Web applications with NodeJS

Part II

Back to HTTP server

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

function onRequest(request, response) {
 }

http.createServer(onRequest).listen(8888);
```

- Node HTTP server needs an asynchronous callback function it can call upon incoming requests.
- If it were synchronous, and if a second user requests the server while it is still serving the first request, that second request could only be answered after the first one is done.
- After all, Node.js is just one single process, and it can run on only one single CPU core.

One thread – advantage or limitation?

“NodeJS model allows to write applications that have to deal with concurrency in an efficient and relatively straightforward manner.“

Test yourself: What is executed first?

```
1.    var http = require("http");

      function onRequest(request, response) {
2.     console.log("Request received.");
3.     response.writeHead(200, {"Content-Type": "text/plain"});
4.     response.write("Hello World");
5.     response.end();
      }


6.    http.createServer(onRequest).listen(8888);

7.    console.log("Server has started.");
```

Task 2

ROUTING REQUESTS

Handling requests

- When the callback function *onRequest()* gets triggered, two parameters are passed into it: *request* and *response*.
- Those are objects, and we can use their methods to handle the details of the HTTP request and to respond to the request (i.e., to actually send something over the wire back to the browser that requested your server).
- In our exampe: whenever a request is received, it uses the *response.writeHead()* function to send an HTTP status 200 and content-type in the HTTP response header, and the *response.write()* function to send the text "Hello World" in the HTTP response body.
- At last, we call *response.end()* to actually finish our response.

Handling requests

```
function onRequest(request, response) {
            console.log("Request received.");
            response.writeHead(200, {"Content-Type": "text/plain"});
            response.write("Hello World");
            response.end();
           }
```

- Whenever a request is received:
  - it uses the *response.writeHead()* function to send an HTTP status 200 and content-type in the HTTP response header.
  - It uses the *response.write()* function to send the text "Hello World" in the HTTP response body.
  - at last, we call *response.end()* to indicate that this is the end of our response.

Digression

WRITING NODE MODULES. END OF GLOBAL VARIABLES

Using modules in Node

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

...

http.createServer(...);
```

- There is a module called "http", we can make use of it in our code by requiring it and assigning the result of the require to a local variable.
- This makes our local variable an object that carries all the public methods the *http* module provides.
- It's common practice to choose the name of the module for the name of the local variable, but we are free to choose whatever we like:

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

...

foo.createServer(...);
```

Creating our own modules: *server.js*

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

function start() {
  function onRequest(request, response) {
    console.log("Request received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}
```

```
exports.start = start;
```

Make function *start* visible outside our module.

Calling our function from another module

- In file *index.js*:

```
var server = require("./server");

server.start();
```

- Now we can put the different parts of our application into different files and wire them together by making them modules.

Global object

- One of the main disadvantages of JavaScript is the Global Object.
- Since web applications may have lot of objects, JavaScript could be a minefield of conflicting global objects.
- Node uses the CommonJS **module system**: variables local to a module are truly local to this module.
- This clean separation between modules prevents the Global Object problem from being a problem.

Handling requests

- Depending on which URL the browser requested from our server, we need to react differently.
- Making different requests point at different parts of our code is called "routing“.
- We need to look into the HTTP requests and extract the requested URL as well as the GET/POST parameters from them.
- We need to feed this data into our router, and based on these, the router then needs to decide which code to execute.

Request object

- All the information we need is available through the *request* object which is passed as the first parameter to our callback function *onRequest()*.
- To interpret this information, we need some additional Node.js modules, namely *url* and *querystring*.

url module

```
                               url.parse(string).query
                                           |
           url.parse(string).pathname      |
                       |                   |
                       |                   |
                     ------ -------------------
http://localhost:8888/start?foo=bar&hello=world
                                ---       -----
                                 |          |
                                 |          |
              querystring(string)["foo"]    |
                                            |
                                            |
                         querystring(string)["hello"]
```

Parsing path name

```
function onRequest(request, response) {
   var pathname = url.parse(request.url).pathname;
   console.log("Request for " + pathname + " received.");
   response.writeHead(200, {"Content-Type": "text/plain"});
   response.write("Hello World");
   response.end();
 }
```

- Our application can now distinguish requests based on the URL path requested.
- This allows to map requests to request handlers based on the URL path using our (yet to be written) router.

(In the context of our application, it simply means that we will be able to have requests for the */start* and */upload* URLs handled by different parts of our code).

router.js

```
function route(pathname) {
  console.log("About to route a request for " + pathname);
}

exports.route = route;
```

Wiring router into a server

- Our HTTP server needs to know about and make use of our router.
- We could hard-wire router object into the server, but instead we are going to **loosely couple** server and router by injecting this dependency (Martin Fowlers post on Dependency Injection).

Passing route as a parameter to server start

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

  route(pathname);

  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
 }

 http.createServer(onRequest).listen(8888);
 console.log("Server has started.");
}
```

index.js

```
var server = require("./server");
var router = require("./router");

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

- And here we are passing an application-specific router.

Digression: The Kingdom of verbs

- In our index file, we could have passed the *router* object into the server, and the server could have called this object's *route* function.
- This way, we would have passed a *thing*, and the server would have used this thing to *do* something.
- But the server doesn't need the thing. It only needs to get something *done*, and to get something done, you don't need things at all, you need *actions*. You don't need *nouns*, you need *verbs*.
- Unlike regular OOP, where everything is a noun, this is a functional programming.

Execution in the Kingdom of Nouns.

Task 3

REQUEST HANDLERS

Module requestHandlers

```
function start() {
  console.log("Request handler 'start' was called.");
}

function upload() {
  console.log("Request handler 'upload' was called.");
}

exports.start = start;
exports.upload = upload;
```

Making handlers extendable

- In a real application, the number of handlers may be very large.
- A varying number of items, each mapped to a string (the requested URL)? Sounds like an associative array.

- JavaScript doesn't provide associative arrays - or does it?

Reminder: JavaScript objects vs. conventional objects

- In C++ or C#, when we’re talking about objects, we're referring to instances of classes or structs.
- Objects have different properties and methods, depending on which templates (that is, classes) they are instantiated from.
- In JavaScript, objects are just collections of name/value pairs - think of a JavaScript object as a dictionary with string keys.
- If JavaScript objects are just collections of name/value pairs, how can they have methods? Well, the values can be strings, numbers etc. - or functions!

Mapping requests to handler functions

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

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

Passing *handle* dictionary to server *start()*

```
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.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }
```

In *router.js*

```
function route(handle, pathname) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname]();
  } else {
    console.log("No request handler found for " + pathname);
  }
}
```

- We check if a request handler for the given pathname exists, and if it does, we simply call the corresponding function, pulled out of the handle dictionary.

Making the request handlers respond

```
function start() {
  console.log("Request handler 'start' was called.");
  return "Hello Start";
}

function upload() {
  console.log("Request handler 'upload' was called.");
  return "Hello Upload";
}
```

Returning content from router based on requested path

```
function route(handle, pathname) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    return handle[pathname]();
  } else {
    console.log("No request handler found for " + pathname);
    return "404 Not found";
  }
}
```

For path “/start”, this will call function *start* which returns string “Hello start”

In server.js

```
response.writeHead(200, {"Content-Type": "text/plain"});
    var content = route(handle, pathname)
    response.write(content);
    response.end();
```

Here, *route* will return a corresponding text, which we store in variable *content* and write the dynamic message back to client.

Do we see the problem with our design?

What if one of the request handlers wants to make use of a non-blocking operation in the future?

Sleep for 10 seconds when start is requested

```
function start() {
  console.log("Request handler 'start' was called.");

  function sleep(milliSeconds) {
    var startTime = new Date().getTime();
    while (new Date().getTime() < startTime + milliSeconds);
  }

  sleep(10000);
  return "Hello Start";
}

function upload() {
  console.log("Request handler 'upload' was called.");
  return "Hello Upload";
}
```

We think that:

- When the function *start()* is called, Node.js waits 10 seconds and only then returns "Hello Start".
- When calling *upload()*, it returns immediately, just like before.

(Of course, you should imagine that instead of sleeping for 10 seconds, there would be a real blocking operation in *start()*, like some sort of a database query.)

That is not what happens

- If you run the app in 2 different browsers, and request */start* in one and */upload* in another, you will notice that:
  - The */start* URL takes 10 seconds to load, as we would expect.
  - But the */upload* URL **also** takes 10 seconds to load, although there is no *sleep()* in the corresponding request handler.

Why?

- Because *start()* contains a blocking operation.
- For the Node execution model expensive operations are ok, but we cannot block the entire single Node.js process with them.
- Instead, whenever expensive operations must be executed, they must be put in the background, and their events must be handled by the event loop.

Introducing a non-blocking operation

```
var exec = require("child_process").exec;
function start() {
  console.log("Request handler 'start' was called.");
  var content = "empty";

  exec("ls -lah", function (error, stdout, stderr) {
    content = stdout;
  });

  return content;
}
```

- What *exec()* does: it executes a shell command from within Node.js.
- In this example, we use it to get a list of all files in the current directory ("ls -lah"), allowing us to display this list in the browser of a user requesting the */start* URL.

New problems

- What the code does is: creates a new variable *content* (with an initial value of "empty"), executes "ls -lah", fills the variable with the result, and returns it.
- If you try it, it loads a beautiful web page that displays the string "empty". Why?

What is the problem?

- *exec()* does its magic in a non-blocking fashion. That's a good thing, because we can execute very expensive shell operations without forcing our application into a full stop as the blocking *sleep* operation did.
- However, *return content* line is executed **before** the asynchronous call to *exec()* finished, and the result is still empty when it is returned.

Why it does not work

- *exec()*, in order to work non-blocking, makes use of a callback function.
- In our example, it's an anonymous function which is passed as the second parameter to the *exec()* function call:

```
exec("ls -lah", function (error, stdout, stderr) {
    content = stdout;
  });
```

- *exec()* does something in the background, while Node.js itself continues with the application
- the callback function we passed into *exec()* will be called only when the *exec* has finished running.

NO to the synchronous code!

- Our own code is being executed synchronously, which means that immediately after calling *exec()*, Node.js continues to execute *return content;*.
- To fix the problem, we need to return the response <u>after</u> *exec* has finished.

Fixing the problem: passing *response* object to the router

```
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);
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}
```

- Instead of expecting a return value from the *route()* function, we pass it a third parameter, our *response* object.
- Furthermore, we removed any *response* method calls from the *onRequest()* handler, because we now expect *route* to take care of that.

In *router.js*: passing response to the handler

```
function route(handle, pathname, response) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response);
  } else {
    …
  }
}
```

- Same pattern: instead of expecting a return value from our request handlers, we pass the *response* object on.

In *requestHandlers.js*: respond inside the callback function

```
var exec = require("child_process").exec;

function start(response) {
  console.log("Request handler 'start' was called.");

  exec("ls -lah", function (error, stdout, stderr) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write(stdout);
    response.end();
  });
}
```

All is non-blocking

Finally, an expensive operation behind */start* will no longer block requests for */upload* from being answered immediately

Longer operation, but still non-blocking

```
var exec = require("child_process").exec;

function start(response) {
  console.log("Request handler 'start' was called.");

  exec("find /",
    { timeout: 10000, maxBuffer: 20000*1024 },
    function (error, stdout, stderr) {
      response.writeHead(200, {"Content-Type": "text/plain"});
      response.write(stdout);
      response.end();
    });
}
```