Web and Mobile Application Development

Nodejs Part III

Material created by:
David Augenblick, Bill Mongan, Dan Ziegler, Samantha Bewley, and Matt Burlick

Scaling our Content

- Ok so now we have a way to server static content as well as a way to respond to requests for dynamically generated content.
- However as our web application grows, there may be tons of different request paths and the logic in each might be extensive.
- Therefore, to adhere to good software engineering principles we'd like to separate this into different files.
- We're going to start making modules of our own which can **emit** events that other code can **listen** for (and respond to).

Server Program

- Moving forward our Nodejs server program will now have a pretty basic job to do:
  - Serve static pages
  - Route dynamic requests to appropriate modules and return their response.
- So now let’s look how to create and import our own modules.

Custom Modules

- By default, code in a different file it considered a module.
- However, for another file to use it, that file must “require it”
- A module has a special object called *exports* that we can dynamically bind functions and values to.
- This object is returned when we require the module from a different file
  - And now we can call the bound functions and values from this returned object.

```
// main.js
// require the hello module (hello.js)
var x = require('./hello');
// run the world function, which is located in the hello module
x.world();
x.bye();
console.log(x.x);
```

```
// hello.js
‘use strict’
exports.world = function() {
  console.log('Hello World');
}
exports.bye = function() {
  console.log(‘Goodbye');
}
exports.x = 5;
```

Custom Class Module

- We can also bind a class to the `exports` object (instead of, or in addition to, various functions).
- Lets review making JavaScript classes (using the ES6 standard)

```
'use strict'
class Hello{
      constructor(msg){
            this.msg = msg;
      }
      print(){
            console.log(this.msg);
            return this.msg;
      }
      static about(){
            return "This is a static hello method";
}
exports.Hello = Hello;
```

Custom Class Module

- From a usage standpoint we can create instances of our exported class and call methods on it or call static methods directly on the class

```
'use strict'
//hello.js
class Hello{
      constructor(msg){
            this.msg = msg;
      }
      print(){
            console.log(this.msg);
            return this.msg;
      }
      static about(){
            return "This is a static hello method";
}
exports.Hello = Hello;
```

```
// main.js
// require the hello module (hello.js)
'use strict'
var x = require('./hello');
// run the world function, which is located in the hello module
x.Hello.about(); //static method call

var myHello = new x.Hello("Whatever");  //create an instance
myHello.print();  //call a method on the instance
```

Server Program

- So now we can start letting our server act like a router….

```
var express = require('express');
var app = express();
app.use(express.static("."));

var hello = require('./hello');
var h = new hello('my message!');
var users = require('./users');  //assuming we made such a module

app.get('/greet', function(req,res){
       var resp=h.print();
       res.write(resp);
       res.end();
});

app.get('/list_users', function(req,res){
       var resp=users.getUsers();  //getUsers must be an exported function
       res.write(resp);
       res.end();
});

app.listen(8080,function(){
       console.log('Server Running...');
});
```

Emitting

- Our own modules may need to wait for asynchronous jobs to be completed before returning their data.
- So we'd like to have our modules to able to send a signal (*emit and event*) that users of it can act on.
- To do this we'll have our modules *inherit* the `EventEmitter` module (which is within the `events` module)
  - Our class then **extends** it
  - Our class's constructor must call the inherited class's constructor via the **super()** method

```
'use strict'
var EventEmitter = require('events').EventEmitter;

class Hello extends EventEmitter{
      constructor(){
            super();
      }
}

exports.Hello=Hello
```

Emitting

- Now our object can **emit** events and other code can **listen** for those emissions
  - We’ll listen to emissions by binding a function to the class’s emission message `once`.
  - This is similar to the `on` function we used with response objects, but here we only want to do this binding once to this object, not over and over again.

```
//setup stuff for server above here…

var helloModule = require('./hello');
var h = new helloModule.Hello();

app.get('/greet', function(req,res){
        var resp=helloModule.Hello.world();
        res.write(resp);
        res.end();
});

app.get('/testing', function(req,res){
        h.once('byebye', function(msg){
                res.write(msg);
                res.end();
        });
        h.test();

});

app.listen(8080,function(){
        console.log('Server Running...');
});
```

```
// hello.js
‘use strict’
var EventEmitter = require('events').EventEmitter;

class Hello extends EventEmitter{
        constructor(){super();}
        static world(){
                console.log('Hello World');
                return 'Hello World';
        }
        test(){
                this.emit('byebye',‘<p>Hello World</p>');
        }
}

exports.Hello = Hello;
```

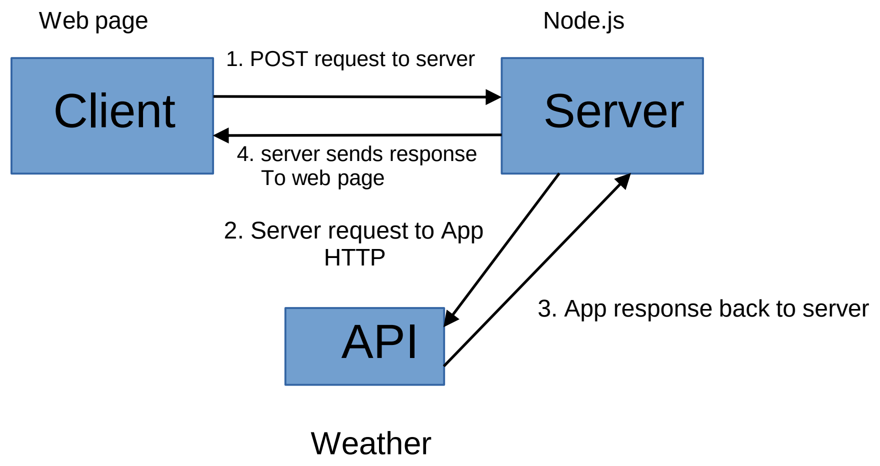
Full Example

- Perhaps your application wants to do an HTTP request to an external site
- We have already see that we *may* be able to do this purely using Javascript+jQuery+AJAX.
  - Without the need for a server.
- However, some sites do not allow client applications to make HTTP requests
  - They must come from a server.
- Therefore, we will show how to make an HTTP request **by the server** and return it to the client.

External HTTP Requests in Node.js

- The flow of this full-fledge example will be:
  1. Client application makes POST request to the local server (via AJAX)
  2. Local server takes this POST data and uses it to make an external HTTP request.
  3. Local server gets back data from it's request (asynchronously)
  4. Local server creates and returns a message using the data acquired from the external HTTP request.
  5. The client application populates a `div` using this AJAX response.
- Notice all the asynchronous events!

Flow Diagram for Weather Program



Client code

- When the input button is clicked, a Javascript function is called makes an AJAX POST request to our local server's `./getWeather` path.
- Included is the data in the `zipcode` field.

```
//Weather.html
<html>
<head>
<title>Weather Node Example</title>
                                        <!----- Jquery includes…. ---->
<script>
       function requestWeather(){
              //SEE NEXT SLIDE
       }
</script>
</head>

<body>
Zip code:<input type=text id="zipcode" /> <input type=button onclick="requestWeather()" value="Get Weather!"/>
<div id=current></div>
</body>
</html>
```

Client code

- In the head is our AJAX call

```
function requestWeather(){
    var zip = $("#zipcode").val();
    params = {
        city: zip
    };
    var URL = "./getWeather";
    $.ajax({
        type: "POST",
        url : URL,
        dataType : "text",
        data : params,
        success : function(msg){
            $("#.current").html(msg);
        },
        error: function(jgXHR, textStatus,errorThrown){
            alert("Error: " + textStatus + " " + errorThrown);
        }
    });

}
```

Server Code

- Now on to the server script…
- Well create a server using express
- We'll also include body-parser to make parsing of the POST request easier.
- We'll attach a response function for handling POST requests to `/requestWeather`
- The handling of the weather stuff will be in a file/module/class `./controllers/weather`
- This class will have a function `getWeather` that will asynchronously get the weather from Open Weather and populate a response.
- And we'll set this server to listen on port 8080

Server Code

```
var express = require('express');
var bodyParser = require("body-parser");

var app = express();
app.use(express.static("."));
app.use(bodyParser.urlencoded({extended:false}));
app.use(bodyParser.json());

var WeatherController = require('./controllers/weather');
var wc = new WeatherController.Weather();

app.post('/getWeather', function (req, res) {
      wc.once('byebye', function(msg){
            res.write(msg);
            res.end();
      });
      wc.getWeather(req);
});
app.listen(8080,function(){
      console.log('Server Running...');
});
```

Weather Module

- The `Weather` module will inherit `EventEmitter` so that it can create an event when it’s gotten all the data from the external weather API.
- It will also read in from a local file the API key.
- The `getWeather` function’s job will be to:
  - Take in the zip code.
  - Make an HTTP request to

    URL = http://api.openweathermap.org/data/2.5/weather?zip=' + req.body.city + '&appid=' + key +'&units=imperial’;
- When the response from the URL is completed, create our own response and emit a message with that response.

Weather Module

```
‘use strict’
var fs = require('fs'); //for reading in key from a file
var request = require(‘request’);  //the request http wrapper module

var EventEmitter = require('events').EventEmitter;

var key = fs.readFileSync(‘./weatherkey.txt','utf8');

class Weather extends EventEmitter{
      constructor(){super();}
      getWeather(request) {
            //next slide
      }
}

exports.Weather = Weather;
```

Weather Module

```
getWeather(req){
      var URL = 'http://api.openweathermap.org/data/2.5/weather';
      URL += '?zip=' + req.body.city + '&appid=' + key +'&units=imperial';

      var self = this;  //need otherwise emission within the response object with be from response
                        //object (that will be local ‘this’)

      request.get(URL, function(error, response, body){
                  var json = JSON.parse(body);
                  var city = json.name;
                  var temp = json.main.temp
                  var html = city+" " + temp +"F‟;
                  console.log(html);
                  self.emit('byebye',html);
      });
}
```

Resources

- For more on the `response` module:

  https://github.com/request/request