Comp 424 - Client-side Web Design

Fall Semester 2016 - Week 12

Dr Nick Hayward

Contents

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

- Data storage
  - *Redis*
  - *MongoDB - Part 1*

Server-side considerations - Node.js

install Node.js

- a number of different ways to install **Node.js**, **npm**, and the lightweight, customisable web server **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`

```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

```javascript
/* 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

```javascript
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`
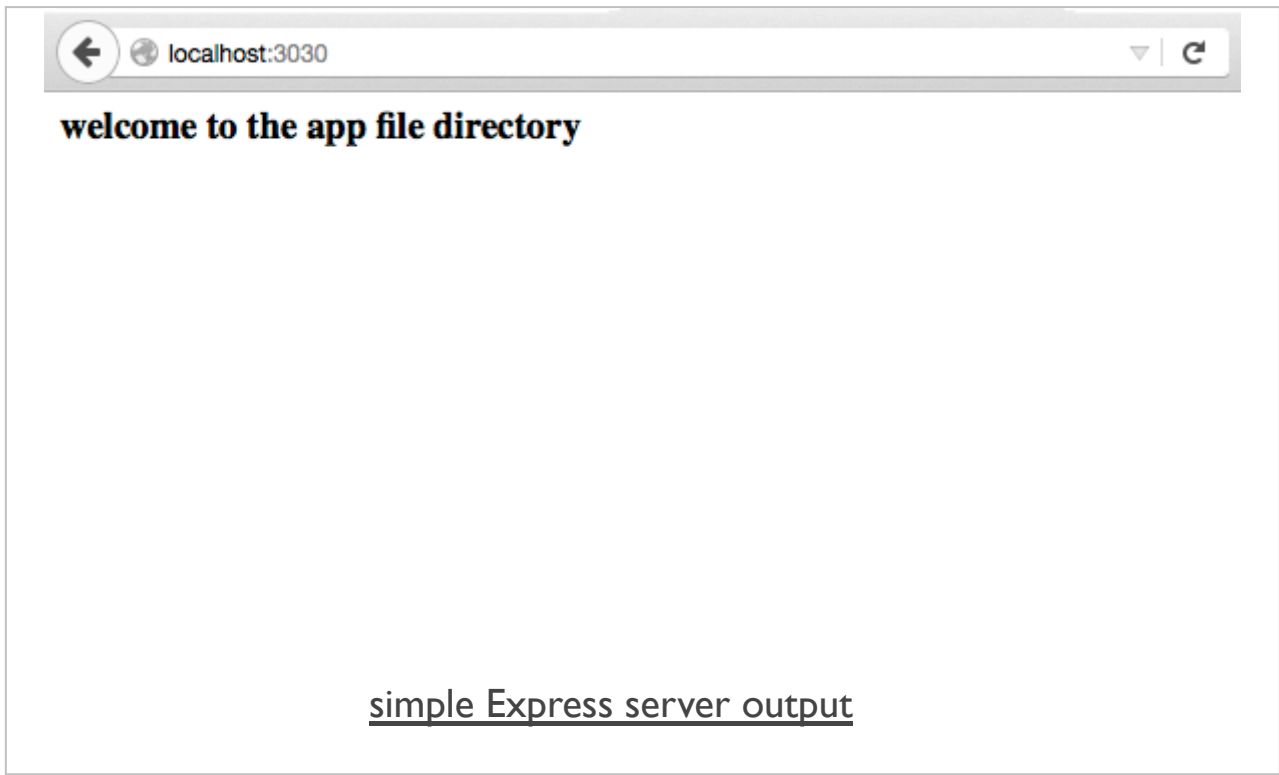
Server-side considerations - Node.js

- 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

```javascript
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
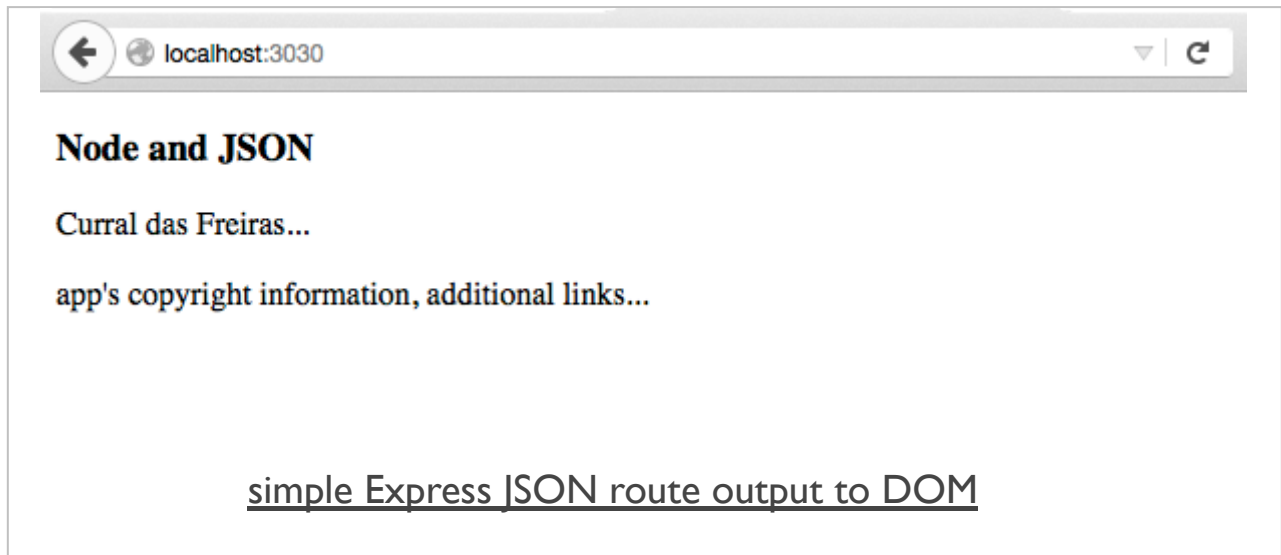
Server-side considerations - Node.js

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

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*

```javascript
$("#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*

```javascript
$(".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

```html
<!-- 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
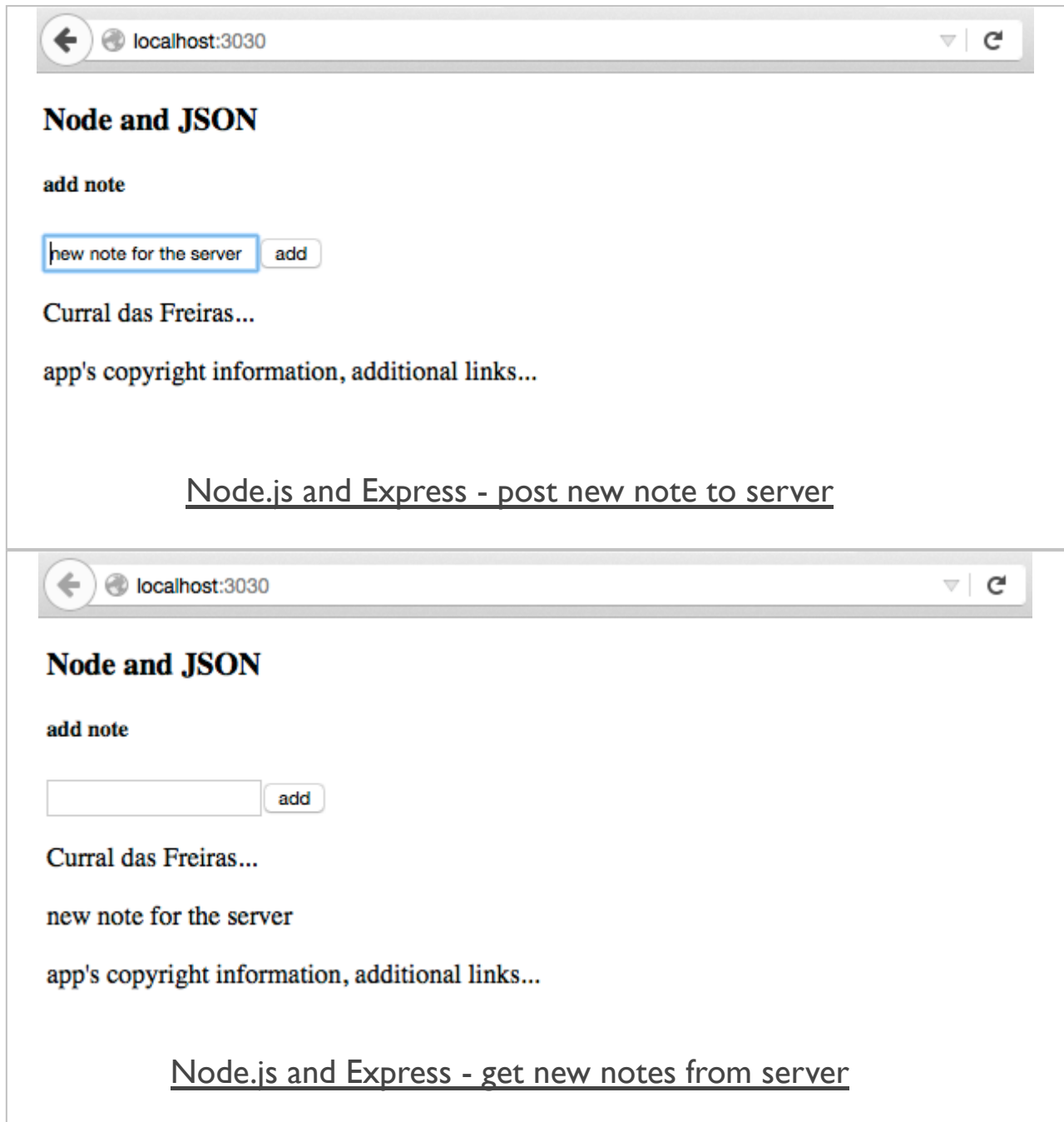
Server-side considerations - data storage

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/*

- 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

```
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>
```

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

```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 = {};
```
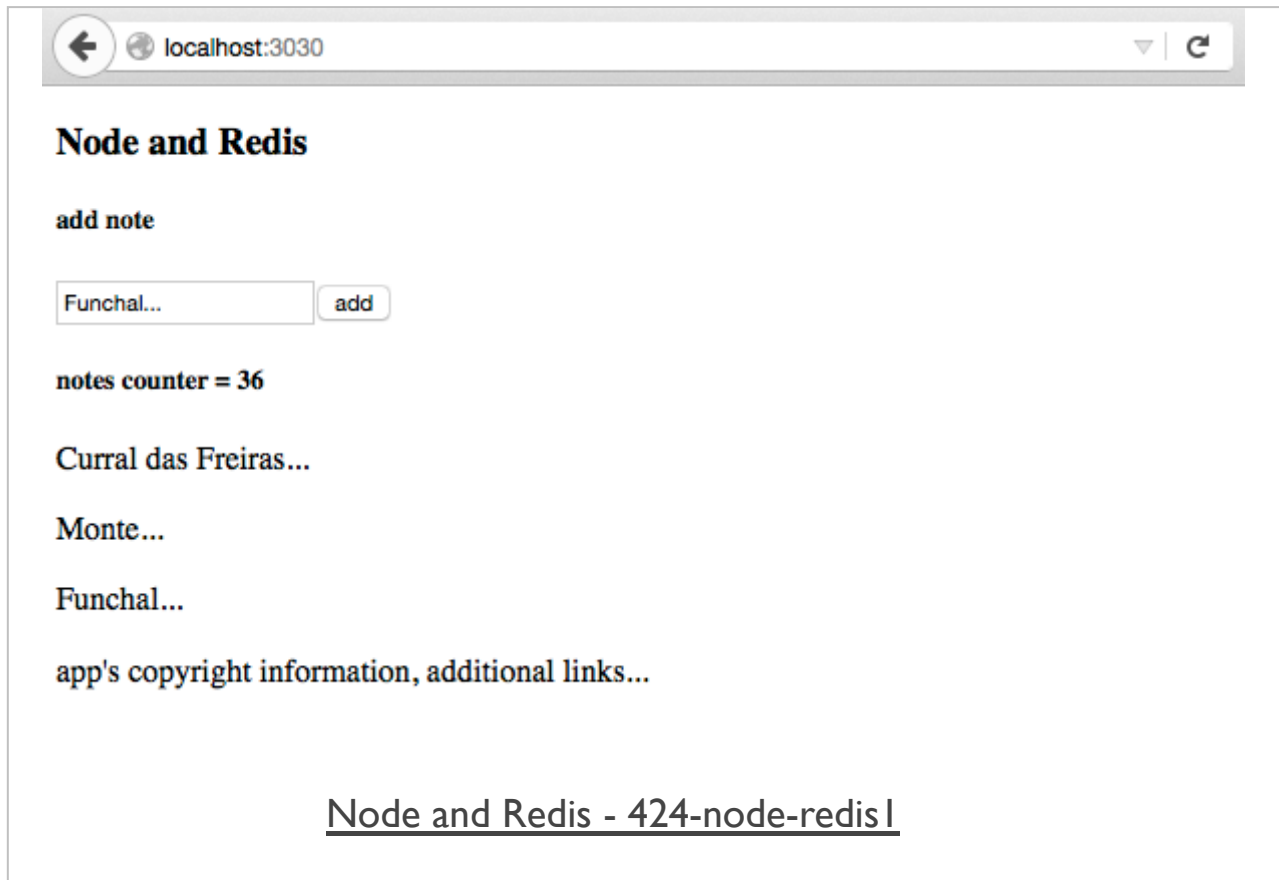
Server-side considerations - data storage

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

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`*

Source code - demos

Node.js

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

Redis

- 424-node-redis1

MongoDB

- 424-node-mongo1

References

- 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*