[HOME](http://theholmesoffice.com/)[SUBSCRIBE](http://theholmesoffice.com/rss/)

**Creating an MVC framework for our Node.js page - getting ready for scalability**

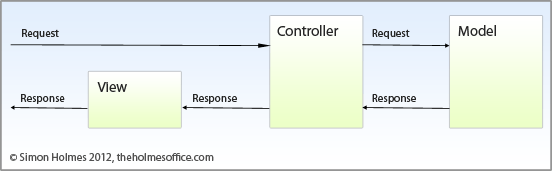
09 JULY 2012 on [mvc](http://theholmesoffice.com/tag/mvc/), [node.js](http://theholmesoffice.com/tag/node-js/), [nodejs](http://theholmesoffice.com/tag/nodejs/)

Having got our [simple webpage working in Node.js](http://theholmesoffice.com/how-to-build-a-simple-webpage-in-node-js/), it’s now time to think bigger. But we can’t very well create an entire website in just one file, so let’s look at how to create a framework upon which we can base our future development.

If you’ve spent much time as a developer you’ll know that the key to maintainability is separation of concerns across your projects. If you’ve been developing a long time, since before such architectures were really discussed, then you probably learned the hard way. Anyhow, what we mean, and what we want to achieve is to separate out our data, logic and presentation.

**MVC – The Model-View-Controller approach**

A very popular approach for web sites & web apps – and the one we’ll be working with – is MVC, which stands for model, view, controller. This makes distinctions between the model (data), view (presentation) and controllers (logic).

The diagram below shows how a standard MVC request/response loop looks:   


In short these are the steps:

1. Request comes in to the Controller
2. Controller makes a request to the model
3. Model returns a response to the controller
4. The controller (normally) performs some operations using the response from the model
5. The controller sends a response to the view
6. The view renders the response

In Node.js, because we are also creating the webserver too, we need to add three steps in at the beginning:

1. Request comes in to server
2. Server sends request to router
3. Router sends request to appropriate controller

Looking at this, we can see that we are going to need the following objects (or collections of objects):

1. Server – to listen to and respond to HTTP requests
2. Router – to send the incoming requests to the correct controller
3. Controllers – to perform operations & interrogate the data
4. Model – to provide the data
5. Views – to provide the HTML rendering we’re going to see in the browser

**Building our framework**

To keep things clean, lets create a new folder to put our site in.

**Creating the node.js server**

Create a file called server.js with the following content:

var http\_IP = '127.0.0.1';

var http\_port = 8899;

var http = require('http');

var server = http.createServer(function(req, res) {

require('./router').get(req, res);

}); // end server()

server.listen(http\_port, http\_IP);

console.log('listening to http://' + http\_IP + ':' + http\_port);

The only new thing there, is that rather than outputting anything directly here, we are calling in a ‘router’ module and sending it the request & response object. This makes sense if you look back at the steps the application needs to take: *server sends request to router.*

**Building the router**

In the same folder as the server.js file, create a new file called router.js and save it with the following content:

var url = require('url');

var fs = require('fs');

exports.get = function(req, res) {

req.requrl = url.parse(req.url, true);

var path = req.requrl.pathname;

if (/.(css)$/.test(path)) {

res.writeHead(200, {

'Content-Type': 'text/css'

});

fs.readFile(\_\_dirname + path, 'utf8', function(err, data) {

if (err) throw err;

res.write(data, 'utf8');

res.end();

});

} else {

if (path === '/' || path === '/home') {

require('./controllers/home').get(req, res);

} else {

require('./controllers/404').get(req, res);

}

}

};

In the router we are doing a few different things:

1. Importing the URL and FileSystem (fs) modules that come as part of your node.js install
2. Exporting a function called “get” – this allows our server file to use this function, passing the request and response objects through
3. Getting the path of the URL request
4. Testing to see whether the request is for a CSS file and loading it in if so. There are a number of different ways of handling static files, but this method is fine for what we’re doing here
5. Sending the request to the correct controller, if it’s not for a CSS file

At this point in our development, it’s number 5 we are interested in. We are checking for two URL paths ‘/’ and ‘/home’ which will route to a controller called ‘home’. We then send any other URL requests to a controller called ’404′. Let’s take a look at these controllers.

**Building the controllers**

Create a folder in the root of your site, called ‘controllers’. Inside this folder create a file called ‘home.js’ with the following content:

var template = require('../views/template-main');

var test\_data = require('../model/test-data');

exports.get = function(req, res) {

var teamlist = test\_data.teamlist;

var strTeam = "",

i = 0;

for (i = 0; i < teamlist.count;) {

strTeam = strTeam + "<li>" + teamlist.teams[i].country + "</li>";

i = i + 1;

}

strTeam = "<ul>" + strTeam + "</ul>";

res.writeHead(200, {

'Content-Type': 'text/html'

});

res.write(template.build("Test web page on node.js", "Hello there", "<p>The teams in Group " + teamlist.GroupName + " for Euro 2012 are:</p>" + strTeam));

res.end();

};

Right at the top of this file you can see that we are including the rest of our MVC structure the *model* and the *view*. Referring back to the diagram we remember that the controller should do two main things:

1. The controller (normally) performs some operations using the response from the model
2. The controller sends a response to the view

**Performing operations on model data**

Our task here is quite simple: get a data object (*teamlist*) and create a HTML string putting the items into an unordered list. In the next step we’ll see how we create this data.

**Sending a response to the view**

We will use a view (think template) called template-main, which we will create in a few minutes. Note that through the template.build function that we are only sending three content items; within the controller there is no reference to how this content will be displayed. In this example you will notice that we are sending through the HTML for the <ul> list, there are arguments against having even this level of markup in a controller, but we’re going for a simple example here so let’s keep it in!

**Getting data from the model**

In a future post we’ll see how we can extend this demo by getting the data out of a database, but for the sake of remaining focused on the point right now we’ll fake it by creating and returning a JSON object.

So, create a folder in the root of your site called ‘model’. Inside this folder create a new file called ‘test-data.js’ with the following content:

var thelist = function() {

var objJson = {

"GroupName": "D",

"count": 4,

"teams": [{

"country": "England"

}, {

"country": "France"

}, {

"country": "Sweden"

}, {

"country": "Ukraine"

}]

};

return objJson;

};

exports.teamlist = thelist();

Through the export function we are exposing the function thelist() which itself is returning a JSON object. This is how – for now – we will return a data object to the teamlist in our controller.

**Using a view template to create the HTML page**

Now that we have our controller pulling in some data from the model and doing something with it, we want to show the outcome in a webpage. We’ve seen that we’re going to send the data to a build function in a view template, so let’s create that now.

Create a folder in the root of your site called ‘views’. Inside this folder create a new file called ‘template-main.js’ with the following content:

exports.build = function(title, pagetitle, content) {

return ['<!doctype html>',

'<html lang="en">nn<meta charset="utf-8">n<title>{title}</title>',

'<link rel="stylesheet" href="/assets/style.css" />n',

'<h1>{pagetitle}</h1>',

'<div id="content">{content}</div>nn']

.join('n')

.replace(/{title}/g, title)

.replace(/{pagetitle}/g, pagetitle)

.replace(/{content}/g, content);

};

Here we are directly exposing the build function, which takes three parameters: title, pagetitle and content.

The first thing we are doing here is joining an array of strings, placing a new line ‘n’ between them. This creates one long string of HTML that will render on separate lines leaving the source code more readable. This string is the basis of our template, and contains a number of placeholders for the content items {title}, {pagetitle} and {content}.

Naturally then, the second step here is to replace these placeholders with the actual content sent through via the function arguments. The three replace functions are doing exactly that, running a very simple regular expression to globally change all occurrences found in the string. This means that you can have {title} in several places in your template, and each will get replaced. Note that we are also chaining the join and replace commands; the separate lines are for readability. Seeing as it works when laid out like this there is little sense in putting them all on one line and making the script harder to read.

**Using a static CSS file**

You may have noticed that in the HTML view template that we are referencing a static CSS file; remember in the router how we created a statement checking for the .css file extension? Now is the time to implement it.

In the root of your site create a folder called ‘assets’. Inside this folder create a text file called ‘style.css’ with the following content:

\* {font-family:arial, sans-serif;}

This isn’t the most complex CSS file in the world, but the change in font is enough for us to know that it is working!

**Creating a 404 page**

The final piece of our site is to create our 404 page. We already have a suitable template – our template-main view – so we’ll re-use that. Our router is also already set to send people to a 404 if it doesn’t recognise the URL path requested. So all that remains for us to do is to create the controller.

In your ‘controllers’ folder create a new file called ’404.js’ with the following content:

var template = require('../views/template-main');

exports.get = function(req, res) {

res.writeHead(404, {

'Content-Type': 'text/html'

});

res.write(template.build("404 - Page not found", "Oh noes, it's a 404", "<p>This isn't the page you're looking for...</p>"));

res.end();

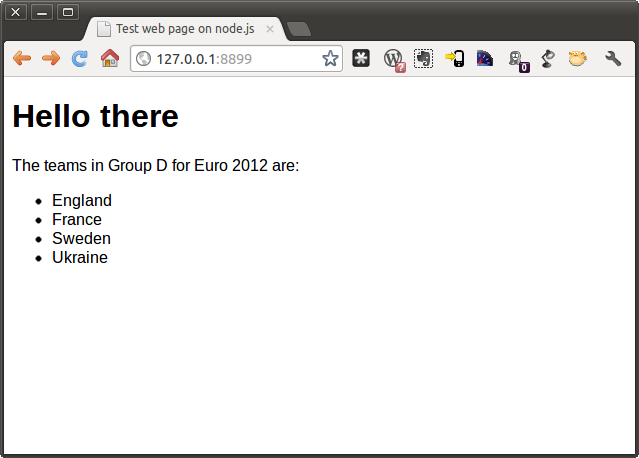
};

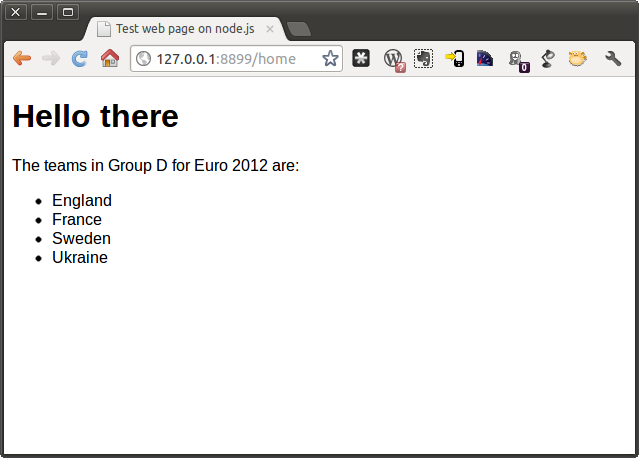
You should be able to see what we’re doing here. Using the same template - *views/template-main* - we send it three items of content. The only difference this time is that we have changed the status code of the response – in the res.writeHead function – from 200 (Ok) to 404 (not found).

**Let’s run the site!**

That’s all the pieces of our MVC framework in place, so let’s try out our site. In terminal cd into the root folder of your site and run it with the command:

node server.js

Now head to 127.0.0.1:8899 in browser and you should see something like this:   


Let’s test our other path from the router, by navigating to 127.0.0.1:8899/home   


Finally, let’s test our 404 page, by going to a URL that we haven’t added to the router, for example 127.0.0.1:8899/test 

**Success!**

So there we have it. We now have a working website in node.js written in a maintainable and scalable MVC framework. In future posts I will take a look at how to remove some of the pain by [using the express.js framework](http://theholmesoffice.com/express-js-framework-node-js/), and also[how to get the data from a MongoDB database](http://theholmesoffice.com/how-to-ge-data-from-mongodb-into-node-js/).

***Get the source code***

*The source code for this and my other tutorials on Node.js is available on GitHub here:*[*https://github.com/simonholmes/knowing-node/*](https://github.com/simonholmes/knowing-node/)

[**Simon Holmes**](http://theholmesoffice.com/author/simon/)

Author of "Getting MEAN" and "Mongoose for Application Development". Lover of JavaScript development. Founder of Full Stack Training.

*UK*[*http://www.fullstacktraining.com*](http://www.fullstacktraining.com/)

**Share this post**

[**The Holmes Office**](http://theholmesoffice.com/) © 2016Proudly published with [**Ghost**](https://ghost.org/)