Link: https://hackhands.com/delving-node-js-express-web-framework/

Posted on November 18th 2014

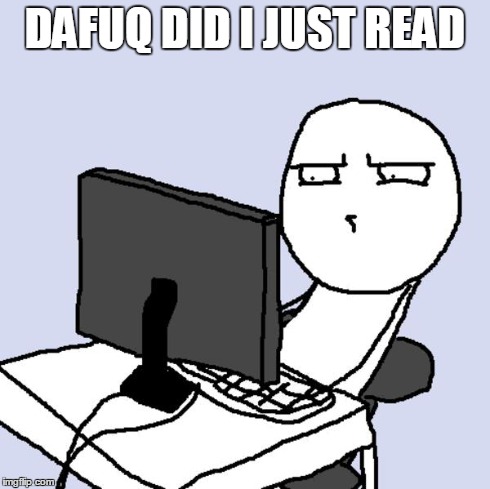
**Delving into Node.js and Express web framework**

**TL;DR**

This is a second post (first one is [here](https://hackhands.com/how-to-get-started-on-the-mean-stack/)) in a series of posts which will teach you how to take advantage of the MEAN stack in becoming a full-stack JavaScript developer. This post will be all about [Node.js](http://nodejs.org/) and[Express web framework](http://expressjs.com/). The third post, which is all about MongoDB is [here](https://hackhands.com/mongodb-crud-mvc-way-with-passport-authentication/).

*Disclaimer: code is based on the*[*MEAN.JS framework*](http://meanjs.org/)*and the tutorial partly follows the structure from*[*MEAN Web Development by Amos Haviv*](http://amzn.to/1cvHyth)*http://ir-na.amazon-adsystem.com/e/ir?t=nikolab-20&l=as2&o=1&a=B00NXWI1BM, the author of the MEAN.JS framework.*

**Aftermath of the first tutorial**

So, if after [the first post](https://hackhands.com/how-to-get-started-on-the-mean-stack/) you were like  
[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/dafuq3.jpg)

then I feel for you, as you haven't put anything on the actual web, you haven't run any meaningful Node.js application, and you haven't even begin to understand what this AngularJS thing is, let alone saved anything meaningful in the MongoDB database. Well, that's why the post was called *How to get started on the MEAN stack*, right? ;)

To, maybe, disappoint you even further, you're not going to save anything to the database in this tutorial either, instead you're going to learn the ins and outs of Node.js and Express web framework. Pushing it all to just one post would make it humongous and most likely harder to digest all at once. If you agree that you need a solid foundation in the tools you use, in order to make something meaningful, then read on.

Finally, as a promise, in the next installment of this tutorial series you'll learn all about the MongoDB and how Node.js and aforementioned Express work together with it (update: the third blog post is live [here](https://hackhands.com/mongodb-crud-mvc-way-with-passport-authentication/)).

**Enter the depths of Node.js**

[Node.js](http://nodejs.org/) was created by Ryan Dahl and it's basically a platform built on Chrome's JavaScript runtime called V8, whose greatest advantage over other JavaScript engines is the compiling of JavaScript code to native machine code before executing it. It uses an *event-driven*, *non-blocking* I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

**Event driven**

Saying that something is event-driven means that you can register some function to some event and that function will then be executed once the event is triggered. A simple example in JavaScript/HTML follows:

<!DOCTYPE html>

<**html**>

<**head**>

<**title**></**title**>

</**head**>

<**body**>

<**input** type="text" id="myInput">

<**input** type="button" id="myButton" value="Show Name">

<**script** type="text/javascript">

**var** myButton = **document**.getElementById('myButton');

**function** **outputText**(){

**var** myInputText = **document**.getElementById('myInput').value;

alert(myInputText);

}

myButton.addEventListener('click', outputText);

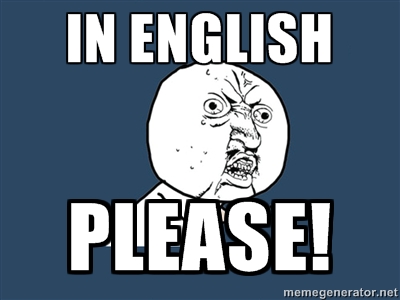
</**script**>

</**body**>

</**html**>

Just copy/paste this code in an **events.html** file and open it up in your browser (for your convenience, here's the [jsFiddle link of this example](http://jsfiddle.net/Hitman666/c9buyjnh/)). If you enter something in the input field and click the button, you will get the alert with the text you entered in the input field. So, in this simple example we've registered a function **outputText** to a buttons' (with an id of *myButton*) **click** event.

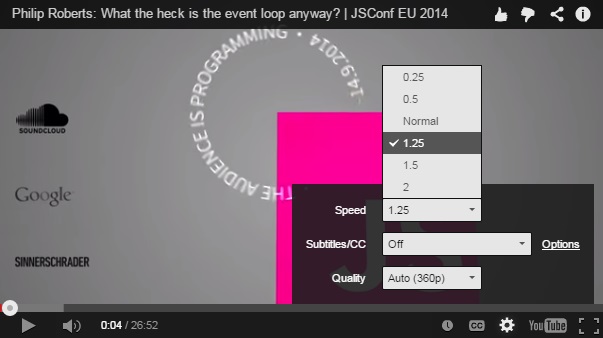
Web browser manages a single thread to run the entire JavaScript code using an inner loop a.k.a **the event loop**, which is basically a single-threaded loop that the browser runs infinitely. When some event gets emitted, the browser adds it to its **event queue**. The event loop then grabs the next event from the event queue in order to execute the **event handlers** registered to that event. After all of the event handlers are executed, the loop grabs the next event from the queue, executes its handlers, grabs the next event from the queue, and you get the picture, right? Well, if you're like

[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/InEnglishPlease.jpg)

then please invest the next 20 minutes in this video by Philip Roberts on "What the heck is the event loop anyway?" and I promise you'll thank me later:

<https://www.youtube.com/watch?v=8aGhZQkoFbQ>

*Wait, what do you mean 20 minutes, if the video clearly states it's 26:52 mins long!? Well, since today we're all about time hacking, you can listen at the speed of 1.25, but I'm sure you already know this, right? ;) And yeah, here's a nifty Google search which calculates*[*how much time you've saved*](https://www.google.com/search?num=50&espv=2&q=%2826minutes+and+52+seconds%29+minus+%2826minutes+and+52+seconds+multiplied+by+0.25%29&oq=%2826minutes+and+52+seconds%29+minus+%2826minutes+and+52+seconds+multiplied+by+0.25%29)*by listening at this speed.*

[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/YoutubeSpeedSetting.jpg)

**! Blocking code**

Say you have this piece of PHP code:

$mysqli = **new** mysqli("host", "user", "pass", "db");

$todos = $mysqli->query('SELECT \* FROM todos');

var\_dump($todos);

It's clear that this code **is blocking** any further code from executing until all the results from the database are loaded. This is called [synchronous programming](http://en.wikipedia.org/wiki/Synchronous_programming_language). Since the browser is single-threaded, using synchronous programming in this example would freeze everything else in the page. We, of course, don't want that so there were attempts to solve this issue by introducing thread pools but that became too hard to maintain especially when the application needed to be real-time.

So, how does the the *single threaded non blocking IO model* work in Node.js then? This [question on StackOverflow](http://stackoverflow.com/questions/14795145/how-the-single-threaded-non-blocking-io-model-works-in-node-js) has an awesome elaboration about it, and the crucial part from it would be this:

Node.js is built upon [libuv](https://github.com/joyent/libuv), a cross-platform library that abstracts APIs for asynchronous (non-blocking) input/output provided by the supported OSes (Unix, OS X and Windows at least). In asynchronous programming model open/read/write operations on devices and resources managed by the file-system ***don't block the calling thread***(as in the typical synchronous model) and **just mark**the process **to be notified** when new data or events are available.

Node.js tackles the problem leveraging JavaScript's language features called **closures**. Every function that requests IO has a signature like function(...parameters..., callback) and needs to be given a callback that will be invoked when the requested operation is completed.

**Javascript closures**

JavaScript's support for closures allows us to use variables that we've defined in the outer (calling) function inside the body of the callback. To help demonstrate the power of closures lets take a look at this example ([link to jsFiddle](http://jsfiddle.net/Hitman666/73vqfr9p/1/)):

**function** **parent**() {

**var** message = 'Howdy World';

**function** **child**() {

alert (message);

}

**return** child;

}

**var** childC = parent();

childC(); //this will alert Howdy World

The parent() function returns the child() function, and the child() function is called **after** the parent() function has already been executed. A closure is **not only the function**, but also the **environment** in which the function was created (which may be counterintuitive because usually the parent() function's local variables should only exist while the function is being executed). In this case, the childC() is a closure object that consists of the child() function and the environment variables that existed when the closure was created, including the message variable.

**Node.js modules**

One of the major JavaScript design flaws is that it shares a **single global namespace**. This means that when you assign a variable in one script, you can accidently overwrite another variable already defined in a previous script. This can easily become a conflict nightmare in larger applications, as errors will be difficult to debug. It could have been a major threat for Node.js evolution as a platform, but luckily a solution was found in the CommonJS modules standard.

[CommonJS](http://wiki.commonjs.org/wiki/CommonJS) is a project started in 2009 to standardize the way of working with JavaScript **outside the browser**. The CommonJS standards specify the following three key components when working with modules:

* **require()** method is used to load the module into your code
* **exports** object is contained in each module and allows you to expose pieces of your code when the module is loaded
* **module** object was originally used to provide metadata information about the module. It also contains the pointer to an exports object as a property

In Node.js's CommonJS module implementation, each module is written in a single  
JavaScript file and has an isolated scope that holds its own variables. The author of the module can expose any functionality through the exports object. To understand it better, let's say we created a module file named **hello.js** that contains the following code snippet:

**var** message = 'Howdy';

exports.sayHello = **function**(){

**console**.log(message);

}

Also, let's say we created an application file named **server.js**, which contains the following lines of code:

**var** hello = **require**('hello');

hello.sayHello();

Different approach, but same output, would be:

**module**.exports = **function**() {

**var** message = 'Howdy';

**console**.log(message);

}

Then, the module would be loaded and called in the server.js file as follows:

**var** hello = **require**('hello');

hello();

Node.js comes pre-bundled with some modules (like for example a file reading module [fs](http://nodejs.org/api/fs.html), [http](http://nodejs.org/api/http.html) module, etc.) which are documented (along with many other modules) in its [documentation](http://nodejs.org/api/), but there is also a vast number of third party modules (as of the date of this post there is a total of **105006** of them and you can check the current numbers on [modulescount.com](http://www.modulecounts.com/)). NPM (node package manager, about which we spoke in the previous tutorial) installs these modules in a folder named **node\_modules** under the root folder of your application and you can just require them as you would normally require a core module.

**Lets build a Node.js web server**

The code for the simples Node.js web server ever:

**var** http = **require**('http');

**var** port = 1337;

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

res.writeHead(200, {

'Content-Type': 'text/plain'

});

res.end("Hello Mitnick, you've just stumbled on the simplest web server ever");

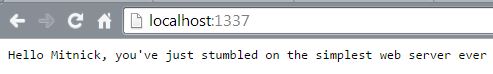
}).listen(port);

**console**.log('Our awesome web server running at http://localhost:'+ port);

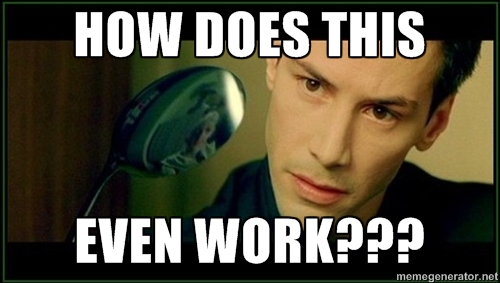
If you save the previous code listing in a file named **server.js** and run it with **node** via command prompt like this (once in the root folder of your application - fancy name for where the actual server.js file is placed):

$ node server.js

then, if you open up your browser and type in [http://localhost:1337](http://localhost:1337/), you will see the following output:

[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/localhostNodejsSimplestWebServer.jpg)

So,

[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/howDoesThisEvenWork.jpg)

First you require the http module and use its **createServer()** method to return a new server object. The**listen()** method then instructs the server to listen on the port 1337. The callback function that is passed as an argument to the createServer() method gets called every time there's a HTTP request sent to the web server. The callback function then does two things:

* It calls the **writeHead()** method from the response object (**res**), which is used to set the response HTTP headers. In this example it will set the Content-Type header value to **text/plain**
* Then, it calls the **end()** method of the response object, which is used to finalize the response

**Connect module**

So, cool, we're creating a web server with low level commands. Great? Well, unless you've got tons of time at your hands, it's not. There are many modules that support web application development but none is as popular as the [Connect](https://github.com/senchalabs/connect) module. The Connect module delivers a set of wrappers around the Node.js low-level APIs to enable the development of rich web application frameworks.

The developers at [Sencha](http://www.senchalabs.org/) made the [Connect module](https://github.com/senchalabs/connect#readme) which uses a modular component called **middleware**, which allows you to simply register your application logic to predefined HTTP requests. Connect middleware are basically a bunch of callback functions, which get executed when a HTTP request occurs. The middleware can then perform some logic, return a response, or call the next registered middleware. While you will mostly write custom middleware to support your application needs, Connect also includes some common middleware to support logging, static file serving, etc.

Express web framework is based on Connect's middleware approach, so in order to understand how Express works, we'll begin with creating a Connect application. In order to use the following examples you'll need to install connect module, and you can do that by running the following command:

$ npm install connect

Connect middleware is just JavaScript function with following three arguments: **req**, **resp**, **next**. Place the following code in **server\_connect.js** file:

**var** port = 1337;

**var** connect = **require**('connect');

**var** app = connect();

**var** howdyWorld = **function**(req, res, next) {

res.setHeader('Content-Type', 'text/plain');

res.end('Howdy World');

};

app.use(howdyWorld);

app.listen(port);

**console**.log('Server running at http://localhost:' + port);

Let's recap on what we did:

* added a middleware function named **howdyWorld()**, which accepts three arguments: req, res, and next
* used the **res.setHeader()** method to set the response Content-Type header in the middleware function
* set the response text with **res.end()** method
* registered our middleware with the Connect application by using **app.use()** method
* set the application to listen on port 1337
* logged some text to the console

One of Connect's greatest features is the ability to register **as many middleware functions** as we want, for example here's how we would add a **logger()** middleware to our server:

**var** port = 1337;

**var** connect = **require**('connect');

**var** app = connect();

**var** logger = **function**(req, res, next) {

**console**.log(req.method, req.url);

next();

};

**var** howdyWorld= **function**(req, res, next) {

res.setHeader('Content-Type', 'text/plain');

res.end('Howdy World');

};

app.use(logger);

app.use('/howdy', howdyWorld);

app.listen(port);

**console**.log('Server running at http://localhost:' + port);

The **logger()** middleware uses the **console.log()** method to log the **method** and **url** of the request (**req**variable) out to the console. The logger() middleware is registered before the **howdyWorld()** middleware because that determines the order in which each middlewares are executed. Another thing to notice is the**next()** call in the logger() middleware, which then calls the howdyWorld() middleware. Removing the next() call would stop the execution of middleware function at the logger() middleware, which would in turn mean that the request would hang forever as the response would never end by calling the **res.end()**method. Finally, the howdyWorld() middleware is bound to respond only to requests made to the **/howdy**path.

**Express**

Since Connect module is open source, it can be extended and one (yeah, you heard that right - one, uno, jedan, ein) developer named TJ Holowaychuk did it better than most when he released a Connect-based web framework known as [Express](http://expressjs.com/). I'll summarize TJ Holowaychuk like this:

[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/legend.jpg)

He has more than **500** [open source projects](https://github.com/tj), people even ask [how does he do it](http://www.quora.com/How-is-TJ-Holowaychuk-so-insanely-productive), anyways as the image suggest - a legend. Ok, enough praise, let's move on fellow mortals...

Express is built on top of Connect and it uses its middleware architecture. It extends Connect to allow a variety of common web application use cases:

* HTML template engines
* various data format outputs
* routing system, etc

Now, some action! Lets start a new project by making a new directory and running **npm init** command in it. After filling some basic information about your application (when it asks you for a **entry point**, enter server.js instead of default index.js), let's install express by running:

$ npm install express --save

As you may remember from the first post, the **--save** switch saves the module dependency in your**package.json** file which was created with the npm init command. Now, create a file named **server.js** and copy/paste the following content in it:

**var** port = 1337;

**var** express = **require**('express');

**var** app = express();

app.use('/', **function**(req, res) {

res.send('Howdy World');

});

app.listen(port);

**console**.log('Server running at http://localhost:' + port);

First, we require the Express module and create a new Express application object. Then, we use the**app.use()** method to bind a middleware function with a **specific path**, and we use the **app.listen()** method to tell the Express application to listen on the port 1337. The app.use() method's second parameter is an anonymous (middleware) function which accepts parameters **req** and **res**, which then uses the **res.send()**method to send the response back. The **res.send()** method is basically an Express wrapper that sets the Content-Type header according to the response object type and then sends a response back using the Connect's res.end() method.

**MVC**

So, for some long time now the state of the best web development practices is like this:

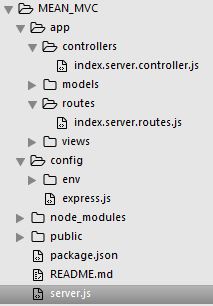
[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/MVC.jpg)

and you just may be like this:

[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/MVClady.jpg)

Don't despair, we're going to demistify this. The Express framework in itself doesn't require any specific pattern or syntax or structure as do some other web frameworks. Applying the MVC pattern to our Express application means that we can create **specific folders** where we place our JavaScript files in a certain logical order. All those files are  
basically **CommonJS modules** that function as logical units.

Sooner or later you'll find yourself wondering how you should arrange your project files and break them into logical units of code. **Horizontal project structure** is the structure that we will use, and it is based on the division of folders and files by their **functional role** rather than by the feature they implement, which means that all the application files are placed inside a main application folder that contains an MVC folder structure. **A vertical project structure** is based on the division of folders and files by the **feature** that they implement, which means that **each feature** has its own folder that contains an MVC folder structure. This is how our app folder structure will look like:

[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/MVCappStructure.jpg)

The **app** folder is where we'll keep our Express application logic and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:

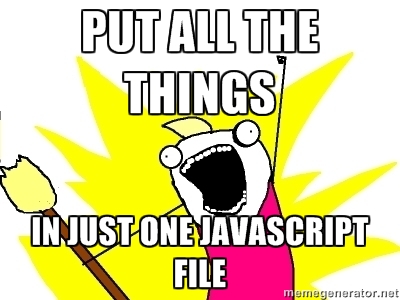
* **controllers** - contains Express application controllers
* **models** - contains Express application models
* **routes** - contains Express application routing middleware
* **views** - contains Express application views

The **config** folder is where we'll keep our Express application configuration files. It will contain several files and folders:

* **env** - contains Express application environment configuration files
* **config.js** file - contains configuration code of our Express application
* **express.js** file - contains initialization code of our Express application

The **public** folder is where we'll keep our static client-side files and it will be also divided into the folders that represent a separation of functionality to comply with the MVC pattern, but we'll take a look at this in more detail in one of the next tutorials when we'll be going through Angular.js MVC folder structure.

So, basicalls, this whole MVC thing is just a way you structure your application in order to help you maintain it later when the project becomes big. Big deal, you might say, and you may do it like this:

[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/allTheThingsInJustOneJsFile.jpg)

but if you're project grows you'll probably end up like this:

[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/useMVCorElse.jpg)

**Lets build our application following the MVC pattern**

If you want to follow along but you dread typing, you can [clone the finished project from Github](https://github.com/Hitman666/MEAN_MVC_2ndTutorial). If you would like to test the project, you have to:

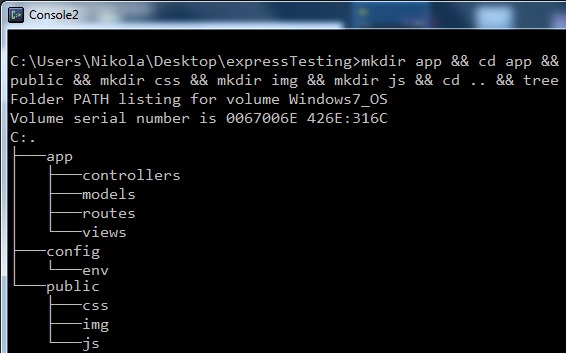
* run **npm install** after cloning
* and run **node server** from the root of the application (where the server.js file is placed)
* open up your browser at [http://localhost:1337](http://localhost:1337/)

**Folder structure**

So, on a premise of you taking my word for it, let's create this infamous MVC folder structure. Just copy paste this in your terminal:

mkdir app && **cd** app && mkdir controllers && mkdir models && mkdir routes && mkdir views && **cd** .. && mkdir config && **cd** config && mkdir env && **cd** .. && mkdir public && **cd** public && mkdir css && mkdir img && mkdir js && **cd** .. && tree

and you should see the following output (also, you will see the **node\_modules** folder structure which was created when you node installed the express library, not shown here for brevity):

[](http://www.nikola-breznjak.com/blog/wp-content/uploads/2014/11/folderStructure.jpg)

**Controller**

Now, in the **app/controllers** folder, create a file named **index.server.controller.js** with the following lines of code:

exports.render = **function**(req, res) {

res.send('Howdy World');

};

Here we're using a CommonJS module pattern to define a function named **render()** which we'll then later use to require this module and use its function.

**Express routing**

Once we've created a controller, we'll need to use Express routing functionality to make use of this controller. Express routing example is shown in the following code snippet:

app.get('/', **function**(req, res) {

res.send('This is a GET request');

});

This tells Express to execute the middleware function for any HTTP request that comes through a GET method and is directed to the root path ('/'). If we would need to deal with POST requests, we would simply use **app.post**.

Express can chain several middlewares in a single routing definition, which means that middleware functions will be called in order. This is usually used to validate requests before executing the response logic. To understand this better, lets take a look at the following example:

**var** port = 1337;

**var** express = **require**('express');

**var** nameExists = **function**(req, res, next) {

**if** (req.param('name')) {

next();

}

**else** {

res.send('What is your name?');

}

};

**var** sayHowdy = **function**(req, res, next) {

res.send('Howdy ' + req.param('name'));

};

**var** app = express();

app.get('/', nameExists, sayHowdy);

app.listen(port);

**console**.log('Server running at http://localhost:' + port);

**edit (21.1.2015):** At the time when this blog post was published the Express version was "4.10.7", and in the currently new version "4.11.0" the '*app.param(callback)*' is deprecated. In the above example you should use **req.param.name** instead if you're using the new version of Express.

Here we have two middleware functions named **nameExists()** and **sayHowdy()**. The nameExists() middleware is looking for the name parameter, and if it finds a defined name parameter it will call the next middleware function. But, if it doesn't then the middleware will handle the response by itself. In our case, the next middleware function would be the sayHowdy() middleware function and this is possible because we've added the middleware function in a row using the app.get() method. The order of the middleware functions determines which middleware function is executed first. So, if you visit [**http://localhost:1337/**](http://localhost:1337/) in your browser (after running the script with node) you would get the output **What is your name?**. If you enter [**http://localhost:1337/?name=Kevin**](http://localhost:1337/?name=Kevin), you would get the output **Howdy Kevin**.

Ok, enough theory, let's continue with our example by creating a new file named **index.server.routes.js** in the **app/routes** folder, with the following code snippet:

**module**.exports = **function**(app) {

**var** index = **require**('../controllers/index.server.controller');

app.get('/', index.render);

};

As we see in this example, CommonJS module pattern supports both the exporting of several functions like we did with our controller example, and the use of a single module function like we did here. Next, we required our index controller and used its render() method as a middleware to GET requests made to the root path.

The routing module function accepts a single argument called app, so when we call this function, we'll need to pass it the instance of the Express application.

In order to glue all this together, in the **config** folder create a new file named **express.js** with the following code snippet:

**var** express = **require**('express');

**module**.exports = **function**() {

**var** app = express();

**require**('../app/routes/index.server.routes.js')(app);

**return** app;

};

First we required the Express module, then we used the CommonJS module pattern to define a module function that initializes the Express application. First, it creates a new instance of an Express application, and then it requires the routing file and calls it as a function passing it the application instance as an argument. The routing file we made before will use the application instance to create a new routing configuration and will call the controller's render() method. The module function ends by returning the application instance. The express.js file is where we configure our Express application. This is where we add everything related to the Express configuration.

To finalize our application, take the **server.js** file that you made before and copy the following code:

**var** port = 1337;

**var** express = **require**('./config/express');

**var** app = express();

app.listen(port);

**module**.exports = app;

**console**.log('Server running at http://localhost:' + port);

So, that's it. If you run the example by executing **node server.js** in the command prompt you'll get the same thing as we did in the beginning and with everything in just one file. So, you may ask 'Why the f$%& all this trouble?'. Well, as I said - it will pay off later in your career when your project grows in complexity.

**Rendering views**

A very common feature of web frameworks is the ability to render views. The basic concept is passing your data to a template engine that will render the final view usually in HTML. In the MVC pattern, your controller uses the model to retrieve the data portion and the view template to render the HTML output. The Express allows the usage of many Node.js template engines (for example [EJS](https://github.com/tj/ejs), [jade](https://github.com/jadejs/jade), [swig](https://github.com/paularmstrong/swig) to name few - lookup te rest [here](https://github.com/Deathspike/template-benchmark)) to achieve this functionality.

Express has two methods for rendering views:

* **app.render()** - used to render the view and then pass the HTML to a callback function
* **res.render()**- used to render the view locally and sends the HTML as a response

You'll probably end up using res.render() most of the time because you usually want to output the HTML as a response. However, if, for an instance, you'd like your application to send HTML e-mails, you will probably use app.render().

We'll dip our toes into templating with Node.js's templating engine mentioned earlier, called [EJS](https://github.com/tj/ejs) - so let's install it

$ npm install ejs --save

then add this to your existing **express.js** file in the **config** folder, just before requiring the route:

app.set('views', './app/views');

app.set('view engine', 'ejs');

EJS views are basically HTML code mixed with EJS tags. EJS templates will be placed in the **app/views**folder and will have the **.ejs** extension. To create your first EJS view, go to your app/views folder, and create a new file named **index.ejs**that contains the following HTML code snippet:

<!DOCTYPE html>

<**html**>

<**head**>

<**title**><**%=** title %></**title**>

</**head**>

<**body**>

<**h1**><**%=** title %></**h1**>

</**body**>

</**html**>

All you have left to do is configure your controller to render this template and automatically output it as an HTML response. To do so, go back to your **app/controllers/index.server.controller.js** file, and change it to look like the following code snippet:

exports.render = **function**(req, res) {

res.render('index', {

title: 'Howdy World'

})

};

The **res.render()** method will use the EJS template engine to look for the file in the views folder that we set in the config/express.js file and will then render the view ('index' - without .ejs extension) using the object variables that we send to it (just **title** for now).

**Using static files**

You will most probably need to show some images in your templates, and the way you can do this is by using the **express.static()**middleware. In the **config/express.js** file add the following line just before the**return app;** statement:

app.use(express.static('./public'));

The express.static() middleware takes one argument to determine the location of the static folder. Lets say you added an image named **logo.jpg** to the **img** folder (inside the public folder). The way you would show it in a view (index.ejs) is by using the following straight forward code:

<**img** src="img/logo.jpg" alt="Hack Hands logo">

**Conclusion**

Yay, we're done! In this tutorial we've covered a lot of ground so don't worry if some of it doesn't sink in at first reading. Reread, explore on your own - that's how you'll get the grip of it the best. Anyways, enough of the self help punch lines, hope you liked the post and see you in the next installment of this post when we'll tackle the MongoDB.