Comp 424 - Client-side Web Design

Fall Semester 2016 - Week 13

Dr Nick Hayward

Contents

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

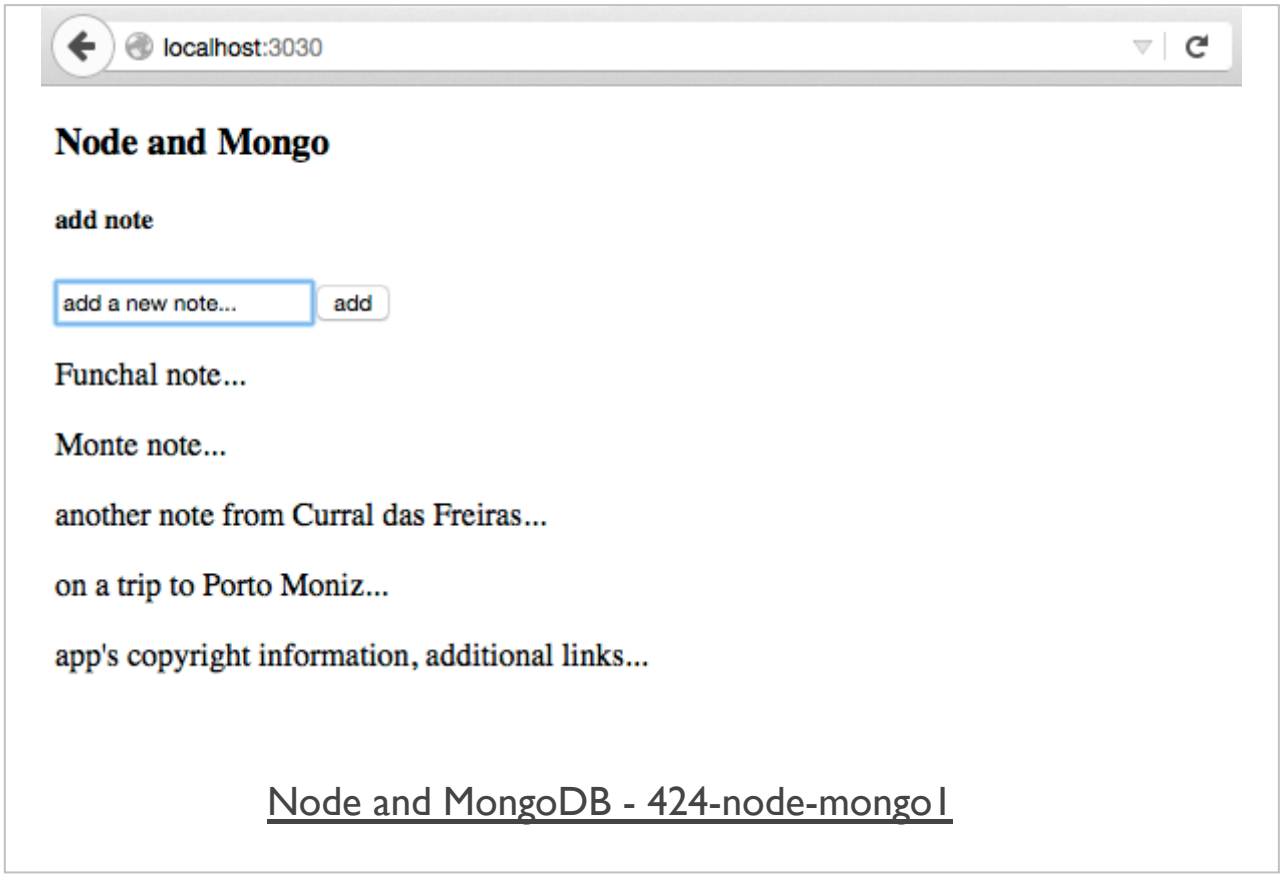
Then we need to update our app's `get` route for serving these notes,

```
//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.

Data visualisation

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.

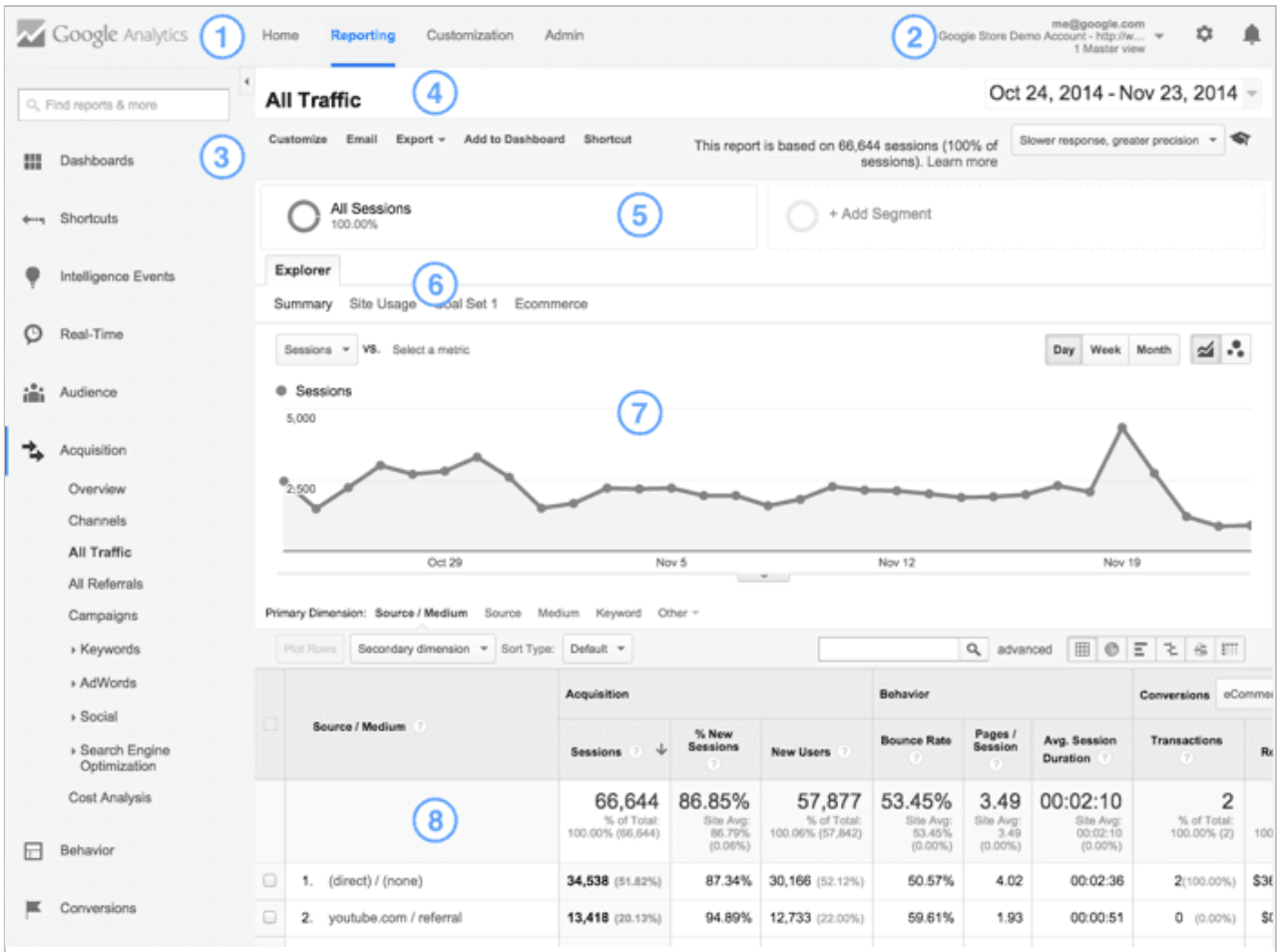
From the data we derive the information, which then helps us define the groupings and zones within our dashboard. Effectively, we are conveying a story with the information derived from the datasets.

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.

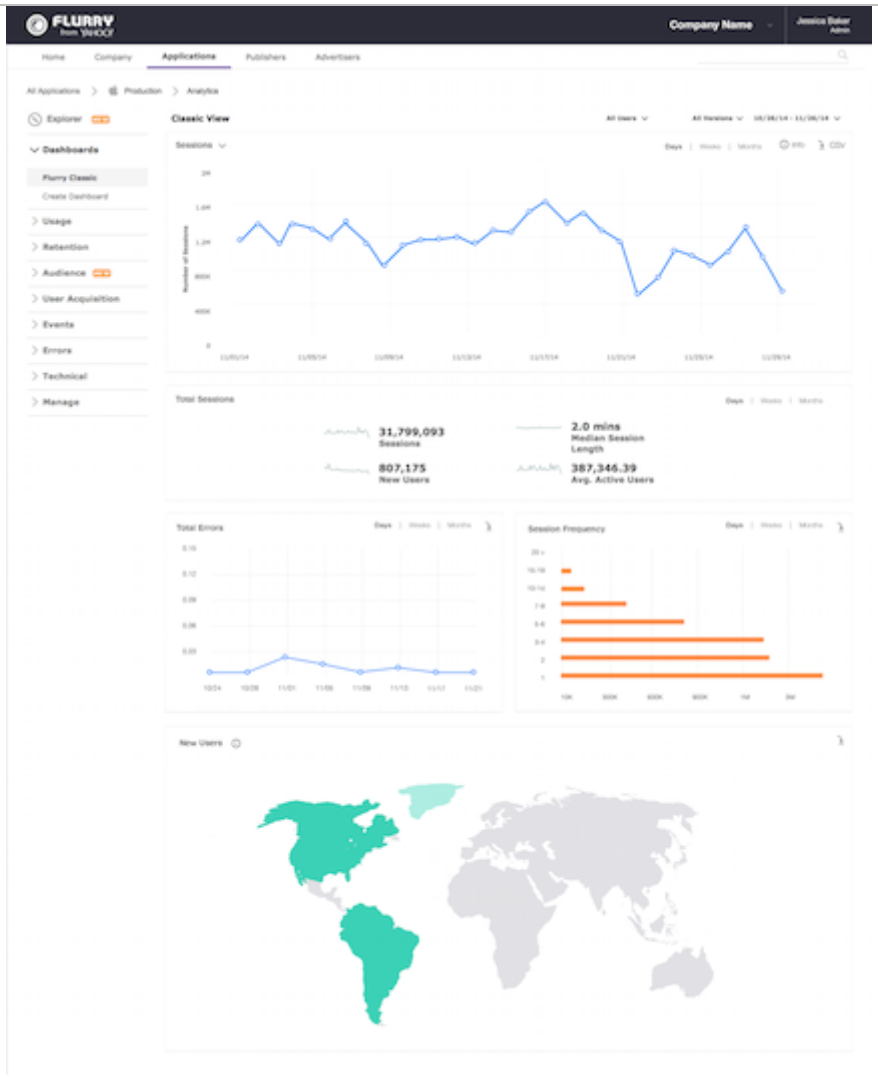
To present the information on our dashboard in a meaningful manner, we need to organise the data into logical units of information. Each section of the dashboard should be organised to help highlight and detect any underlying or prevailing issues, and then present them to the user.

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 visualisation - D3

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...

Data visualisation - D3

Selections - intro

Selection is one of the key tasks required within D3 to manipulate and visualise our data. It simply allows us to target certain visual elements on a given page.

Selector support is now standardised upon the W3C specification for the selector API. It is supported by all of the modern web browsers. However, its limitations are particularly noticeable for work with visualising data on the client side.

In effect, this selector API only provides support for selector and not selection. This means that we are able to select an element in the document, but to manipulate or modify its data we need to implement a standard loop etc. To select all *p* elements in a document, we need to loop through the document, select all of the *p*

elements, and then manipulate each within a standard loop construct.

D3 introduced its own selection API in an attempt to address these issues and perceived shortcomings. This API is built on the level-3 selector support from the original W3C API, which is commonly known as CSS3 selector support.

For example, the ability to select elements by ID or class, its attributes, set element IDs and class, and so on...

Data visualisation - D3

Selections - single element

To select a single element within our page, we can use the following code,

```
d3.select("p");
```

This will now select the first *p* element on the page, and then allow us to modify as necessary. For example, we could simply add some text to this element.

```
d3.select("p")
.text("Hello World");
```

This D3 command simply performs a selection of a single specified element. This could be a generic element, such as <p>, or a specific element defined by targeting its ID.

We can use additional modifier functions, such as `attr`, to perform a given modification on the selected element. For example,

```
//set an attribute for the selected element
d3.select("p").attr("foo");
//get the attribute for the selected element
d3.select("p").attr("foo");
```

We can also add or remove classes on the selected element,

```
//test selected element for specified class
d3.select("p").classed("foo")
//add a class to the selected element
d3.select("p").classed("goo", true);
//remove the specified class from the selected element
d3.select("p").classed("goo", function(){ return false; });
```

There are many others as well, including setting specific styles for the selected element, its text or HTML, and so on.

Data visualisation - D3

Selections - multiple elements

We can also select all of the specified elements using D3. For example,

```
d3.selectAll("p")
.attr("class", "para");
```

The way we use and implement multiple element selection is, in general principle, the same as single selection. We can also use the same modifier functions, which allows us to modify each element's attributes, style, class, and so on.

Data visualisation - D3

Selections - iterating through a selection

D3 provides us with a selection iteration API, which allows us to iterate through each selection. We can then modify each selection relative to its position. This is very similar to the way we normally loop through data, for example a simple array.

```
d3.selectAll("p")
.attr("class", "para")
.each(function (d, i) {
    d3.select(this).append("h1").text(i);
});
```

In D3, selections are essentially like arrays with some enhancements. So, we can still iterate through our selections, including those not yet created, by using the built-in iterator for the selection API methods.

```
d3.selectAll('p')
.attr("class", "para2")
.text(function(d, i) {
    return i;
});
```

We can use this pattern with many chained methods.

Data visualisation - D3

Selections - performing sub-selection

As we work on visualisations, it will often be necessary to perform specific scope requests. For example, selecting all <p> elements for a given element.

Therefore, we are selecting all *p* elements at the local scope of the specified *div* elements. For example,

```
//direct css selector (selector level-3 combinators)
d3.select("div > p")
    .attr("class", "para");

//d3 style scope selection
d3.select("div")
    .selectAll("p")
    .attr("class", "para");
```

The above two examples produce the same effect and output, but use very different selection techniques. The first example uses the CSS3, level-3, selectors to get direct access to the `div` and then the sub `p` elements. This type of syntax, `div > p`, is known as combinators.

Level-3 selector support offers a number of different kinds of structural combinators.

Data visualisation - D3

Selections - combinators

A brief update on usage and options for combinators.

1. descendant combinator

This combinator uses the pattern of *selector selector*. Effectively, it is describing a loose parent-child relationship for the selected elements within our document. It is defined as *loose* because this combinator does not care whether the parent-child relationship is child, grandchild etc relative to the selected parent. For example,

```
d3.select("div p");
```

This will select the <p> element as a child of the parent element. It does not care if the <p> element is a child of another element, as along as the <p> element is a descendant of the parent element.

2. child combinator This combinator uses the same style of syntax, *selector* > *selector*, but it is able to describe a more restrictive *parent-child* relationship between two elements. For example,

```
d3.select("div > p");
```

It will only find the <p> element if it is a direct child to the element.

Data visualisation - D3

Selections - D3 sub-selection

Another option for sub-selection is D3's built-in selection of child elements. D3 provides a simple option to select an element, and then chain another selection to get the child element. This type of chained selection defines a scoped selection within D3.

In effect, we are selecting a *p* element nested within our selected *div* element.

The advantage of this type of selection is that each selection is, effectively, independent. Therefore, we can perform actions etc on the initial *div* selection before selecting the *p* element.

The D3 API is, effectively, built around the inherent concept of function chaining. Therefore, it can almost be considered a *Domain Specific Language* for building HTML/SVG elements dynamically.

One of the benefits of chaining is that it allows us to easily produce concise and readable code that produces dynamic visual content for our document.

For example,

```
var body = d3.select("body");


body.append("div")
    .attr("id", "div1")
    .append("p")
    .attr("class", "para")
    .append("h5")
    .text("this is a paragraph heading...");
```

This will simply select the main body of our document, append a *div* element, set its ID to *div1*, then append a *p* element with a class of *para*, and then add some text to the appended *h5* heading.

Data visualisation - D3

Data Intro - page elements

With D3, the generation of new DOM elements will normally fit either circles, rectangles, or some other visual form that represents the data. However, we can also create generic structural elements in HTML, as we've just seen with a *div*, *p*, and so on.

For example, we can append a standard *p* element to our new page as follows

```
d3.select("body").append("p").text("sample text...");
```

With this simple code, we have used D3 to select the *body* element, and then append a new *p* element with the text "new paragraph". The one thing we can notice straight away is that D3 supports *chain syntax*, which allowed us to *select*, *append*, and add *text* in one statement.

Effectively, in one statement we were able to select the element, append a new element, and

then bind some data. The data was a string of textual characters in this example.

NB: JavaScript, like HTML, does not care about whitespace, line breaks, so we could have written the previous D3 code on separate lines for each function.

Data visualisation - D3

Data Intro - page elements

```
d3.select("body").append("p").text("sample text...");
```

So, the above D3 can be understood as follows

- D3 - references the D3 object, so we can access its built-in methods
- .select("body") - this method accepts a CSS selector and returns the first instance of the matched selector in the document's DOM
  - *.selectAll() - **NB:** this method is a variant of the single select, and will return all of the matched CSS selectors in the DOM*
- .append("p") - creates the specified new DOM element, and appends it to the end of the defined select CSS selector, just before the closing tag. So, in our example, it simply appends a *p* element to the end of the defined *body*.
- .text("new paragraph") - takes the defined string, "new paragraph", and adds it to the newly created *p* DOM element.

The semi-colon naturally indicates the end of the statement block.

NB: when chaining methods in D3, order of precedence matters. Also the output type of one method needs to match the expected input type of the next method.

Data visualisation - D3

Binding data - making a selection

With our new data, we then need to decide upon a selector within our document. For example, we could select all of the paragraphs in our document

```
d3.select("body").selectAll("p");
```

However, what happens if the element we require does not yet exist. We then need to use a method called *enter()*. For example,

```
d3.select("body").selectAll("p").data(dataset).enter().append("p").text("new paragraph");
```

If we ran this code, we would get new paragraphs that match the total number of values currently available in the *dataset*. Effectively, its akin to looping through an array, and outputting a new paragraph for each value in the array.

So, how does the **enter()** function work. Well, to create new, data-bound elements we need to use *enter()*. This method checks the current DOM selection, and the data being assigned to it. If there are more data values than matching DOM elements, it will simply create a new placeholder element for the data value. It then passes this placeholder on to the next step in the chain, which is *append()* in our code.

If we run the above code we can see the output text within our new paragraphs. However, the data from our dataset has also been assigned to the new paragraphs. This data value is assigned to each paragraph's ***data*** attribute.

NB: when D3 binds data to a DOM element, it does not exist in the DOM itself, but it does exist in the memory as a ***data*** attribute of the given element. So, in this case it is simply assigned in memory to each *p* element.

Data visualisation - D3

Binding data - using the data

If we change our last code example as follows,

```
d3.select("body").selectAll("p").data(dataset).enter().append("p").text(function(d) { return d; });
```

and then load our HTML, we will now see the dataset values output instead of the previous text. Effectively, anytime in the chain after calling the `data()` method, we can then access the current data using `d`. If we are using a loop, the value of *d* will also change corresponding to the position in the loop.

NB: the above function is called an **anonymous** function because we have not assigned a specific function name. If we want to use the above concept of `d`, we need to assign it to a function, anonymous or otherwise, so that it has a value.

We can also bind other things to elements with D3, for example CSS selectors, styles, etc.

```
.style("color", "blue");
```

If we chained the above to the end of our existing code, we would now bind an additional css style attribute to each *p* element, thereby turning the font colour blue.

We could also extend this code to include a conditional statement that checks the value of the data, and then only assigns this CSS styling if it is even or odd. A very simplistic striped colour option, for example.

```
.style("color", function(d) {
if (d % 2 == 0) {
return "green";
} else {
 return "blue";
}
});
```

- DEMO - D3 basic elements

Image - D3 Basic Elements

D3 - basic elements

Source code - demos

D3.js

- D3 basic elements

MongoDB

- 424-node-mongo1

References

- Chocolatey for Windows
  - *Chocolatey package manager for Windows*
- D3.js
  - *D3 - API reference*
  - *D3 - Wiki*
- Homebrew for OS X
  - *Homebrew - the missing package manager for OS X*
- Kirk, A. *Data Visualisation: A successful design process*. Packt Publishing. 2012.
- 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*
- W3 Selector API