Web applications with NodeJS

Part III

Designing NodeJS applications

- We learned in the previous lessons:

Dictionary of actions

- Mapping requests to handler functions

```
var handle = {};
handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;
```

Functions as values in a dictionary

- This dictionary of available actions is passed to

```
server.start(router.route, handle);
```

Function as a parameter

Dictionary of functions

requestHandler.js functions: first attempt

```
function start() {
        console.log("Hello start");
}

function upload() {
        console.log("Hello upload");
}
```

router.js

- Router is calling a corresponding function from the action dictionary, depending on url path

```
function route(handle, path) {
        handle[path]();
}
```

First, we were writing response in *server.js*

- We need to return specific content to server.js so it can write it using the response object

```
function start ()
{
        …
        content = route (handle, pathname);
        response.write (content);
```

router.js was changed: function *route()* returns value to *server.js*

```
function route (path)
{
        …
        return handle[path]();
```

requestHandler.js
functions changed to also return values

```
function start() {
        return “Hello start”;


function upload() {
        return “Hello upload”;
```

And then we encountered a problem

- Our application was able to transport the content (which the request handlers would like to display to the user) from the request handlers to the HTTP server by returning it up through the layers of the application (request handler -> router -> server).
- But heavy **synchronous** I/O in start()
  blocks all the other requests

If we block *start()*…

```
function start()
        sleep (10000);
        return “Hello start”;

function upload()
        return “Hello upload”;
```

upload() is blocked as well

We changed: making heavy I/Os **asynchronous – non-blocking**

```
function start()
        Asynchronous I/O with callback
        return result of asynchronous operation

function upload()
        return “Hello upload”;
```

In order to send response asynchronously

- We need to write response in responseHandler inside callback function, when asynchronous operation is complete
- Redesigned: no return values, passing ***response*** object to the handler: server -> router ->handler
- We passed the *response* object (from our server's callback function *onRequest()*) through the router into the request handlers.
- The handlers are now able to use this object's methods to respond to requests themselves and in the appropriate time.

server.js

- We pass *response* from server.js, so it can be used when processing is done

```
start ()
        …
        route (handle, pathname, response);
```

router.js

- Router pass *response* further to requestHandler

```
function route()

        …
        handle[pathname](response);
```

requestHandler.js

- Handler writes the response when it is done processing I/O

```
function start(response)
        Asynchronous I/O
        inside callback function:
                response.write(result)

function upload(response)
        response.write("Hello upload") ;
```

Part III

FINISHING CASE STUDY

Handling incoming requests

- GET
- POST

Reminder: handling GET requests

http://localhost:8888/start?foo=bar&hello=world

```
var url = require ("url");
var url_parts = url.parse(request.url, true);
var pathname = url_parts.pathname; /start
var query = url_parts.query; foo=bar&hello=world
var foovalue = query.foo; bar
var hellovalue = query.hello; world
```

Handling POST requests

- Simple example: reading content of a *textarea* filled by the user and submitted to the server via POST request.
- Upon receiving and handling this request, we will display the content of the *textarea* back to the user.

Displaying form with textarea on *start()* in *requestHandler.js*

```
function start(response) {
  var body = '<html>'+
   '<head>'+
   '<meta charset="UTF-8" />'+
   '</head>'+
   '<body>'+
    '<form action="/upload" method="post">'+
     '<textarea name=" txtArea" rows="20" cols="60"></textarea>'+
     '<input type="submit" value="Submit text" />'+
    '</form>'+
   '</body>'+
   '</html>';
   response.writeHead(200, {"Content-Type": "text/html"});
   response.write(body);
   response.end();
}
```

POST requests are handled asynchronously

- The POST request will hit our */upload* request handler when the user submits this form.
- Handling POST data is done in a non-blocking fashion, by using asynchronous callbacks.
- This makes sense, because POST requests can potentially be very large - nothing stops the user from entering text that is multiple megabytes in size. Handling the whole bulk of data in one go would result in a blocking operation.

Non-blocking POSTs

- To make the whole process non-blocking, Node.js serves the POST data in small chunks
- Callback functions are called upon certain events:
  - *data* (an new chunk of POST data arrives)
  - *end* (all chunks have been received)

Listening to data transfer events

```
request.addListener("data", function(chunk) {
  // called when a new chunk of data was received
});

request.addListener("end", function() {
  // called when all chunks of data have been received
});
```

- We need to tell Node.js which functions to **call back** to when these events occur.
- This is done by adding *listeners* to the *request* object that is passed to our *onRequest* callback whenever an HTTP request is received.

Collecting POST data in chunks - server.js

```
function start(route, handle) {
  function onRequest(request, response) {
    var postData = "";
    var pathname = url.parse(request.url).pathname;
    request.setEncoding("utf8");
    request.addListener("data", function(postDataChunk) {
        postData += postDataChunk;
        console.log("Received POST data chunk '"+postDataChunk +
                                                          "'.");
    });
    request.addListener("end", function() {
        route(handle, pathname, response, postData);
    });
  }
```

Collecting **all** POST data and passing it to the router

```
function start(route, handle) {
  function onRequest(request, response) {
    var postData = "";
    var pathname = url.parse(request.url).pathname;
    request.setEncoding("utf8");
    request.addListener("data", function(postDataChunk) {
        postData += postDataChunk;
        console.log("Received POST data chunk '"+postDataChunk + "'.");
    });
    request.addListener("end", function() {
        route(handle, pathname, response, postData);
    });
  }
```

Passing postData in *router.js*

```
function route(handle, pathname, response, postData)
{
    handle[pathname](response, postData);
}
```

"/upload"

Collected in server.js, and handle is called when all data has been received

requestHandler.js

- in *requestHandlers.js*, we include the data in our response of the */upload* request handler:

```
function upload(response, postData) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("You've sent: " + postData);
    response.end();
}
```

Parsing out individual fields of *postData*

- To parse individual fields of posted data: we use the *querystring* module:

```
var querystring = require("./querystring");
querystring.parse(postData).txtArea
```

Name of textarea control

Serving images. File system

Module **fs** – file system

- In order to serve image file to be displayed on request we add a new handler in *requestHandlers.js*

```
function show(response) {
  fs.readFile("/tmp/test.png", "binary", function(error, file) {
        response.writeHead(200, {"Content-Type": "image/png"});
        response.write(file, "binary");
        response.end();
  }
 });
}
```

Adding a new action to a dictionary

```
var handle = {}
handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;
handle["/show"] = requestHandlers.show;
```

- Made easy due to loose coupling

Result

By restarting the server and opening http://localhost:8888/show in the browser, the image file saved at */tmp/test.png* should be displayed.

Handling file uploads

- All the details of parsing incoming file data are abstracted in *node-formidable* module by Felix Geisendörfer.

```
npm install formidable
```

Using *formidable* library

```
var formidable = require("formidable");
```

- Next we need to create a new *IncomingForm* object, which is an abstraction of the submitted form
- This object can then be used to parse the *request* object of our HTTP server for the fields and files that were submitted through this form.

The example code (from the node-formidable project page)

- Referencing libraries

```
var formidable = require('formidable'),
        http = require('http'),
        sys = require('sys');
```

The example code

- Showing a file upload form

```
http.createServer(function(req, res) {


...
  // by default
  res.writeHead(200, {'content-type': 'text/html'});
  res.end(
    '<form action="/upload" enctype="multipart/form-data" '+
                                                  'method="post">'+
    '<input type="text" name="title"><br>'+
    '<input type="file" name="upload" multiple="multiple"><br>'+
    '<input type="submit" value="Upload">'+
    '</form>'
  );
}).listen(8888);
```

The example code

- Inspecting and parsing uploaded form

```
http.createServer(function(req, res) {
 if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
        var form = new formidable.IncomingForm();
        form.parse(req, function(err, fields, files) {
                res.writeHead(200, {'content-type': 'text/plain'});
                res.write('received upload:\n\n');
                res.end(sys.inspect({fields: fields, files: files}));
  });
  return;
 }


 …
}).listen(8888);
```

Adding file upload functionality to our code

1. Add a file upload element to the form which is served at */start*,
2. Integrate node-formidable into the *upload* request handler, in order to save the uploaded file to */tmp/test.png*,
3. Embed the uploaded image into the HTML output of the */upload* URL.

Changes in design

- We want to handle the file upload in our */upload* request handler, and there, we will need to pass the *request* object to the form.parse call of node-formidable.

- But all we have is the *response* object and the postData. Looks like we will have to pass the request object all the way from the server -> to the router -> to the request handler.

server.js

- Let's start with *server.js* - we remove the postData handling, and we pass *request* to the router instead:

```
function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    route(handle, pathname, response, request);
  }


...
}
```

router.js

- We don't need to pass *postData* on anymore, and instead pass *request*:

```
function route(handle, pathname, response, request) {
    handle[pathname](response, request);
}
```

What we have learned

1. Storing functions in a dictionary, expanding functionality with adding new functions (loosely coupled objects JavaScript way)
2. Programming with non-blocking functions using callbacks
3. Writing web server with the full handling of GET and POST requests. Passing on request and response objects is probably a good idea, to ensure the asynchronous treatment.
4. Using Node libraries to handle difficult tasks

- This tutorial is based on the book:

http://www.nodebeginner.org/

- The code is available at:

https://github.com/ManuelKiessling/NodeBeginnerBook/tree/master/code/application

- The Node.js community wiki and the NodeCloud directory are probably the next points for more information.