

Comp 324/424 - Client-side Web Design

Spring Semester 2017 - Week 10

Dr Nick Hayward

Contents

- Complementary Server-side considerations
 - *Node.js*

JS Server-side considerations - save data

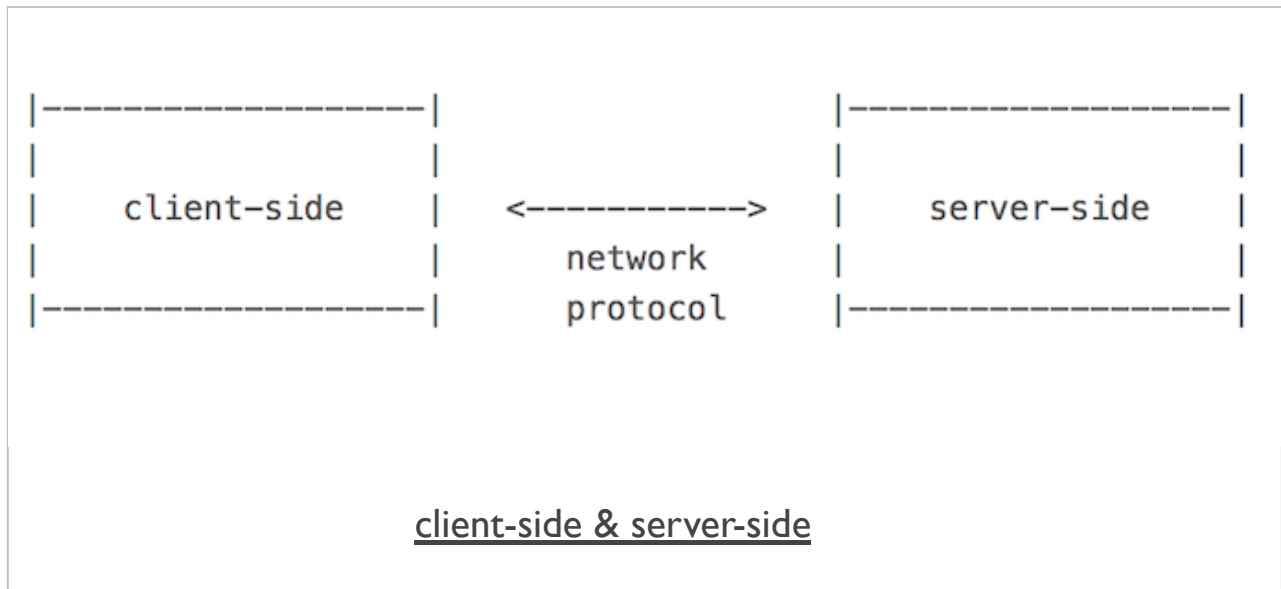
save JSON in travel notes app

- need to be able to save our simple notes
- now load from a JSON file as the app starts
 - *also we can add new notes, delete existing notes...*
- not as simple as writing to our existing JSON file direct from JS
 - *security implications if that was permitted directly from the browser*
- need to consider a few server-side options
- could use a combination of PHP on the server-side
 - *with AJAX jQuery on the client-side*
 - *traditional option with a simple ajax post to a PHP file on the server-side*
- consider JavaScript options on the client and server-side
- brief overview of working with **Node.js**

Server-side considerations - intro

- normally define computer programs as either client-side or server-side programs
- server-side programs normally abstract a resource over a network
 - *enabling many client-side programs to access at the same time*
 - *a common example is file requests and transfers*
- we can think of the client as the web browser
- a web server as the remote machine abstracting resources
- abstracts them via **hypertext transfer protocol**
 - *HTTP for short*
- designed to help with the transfer of HTML documents
 - *HTTP now used as an abstracted wrapper for many different types of resources*
 - *may include documents, media, databases...*

Image - Client-side and server-side computing



Server-side considerations - Node.js

intro - what is Node.js?

- Node.js is, in essence, a JavaScript runtime environment
 - *designed to be run outside of the browser*
- designed as a general purpose utility
- can be used for many different tasks including
 - *asset compilation*
 - *monitoring*
 - *scripting*
 - *web servers*
- with Node.js, role of JS is changing
 - *moving from client-side to a support role in back-end development*

Server-side considerations - Node.js

intro - speed of Node.js

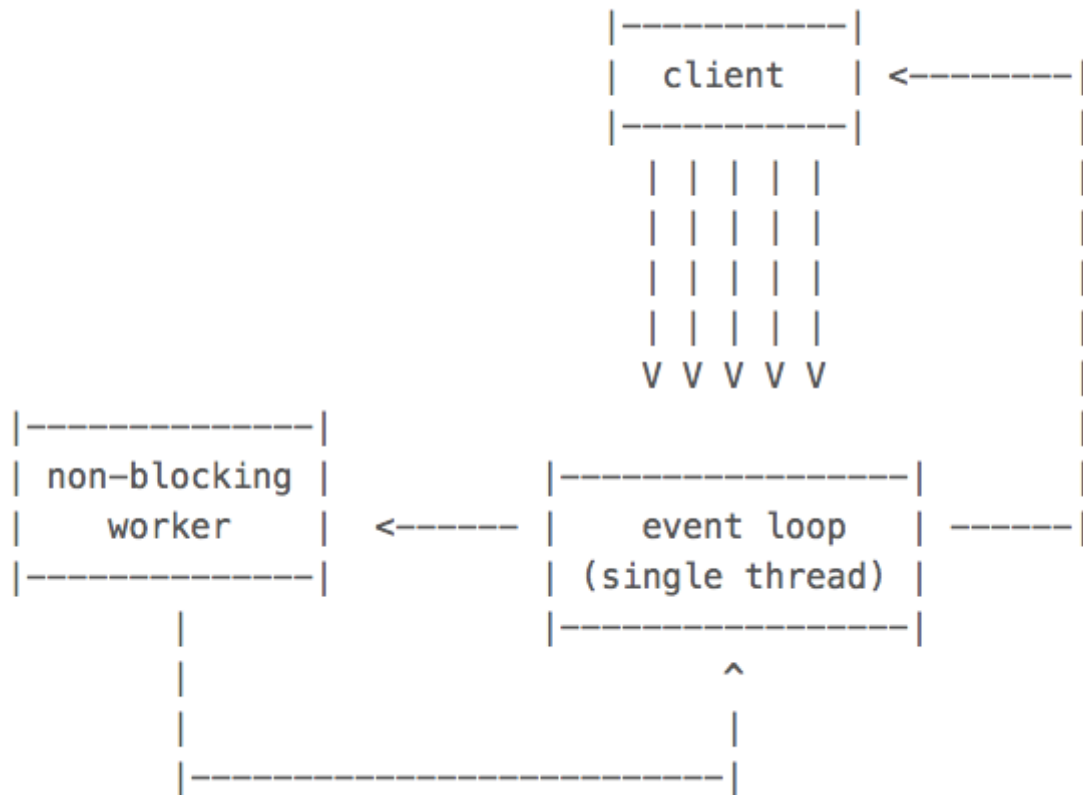
- a key advantage touted for Node.js is its speed
- many companies have noted the performance benefits of implementing Node.js
 - *including PayPal, Walmart, LinkedIn...*
- a primary reason for this speed boost is the underlying architecture of Node.js
- Node.js uses an **event-based** architecture
- instead of a threading model popular in compiled languages
- Node.js uses a single event thread by default
- all I/O is asynchronous

Server-side considerations - Node.js

intro - conceptual model for processing in Node.js

- how does Node.js, and its underlying processing model, actually work?
- client sends a hypertext transfer protocol, HTTP, request
 - *request or requests sent to Node.js server*
- event loop is then informed by the host OS
 - *passes applicable request and response objects as JavaScript closures*
 - *passed to associated worker functions with callbacks*
- long running jobs continue to run on various assigned worker threads
- responses are sent from the non-blocking workers back to the main event loop
 - *returned via a callback*
- event loop returns any results back to the client
 - *effectively when they're ready*

Image - Client-side and server-side computing



Node.js - conceptual model for processing

Server-side considerations - Node.js

intro - threaded architecture

- concurrency allows multiple things to happen at the same time
- common practice on servers due to the nature of multiple user queries
- Java, for example, will create a new thread on each connection
 - *threading is inherently resource expensive*
- size of a thread is normally around 4MB of memory
- naturally limits the number of threads that can run at the same time
- also inherently more complicated to develop platforms that are thread-safe
 - *thereby allowing for such functionality*
- due to this complexity
 - *many languages, eg: Ruby, Python, and PHP, do not have threads that allow for real concurrency*
 - *without custom binaries*
- JavaScript is similarly single-threaded
 - *able to run multiple code paths in parallel due to **events***

Server-side considerations - Node.js

intro - event-driven architecture

- JavaScript originally designed to work within the confines of the web browser
- had to handle restrictive nature of a single thread and single process for the whole page
- synchronous blocking in code would lock up a web page from all actions
 - *JavaScript was built with this in mind*
- due to this style of I/O handling
 - *Node.js is able to handle millions of concurrent requests on a single process*
- added, using libraries, to many other existing languages
 - *Akka for Java*
 - *EventMachine for Ruby*
 - *Twisted for Python*
 - ...
- JavaScript syntax already assumes events through its use of callbacks
- **NB:** if a query etc is CPU intensive instead of I/O intensive
 - *thread will be tied up*
 - *everything will be blocked as it waits for it to finish*

Server-side considerations - Node.js

intro - callbacks

- in most languages
 - *send an I/O query & wait until result is returned*
 - *wait before you can continue your code procedure*
- for example, submit a query to a database for a user ID
 - *server will pause that thread/process until database returns result for ID query*
- in JS, this concept is rarely implemented as standard
- in JS, more common to pass the I/O call a **callback**
- in JS, this **callback** will need to run when task is completed
 - *eg: find a user ID and then do something, such as output to a HTML element*
- biggest difference in these approaches
 - *whilst the database is fetching the user ID query*
 - *thread is free to do whatever else might be useful*
 - *eg: accept another web request, listen to a different event...*
- this is one of the reasons that Node.js returns good benchmarks and is easily scaled
- **NB:** makes Node.js well suited for I/O heavy and intensive scenarios

Server-side considerations - Node.js

install Node.js

- a number of different ways to install **Node.js**, **npm**, and the lightweight, customisable web framework **Express**
- run and test Node.js on a local Mac OS X or Windows machine
- download and install a package from the following URL
 - *Node.js - download*
- install the Node module, **Express**
- Express is a framework for web applications built upon Node.js
 - *minimal, flexible, & easily customised server*
- use *npm* to install the Express module

```
npm install -g express
```

- *-g* option sets a global flag for Express instead of limited local install
- installs Express command line tool
 - *allows us to start building our basic web application*
- now also necessary to install Express application generator

```
npm install -g express-generator
```

Server-side considerations - Node.js

NPM - intro

- **npm** is a package manager for Node.js
- Developers can use **npm** to share and reuse modules in Node.js applications
- **npm** can also be used to share complete Node.js applications
- example modules might include
 - *Markup, YAML etc parsers*
 - *database connectors*
 - *Express server*
 - ...
- **npm** is included with the default installers available at the Node.js website
- test whether **npm** is installed, simply issue the following command

```
npm
```

- should output some helpful information if **npm** is currently installed
- **NB:** on a Unix system, such as OS X or Linux
 - *best to avoid installing **npm** modules with `sudo` privileges*

Server-side considerations - Node.js

NPM - installing modules

- install existing **npm** modules, use the following type of command

```
npm install express
```

- this command installs module named `express` in the current directory
- it will act as a local installation within the current directory
- installing in a folder called `node_modules`
 - *this is the default behaviour for current installs*
- we can also specify a global install for modules
 - eg: we may wish to install the **express** module with global scope

```
npm install -g express
```

- again, the `-g` flag specifies the required global install

Server-side considerations - Node.js

NPM - importing modules

- import, or effectively add, modules in our Node.js code
 - *use the following declaration*

```
var module = require('express');
```

- when we run this application
 - *Node.js looks for the required module library and its source code*

Server-side considerations - Node.js

NPM - finding modules

- official online search tool for **npm** can be found at
 - *npmjs*
- top packages include options such as
 - *browserify*
 - *express*
 - *grunt*
 - *bower*
 - *karma*
 - ...
- also search for Node modules directly
 - *search from the command line using the following command*

```
npm search express
```

- returns results for module names and descriptions

Server-side considerations - Node.js

NPM - specifying dependencies

- ease Node.js app installation
 - *specify any required dependencies in an associated `package.json` file*
- allows us as developers to specify modules to install for our application
 - *which can then be run using the following command*

```
npm install
```

- helps reduce the need to install each module individually
- helps other users install an application as quickly as possible
- our application's dependencies are stored in one place
- example `package.json`

```
{
  "name": "app",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.2.x",
    "underscore": "-1.2.1"
  }
}
```

Server-side considerations - Node.js

initial Express usage

- now use Express to start building our initial basic web application
- Express creates a basic shell for our web application
 - *cd to working directory and use the following command*

```
express /node/test-project
```

- command makes a new directory
 - *populates with required basic web application directories and files*
- cd to this directory and install any required dependencies,

```
npm install
```

- then run our new app,

```
npm start
```

- or run and monitor our app,

```
nodemon start
```

Server-side considerations - Node.js

initial Express server - setup

- we've now tested **npm**, and installed our first module with **Express**
- test **Express**, and build our first, simple server
- initial directory structure

```
| - .  
  | - 424-node  
    | - node_modules
```

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

```
| - .  
  | - 424-node  
    | - node_modules  
    | - server.js
```

- start adding our Node.js code to create a simple server

Server-side considerations - Node.js

initial Express server - server.js - part I

- 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
```

Server-side considerations - Node.js

initial Express server - server.js - part 2

- open our web browser, and use the following URL

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

- this is the route of our new server
 - *to get our newly created route use the following URL*

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

- this will now return our specified route, and output message
- update our `server.js` file to support root directory level routes

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

- now load our server at the root URL

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

- stop server from command line using CTRL and c

Server-side considerations - Node.js

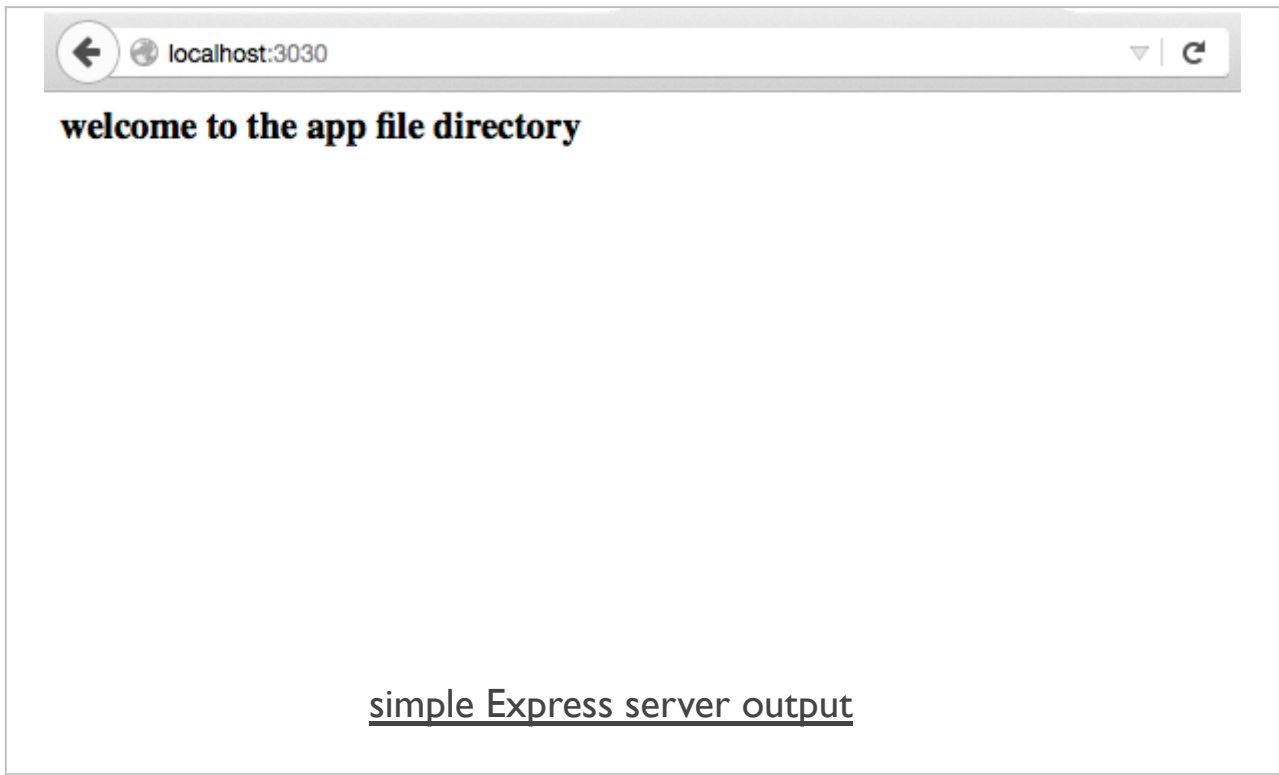
initial Express server - server.js - part 3

- currently, initial Express server is managing some static routes for loading content
 - *we simply tell the server how to react when a given route is requested*
- what if we now want to serve some HTML pages?
 - *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"));
```

- now defining Express as a static file server
 - *enabling us to publish our HTML, CSS, and JS files*
 - *published from our default directory, /app*
- if requested file not available
 - *server will check other available routes*
 - *or report error to browser if nothing found*
- DEMO - 424-node

Image - Client-side and server-side computing



Server-side considerations - Node.js

working with data - JSON

- let us now work our way through a basic Node.js app
- serve our JSON, then read and load from a standard web app
- create our initial `server.js` file

```
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);
});
```

Image - Client-side and server-side computing



Server-side considerations - Node.js

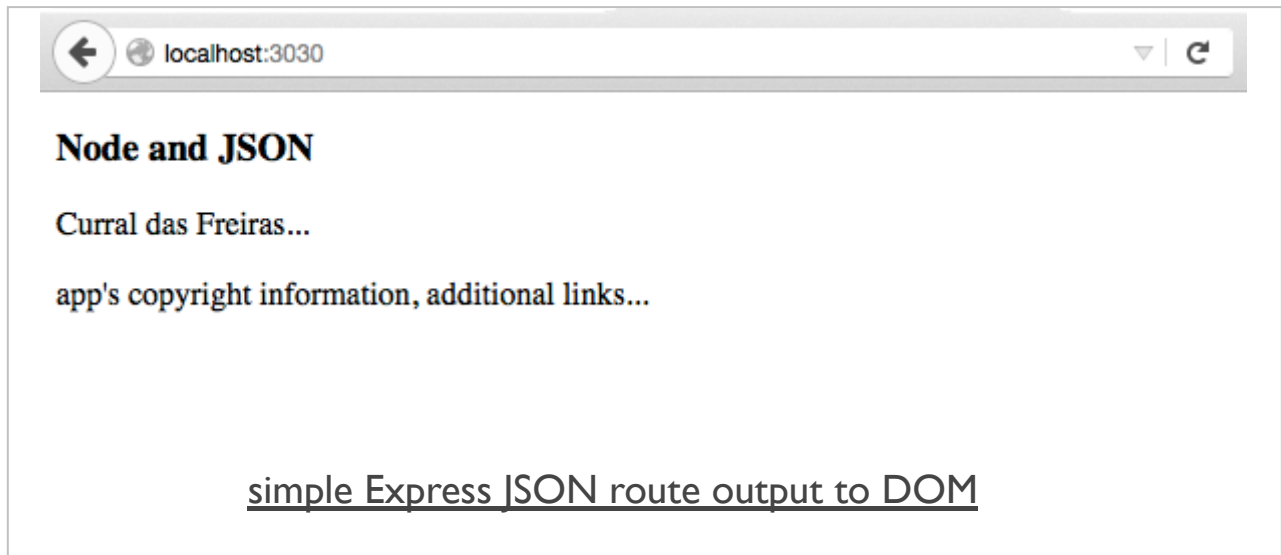
working with data - JSON

- now have our get routes setup for JSON
- now add some client-side logic to read that route
- render to the browser
- same basic patterns we've seen before
 - using jQuery's `.getJSON()` function

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

- response object from our JSON
 - *this time from the server and not a file or API*
- use our familiar functions to create and render each note
 - *call our normal `buildNote()` function*
- DEMO - 424-node-json1

Image - Client-side and server-side computing



Server-side considerations - Node.js

working with data - post data

- we've seen examples that load JSON data
 - using jQuery's `.getJSON()` function
- now consider jQuery's `post` function
 - allow us to easily send JSON data to the server
 - simply called *post*
- 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"  
  });  
});
```

Server-side considerations - Node.js

working with data - post data

- may look similar to our earlier `get` routes
 - *difference due to browser restrictions*
 - *can't simply request direct route using our browser*
 - *as we did with `get` routes*
- need to change JS we use for the client-side
 - *allows us to post new route*
 - *then enables view of the returned message*
- update our test app to store data on the server
 - *then initialise our client with this stored data*

Server-side considerations - Node.js

working with data - post data

- start with a simple check that the post route is working correctly
 - *add a button, submit a request to the post route, and then wait for the response*
 - *add event handler for a button*

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

- submit a post request
 - *specify the route for the post to the Node.js server*
 - *then specify the data to post - an empty object in this example*
 - *the specify a callback for the server's response*
- test returns the following output to the browser's console,

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

Server-side considerations - Node.js

working with data - post data

- now send some data to the server
 - *add new note to our object*
- update the server to handle this incoming object
 - *process the submitted jQuery JSON into a JavaScript object*
 - *ready for use with the server*
- use the **Express** module's body-parser plugin
- update `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 }));
...
```

- as server receives new JSON object
 - *it will now parse, or process, this object*
 - *ensures it can be stored on the server for future use*

Server-side considerations - Node.js

working with data - post data

- now update our test button's event handler
 - *send a new note as a JSON object*
- note will retrieve its new content from the input field
 - *gets the current time from the node server*

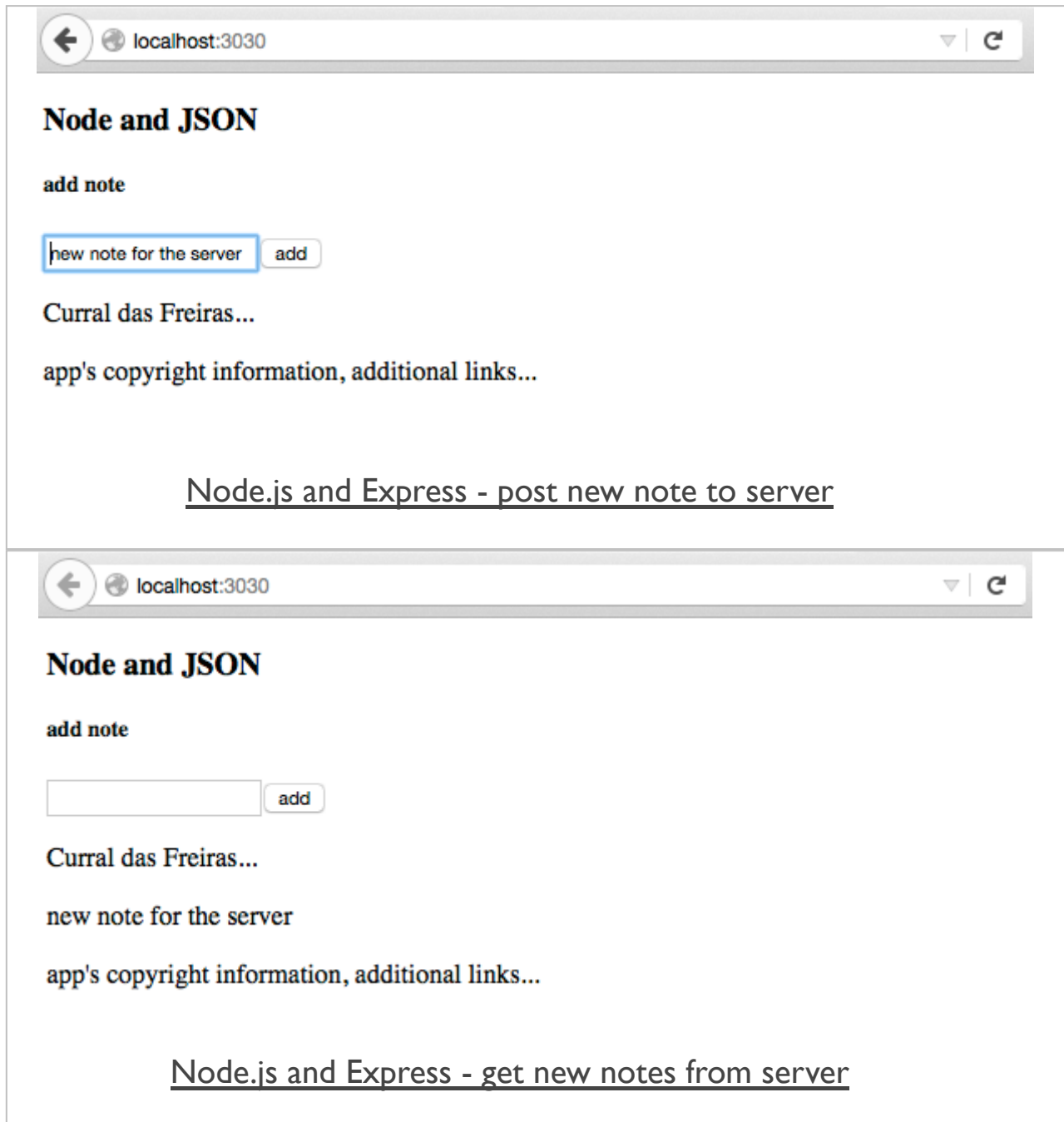
```
$(".note-input button").on("click", function() {  
    //get values for new note  
    var note_text = $(".note-input input").val();  
    var created = new Date();  
    //create new note  
    var newNote = {"created":created, "note":note_text};  
    //post new note to server  
    $.post("notes",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

- input field and button follow the same pattern as previous examples

```
<!-- note input -->  
<section class="note-input col-6">  
    <h5>add note</h5>  
    <input><button>add</button>  
</section>
```

- DEMO - 424-node-json2

Image - Client-side and server-side computing



Demos

Node.js

- 424-node
- 424-node-json1
- 424-node-json2

References

- Node.js
 - *Node.js home*
 - *Node.js - download*
 - *ExpressJS*
 - *ExpressJS body-parser*