Node Chat tutorial

<insert introduction>

Table of Contents

Chapter 1

1.1 Hello world

To start build the node chat application you will need a set of tools to get started.

- Windows Azure SDK for node.js
- A Text Editor of choice. Sublime Text comes recommended.
- Windows operating system. I use windows 7.

That's it really, once you have the azure SDK and a text editor your ready to go. Note that the azure sdk should install IIS for you aswell.

You may want to read the Azure SDK for node tutorial first

1.1.1 Setup Azure

To get started open up powershell, navigate to your folder of choice to store your applications (for me it's C:\node). And run [github]

New-AzureService so642

This gives us an empty azure service, setting up the default azure service files.

Now before we can run anything we have to add an azure web role. Azure web roles allow us to run node on IIS using iisnode. There are a ton of advantages to using iisnode <insert them>

So to get a web roles setup, run [github]

Add-AzureWebRole

The azure webrole has created a server.js role for us in so642/WebRole1/server.js

```
var http = require('http');
var port = process.env.port || 1337;
http.createServer(function (req, res) {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello World\n');
}).listen(port);
```

The above snippet looks almost exactly like the hello world server from the node.js website except we listen on `process.env.port` if it exists. This allows iisnode to pass in an arbitary port for our webserver to run on. It basically means it can pass different ports to multiple node applications and run them behind a single load balancer.

1.1.2 Start the server

You can start the server with

node .\server.js

This will get node to interpret your program and run it, it should be running on port 1337 so you can open your browser on localhost:1337 and check if it worked. To end a node program use Ctrl+C in the command line.

You can also start the server through the azure emulator with

Start-AzureEmulator

Which takes a bit longer to run because it has to boot up the azure emulator, IIS and nodeiis. You can check your same program in the browser at localhost:81

1.2 Developing with tests first

Tests are important, really important. I wholly recommend that before you start writing any code you write unit tests and integration tests.

That’s why the first thing we’re going to do is write HTTP integration tests.

However before we continue with the tests lets have a quick aside about version control

1.2.1 Version control

In the node community people use git and github for their version control and code repositories. Personally I installed msysgit and use their Git bash tool for windows. Github has some git cheatsheets to help out with learning git.

One of the features of git is having a .gitignore file which specifies files you do not want to have in your git repository. Personally I've added local_package.csx and WebRole1\server.js.logs to my .gitignore file [github]

1.2.2 Package.json

But let's continue with the testing, the first thing we need to do is add a package.json file to our project so we can use npm to install our dependencies for you. The way npm works is that it looks at your package.json file for a list of dependencies and installs them from there. This means that rather then installing every dependencies manually we can have this done automatically from a central location.

Our first package.json file should look like [github]

```
{
      "name": "so-642",
      "version": "0.0.1",
      "devDependencies": {
            "nodeunit": "0.6.4"
      },
      "private": true
}
```

Here we have name and version because npm requires them. We have the private flag set to true so that we cannot accidentally publish this module on the npm registry and we have nodeunit in our devDependencies. We have placed nodeunit in devDependencies rather then dependencies because it's only needed for development and not for deployment.

To install nodeunit using npm run

```
npm install -d
```

In the directory with package.json, npm will automatically install your dependencies, and because we use the **-d** flag it will also install your dev dependencies.

Just to confirm that you've installed this module correctly you can open the node REPL using

node

And then run the following code

> require('nodeunit') !== undefined

This will tell you that the nodeunit module is not undefined, so you've installed it correctly. (Ctrl + C to exit the REPL).

1.2.3 Setting up the first test script

So we've got our testing framework installed, we now just need to write a test. The first test will be a http test to test the current hello world server.
Personally I create a test folder and a http folder inside test so my file structure looks like

```
WebRole1
  - test
    - http
      main.js
```

main.js contains our first test. For this test we will have to require the http module to make http calls and we will have to export a module for nodeunit to run.

nodeunit expects your test files to export objects where every method is a test to be run. Each method takes a single argument, which is the test object. The test object has a set of utility methods that reflect the node assert module. So our file starts out as *[github]*

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

module.exports = {};
```

We now need a test method *[github]*

```
"test /": function (test) {

}
```

We can double check that this works by using the nodeunit CLI to run our unit tests. First you have to install the nodeunit CLI using npm

npm install nodeunit -g

Then you just run

nodeunit ./test/http

And this should show an error message saying that the tests are not done. This is because with nodeunit you have to tell the test that it's finished using `test.done();` [github]

After doing so you can run the unit tests again and this time they should pass with "OK 0 assertions". That's not very useful of course, we need to add two useful assertions.

1.2.4 Adding HTTP to the test

We will add a `test.expect` call to tell nodeunit that we expect two assertions to run, this allows nodeunit to tell us whether all our expected assertions run and we will add a http request using the node http API [github]

```
test.expect(2);

var req = http.request({
     host: "localhost",
     port: 81,
     path: "/"
}, handleResponse);

req.on("error", throwError);

req.end();

function handleResponse() {

}
```

Here we are using the `http.request` method to make a HTTP request to localhost:81 where our azure emulation is running our server. We have to attach an error handler to the request object so that we can handle any http level errors. And we have to call `req.end` to send the actual HTTP request.

Of course this doesn't actually do anything, we have to implement `handleResponse`. [github]

```
function handleResponse(res) {
      var body = "";
      test.equal(res.statusCode, 200,
            "statusCode is not 200");

      res.on("data", appendDataToBody);

      res.on("end", testBody);

      function appendDataToBody(chunk) {
            body += chunk;
      }

      function testBody() {
            test.equal(body, "Hello World\n",
                  "body is not hello world");
            test.done();
      }
}
```

Our `handleResponse` function tests that the HTTP status code is 200 using the `test.equal` which is the same as assert.equal

It also listens on data coming in from the response and aggregates it in a body object which it then uses once the response has ended. Our second assertion then tests that the HTTP body is the hello world text and as follows calls `test.done` to let nodeunit know this test method has finished.

We can then use nodeunit again to run these test

```
nodeunit .\test\http
```

As long as our azure emulation is still running the tests should pass.

1.2.5 Refactoring using request

Note that the http API we used is quite low level. This particular example is manageable but if we want to use more advanced features like auto-redirect or session emulation then we would have to write boilerplate code for it. This is why it's best to use an existing module for http request sugar. In this case I will use request

This allows us to refactor our code as follow [github]

```
request(uri, handleResponse);

function handleResponse(err, res, body) {
      if (err) throw err;

      test.equal(res.statusCode, 200,
            "statusCode is not 200");

      test.equal(body, "Hello World\n",
            "body is not hello world");
      test.done();
}
```

Note that you will have to add request to your package.json file and then npm install it. [github]

1.2.6 Refactoring the server

Now we can refactor our application into a nicer event driven setup. The core of the application architecture will revolve around a single mediator using the mediator module.

This means refactoring the server to be [github]

```
var mediator = require("mediator"),
      path = require("path");

mediator.on('boot.error', throwError);

load(path.join(__dirname, "routines"), routinesLoaded);

function throwError(err) {
      throw err;
}

function routinesLoaded(err) {
      if (err) {
            return mediator.emit('boot.error', err);
      }

      mediator.emit('boot.ready');
}

function load(directory, callback) {

}
```

Here we are setting up our mediator by attaching an error thrower to our boot.error event. This would be replaced in production to log errors instead. Then we call load to load all our routines in the routines folder and handle a callback when the routines have loaded.

Our callback emits the boot.ready event to let routines know that we are ready to boot. In this case a routine can be anything. We stop telling objects to do something directly and instead let every routine grab a handle on the mediator, they can then listen and react to any event they want. This encourages an aggressively loosely coupled architecture where every routine implements some mediator api.

Note how we pass errors through the mediator so that someone else can worry about how we deal with them.

Next we need to implement load so it can load the routines. for this we will need to require fs and after

```
      fs = require("fs"),
      after = require("after"),
```

The file system is needed to recursively walk folders and files and after is needed for asynchronous flow control. [github] (patch1: [github])

```
      function load(directory, callback) {
            fs.readdir(directory, handleFiles);

            function handleFiles(err, files) {
                  if (err) return mediator.emit('boot.error', err);

                  var next = after(files.length, callback);
                  files.forEach(handleFile);

                  function handleFile(file) {
                        var uri = path.join(directory, file);
                        if (file.substr(-3, 3) === ".js") {
                              require(uri);
                              next();
                        } else {
                              load(uri, callback);
                        }
                  }
            }
      }
```

So basically load now loads the directory. For every file in the directory it will either require it if it ends in ".js" or load the directory itself. Note I'm using after here to say "run the callback after this next function has been called files.length times". This ensure that only after we've dealt with all files will the callback be fired.

1.2.7 Adding routers and controller

Along with setting up the function that loads all our routines we will need to implement some routines. The first routine to implement is the server routine that creates a http server and listens to it. In this case we will be using express so your package.json has to be updated to include express [github]

I created a file at routines/server/boot.js that listens on the boot ready event and creates a server object. It then fires an event down the mediator saying that the server has been created. This means that anything that needs a handle on the server can just listen to that event rather then have to be passed the server explicitely.

This routine also tells the server to start by listening on a port. In this case we are using express's createServer mechanism which is similar to http since express inherits from http. [github]

```
var mediator = require("mediator"),
      express = require("express");

mediator.on("boot.ready", createServer);

function createServer() {
      var server = express.createServer(),
            port = process.env.port || 4000;

      mediator.emit("boot.server.created", server);

      server.listen(port);

      mediator.emit("boot.server.listening", port);
}
```

Now our server doesn't actually do anything because we have no routes. In express adding a route is a matter of calling a method on the server like `server.get(uri, handler)`.

So we need to implement logic to handle the routes. This can either be automated or done by hand, I personally do it by hand by implementing the following in routines/routes/home.js [github]

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

mediator.on("boot.server.created", attachRoutes);

function attachRoutes(server) {
      server.get("/", handleMain);

      function handleMain(req, res) {
            mediator.emit("controller.home.main", req, res);
      }
}
```

Here we simply listen on the server creation event and attach a route handler to “GET /”. In this case all the router does is emit an event for a controller to handle.

This means we have to implement a controller. The reason we split it up into router and controler is to allow the controller to know *nothing* about the details of the HTTP server. In this case we could easily swap out express for flatiron, rewrite the routers and use the same controllers.

In this particular case the controller for home (routines/controllers/home) is rather simple *[github]* *(patch1 [github])*

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

mediator.on("controller.home.main", handleMain);

function handleMain(req, res) {
      res.end("Hello World\n");
}
```

This controller only really listens to the home.main event and sends hello world down the response. Note that express will write the correct headers under the hood for us.

If we run our unit tests again using

```
nodeunit .\test\http
```

You will notice that they all pass again. However we now need to go back and add tests for all the layers we’ve added.

We will need to write tests for the servers, routers and controllers

1.2.8 Writing tests for the Server, controllers and routers

When writing unit tests for routines we want to test the mediator interface implements. This means testing that they listen to the correct events and emit the correct events. Apart from the mediator communication you don't care or test any of the other details of the routines.

1.2.8.1 Testing the server

So a unit test for say server/boot would start like

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

require("../../routines/server/boot");

module.exports = { /* tests */ };
```

All your doing is requiring the routine and letting it attach itself to the mediator then you talk to the routine through the mediator.

I've implemented the testing of server creation method as follows: [github]

```
"test server creation": function (test) {
      var server;

      test.expect(1);

      mediator.once("boot.server.created", testServer);
      mediator.once("boot.server.listening", cleanUp);

      mediator.emit("boot.ready");

      function testServer(value) {
            server = value;
            test.ok(server, "server not created");
      }

      function cleanUp() {
            server.close();

            test.done();
      }
},
```

Here we listen to the server created event, fire the boot ready event and test that the server is actually created. Note that we bind to the listening event so we can close the server down again. The server listening test is similar.

Note that you have to run this test with

nodeunit .\test\server

And we can't run **nodeunit .\test** because a limitation with nodeunit not searching for tests recursively. Thankfully however we already have a recursive file loading function. With a minor amount of refactoring *[github]* we can re-use this function to run all our tests.

1.2.8.2 Writing our own test runner

So to make unit testing easier we should write a small test running script [github]

```
var load = require("./utils/load"),
	nodeunit = require("nodeunit");
	path = require("path");

load(path.join(__dirname, "test"), runTests);

function runTests(err, files) {
	if (err) {
		throw err;
	}

	var obj = {};

	for (var i = 0, len = files.length; i < len; i++) {
		obj[i] = files[i];
	}

	nodeunit.reporters.default.run(obj);
}
```

Here we load all the tests in the test folder and convert them from an array into an object. THis is because the nodeunit reporter needs an object and assumes arrays are file uris.

Note we’ve also added

```
"scripts": {
	"test": "node ./testrunner.js"
},
```

To our package.json file. This allows us to use

npm test

To run our tests, since npm test will invoke the test script which invokes the test runner.

Since we’ve added a test for the server routine we need add two further tests for the routes and the controllers.

1.2.8.3 Testing the routes

Personally I implemented the following in test/routes/home [github]

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

require("../../routines/routes/home.js");

module.exports = {
	"test controller home main": function (test) {
		var req = {}, res = {};

		test.expect(2);

		mediator.once("controller.home.main", testHome)

		mediator.emit("boot.server.created",{
			get: function (_, cb) {
				cb(req, res);
			}
		});

		function testHome(request, response) {
			test.equal(request, req,
				"request objects are not the same");
			test.equal(response, res,
				"response objects are not the same");
			test.done();
		}
	}
}
```

Testing the routes is a simple matter of listening on the controller home event. And then checking that the router forwards the req and res properly for its route handler.

Note that we mock the normal server with an anonymous object that implements get which just calls the callback directly with a dummy req/res object.

1.2.8.4 Testing the controllers

For the controller test I implemented the following *[github]*

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

```
require("../../routines/controllers/home.js")

module.exports = {
      "test controller home main": function (test) {
            test.expect(1);

            mediator.emit("controller.home.main", null, {
                  end: function (data) {
                        test.ok(data,
                              "end was not called with data");
                        test.done();
                  }
            });
      }
};
```

For those coming from other languages/platforms notice how easy it is to mock tests in javascript. Here I'm passing in an anonymous object that mocks the server.end method.

The controller is pretty easy to test, emit the controller event and test that the controller actually ends the response.

Now if you run the tests you will notice they fail. If you take a look at the details, the reason they fail is because we have two tests that manipulate the same mediator event and our mediator is a process global.

Now there are two ways you fix testing global state. You either remove global state or you tip toe around it and hope it doesn't break.

We are going to remove global state by injecting the mediator into every routine. Thankfully we already have unit tests that we can re-run when our code has been refactored

1.2.9 Refactoring to inject mediators into routines

The first step is to fix up the package.json file. We are going to remove the mediator dependencies and add in pd. pd will give us utilities needed for making the code easier [github]

```
"dependencies": {
        "after": "0.2.0",
        "pd": "0.3.9",
        "express": "2.5.2"
    },
```

Now we just fix the routines one by one starting with the server [github]

```
    var express = require("express"),
        pd = require("pd");

    module.exports = pd.bindAll({
        start: start,
        createServer: createServer
    });

    function start(mediator) {
        this.mediator = mediator;
        this.mediator.on("boot.ready", this.createServer);
    }

    function createServer() {
        var server = express.createServer(),
            port = process.env.port || 4000;

        this.mediator.emit("boot.server.created", server);

        server.listen(port);

        this.mediator.emit("boot.server.listening", port);
    }
```

Here we are now exporting a module with a start method. The start method will be invoked by our server setup with a mediator. Note that we store the mediator on the exported object. Note we also call `pd.bindAll` which fixes the methods on the object to be bound to the object. This allows us to pass the methods around as callbacks without losing the correct this value. Of course we also have to patch the server so that it actually injects the mediator into the routines [github]

```
    var EventEmitter = require("events").EventEmitter,
        mediator = new EventEmitter(),

    /* snip */
```

```
function routinesLoaded(err, files) {
      if (err) {
            return mediator.emit('boot.error', err);
      }
      Object.keys(files).forEach(startWithMediator);
      mediator.emit('boot.ready');

      function startWithMediator(name) {
            files[name].start(mediator);
      }
}
```

Here we simply create our own mediator by creating an event emitter instance. We then start all our modules using the mediator.

Now we also have to fix up our

- Controllers [github]
- Routes [github]
- Test [github]

Now that we have fixed all that and start our server again in the azure emulator you will notice that all the tests pass using

Start-AzureEmulator
npm test

Now I have noticed that restarting the azure emulator is not fun. It's time consuming and feels like it should be automated.

Since we have a modular architecture with routines we should be able to add a hot code reload routine into our project that reloads our routines when their respective files change. This will require further architecture changes to allow our routines to be attached and detached from our application core.

Since we've mentioned the application core we will actually have to write a core. I wrote mine in utils/core [github] (patch1: [github])

```
var EventEmitter = require("events").EventEmitter.prototype,
      pd = require("pd");

var Core = pd.bindAll({
      attach: attach,
      modules: {}
  });

pd.extend(Core, EventEmitter);

module.exports = Core;

function attach(name, module) {
      if (typeof name !== "string") {
          return Object.keys(name).forEach(invokeAttach, this);
      }

      if (this.modules[name]) {
          this.modules[name].detach(this);
          delete this.modules[name];
      }

      this.modules[name] = module;

      module.start(this);

      function invokeAttach(key) {
          this.attach(key, name[key]);
      }
}
```

So here we create a Core object with an attach method to attach modules. It also has a modules array to store references to loaded modules.

We use pd to mixin the EventEmitter methods with the core and then export the core.

The attach method is overloaded to allow both `attach(name, module)` and `attach(hashOfNamesAndModules)`. This is a common technique of method overloading in javascript. Apart from that it just remove the module if it exists, and then starts the module with the core object.
Note that the core is an event emitter. We've just made the core our mediator.

We now refactor the server to use our new core modules *[github]*

```
    var load = require("./utils/load"),
        core = require("./utils/core"),
        path = require("path");
```

```
core.on('boot.error', throwError);

load(path.join(__dirname, "routines"), routinesLoaded);

function throwError(err) {
	throw err;
}

function routinesLoaded(err, files) {
	if (err) {
		return core.emit('boot.error', err);
	}

	core.attach(files);
	core.emit('boot.ready');
}
```

Note how all we have to do is attach all our routines to the core and emit the ready event.