Extra notes - Client-side Design and Development

- Dr Nick Hayward

An Introduction to Node.js and Express

A brief introduction to Node.js, and the Express web framework.

Contents

Intro and outline

We can use Node.js and the Express web application framework to create a HTTP based application.

Initial Node.js and Express usage

We can use Express to start building our initial basic application. Express creates a shell for our application with the following command,

```
express nodetest
```

This command makes a new directory in the current working directory, and populates it with the required basic web application directories and files. We can then `cd` to this directory and install any required dependencies,

```
npm install
```

We can then run our new app,

```
npm start
```

or use 'Nodemon' to constantly monitor and update our app.

```
nodemon start
```

We've now tested **npm**, and we've installed our first module with **Express**. Let's now test Express, and build our first, simple server. We'll be working within a newly created test directory, for example

```
|- .
   |- nodetest
      |- node_modules
```

The first thing we need to do is create a JS file to store our server code, so we'll add `server.js`

```
|- .
   |- nodetest
      |- node_modules
      |- server.js
```

We can then start adding our Node.js code to create a simple server.

server.js

We can now add some initial code to get our server up and running.

```
/* a simple Express server for Node.js*/
var express = require("express"),
    http = require("http"),
    appTest;

// create our server - listen on port 3030
appTest = express();
http.createServer(appTest).listen(3030);

// set up routes
appTest.get("/test", function(req, res) {
  res.send("welcome to the 424 test app.");
});
```

Then start and test this server as follows at the command line,

```
node server.js
```

We can open this initial test server in our web browser using the following URL,

```
http://localhost:3030
```

This is the route of our new server. However, to get our newly created route, we can use the following URL,

```
http://localhost:3030/test
```

This will now return our specified route, and output message. We can update our `server.js` file to support root directory level routes. We need to add the following to our server code,

```
appTest.get("/", function(req, res) {
  res.send("Welcome to the 424 server.")
});
```

We can now load our server at the root URL,

```
http://localhost:3030
```

We can also stop our server from the command line with a key combination of `CTRL` and `c`.

Static files and routes

At the moment, our initial Express server is helping us manage some static routes for loading content. In effect, we simply tell the server how to react when a given route is requested.

However, what if we now want to serve some HTML pages. Thankfully, Express allows us to set up routes for static files.

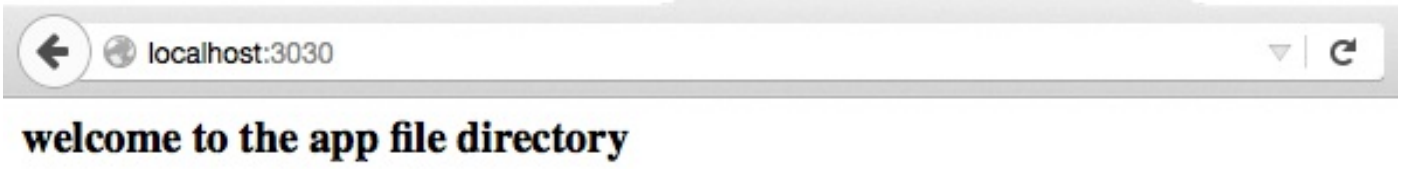
```
//set up static file directory - default route for server
appTest.use(express.static(__dirname + "/app"));
```

In essence, we are now defining Express as a static file server, thereby enabling us to publish our HTML, CSS, and JS files and code from our default directory, `/app`.

So, if we add a new `index.html` file to this directory, and then load our server at `http://localhost:3030/` our server will try to load the default `index.html` file from the `/app` directory.

If the requested file, default or explicit, is not available at the specified default route, the server will then check other available routes. Then, if nothing is still found, it will simply fail and report to the browser.

For example,



Test app - working with JSON

We can now work our way through a basic Node.js app. It will serve our JSON, and then we can read and load them from a standard web app.

Let's start with a new app directory, and setup Node.js and our files. We can use a sample directory structure as follows,

```
|- .
   |- node-test-json
      |- node_modules
      |- server.js
```

Within our app directory, we're going to update our earlier `server.js` file to allow us to serve a route for JSON. We can then use this to read our content for publication.

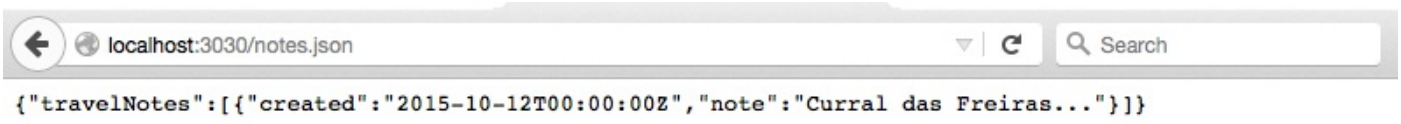
So, our test `server.js` is as follows

```
var express = require('express'),
    http = require("http"),
    jsonApp = express(),
    notes = {
      "travelNotes": [{
      "created": "2015-10-12T00:00:00Z",
```

```
      "note": "Curral das Freiras..."
      }]
    };

jsonApp.use(express.static(__dirname + "/app"));

http.createServer(jsonApp).listen(3030);

//json route
jsonApp.get("notes.json", function(req, res) {
  res.json(notes);
});
```

We're not doing much at the moment, but we can still load the app in the browser, and it will serve the files from the `/app` directory. For example, our `index.html` file.

For example,



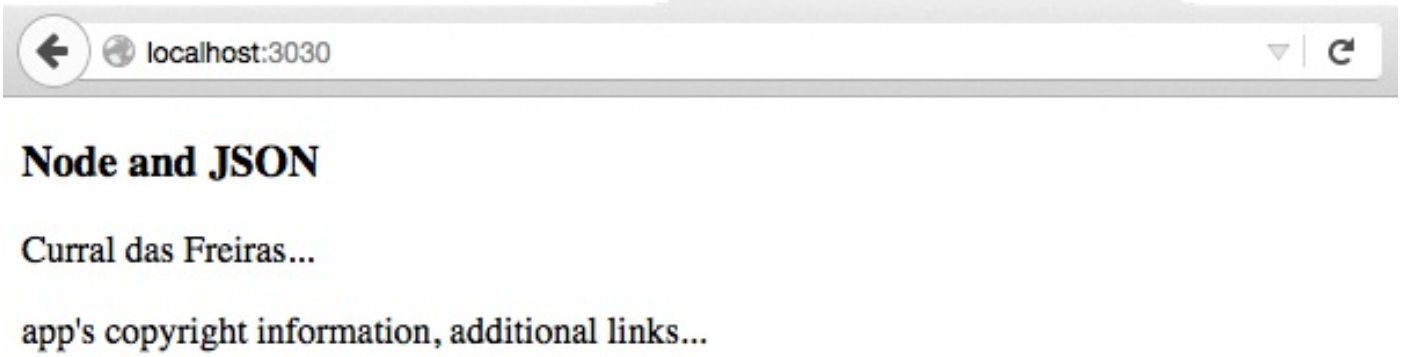
Get JSON route

We now have our `get` routes setup for JSON. So, we need to add some client-side logic to read that route, and render to the browser. We'll simply use a deferred pattern, a Promise object, for reading this exposed route thanks to jQuery's `.getJSON()` function,

```
...
  $.getJSON("notes.json", function (response) {
    console.log("response = "+response.toSource());
    buildNote(response);
  })
...
```

With the response object from our JSON, this time from the server and not a file or API, we can use our familiar helper functions to create and render each note. So, with the response object we can then call a helper function for our custom app.

For example,



Post route

So far, we've seen examples that load JSON data. We've been using jQuery's `.getJSON` function as a way to return our data, and then insert it in the DOM of our application.

However, we can now consider jQuery's reciprocal function to allow us to easily send JSON data to the server.

This update process, whereby we send JSON data to the server over a HTTP protocol, is simply called `post`.

As you might imagine, we begin our updates by creating a new route in our Express server. One that will handle the `post` route.

```
jsonApp.post("/notes", function(req, res) {
  //return simple JSON object
  res.json({
    "message": "post complete to server"
  });
});
```

Whilst this may look similar to our earlier `get` routes, there is a subtle difference. This is inherently due to browser restrictions. In effect, we can't simply request the direct route using our browser, as we did with the `get` routes.

Instead, we have to change the JS we use for the client-side to post to this new route, which then enables us to view the returned message. So, let us update our test app to store data on the server, and then initialise our client with this stored data.

We can start with a simple check that the post route is working correctly. We can add a button, and submit a request to the post route, and then wait for the response. For example, we'll add the following event handler for a button,

```
$("#post").on("click", function() {
  $.post("notes", {}, function (response) {
    console.log("server post response returned..." + response.toSource());
  })
});
```

When we submit a `post` request, we specify the route for the post, then the data as an object, and then a callback for the server's response. This will then return the following output to the browser's console,

```
server post response returned...({message:"post complete to server"})
```

This simply returns the specified post JSON in the Node.js server file.

Post data to the server

We can now send some data to the server, basically populating our object. We need to update the server to handle

this incoming object, in effect making it usable within our application.

What we need to do is process the submitted jQuery JSON into a JavaScript object that the server can use for processing and storing. Thankfully, we can use the **Express** module's `body-parser` plugin.

So, we can update our `server.js` as follows,

```
//add body-parser for JSON parsing etc...
var bodyParser = require("body-parser");
...
//Express will parse incoming JSON objects
jsonApp.use(bodyParser.urlencoded({ extended: false }));
...
```

Effectively, as the server receives a JSON object, it will now parse, or process, this object to ensure that it can be stored on the server.

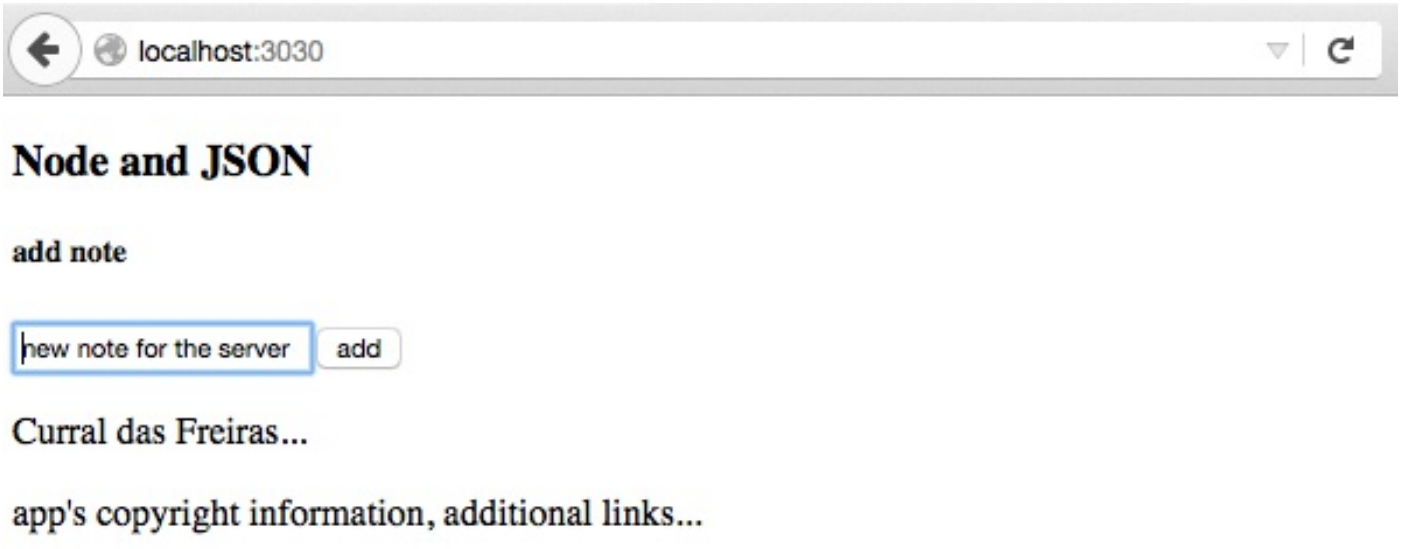
We can now update our test button's event handler to send a new note as a JSON object. This note will retrieve its new content from the input field, and then get the current time from the node server.

```
$(".note-input button").on("click", function() {
  //get values for new note
  var note_text = $(".note-input input").val();
  var created = new Date();
  //create new note
  var newNote = {"created":created, "note":note_text};
  //post new note to server
  $.post("notes", newNote, function (response) {
    console.log("server post response returned..." + response.toSource());
  })
});
```

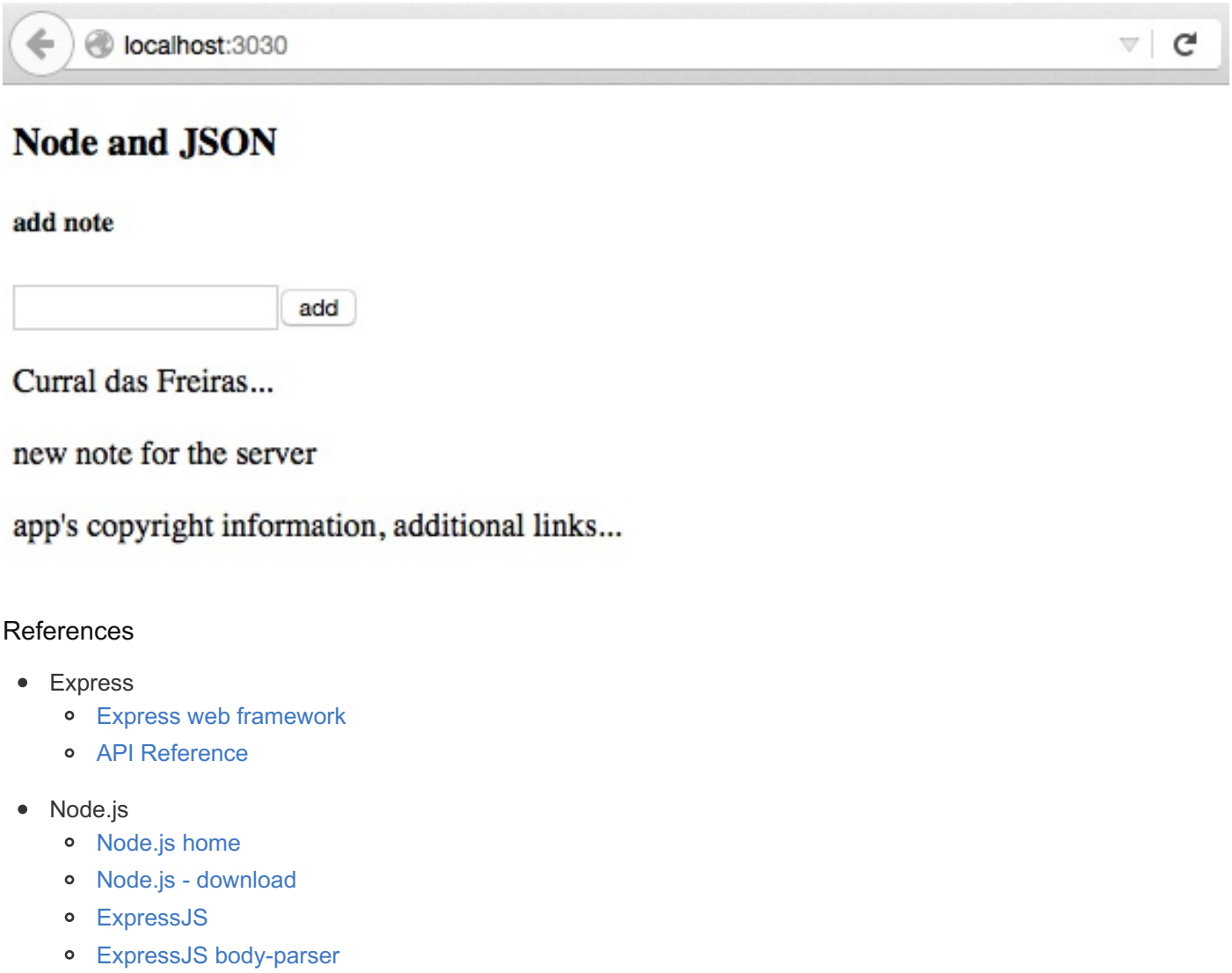
As we post new notes to the server, they will now be stored on the server whilst it remains live.

So, our server will now post new notes to the server, store them, and then get them for rendering. It will persist our notes until the server is restarted. This is a step forward for our test apps, but we still need a way to persist the data beyond the uptime of the server.

For example,



and,



References

- Express
  - Express web framework
  - API Reference
- Node.js
  - Node.js home
  - Node.js - download
  - ExpressJS
  - ExpressJS body-parser