponse response2 = httpclient.execute(httpPost);
65            HttpEntity entity2 = response2.getEntity();
66            // do something useful with the response body and ensure it is fully consumed
67            EntityUtils.consume(entity2);
68          } catch ( Exception e) {
69              e.printStackTrace();
70        }
71    }
72    public static void connectGet(){
73     try {
74        CloseableHttpClient httpclient = HttpClients.createDefault();
75        HttpGet httpGet = new HttpGet("http://localhost:8080");
76        CloseableHttpResponse response = httpclient.execute(httpGet);
77        if (response.getStatusLine().getStatusCode() != 200) {
78            throw new RuntimeException("Failed : HTTP error code : "
79               + response.getStatusLine().getStatusCode());
80        }
81        BufferedReader br = new BufferedReader(
82                         new InputStreamReader((response.getEntity().getContent())));
83        String output;
84        String info = "";
85        while ((output = br.readLine()) != null) {
86            info = info + output;
87        }
88        System.out.println(info);
89      } catch ( Exception e) { e.printStackTrace(); }
90    }
91    public static void work() {
92        connectGet();
93        sendPut("{\"value\":28}", "http://localhost:8080");
94        connectGet();
95    }
96    public static void main (String args[]) throws InterruptedException{
97        work();
98    }
99 }
```

Listing 1.94. `RestClientHttp.java`

The output is:

```
1 Server answer tp GET:20
2 Server answer to PUT:28
3 Server answer tp GET:28
```

- *If you wish to converse with me, define your terms.* (Voltaire).
- *Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.* (Conway's Law)
- *Be of the web, not behind the web* (Ian Robinson)
- REST (RPC)
- Hexagonal monolithic
- Message passing
- Events
- Mechanisms / architectures
- Applications building and testing