Comp 424 - Client-side Web Design

Fall Semester 2016 - Week 12 Notes

Dr Nick Hayward

Contents

- Complementary Server-side considerations
  - *Node.js*
- Data storage
  - *Redis*
  - *MongoDB*
- Data visualisation

Server-side considerations - Node.js

install Node.js

There are a number of different ways to install *Node.js*, *npm*, and the lightweight, customisable web server **Express**.

To run and test Node.js on a local Mac OS X or Windows machine, simply download and install a package from the following URL,

- Node.js - download

We can also install the Node module, **Express**. Express is a framework for web applications built upon Node.js, and is minimal and flexible.

We can use *npm* to install the *express* module. The –g option sets a flag for Express to global instead of a limited local install.

```
npm install -g express
```

This installs the *express* command line tool, which allows us to start building our basic web application. It is now also necessary to install the *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 easily 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.

To test whether **npm** is installed, simply issue the following command

```
npm
```

This should output some helpful information if **npm** is currently installed.

NB: on a Unix system, such as OS X or Linux, it is best to avoid installing **npm** modules with `sudo` privileges.

Server-side considerations - Node.js

NPM - installing modules

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

```
npm install express
```

This will install the 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.

As mentioned a few moments ago, we can also specify a global install for modules. For example, 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

To import, or effectively add, modules in our Node.js code we can use the following declaration,

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

When we run this application, it will look for the required module library and its source code.

Server-side considerations - Node.js

NPM - finding modules

The official online search tool for **npm** can be found at

- npmjs

Top packages include options such as

- browserify (helps us bundle require modules in the browser...)
- express
- grunt (a task runner to help with automation of various development processes...)
- bower (a package manager for web development...)
- karma (a JS test runner...)

and many more...

We can also search for node modules directly from the command line using the following command,

```
npm search express
```

This will return results for module names and descriptions.

Server-side considerations - Node.js

NPM - specifying dependencies

For Node.js applications, we can ease their installation by specifying any required dependencies in an associated `package.json` file. This allows us as developers to simply specify the modules to install for our application, which can then be run using the following command

```
npm install
```

This helps reduce the need to install each module individually, and helps other users install an application as quickly as possible. Also, our application's dependencies are stored in one place.

An example `package.json` file might be as follows,

```
{
"name": "app",
"version": "0.0.1",
"dependencies": {
```

```
  "express": "4.2.x",
  "underscore": "-1.2.1"
}
}
```

Server-side considerations - Node.js

initial Express usage

We can now use Express to start building our initial basic web application.

Express creates a basic shell for our web application with the following command,

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

This command makes a new directory, and populates it with the required basic web application directories and files.

We 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
```

Server-side considerations - Node.js

initial Express server - setup

So, 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

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

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

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

Server-side considerations - Node.js

initial Express server - server.js - part 1

We can now add some initial code to get out 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.");
});
```

We can then start and test this server as follows at the command line,

```
node server.js
```

initial Express server - server.js - part 2

Then we open our web browser, and use 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.

Server-side considerations - Node.js

initial Express server - server.js - part 3

So, 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"));
```

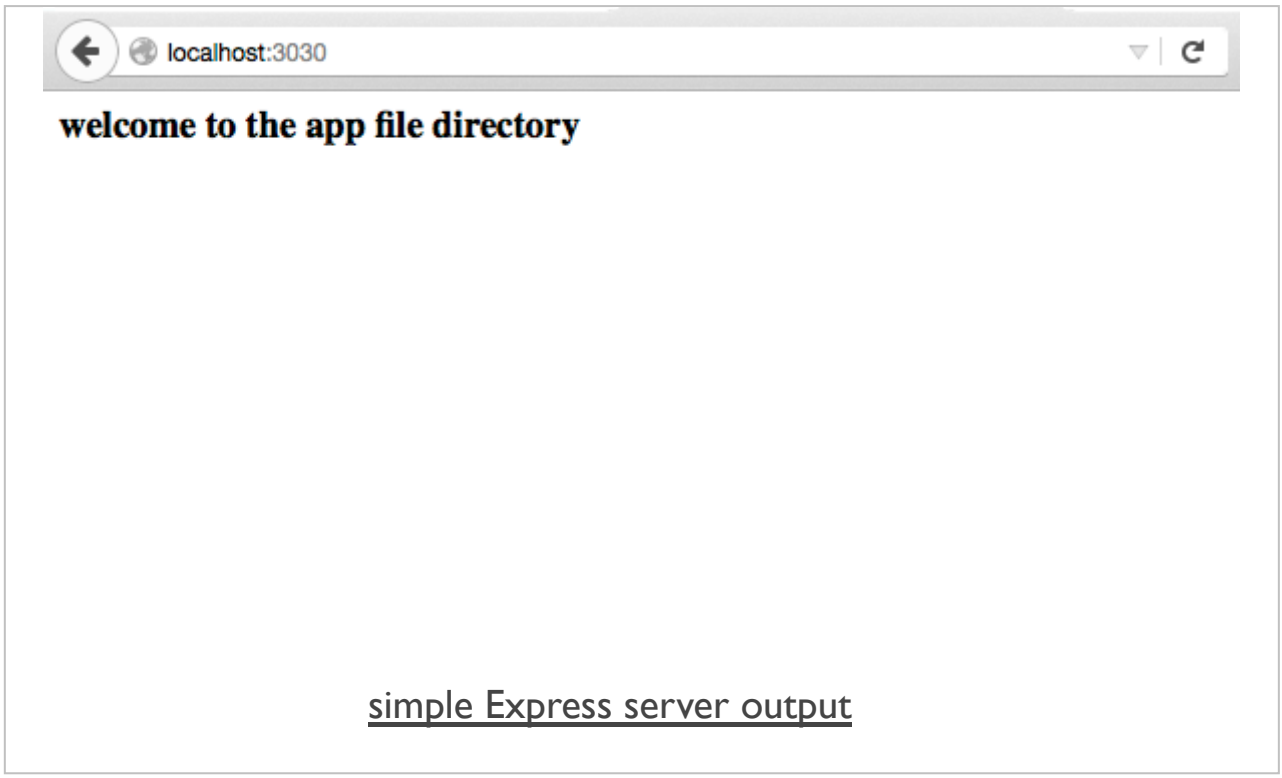
In essence, we are now defining Express as a static file server, thereby enabling us to publish our HTML, CSS, and JS files and code from our default directory, `/app`.

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.

Image - Client-side and server-side computing



simple Express server output

Server-side considerations - Node.js

working with data - JSON

So, let us 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 should have a directory structure as follows,

```
|- .
   |- 424-node-json1
      |- node_modules
      |- server.js
```

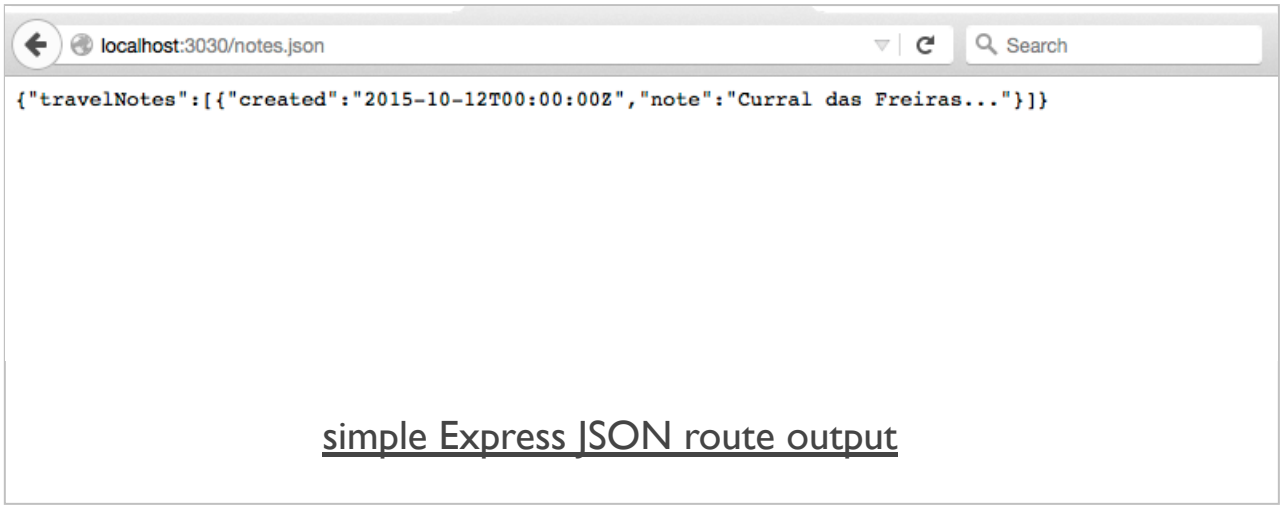
Within our `424-node-json1` app directory, we're going to update our earlier `server.js` file to allow us to serve a route for JSON. We can then use this to read our content for publication.

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.

Image - Client-side and server-side computing



simple Express JSON route output

Server-side considerations - Node.js

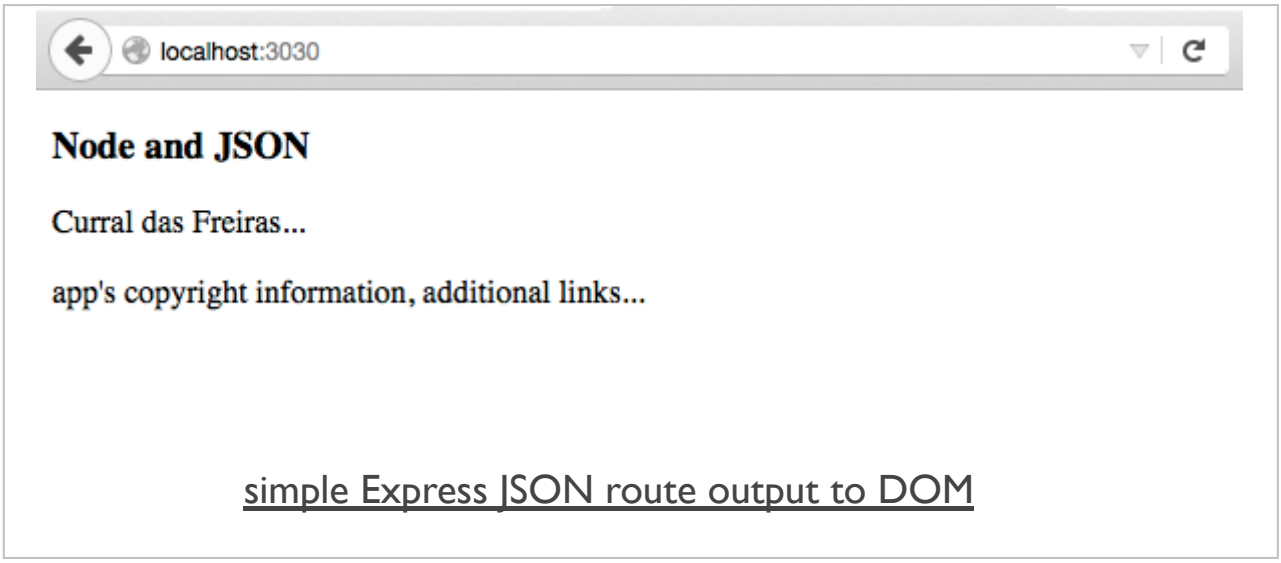
working with data - JSON

We now have our `get` routes setup for JSON. So, we can now add some client-side logic to read that route, and render to the browser. We'll simply use the same basic patterns we've seen before, 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 our normal `buildNote()` function.

Image - Client-side and server-side computing



simple Express JSON route output to DOM

Server-side considerations - Node.js

working with data - post data

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"
  });
});
```

working with data - post data

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.

Server-side considerations - Node.js

working with data - post 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. 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.

Server-side considerations - Node.js

working with data - post data

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.

Server-side considerations - Node.js

working with data - post data

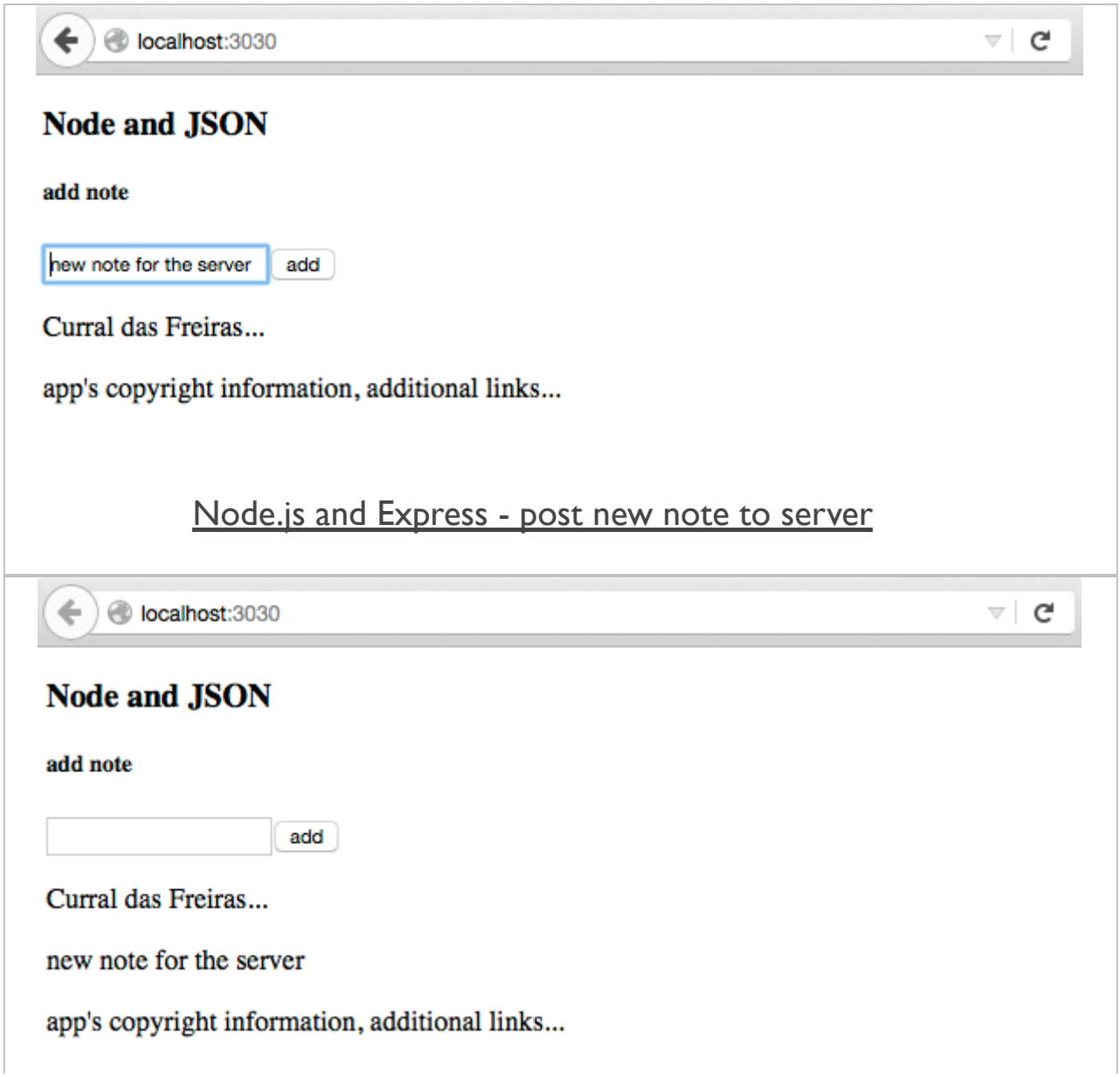
We can now update our test button's event handler to send a new note as a JSON object. This note will retrieve its new content from the input field, and then get the current time from the node server.

```
$(".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.

Image - Client-side and server-side computing

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.



Node.js and Express - get new notes from server

Server-side considerations - data storage

intro

So far, we've tested Node.js, created a server for hosting our files and routes with ExpressJS, read JSON from the server, and updated our JSON on the server-side.

This works well assuming we don't need to restart, repair, or update our server. If we do, we lose the updates posted to the server, and will return to the default data stored.

To help us solve this obvious issue, we'll need to consider persistent data storage that runs independently from our application. We'll now work our way through a couple of NoSQL options, in particular Redis and MongoDB, and we'll see how we can integrate both within our Node.js applications.

Server-side considerations - data storage

SQL or NoSQL

When we think of databases, we have often thought solely in terms of SQL, or structured query language. It is a language used to query data in a relational format. Such relational databases, for example MySQL or PostgreSQL, store their data in tables, which then provide a semblance of structure through rows and cells. We can then easily cross-reference, or relate, these table rows with rows in other tables.

So, we might use this relational structure to map authors to books, players to teams, and so on. One of the primary benefits of using a relational database is this inherent ability to store information, thereby dramatically reducing redundancy, and hopefully required storage space as well.

As storage restrictions have continued to ease in recent years, we now see a shift in thinking, and database design in general. We've started to

see the introduction of non-relational databases, often referred to simply as **NoSQL**. With this type of database, redundant data may be stored, but such designs often provide increased ease of use for developers. Some of these databases and stores have also been written with specific use-cases in mind, for example providing more efficient reading of data compared to writing. In effect, highly efficient data storage for specialised environments and scenarios.

We'll now work our way through Redis and MongoDB, two popular examples of NoSQL data storage.

Server-side considerations - data storage

Redis - intro

Redis provides an excellent example of NoSQL based data storage.

It has been designed for fast access to frequently requested data. As such, this improvement in performance is often due to a reduction in perceived reliability due to in-memory storage instead of writing to a disk.

Redis is able to flush data to disk, and will perform this task at given points during uptime, but for the majority of cases it can be considered an in-memory data store.

Redis stores this data in a **key-value** format, similar in nature to standard object properties in JavaScript. Due to this underlying structure, developers often perceive Redis as a natural extension of conventional data structures, including hashes, lists, sets, and so on. It has also become a good option for quick access to data,

or optionally caching temporary data for frequent access. Therefore, Redis has become particularly useful for improving response times within suitable applications.

Server-side considerations - data storage

Redis - installation

On OS X, the easiest option for installing Redis is to use the Homebrew package manager. Once that's installed, simply issue the following terminal command,

```
brew install redis
```

On Windows, there is a port maintained by the Microsoft Open Tech Group. Further details can be found on the projects page, MSOpenTech - Redis. Or, you can use the Chocolatey package manager to download and maintain Redis. With the integration of Bash, it will become even easier.

For Linux, simply 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

Once installed, we can start the Redis server with the following command,

```
redis-server
```

and then interact with our new server directly using the CLI tool,

```
redis-cli
```

We can then store some data in Redis using the `set` command. For example, we could create a new key for `notes`, and then set its value to `0`.

```
set notes 0
```

If there is no problem with the command, Redis will respond with `OK`. We can then retrieve a value using the `get` command,

```
get notes
```

which will now return our set value of 0.

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

We can also manipulate existing values for a given key. For example, we may wish to 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

We've now installed and tested Redis using the command line. So, we'll now add it to our test Node.js application.

We'll call this new test app, `424-node-redis1`.

Integrating Redis with Node.js is, thankfully, pretty straightforward. The first thing we need to do is create a `package.json` file, which we'll store at the root of our node app. Our app is now structured as follows,

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

We use the `package.json` file to help us manage and track dependencies within our app. It also helps us track other aspects of our

application, including name, description, version, and so on.

Server-side considerations - data storage

Redis and Node.js - package.json

So, our current `package.json` file is as follows

```
{
  "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 could create `package.json` by hand or use the following command to walkthrough various options

```
npm init
```

This creates a `package.json` file for us, and then we can add any required dependencies as follows,

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

This will update our `package.json` file.

Server-side considerations - data storage

Redis and Node.js - set notes value

Let's now add Redis to our earlier test app. The first thing we need to do is 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");
...
```

Then, we need to create a client to allow us to connect to Redis from Node.js.

```
redisConnect = redis.createClient();
```

We can then use Redis, for example, to store the number of times our notes are requested from the server.

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

We can add a call to our earlier `get` route in the server.js file. If we then check Redis, we'll see that the value of our `notes` has now, of course, been incremented.

Server-side considerations - data storage

Redis and Node.js - get notes value

So, we've now set the counter value for our notes. Let's add our counter to the application, so we know how many times the notes have been accessed.

We can use the `get` command with Redis to retrieve the incremented values for the `notes` key in Redis.

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

So, if we examine this code, we can see that `get` accepts two parameters. One for `error`, and the other for the value. However, due to the fact that Redis stores its values as strings, we need to process the return value to an integer for correct output. We could simply use the return string, but we might as well check how to convert the types. We can use the

`parseInt()` function, which has two parameters. The first is obviously the return value, and the second is known as a `radix`. This means that we require the `base-10` value of the specified number. If we try to parse a value that is not a number we will simply get `NaN`.

Our count value is now being stored in a global variable `notesTotal`, which we can declare at the top of our `server.js` file with the other variables.

```
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

Once we have started storing our notes counter value in Redis, we can get it for use in our application. To return it for use with the app, we can create a new route to monitor the JSON values. For example,

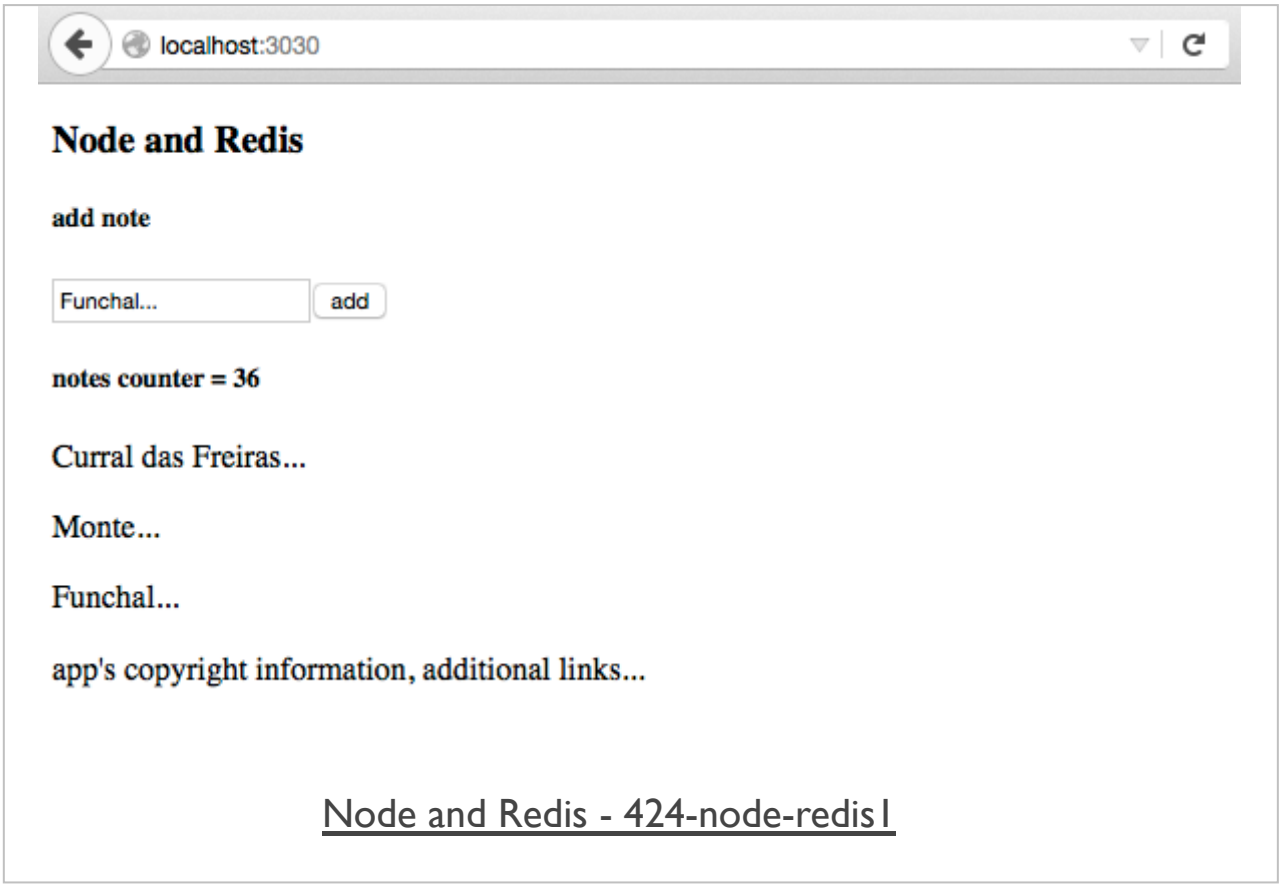
```
//json get route
jsonApp.get("/notesTotal.json", function(req, res) {
  res.json(notesTotal);
});
```

We can then subscribe to this route in our application, and manipulate the JSON as usual.

So, we could render the count total to the app's DOM, store it as an internal record, and so on. We might link it to the creation of a new note within the add button's event handler, for example.

- DEMO - 424-node-redis1

Image - Client-side and server-side computing



Node and Redis - 424-node-redis1

Server-side considerations - data storage

MongoDB - intro

Our next option for persistent data storage is MongoDB, which is another example of a NoSQL based data store. It is a database that enables us to store our data on disk, but unlike MySQL, for example, it is not in a relational format.

MongoDB is best characterised as a **document-oriented** database, which conceptually may be considered as storing objects in collections. It stores its data using the BSON format, which we'll look at in a moment, but ostensibly for our purpose we can consider this as comparable to JSON. The best part is that we can easily interact with MongoDB using JavaScript.

So, we'll have a look at some of the basics of MongoDB, then we'll install and set it up, and then work our way through an example with Node.js.

Server-side considerations - data storage

MongoDB - document oriented

In a traditional SQL database, data is stored in tables and rows. MongoDB, by contrast, uses *collections* and *documents*. A comparison is often made between a collection and a table. However, a document is quite different from a table.

A document can contain a lot more data than a table. For example, in a SQL database we may decide to store user details, including `user_id`, `email`, `name` and so on, in multiple tables. We then join these tables to create a user record. However, with documents we simply store the data, id, email etc, and this will now be the only item in the database to store this data.

A noted concern with this document approach is duplication of data for each user. This, however, is one of the trade-offs between NoSQL (MongoDB) and SQL.

In SQL, the goal of data structuring is to normalise as much as possible, thereby avoiding duplicated information. With MongoDB, there is a notably different goal. We are trying to provision a data store which is as easy as possible for the application to use, and by association the developer.

Server-side considerations - data storage

MongoDB - BSON

BSON is the format used by MongoDB to store its data. It is, effectively, JSON stored as binary with a few notable differences. One of these differences is the `ObjectId` values, which is a data type used in MongoDB to uniquely identify documents. It is created automatically on each document in the database, and can be considered as analogous to a primary key in a SQL database.

The `ObjectId` is a large pseudo-random number. For nearly all practical occurrences, we can assume that this number will be unique. The only situation where we might have a collision or clash in these numbers is if we were generating a large number and the database would not be able to keep pace. Therefore, we can assume they're always unique.

The other interesting aspect of `ObjectId` is that they are partially based on a timestamp.

This allows us to determine when they were created.

Server-side considerations - data storage

MongoDB - general hierarchy of data

In general, MongoDB has three tiers to its hierarchy of data. These include,

- Database - normally one database per app, but it is possible to have multiple per server. It functions the same basic role as a database in SQL.
- Collection - a grouping of similar pieces of data. So, MongoDB stores its documents in collections. Its name is usually a noun, and it resembles in concept a table in SQL. However, collections do not require their documents to share the same schema.
- Document - a single item in the database, for example an individual user record. A document is a data structure that uses field and value pairs, and is similar in nature to JSON objects.

Server-side considerations - data storage

MongoDB - install and setup

So, let's now install and setup MongoDB.

- install on Linux
- install on Mac OS X
  - *again, we can use **homebrew** to install Mong*

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

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

Server-side considerations - data storage

MongoDB - a few shell commands

Then test MongoDB from the command line.

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

- switch to or create a new DB, if not available, as follows

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

However, whilst you'll be switched to this new DB, it will not be created permanently until you create and insert a record in that database. In effect, we don't just create empty DBs, save and then populate later. We create the DB, populate, and then MongoDB will save it because it now has something to save. The only permanent DB is the default `test` DB.

Server-side considerations - data storage

MongoDB - a few shell commands

Let's now 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

We can now create a new test app for use with MongoDB. We'll set it up as before, as we did for testing with Redis, except we obviously don't need Redis this time.

To connect to MongoDB, and create a schema for working with our basic DB, we'll add Mongoose to the application.

So, we'll update our `package.json` for the app, and install Mongoose using npm. I'll go through Mongoose in a moment.

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

Then we can quickly test our app and server with the usual startup command in the app's working directory,

```
node server.js
```

Server-side considerations - data storage

MongoDB - Mongoose schema

To help us work with Node.js and MongoDB, we're going to use **Mongoose** as a type of bridge between these two technologies. In effect, this Node.js module serves two useful purposes. In a similar manner to **node-redis**, Mongoose works as a client from our Node.js application to MongoDB. It also serves as a useful data modeling tool, allowing us to represent our documents as objects in the application.

So, for our purposes, what is a data model. In essence, we can simply consider it as an object representation of a document collection within our given data store. It helps us specify required fields for each collection's document.

A data model in Mongoose is a schema, which we use to describe the underlying structure for all objects of a given type. So, for our notes, we can create a data model for a collection of

notes. We can start by specifying the schema for a note,

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

After creating our schema, we can programmatically build a model. As a convention, we tend to use an initial uppercase letter for the name of a data model object,

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

With our new model, we can start creating objects of this model type simply by using JavaScript's `new` operator. For example, if we wanted to add a new note to our TravelNotes,

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

We can 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, we can now start to add notes to our DB in Mongo, `424db1`.

In our `server.js` file, we need to connect Mongoose to our DB in MongoDB. Then, we define our schema for our notes. This allows us to then model a note, which we can use to create each note for saving to the database.

```
...
//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

We can then update our app's `post` route for saving these notes,

```
//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

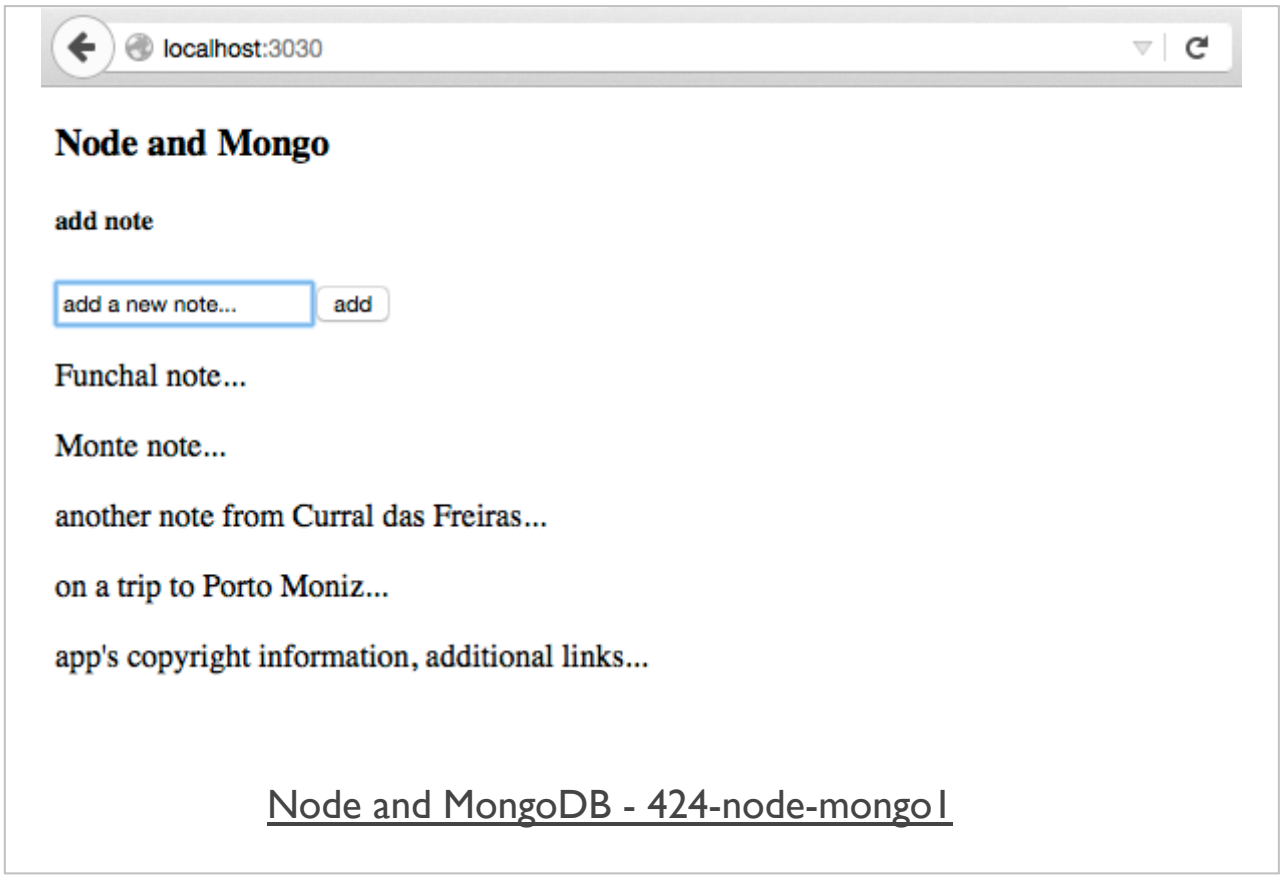
Then we need to update our app's `get` route for serving these notes,

```
//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);
  });
});
```

So, with our test app,

- 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

Data visualisation

intro - part 1

Data visualisation refers to the study of how to visually communicate and analyse data. It is still a relatively young discipline, relative to data in computer science and data science.

Data visualisation also covers many disparate aspects, including infographics, exploratory tools, dashboards, and so on.

However, there are already some notable definitions of the discipline. One of the better known and accepted examples is as follows,

> *"Data visualisation is the representation and presentation of data that exploits our visual perception in order to amplify cognition."*
>
> *(Kirk, A. "Data Visualisation: A successful design process." Packt Publishing. 2012.)*

Several variants of this general theme exist, however the underlying premise remains the same. Data visualisation is a visual representation of the underlying data. The

visualisation aims to impart a better understanding of this data and, by association, its relevant context.

Data visualisation

intro - part 2

There is, of course, an inherent flip-side to data visualisation. Without a correct understanding of its application, it can simply impart a false perception, and by association understanding, on the dataset. We run the risk of creating many examples of the standard *areal unit* problem, where a perception is based upon the creator's base standard and potential bias.

Our brains are inherently good at seeing what they want to see, and without due care and attention our visualisations can simply provide false summations of the data.

Data visualisation

types - part 1

There are many different ways to visualise datasets, and as many ways to customise a standard infographic.

However, there are some standard examples that allow us to consider the nature of visualisations. These can often be grouped into infographics, exploratory visualisations, and dashboards.

Therefore, it is perceived that data visualisation, in its many disparate forms, is simply a variation between infographics, exploratory tools, charts, and some data art.

For example,

- *1. Infographics - well suited for representing large datasets of contextual information. These will often be used within projects more inclined to exploratory data analysis, which tend to be more interactive for the user.*

There are some in the data science community who do not perceive infographics as proper data visualisation because they are designed to guide a user through a story with the main facts already highlighted. This is often seen in contrast to chart-based data visualisation, which present the story and the facts for the user to discover.

NB: such classifications, whilst generally useful, still only provide tangible reference points.

Data visualisation

types - part 2

- *2. Exploratory visualisations - this aspect of data visualisation is more interested in the provision of tools to explore and interpret datasets. The visualisations can be represented either static or interactive. Therefore, from a user perspective these charts can be viewed either carefully, or simply become interactive representations to help discover new and interesting concepts. This interactivity may include the option for the user to filter the dataset, and thereby interact with the visualisation via manipulation of the data, and modify the resultant information represented from the data. This kind of project is often perceived as more objective and data oriented than other forms.*

- *3. Dashboards - these are dense displays of charts. Commonly used to represent and quickly understand a given issue or domain.*

A good example is the display of server logs, website users, and business data within such dashboards.

Data visualisation

Dashboards - intro

Dashboards are dense displays of charts. They are used to allow us to represent and understand the key *metrics* of a given issue as quickly and effective as possible.

For example, consider the display of server logs, website users, and business data within such dashboards.

One definition of a dashboard is as follows,

> *"A dashboard is a visual display of the most important information needed to achieve one or more objective; consolidated and arranged on a single screen so the information can be monitored at a glance."*
>
> *Few, Stephen. Information Dashboard Design: The Effective Visual Communication of Data. O'Reilly Media. 2006.*

So, dashboards are visual displays of information. They can contain text elements, but are primarily a visual display of data rendered as meaningful information.

Data visualisation

Dashboards - intro

The information needs to be consumed quickly, and there is often simply no available time to read long annotations or repeatedly click controls. The information needs to be visible, and ready to be consumed. Dashboards are normally presented as a complementary environment and option to other tools and analytical/exploratory options.

One of the design issues presented by dashboards is how to effectively distribute the available space. Compact charts that permit quick data retrieval are normally preferred.

Therefore, dashboards should be designed with a purpose in mind. Generalised information within a dashboard is rarely useful. They should display the most important information that is necessary to achieve their defined purpose. In effect, a dashboard becomes a central view for

collated data represented as meaningful, useful information.

Data visualisation

Dashboards - good practices

To help promote our information, we need to design the dashboard to exploit all of the available screen space. We need to use this space to help users absorb as much information as possible.

Some visual elements are more easily perceived and absorbed by users than others. Likewise, some naturally convey and communicate information more effectively than others. Such attributes are known as **preattentive attributes of visual perception**.

These include, for example, colour, form, and position.

Dashboards - visual perception

pre-attentive attributes of visual perception

For example,

- **Colour** - there are many different colour models currently available, but the most useful relevant to dashboard design is the *HSL* model. This model describes colour in terms of three attributes,
  - *hue - this is what we normally call colour*
  - *saturation - intensity of colour*
  - *lightness (in effect, brightness) - Our perception of colour will, more often than not, depend upon its context. For example, a light colour will draw attention if it is surrounded by a dark colour. So, we can use colour within a dashboard design to attract a user's attention to areas of the screen that require our user's attention and focus.*
- **Form** - correct use of length, width, and general size can convey quantitative dimensions, each with varying degrees of precision. We can also use the Laws of Pragnanz to manipulate groups of similar shapes and designs, thereby easily grouping like data and information for the user.
- **Position** - the relative positioning of elements can help to communicate the information within a dashboard. Again, the laws of pragnanz teach us that position can often infer a perception of relationship and similarity. Higher items are often perceived as being better, and items on the left of the screen will traditionally be seen first by a western user.

Naturally, we can use these design guidelines and suggestions to inform our decisions relative to the layout of a dashboard. To effectively use all of the available screen space, we need to ensure that we select only the information that counts, and design compact charts and general graphics. Such charts and graphics need to be clear, concise, and direct, thereby reducing the need for explicit decoding to a bare minimum.

Zoning and organisation of information should follow a logical pattern to guide and inform the user.

- pre-attentive attributes of visual perception

1. *Colour*
   - *many different colour models currently available*
   - *most useful relevant to dashboard design is the HSL model*
   - *this model describes colour in terms of three attributes*
     - *hue*
     - *saturation*
     - *lightness*
   - *perception of colour often depends upon context*
2. *Form*
   - *correct use of length, width, and general size can convey quantitative dimensions*
   - *each with varying degrees of precision*

- *use the Laws of Prägnanz to manipulate groups of similar shapes and designs*
- *thereby easily grouping like data and information for the user*

3. *Position*

- *relative positioning of elements helps communicate dashboard information*
- *laws of Prägnanz teach us*
- *position can often infer a perception of relationship and similarity*
- *higher items are often perceived as being better*
- *items on the left of the screen traditionally seen first by a western user*

Data visualisation

Building a dashboard

To begin, we need to clearly determine the questions that need to be answered with the information collated and presented within the dashboard. We also need to ensure that any problems can be detected on time, and be certain why we actually need a dashboard for the current dataset.

By determining these questions and problems, we can then begin to collect the requisite data to help us answer such questions. Naturally, the data can be sourced from multiple, disparate datasets.

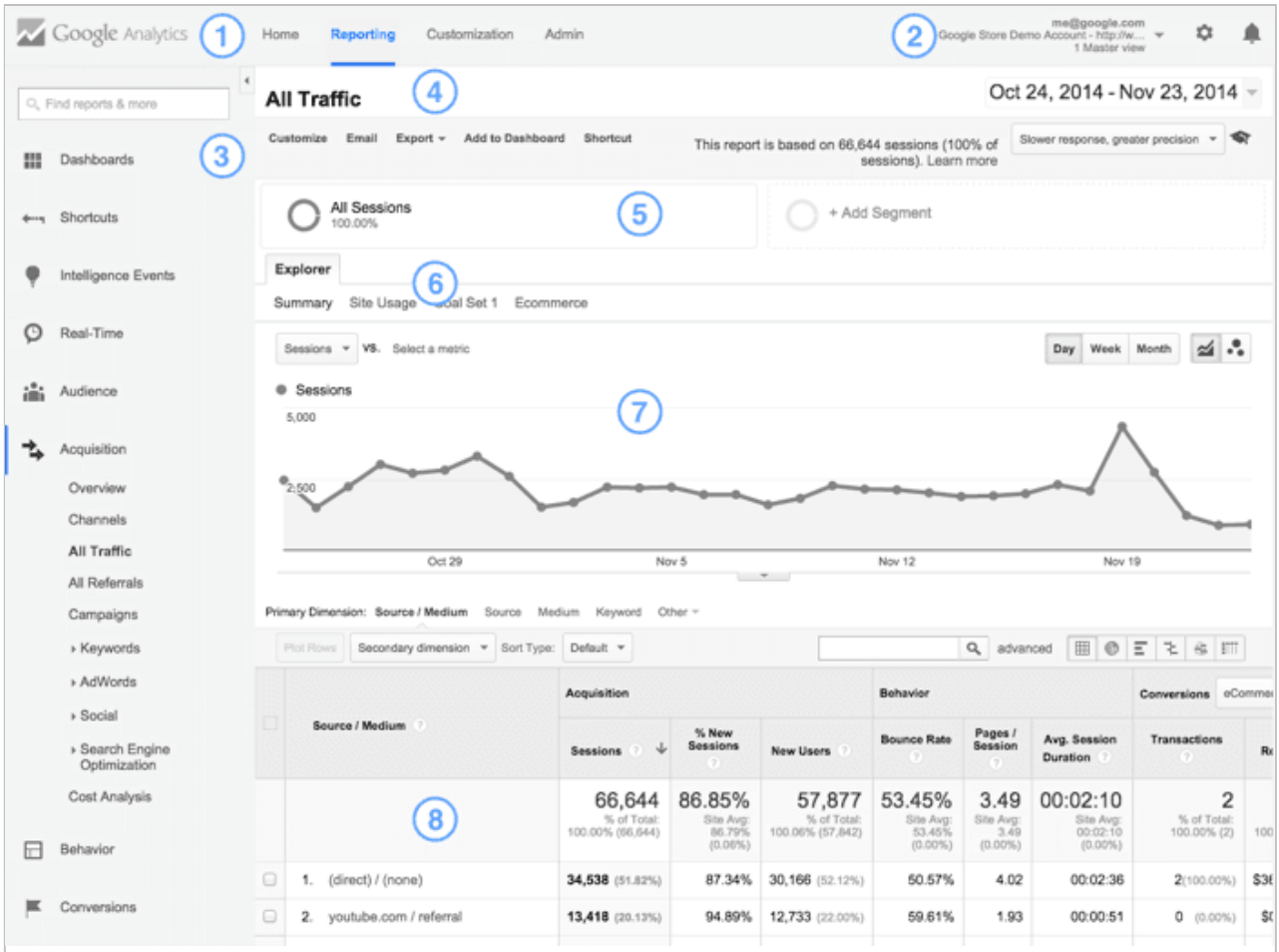
From the data we derive the information, which then helps us define the groupings and zones within our dashboard. Effectively, we are conveying a story with the information derived from the datasets.

The chosen visualisations help us to tell this story more effectively, and present it in a manner appealing to our users. We need to consider information visualisations that will be familiar to our users, and reduce the potential for cognitive load.

We also need to carefully consider the organisation of our data and information.

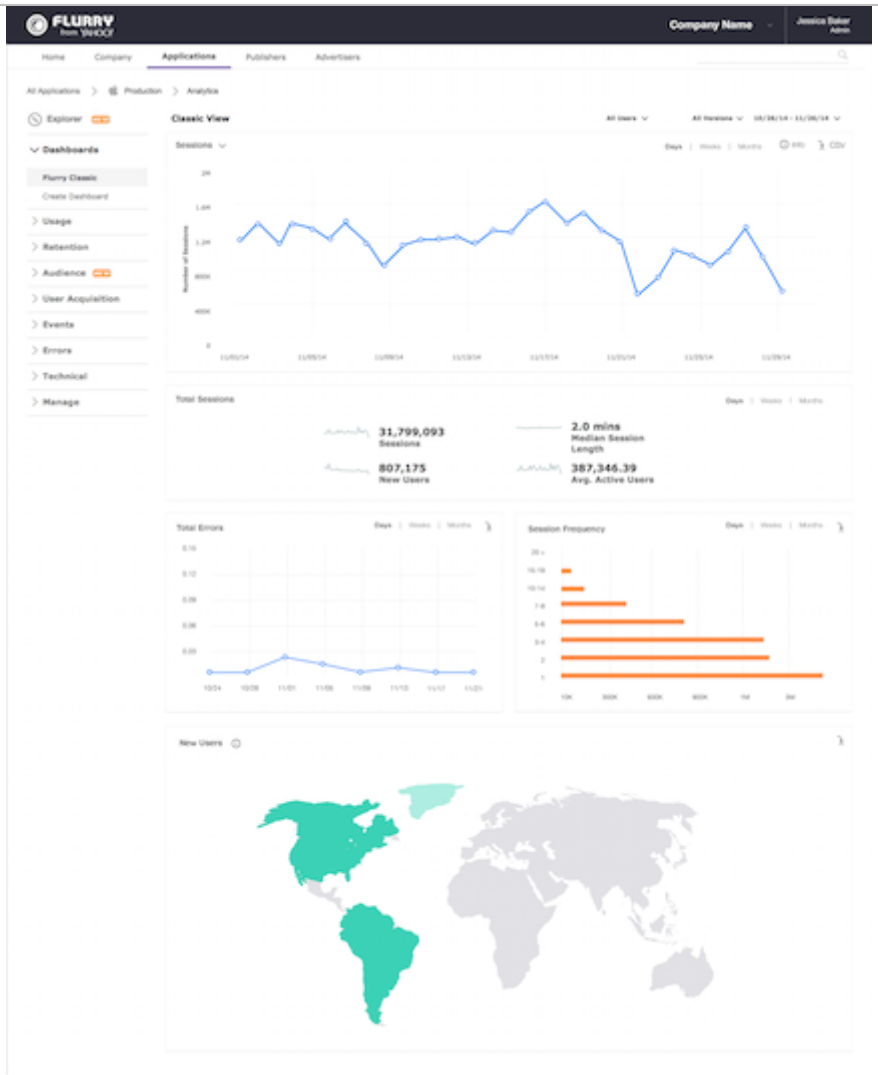
To present the information on our dashboard in a meaningful manner, we need to organise the data into logical units of information. Each section of the dashboard should be organised to help highlight and detect any underlying or prevailing issues, and then present them to the user.

Image - Google Analytics



Dashboard - Google Analytics

Image - Yahoo Flurry



Dashboard - Yahoo Flurry

Image - Mint

Dashboard - Mint

Data visualisation - D3

Intro - part 1

D3 is a custom JavaScript library designed for the manipulation of data centric documents. D3 uses a custom library with HTML, CSS, and SVG to create graphically rich, informative documents for the presentation of data.

Essentially, it uses a data-driven approach to manipulate the document object model (DOM). Hence 'data driven documents', or D3.

Setup and configuration of D3 is pretty straightforward. The most involved initial aspect is often the configuration of a web server, assuming you do not have an existing setup in place.

D3.js works with standard HTML files and, therefore, requires a web server simply capable of parsing and rendering HTML and associated files.

One of the requirements for parsing D3 within our HTML is a UTF-8 encoding reference within a *meta* element in the *head* section of our file.

As with other Javascript libraries, such as jQuery, we simply reference the D3 library with a standard script element in our HTML file.

Data visualisation - D3

intro - part 2

D3 has been designed and built to follow a function style of JavaScript. It differs in its implementation and focus for JavaScript with a conscious focus and priority upon the application of data to documents.

Functional JavaScript

D3 Wiki describes the underlying functional concepts of D3 as follows,

> *D3's functional style allows code reuse through a diverse collection of components and plugins.*
>
> *D3 Wiki*

In JavaScript, functions are objects. As with other objects, a function is a collection of a *name and value pair*. The real difference between a function object and a regular object is that a function can be invoked, and associated, with two hidden properties. These include a function *context* and function *code*.

Variable resolution in D3 relies on variable searching being performed locally first. So, relative to a variable reference, if a variable declaration is not found, the search will continue to the parent object, and continue recursively to the next static parent until it reaches global variable definition. If this is not found, then a reference error will be generated for this variable.

Therefore, it is important to keep this static scoping rule in mind when dealing with D3.

Data visualisation - D3

Data Intro - part 1

Data is structured information with an inherent perceived potential for meaning.

When we consider data relative to D3, we need to know how data can be represented both in programming constructs and its associated visual metaphor.

So, as a brief segue, what is the basic difference between data and information,

> *Data are raw facts. The word raw indicates that the facts have not yet been processed >>> to reveal their meaning...Information is the result of processing raw data to reveal >>> its meaning.*
>
> *Rob, Morris, and Coronel. 2009*

However, this is the general concept of data and information. If we consider them relative to visualisation, we necessarily impart a richer interpretation. Information, in this context, is no longer the simple result of processed raw

data or facts. Instead, it becomes a visual metaphor of the facts.

The same data set can generate any number of visualisations. These may lay equal claim in terms of its validity. In effect, visualisation becomes more about communicating the creator's insight into data than anything else.

Data visualisation - D3

Data Intro - part 2

Relative to development for visualisation, data will often be stored simply in a text or binary format. This is not simply textual data, though, but can also include data representing images, audio, video, streams, archives, models, and so on.

However, for the purposes of D3 this concept may often simply be restricted to textual data, or text-based data. Effectively, any data that can be represented as a series of numbers and strings of alpha numeric characters. For example, textual data stored as a comma-separated value file, or .csv, or JSON document, .json, or a plain text file, .txt, can be used with D3.

This data can then be *bound* to elements within the DOM of a page. This is the inherent pattern for D3.

Data visualisation - D3

Data Intro - Enter-Update-Exit Pattern

In D3, the connection between data and its visual representation is usually referred to as the **enter-update-exit** pattern.

This concept is starkly different from the standard imperative programming style.

So, this pattern includes an

- enter mode
- update mode
- exit mode

Data visualisation - D3

Data Intro - Enter-Update-Exit Pattern

Enter mode

The `enter()` function returns all of the specified data that has not yet been represented in the visual domain. This standard modifier function can then be chained to a selection method to create new visual elements representing the given data elements.

ie: we can keep updating the array, and outputting the new data bound to elements.

Update mode

`selection.data(data)` function, on a given selection, establishes the connection between the data domain and the visual domain. The returned result of this intersection of data and visual will be a *data-bound* selection. We can now invoke a modifier function on this newly created selection to update all existing

elements. This is what we mean by an *update* mode.

ie: as soon as this connection is established we can update the selections in many different ways.

Exit mode

If we invoke `selection.data(data).exit` function on a data-bound selection, the function will compute a new selection which contains all visual elements that are no longer associated with any valid data element.

For example, if we created a bar chart with 25 data points, and then updated it to 20, we would now have 5 left over. With this *exit mode*, we can now remove the excess elements for the 5 spare data points.

Data Intro - binding data - part 1

So, if we consider standard patterns for working with data.

We can, of course, iterate through an array, and then bind the data to an element, but the most common option in D3 is to use the **enter-update-exit** pattern.

We can follow the same basic pattern for binding object literals as data. We use the same basic methodology as an array, except our array is now filled with objects. So, to access our data we call the required attribute of the supplied data. For example,

```
var data = [
    {height: 10, width: 20},
    {height: 15, width: 25}
];

function (d) {
    return (d.width) + "px";
}
```

We can obviously access the *height* attribute per object in the same manner.

Similarly, we can also bind functions as data. D3 allows functions to be treated as data as well. This allows us to create data dynamically, or modify existing data before being bound to elements, and so on. There is a lot we can do if we consider data as a function, and vice-versa.

Data visualisation - D3

Data Intro - binding data - part 2

So, D3 enables us to bind data to elements in the DOM. Binding is, effectively, associating data to specific elements. This allows us to reference those values later, so that we can apply required mapping rules.

So, we use D3's `selection.data()` method to bind our data to DOM elements. However, we obviously need some data to bind, and a selection of DOM elements.

D3 is particularly flexible with data, and will happily accept various types. It will accept various types of arrays of numbers, strings, or object, and these can also include multidimensional arrays. It can also handle JSON, including GeoJSON, and will even accept CSV files.

D3 also has a built-in function to handle loading JSON data, or we can simple continue to use

deferred objects with jQuery, for example.

```
d3.json("testdata.json", function(json) {
    console.log(json); //do something with the json...
});
```

Data visualisation - D3

Data Intro - working with arrays - options

D3 provides a particularly rich set of functions and utilities for working with arrays, which makes the task of manipulating array data considerably easier.

A few of the options are as follows,

- min and max = return the min and max values in the passed array

```
d3.select("#output").text(d3.min(ourArray));
d3.select("#output").text(d3.max(ourArray));
```

- extent = retrieves both the smallest and largest values in the the passed array

```
d3.select("#output").text(d3.extent(ourArray));
```

- sum

```
d3.select("#output").text(d3.sum(ourArray));
```

- median

```
d3.select("#output").text(d3.median(ourArray));
```

- mean

```
d3.select("#output").text(d3.mean(ourArray));
```

- asc and desc

```
d3.select("#output").text(ourArray.sort(d3.ascending));
d3.select("#output").text(ourArray.sort(d3.descending));
```

Data visualisation - D3

Data Intro - working with arrays - nest

D3's nest function can be used to build an algorithm to transform a flat array data structure into a hierarchical nested structure. This function can be configured using the key function chained to *nest*.

In effect, nesting allows elements in an array to be grouped into a hierarchical tree structure. It's similar in concept to the *group by* option in SQL. The main difference is that *nest* in D3 allows multiple levels of grouping and, of course, the result is a tree rather than a flat table.

The levels in the tree are defined by the *key* function, and the leaf nodes of the tree can be sorted by value. The internal nodes of the tree can be sorted by key.

So, sorting data into time contexts is a good use of this function, For example, year, month, week, and so on...

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*