Comp 324/424 - Client-side Web Design

Spring Semester 2020 - Week 12

Dr Nick Hayward

JS Server-side considerations - save data

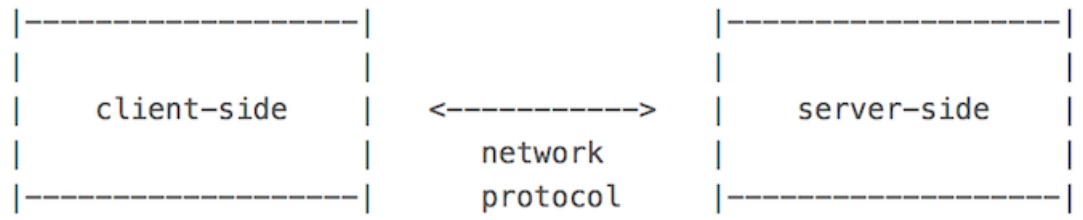
save JSON in travel notes app

- 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



client-side & server-side

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

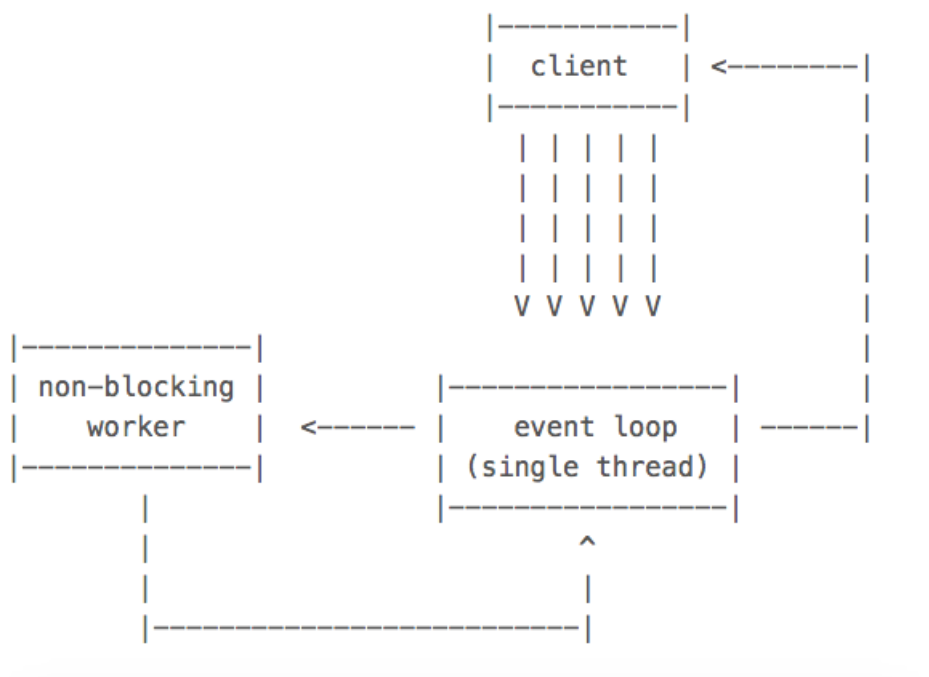
- a key advantage touted for Node.js is its speed
- many companies have noted the performance benefits of implementing Node.js
  - *including PayPal, Walmart, LinkedIn…*
- a primary reason for this speed boost is the underlying architecture of Node.js
- Node.js uses an **event-based** architecture
- 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*
- example 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

CommonJS modules - custom design and usage

- extra notes available on CommonJS module usage
  - *custom design and usage*
  - *library structure and development*
- extra source code examples available
  - *general usage*
  - *custom modules*
  - *custom library example*

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 1

- 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

welcome to the app file directory

simple Express server output

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

simple Express JSON route output

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

simple Express JSON route output to DOM

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*

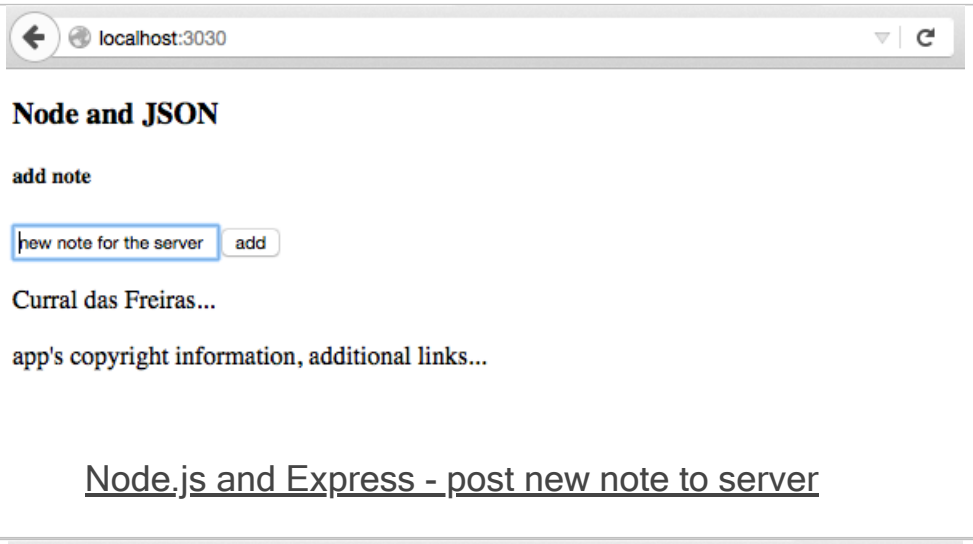
```
$(".note-input button").on("click", function() {
  //get values for new note
  var note_text = $(".note-input input").val();
  var created = new Date();
  //create new note
  var newNote = {"created":created, "note":note_text};
  //post new note to server
  $.post("notes", newNote, function (response)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

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

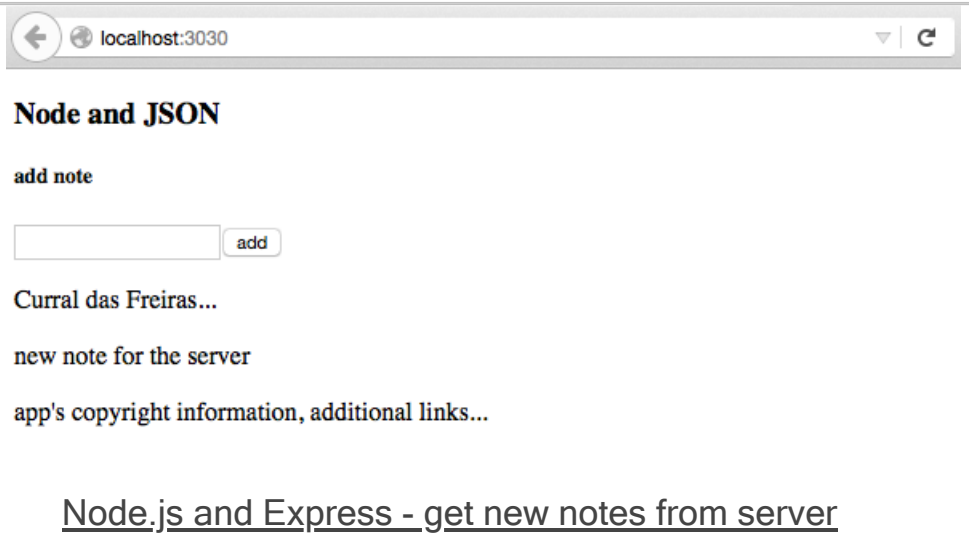
```
<!-- note input -->
<section class="note-input col-6">
    <h5>add note</h5>
    <input><button>add</button>
</section>
```

- DEMO - 424-node-json2

Image - Client-side and server-side computing



Node.js and Express - post new note to server



Node.js and Express - get new notes from server

Node.js extras - API examples

- various custom API examples
  - *ToDos & ToDos with testing*
  - *authentication examples*
  - *Notetaking*
  - *…with Socket.io*
  - *…*
- Twitter with Node.js custom server
  - *user queries &c.*
  - *OAuth based login and authentication*
- Yelp with Node.js custom server
  - *custom server and remote API query*
  - *sample handling of local API for queries*

JavaScript - modular design

ES Module pattern - intro

- simpler and easier to work with than CommonJS
  - *in most examples...*
- JavaScript `strict` mode is enabled by default
- `strict` mode helps with language usage - check for poor usage
  - *stops hoisting of variables*
  - *variables must be declared*
  - *function parameters must have unique name*
  - *assignment to read-only properties throws errors*
  - *...*
- modules are exported with `export` statements
- modules are imported with `import` statements

JavaScript - modular design

ES Module pattern - `export` statements

- ES6 modules are individual files
  - *expose an API using `export` statements*
- declarations are scoped to the local module
- e.g. variables declared inside a module
  - *not available to other modules*
  - *need to be explicitly exported in module API*
  - *need to be imported for usage in another module*
- export statements may only be added to *top-level* of a module
  - *e.g. not in function expression &c.*
- cannot dynamically define and expose API using methods
  - *unlike CommonJS module system - Node.js &c.*

JavaScript - modular design

ES Module pattern - `export default`

- common option is to export a default binding, e.g.

```
export default `hello world`
```

```
export default {
    name: 'Alice',
    place: 'Wonderland'
}
```

```
export default [
    'Alice', 'Wonderland'
]
```

```
export default function name() {
    ...
}
```

JavaScript - modular design

ES Module pattern - bindings

- ES modules export **bindings**
  - *not values or references*
- e.g. an export of `count` variable from a module
  - *`count` is exported as a binding*
  - *export is bound to `count` variable in the module*
  - *value is subject to changes of `count` in module*
- offers flexibility to exported API
  - *e.g. `count` might originally be bound to an object*
  - *then changed to an array…*
- other modules consuming this export
  - *they would see change as `count` is modified*
  - *modified in module and exported…*
- n.b. take care with this usage pattern
  - *useful for counters, logs &c.*
  - *can cause issues with API usage for a module*

JavaScript - modular design

ES Module pattern - named export

- we may define bindings for export
- instead of assigning properties to implicit export object
  - *e.g.*

```
export let counter = 0
export const count = () => counter++
```

- cannot refactor this example for named export
  - *syntax error will be thrown*
  - *e.g.*

```
let counter = 0
const count = () => counter++
export counter // this will return syntax error
export count
```

- rigid syntax helps with analysis, parsing
  - *static analysis for ES modules*

JavaScript - modular design

ES Module pattern - export lists

- lists provide a useful solution to previous refactor issue
- syntax for list export easy to parse
- export lists of named *top-level* declarations
  - *variables &c.*
- e.g.

```
let counter = 0
const count = () => counter++
export { counter, count }
```

- also rename binding for export, e.g.

```
let counter = 0
const count = () => counter++
export { counter, count as increment }
```

- define `default` with export list, e.g.

```
let counter = 0
const count = () => counter++
export { counter as default, count as increment }
```

JavaScript - modular design

ES Module pattern - `export from ...`

- expose another module's API using `export from...`
  - *i.e. a kind of pass through...*
- e.g.

```
export { increment } from './myCounter.js'
```

- bindings are not imported into module's local scope
- current module acts as conduit, passing bindings along export/import chain...
- module does not gain direct access to `export from ...` bindings
  - *e.g. if we call `increment` it will throw a `ReferenceError`*
- aliases are also possible for bindings with `export from...`
  - *e.g.*

```
export { increment as addition } from './myCounter.js'
```

JavaScript - modular design

ES Module pattern - `import` statements

- use `import` to load another module
- `import` statement are only allowed in top level of module definition
  - *same as `export` statements*
  - *helps compilers simplify module loading &c.*
- import default exports
  - *give default export a name as it is imported*
  - *e.g.*

```
import counter from './myCounter.js'
```

- importing binding to `counter`
- syntax different from declaring a JS variable

JavaScript - modular design

ES Module pattern - `import` named exports

- also imported any named exports
  - *import more than just default exports*
- named import is wrapped in braces
  - *e.g.*

```
import { increment } from './myCounter.js'
```

- also import multiple named exports
  - *e.g.*

```
import { increment, decrement } from './myCounter.js'
```

- import aliases are also supported
  - *e.g.*

```
import { increment as addition } from './myCounter.js'
```

- combine default with named
  - *e.g.*

```
import counter, { increment } from './myCounter.js'
```

JavaScript - modular design

ES Module pattern - `import` with wildcard

- we may also import using the *wildcard* operator
  - *e.g.*

```
import * as counter from './myCounter.js'
counter.increment()
```

- name for wildcard import acts like object for module
- call module exports on wildcard

```
import * as counter from './myCounter.js'
counter.increment()
```

- common pattern for working with libraries &c.

JavaScript - modular design

ES Module pattern - benefits & practical usage

- offers ability to explicitly publish an API
  - *keeps module content local unless explicitly exported*
- similar function to *getters* and *setters*
  - *explicit way in and out of modules*
  - *explicit options for reading and updating values…*
- code becomes simpler to write and manage
  - *module offers encapsulation of code*
- import binding to variable, function &c.
  - *then use it as normal…*
- removes need for encapsulation in main JS code
  - *e.g. with patterns such as IIFE…*
- *n.b.* need to be careful how we use modules
  - *e.g. priority for access, security, testing &c.*
  - *all now moved to individual modules…*

JavaScript - modular design

ES Module pattern - Lib structure

- Modules in JavaScript are not a new concept
  - *e.g. CommonJS is a popular option for modular development with Node.js*
- a built-in option for plain JavaScript, *ES Modules*.
- use this option to develop and structure custom module libraries
- e.g.
  - *abstract utility modules*
  - *custom draw libraries*
  - *game renderers*
  - *...*

JavaScript - modular design

ES Module pattern - JS library

- an example JS library - define the following directory structure

```
.
|-- lib
|   |-- spire
|   |   |-- helpers
|   |   |   |__ log.js
|   |   |__ spire.js
|__ main.js
| ...
```

- `lib` directory contains custom JS libraries, which may then be imported for use within an app
- for app usage, we might structure it as follows

```
.
|-- lib
|   |-- spire
|   |   |-- helpers
|   |   |   |__ log.js
|   |   |__ spire.js
|__ index.html
|__ main.js
| ...
```

JavaScript - modular design

ES Module pattern - JS library - `main.js`

- `main.js` file is loaded from the `index.html` file
  - *acts as the loader file for JS in an example app*
- also import example *Spire* JS library into an app using this main loader file, e.g.

```
import Spire from './lib/spire/spire.js';
```

- `Spire` object is the access point to the exported methods and variables for custom JS library

JavaScript - modular design

ES Module pattern - JS library - basic usage

- a custom JS library may then be accessed using this `Spire` object
- e.g. we might call a method from the library

```
const greeting = 'greetings from the planet Earth';
// basic log to console
Spire.log(`${greeting}...we wish you well`);
```

- custom method `log()` provides a reusable method
  - *e.g. use for various logging options in the application*
- might also call the following method using the same pattern

```
Spire.dir({'name': 'test dir logger...'});
```

JavaScript - modular design

ES Module pattern - JS library - module usage

- sample usage might include such *helpers*
  - *we may package in the directory `spire/helpers/`*
  - *e.g., we currently have a `log.js` module for various custom loggers*

```
// basic logger to console
function log(value, ...values) {
  const logValue = console.log(value, ...values);

  return logValue;
}

// directory logger to console
function dir(value, ...values) {
  const dirValue = console.dir(value, ...values);

  return dirValue;
}
```

- we may then simply export these methods from the `log.js` module, e.g.

```
export {
    log,
    dir
}
```

- interface for this module has now been defined relative to the above exported modules

JavaScript - modular design

ES Module pattern - JS library - import modules

- import this module
  - *allow a module to use these exported methods*
  - *interact with the exposed interface*
- as part of the JS library structure we may define
  - *a root module for organising a unified interface for the overall library*
- e.g. use the module `spire.js` to import required modules and their interfaces

```
import * as loggers from './helpers/log.js';
```

- then define a `Spire` object for the overall library, e.g.

```
const Spire = {
    log: loggers.log,
    dir: loggers.dir,
}
```

- this is then exported as the general interface for the Spire JS library, e.g.

```
export default Spire;
```

Responsive Design & Development - Modular Designs

Fun Exercise

Three responsive designs,

- Modular designs - http://linode4.cs.luc.edu/teaching/cs/demos/424/gifs/modular/
  - *Home Design*
  - *Reminders*
  - *Watches*

For each design, consider the following

- define perceived modules for each app
  - *where might you use a module?*
- what type of modules can you define in each app?
  - *e.g. logical, structural, design, performance…*
- from a developer perspective
  - *consider primary modular groupings*
  - *does each module purpose help with testing?*
  - *can each module be decoupled from app?*
    - e.g. test and use outside of current app…

~ 10 minutes

Server-side considerations - data storage

intro

- tested Node.js, created a server for hosting our files and routes with ExpressJS
  - *read JSON from the server*
  - *updated our JSON on the server-side*
- works well as long as we do not need to restart, repair, update etc our server
- data lost with restart etc...
- need to consider a persistent data storage
  - *independent from the application*
- NoSQL options such as Redis and MongoDB
- integration with Node.js

Server-side considerations - data storage

SQL or NoSQL

- common database usage and storage
  - *often thought solely in terms of SQL, or structured query language*
- SQL used to query data in a relational format
- relational databases, for example MySQL or PostgreSQL, store their data in tables
  - *provides a semblance of structure through rows and cells*
  - *easily cross-reference, or relate, rows across tables*
- a relational structure to map authors to books, players to teams...
  - *thereby dramatically reducing redundancy, required storage space...*
- improvement in storage capacities, access...
  - *led to shift in thinking, and database design in general*
- started to see introduction of non-relational databases
  - *often referred to simply as **NoSQL***
- with NoSQL DBs
  - *redundant data may be stored*
  - *such designs often provide increased ease of use for developers*
- some NoSQL examples for specific use cases
  - *eg: fast reading of data more efficient than writing*
  - *specialised DB designs*

Server-side considerations - data storage

Redis - intro

- Redis provides an excellent example of NoSQL based data storage
- designed for fast access to frequently requested data
- improvement in performance often due to a reduction in perceived reliability
  - *due to in-memory storage instead of writing to a disk*
- able to flush data to disk
  - *performs this task at given points during uptime*
  - *for majority of cases considered an in-memory data store*
- stores this data in a **key-value** format
  - *similar in nature to standard object properties in JavaScript*
- Redis often a natural extension of conventional data structures
- Redis is a good option for quick access to data
  - *optionally caching temporary data for frequent access*

Server-side considerations - data storage

Redis - installation

- On OS X, use the Homebrew package manager to install Redis

```
brew install redis
```

- Windows port maintained by the Microsoft Open Tech Group - Redis
  - *or use Windows package manager - https://chocolatey.org/*
  - *try WSL*
  - *n.b. Redis on Windows is not recommended…*
- for Linux - download, extract, and compile Redis

```
$ wget http://download.redis.io/releases/redis-3.0.5.tar.gz
$ tar xzf redis-3.0.5.tar.gz
$ cd redis-3.0.5
$ make
```

Server-side considerations - data storage

Redis - server and CLI

- start the Redis server with the following command,

```
redis-server
```

- interact with our new server directly using the CLI tool,

```
redis-cli
```

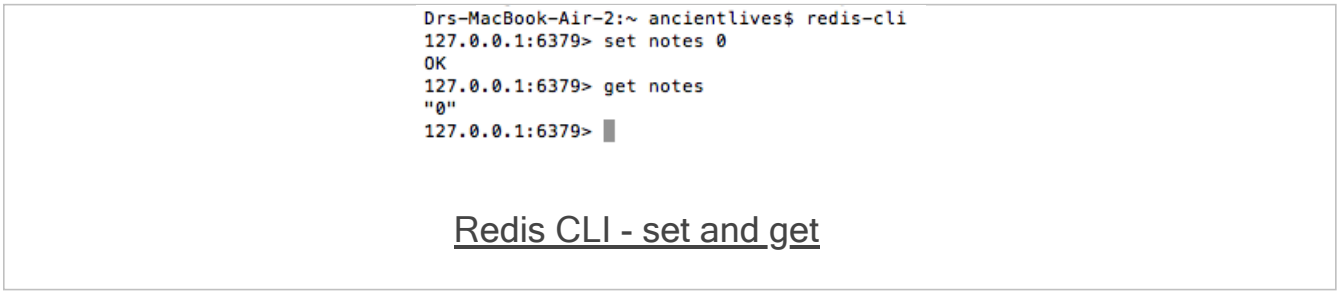
- store some data in Redis using the `set` command
  - *create a new key for `notes`, and then set its value to `0`*
  - *if value is set, Redis returns `OK`*

```
set notes 0
```

- retrieve a value using the `get` command
  - *returns our set value of `0`*

```
get notes
```

Image - Client-side and server-side computing



Redis CLI - set and get

Server-side considerations - data storage

Redis - server and CLI

- also manipulate existing values for a given key
  - *eg: increment and decrement a value, or simply delete a key*
- increment key `notes` value by 1

```
incr notes
```

- decrement key `notes` value by 1

```
decr notes
```

- we can then increment or decrement by a specified amount

```
// increment by 10
incrby notes 10
// decrement by 5
decrby notes 5
```

- delete our key

```
// single key deletion
del notes
// multiple keys deletion
del notes notes2 notes3
```

Image - Client-side and server-side computing

```
Drs-MacBook-Air-2:~ ancientlives$ redis-cli
127.0.0.1:6379> set notes 0
OK
127.0.0.1:6379> get notes
"0"
127.0.0.1:6379> incr notes
(integer) 1
127.0.0.1:6379> incr notes
(integer) 2
127.0.0.1:6379> get notes
"2"
127.0.0.1:6379> decr notes
(integer) 1
127.0.0.1:6379> get notes
"1"
127.0.0.1:6379> incrby notes 10
(integer) 11
127.0.0.1:6379> get notes
"11"
127.0.0.1:6379> decrby notes 5
(integer) 6
127.0.0.1:6379> get notes
"6"
```

Redis CLI - increment and decrement

Server-side considerations - data storage

Redis and Node.js setup

- test Redis with our Node.js app
- new test app called `424-node-redis1`

```
|- 424-node-redis1
   |- app
      |- assets
   |- node_modules
   |- package.json
   |- server.js
```

- create new file, `package.json` to track project
  - *eg: dependencies, name, description, version...*

Server-side considerations - data storage

Redis and Node.js - package.json

```
{
  "name": "424-node-redis1",
  "version": "1.0.0",
  "description": "test app for node and redis",
  "main": "server.js",
  "dependencies": {
    "body-parser": "^1.14.1",
    "express": "^4.13.3",
    "redis": "^2.3.0"
  },
  "author": "ancientlives",
  "license": "ISC"
}
```

- we can write the `package.json` file ourselves or use the interactive option

```
npm init
```

- then add extra dependencies, eg: Redis, using

```
npm install redis --save
```

- use `package.json` to help with app management and abstraction...

Server-side considerations - data storage

Redis and Node.js - set notes value

- add Redis to our earlier test app
- import and use Redis in the `server.js` file

```
...
var express = require("express"),
    http = require("http"),
    bodyParser = require("body-parser"),
    jsonApp = express(),
    redis = require("redis");
...
```

- create client to connect to Redis from Node.js

```
//create client to connect to Redis
redisConnect = redis.createClient();
```

- then use Redis, for example, to store access total for notes on server

```
redisConnect.incr("notes");
```

- check Redis command line for change in `notes` value

```
get notes
```

Server-side considerations - data storage

Redis and Node.js - get notes value

- now set the counter value for our notes
  - *add our counter to the application to record access count for notes*
- use the `get` command with Redis to retrieve the incremented values for the `notes` key

```
redisConnect.get("notes", function(error, notesCounter) {
  //set counter to int of value in Redis or start at 0
  notesTotal.notes = parseInt(notesCounter,10) || 0;
});
```

- `get` accepts two parameters - `error` and return value
- Redis stores values and strings
  - *convert string to integer using `parseInt()`*
  - *two parameters - return value and `base-10` value of the specified number*
- value is now being stored in a global variable `notesTotal`
  - *declared in `server.js`*

```
var express = require("express"),
    http = require("http"),
    bodyParser = require("body-parser"),
    jsonApp = express(),
    redis = require("redis"),
    notesTotal = {};
```

Server-side considerations - data storage

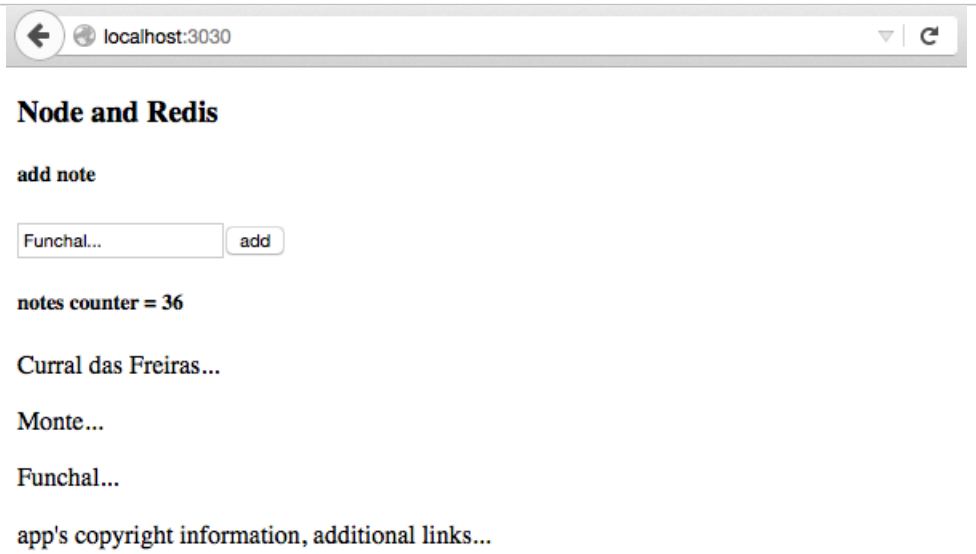
Redis and Node.js - get notes value

- store notes counter value in Redis
- create new route in `server.js`
  - *monitor the returned JSON for the counter*

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

- start using it with our application
  - *load by default, within event handler...*
- render to DOM
- store as a internal log record
- link to create note event handler...
- DEMO - 424-node-redis1

Image - Client-side and server-side computing



Node and Redis - 424-node-redis1

Server-side considerations - data storage

MongoDB - intro

- MongoDB is another example of a NoSQL based data store
  - *a database that enables us to store our data on disk*
- unlike MySQL, for example, it is not in a relational format
- MongoDB is best characterised as a **document-oriented** database
- conceptually may be considered as storing objects in collections
- stores its data using the BSON format
  - *consider similar to JSON*
  - *use JavaScript for working with MongoDB*

Server-side considerations - data storage

MongoDB - document oriented

- SQL database, data is stored in tables and rows
- MongoDB, by contrast, uses **collections** and **documents**
- comparison often made between a collection and a table
  - *NB: a document is quite different from a table*
  - *a document can contain a lot more data than a table*
- a noted concern with this document approach is duplication of data
- one of the trade-offs between NoSQL (MongoDB) and SQL
- SQL - goal of data structuring is to normalise as much as possible
  - *thereby avoiding duplicated information*
- NoSQL (MongoDB) - provision a data store, as easy as possible for the application to use

Server-side considerations - data storage

MongoDB - BSON

- BSON is the format used by MongoDB to store its data
- effectively, JSON stored as binary with a few notable differences
  - *eg: `ObjectId` values - data type used in MongoDB to uniquely identify documents*
  - *created automatically on each document in the database*
  - *often considered as analogous to a primary key in a SQL database*
- `ObjectId` is a large pseudo-random number
- for nearly all practical occurrences, assume number will be unique
- might cease to be unique if server can't keep pace with number generation...
- other interesting aspect of `ObjectId`
  - *they are partially based on a timestamp*
  - *helps us determine when they were created*

Server-side considerations - data storage

MongoDB - general hierarchy of data

- in general, MongoDB has a three tiered data hierarchy
  1. database * normally one database per app * possible to have multiple per server * same basic role as DB in SQL
  2. collection * a grouping of similar pieces of data * documents in a collection * name is usually a noun * resembles in concept a table in SQL * documents do not require the same schema
  3. document * a single item in the database * data structure of field and value pairs * similar to objects in JSON * eg: an individual user record

Server-side considerations - data storage

MongoDB - install and setup

- install on Linux
- install on Mac OS X
  - *again, we can use **Homebrew** to install MongoDB*

```
// update brew packages
brew update
// install MongoDB
brew install mongodb
```

- then follow the above OS X install instructions to set paths...
- install on Windows

Server-side considerations - data storage

MongoDB - a few shell commands

- issue following commands at command line to get started - OS X etc

```
// start MongoDB server - terminal window 1
mongod
// connect to MongoDB - terminal window 2
mongo
```

- switch to, create a new DB (if not available), and drop a current DB as follows

```
// list available databases
show dbs
// switch to specified db
use 424db1
// show current database
db
// drop current database
db.dropDatabase();
```

- DB is not created permanently until data is created and saved
  - *insert a record and save to current DB*
- only permanent DB is the local `test` DB, until new DBs created...

Server-side considerations - data storage

MongoDB - a few shell commands

- add an initial record to a new `424db1` database.

```
// select/create db
use 424db1
// insert data to collection in current db
db.notes.insert({
...   "travelNotes": [{
...   "created": "2015-10-12T00:00:00Z",
...   "note": "Curral das Freiras..."
...   }]
... })
```

- our new DB `424db1` will now be saved in Mongo
- we've created a new collection, `notes`

```
// show databases
show dbs
// show collections
show collections
```

Server-side considerations - data storage

MongoDB - test app

- now create a new test app for use with MongoDB
- create and setup app as before
  - *eg: same setup pattern as Redis test app*
- add **Mongoose** to our app
  - *use to connect to MongoDB*
  - *helps us create a schema for working with DB*
- update our `package.json` file
  - *add dependency for Mongoose*

```
// add mongoose to app and save dependency to package.json
npm install mongoose --save
```

- test server and app as usual from app's working directory

```
node server.js
```

Server-side considerations - data storage

MongoDB - Mongoose schema

- use **Mongoose** as a type of bridge between Node.js and MongoDB
- works as a client for MongoDB from Node.js applications
- serves as a useful data modeling tool
  - *represent our documents as objects in the application*
- a data model
  - *object representation of a document collection within data store*
  - *helps specify required fields for each collection's document*
  - *known as a schema in Mongoose, eg:* `NoteSchema`

```
var NoteSchema = mongoose.Schema({
  "created": Date,
  "note": String
});
```

- using schema, build a model
  - *by convention, use first letter uppercase for name of data model object*

```
var Note = mongoose.model("Note", NoteSchema);
```

- now start creating objects of this model type using JavaScript

```
var funchalNote = new Note({
"created": "2015-10-12T00:00:00Z",
"note": "Curral das Freiras..."
});
```

- then use the Mongoose object to interact with the MongoDB
  - *using functions such as* `save` *and* `find`

Server-side considerations - data storage

MongoDB - test app

- with our new DB setup, our schema created
  - *now start to add notes to our DB, `424db1`, in MongoDB*
- in our `server.js` file
  - *need to connect Mongoose to `424db1` in MongoDB*
  - *define our schema for our notes*
  - *then model a note*
  - *use model to create a note for saving to `424db1`*

```
...
//connect to 424db1 DB in MongoDB
mongoose.connect('mongodb://localhost/424db1');
//define Mongoose schema for notes
var NoteSchema = mongoose.Schema({
  "created": Date,
  "note": String
});
//model note
var Note = mongoose.model("Note", NoteSchema);
...
```

Server-side considerations - data storage

MongoDB - test app

- then update app's `post` route to save note to `424db1`

```
//json post route - update for MongoDB
jsonApp.post("/notes", function(req, res) {
  var newNote = new Note({
    "created":req.body.created,
    "note":req.body.note
  });
  newNote.save(function (error, result) {
    if (error !== null) {
      console.log(error);
      res.send("error reported");
    } else {
      Note.find({}, function (error, result) {
        res.json(result);
      })
    }
  });
});
```

Server-side considerations - data storage

MongoDB - test app

- update our app's `get` route for serving these notes

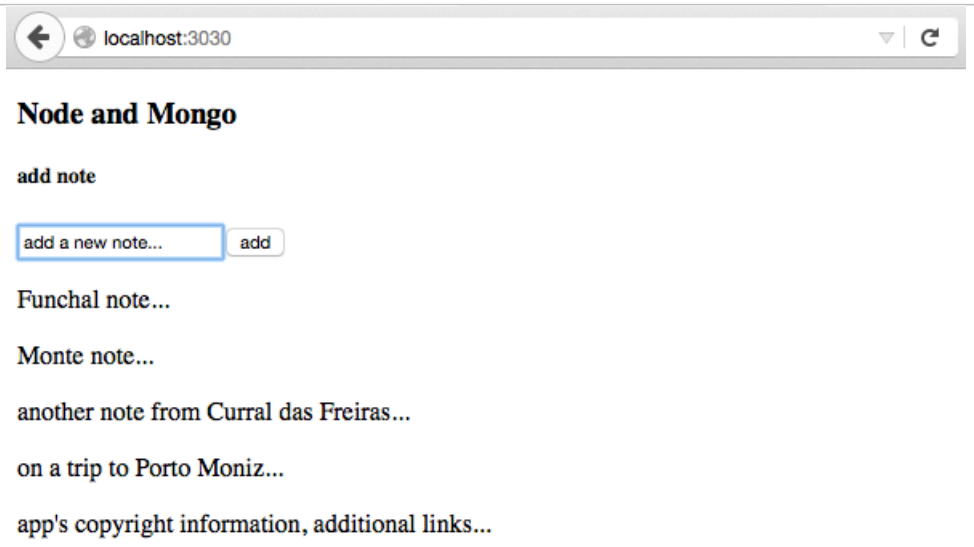
```
//json get route - update for mongo
jsonApp.get("/notes.json", function(req, res) {
  Note.find({}, function (error, notes) {
   //add some error checking...
   res.json(notes);
  });
});
```

- modify `buildNotes()` function in `json_app.js` to get return correctly

```
...
//get travelNotes
var $travelNotes = response;
...
```

- now able to enter, save, read notes for app
- notes data is stored in the `424db1` database in MongoDB
- notes are loaded from DB on page load
- notes are updated from DB for each new note addition
- DEMO - 424-node-mongo1

Image - Client-side and server-side computing



Node and MongoDB - 424-node-mongo1

Client-side - Data - Node, Express, MongoDB &c.

extra notes

- Heroku
  - *Heroku & Git*
  - *Heroku & MongoDB*
  - *Heroku & Postman*
- Node.js
  - *Node.js outline*
  - *Node.js updating*
- Node.js & Express
  - *Node.js and Express*
  - *Node.js & Express starter*
- Node.js, Express, and MongoDB
  - *Node.js and MongoDB*
- Node.js API
  - *Data stores & APIs - MongoDB and native driver*
  - *Node Todos API*
  - *Testing - Node Todos API*
- Node.js & Web Sockets
  - *Node.js & Socket.io*

Resources

- Chocolatey for Windows
  - *Chocolatey package manager for Windows*
- Homebrew for OS X
  - *Homebrew - the missing package manager for OS X*
- MongoDB
  - *MongoDB - For Giant Ideas*
  - *MongoDB - Getting Started (Node.js driver edition)*
  - *MongoDB - Getting Started (shell edition)*
- Mongoose
  - *MongooseJS Docs*
- Node.js
  - *Node.js home*
  - *Node.js - download*
  - *ExpressJS*
  - *ExpressJS body-parser*
- Redis
  - *redis.io*
  - *redis commands*
  - *redis - npm*
  - *try redis*
  - *Windows support*