

Comp 324/424 - Client-side Web Design

Spring Semester 2020 - Week 14

Dr Nick Hayward

Final Demo and Presentation

- presentation and demo - live working app...
 - *final demo*
 - due on Monday 20th April 2020 @ 4.15pm
 - *final report*
 - due on Monday 27th April 2020 @ 6.45pm
 - *NO content management systems (CMSs) such as Drupal, Joomla, WordPress...*
 - NO PHP, Python, Ruby, C# & .Net, Java, Go, XML...
 - NO CSS frameworks such as Bootstrap, Foundation, Materialize...
 - NO CSS preprocessors such as Sass...
 - NO template tools such as Handlebars.js &c.
 - must implement data from either
 - self hosted (MongoDB, Redis...)
 - APIs
 - cloud services (Firebase...)
 - NO SQL...e.g. (you may NOT use MySQL, PostgreSQL &c.)
 - *explain design decisions*
 - describe patterns used in design of UI and interaction
 - layout choices...
 - *show and explain implemented differences from DEV week*
 - where and why did you update the app?
 - perceived benefits of the updates?
 - *how did you respond to peer review?*
- anything else useful for final assessment...
- consider outline of content from final report outline
- ...

All project code must be pushed to a repository on GitHub.

n.b. present your own work contributed to the project, and its development...

Final Report

Report due on Monday 27th April 2020 @ 6.45pm

- final report outline - coursework section of website
 - *PDF*
 - *group report*
 - *extra individual report - optional*
- include repository details for project code on GitHub

Client-side - Data - Firebase

Firestore - intro

- Firestore is hosted platform, acquired by Google
 - *provides options for data storage, authentication, real-time database querying...*
- it provides an API for data access
 - *access and query JavaScript object data stores*
 - *query in real-time*
 - *listeners available for all connected apps and users*
 - *synchronisation in milliseconds for most updates...*
 - *notifications*

Client-side - Data - Firebase

Firebase - authentication

- authentication with Firebase provides various backend services and SDKs
 - *help developers manage authentication for an app*
 - *service supports many different providers, including Facebook, Google, Twitter &c.*
 - *using industry standard OAuth 2.0 and OpenID Connect protocols*
- custom solutions also available per app
 - *email*
 - *telephone*
 - *messaging*
 - *...*

Client-side - Data - Firebase

Firestore - cloud storage

- **Cloud Storage** used for uploading, storing, downloading files
 - *accessed by apps for file storage and usage...*
 - *features a useful safety check if and when a user's connection is broken or lost*
 - *files are usually stored in a Google Cloud Storage bucket*
 - *files accessible using either Firestore or Google Cloud*
 - *consider using Google Cloud platform for image filtering, processing, video editing...*
 - *modified files may then become available to Firestore again, and connected apps*
 - *e.g. Google's Cloud Platform*

Client-side - Data - Firebase

Firestore - Real-time database

- **Real-time Database** offers a hosted NoSQL data store
 - *ability to quickly and easily sync data*
 - *data synchronisation is active across multiple devices, in real-time*
 - *available as and when the data is updated in the cloud database*
- **other services and tools available with Firebase**
 - *analytics*
 - *advertising services such as adwords*
 - *crash reporting*
 - *notifications*
 - *various testing options...*

Client-side - Data - Firebase

Firestore - basic setup

- start using Firestore by creating an account with the service
 - *using a standard Google account*
 - *Firestore*
- login to Firestore
 - *choose either Get Started material or navigate to Firestore console*
- at *Console* page, get started by creating a new project
 - *click on the option to Add project*
 - *enter the name of this new project*
 - *and select a region*
- then redirected to the *console dashboard* page for the new project
 - *access project settings, config, maintenance...*
- reference documentation for the Firestore Real-Time database,
 - *<https://firebase.google.com/docs/reference/js/firebase.database>*

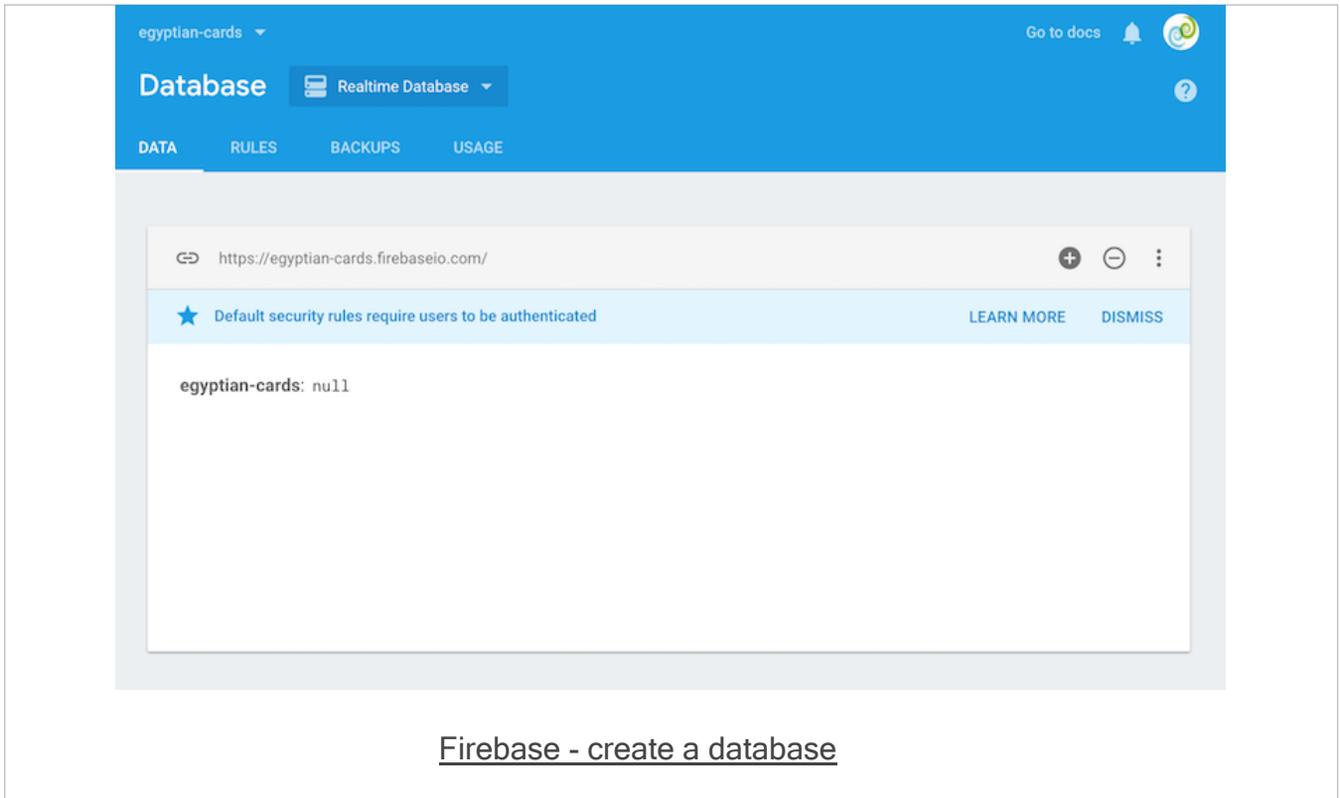
Client-side - Data - Firebase

Firestore - create real-time database

- now setup a database with Firestore for a test app
- start by selecting *Database* option from left sidebar on the Console Dashboard
 - *available under the DEVELOP option*
- then select *Get Started* for the real-time database
- presents an empty database with an appropriate name to match current project
- data will be stored in a JSON format in the real-time database
- working with Firestore is usually simple and straightforward for most apps
- get started quickly direct from the Firestore console
 - *or import some existing JSON...*

Image - Firebase

create a database



Firebase - create a database

Client-side - Data - Firebase

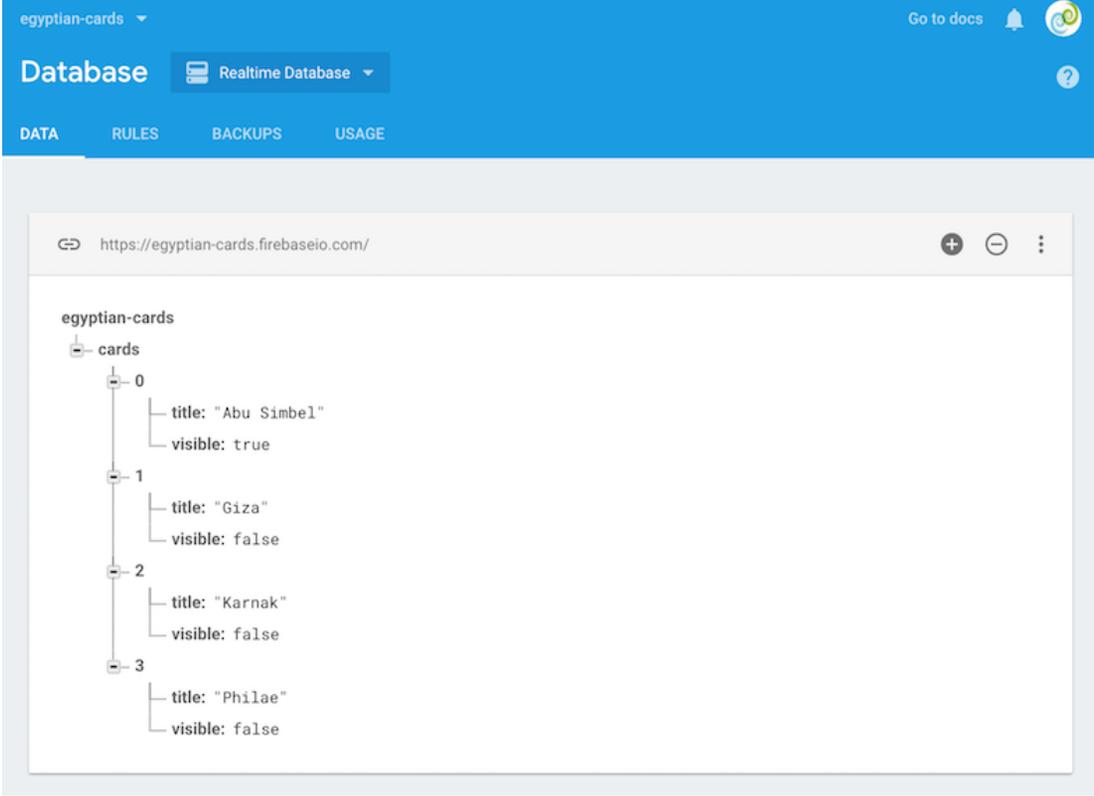
Firestore - import JSON data

- we might start with some simple data to help test Firestore
- import JSON into our test database
 - *then query the data &c. from the app*

```
{
  "cards": [
    {
      "visible": true,
      "title": "Abu Simbel",
      "card": "temple complex built by Ramesses II"
    },
    {
      "visible": false,
      "title": "Amarna",
      "card": "capital city built by Akhenaten"
    },
    {
      "visible": false,
      "title": "Giza",
      "card": "Khufu's pyramid on the Giza plateau outside Cairo"
    },
    {
      "visible": false,
      "title": "Philae",
      "card": "temple complex built during the Ptolemaic period"
    }
  ]
}
```

Image - Firebase

JSON import



The screenshot displays the Firebase Realtime Database interface for a project named 'egyptian-cards'. The top navigation bar is blue and includes the project name, 'Database', 'Realtime Database', and navigation links for 'DATA', 'RULES', 'BACKUPS', and 'USAGE'. The main content area shows a tree view of the database structure. The root node is 'egyptian-cards', which contains a child node 'cards'. The 'cards' node is expanded to show four indexed entries (0, 1, 2, 3). Each entry contains a 'title' and a 'visible' property.

```
egyptian-cards
├── cards
│   ├── 0
│   │   ├── title: "Abu Simbel"
│   │   └── visible: true
│   ├── 1
│   │   ├── title: "Giza"
│   │   └── visible: false
│   ├── 2
│   │   ├── title: "Karnak"
│   │   └── visible: false
│   └── 3
│       ├── title: "Philae"
│       └── visible: false
```

[Firebase - import JSON file](#)

Client-side - Data - Firebase

Firestore - permissions

- initial notification in Firebase console after creating a new database
 - *Default security rules require users to be authenticated*
- permissions with Firestore database
 - *select RULES tab for current database*
- lots of options for database rules
 - *Firestore - database rules*
- e.g. for testing initial app we might remove authentication rules
- change rules as follows

from

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```

to

```
{
  "rules": {
    ".read": "true",
    ".write": "true"
  }
}
```

Client-side - Data - Firebase

add data with plain JS objects

- plain objects as standard Firebase storage
 - *helps with data updating*
 - *helps with auto-increment pushes of data...*

```
{
  "egypt": {
    "code": "eg",
    "ancient_sites": {
      "abu_simbel": {
        "title": "abu simbel",
        "kingdom": "upper",
        "location": "aswan governorate",
        "coords": {
          "lat": 22.336823,
          "long": 31.625532
        },
        "date": {
          "start": {
            "type": "bc",
            "precision": "approximate",
            "year": 1264
          },
          "end": {
            "type": "bc",
            "precision": "approximate",
            "year": 1244
          }
        }
      },
      "karnak": {
        "title": "karnak",
        "kingdom": "upper",
        "location": "luxor governorate",
        "coords": {
          "lat": 25.719595,
          "long": 32.655807
        },
        "date": {
          "start": {
            "type": "bc",
            "precision": "approximate",
            "year": 2055
          },
          "end": {
            "type": "ad",
            "precision": "approximate",

```

```
"year": 100
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

Image - Firebase

JSON import



[Firebase - import JSON file](#)

Client-side - Data - Firebase

add to app's index.html

- start testing setup with default config in app's index.html file
 - *e.g.*

```
<!-- JS - Firebase app -->
<script src="https://www.gstatic.com/firebasejs/5.5.8/firebase.js"></script>
<script>
  // Initialise Firebase
  var config = {
    apiKey: "YOUR_API_KEY",
    authDomain: "422cards.firebaseio.com",
    databaseURL: "https://422cards.firebaseio.com",
    projectId: "422cards",
    storageBucket: "422cards.appspot.com",
    messagingSenderId: "282356174766"
  };
  firebase.initializeApp(config);
</script>
```

- example includes initialisation information so the SDK has access to
 - *Authentication*
 - *Cloud storage*
 - *Realtime Database*
 - *Cloud Firestore*

n.b. don't forget to modify the above values to match your own account and database...

Client-side - Data - Firebase

customise API usage

- possible to customise required components per app
- allows us to include only features required for each app
- *e.g. the only required component is*
- *firebase-app - core Firebase client (required component)*

```
<!-- Firebase App is always required and must be first -->  
<script src="https://www.gstatic.com/firebasejs/5.5.8/firebase-app.js"></script>
```

- we may add a mix of the following optional components,
 - *firebase-auth - various authentication options*
 - *firebase-database - realtime database*
 - *firebase-firestore - cloud Firestore*
 - *firebase-functions - cloud based function for Firebase*
 - *firebase-storage - cloud storage*
 - *firebase-messaging - Firebase cloud messaging*

Client-side - Data - Firebase

modify JS in app's `index.html`

```
<!-- Add additional services that you want to use -->
<script src="https://www.gstatic.com/firebasejs/5.5.3/firebase-auth.js"></script>
<script src="https://www.gstatic.com/firebasejs/5.5.3/firebase-database.js"></script>
<script src="https://www.gstatic.com/firebasejs/5.5.3/firebase-firestore.js"></script>
<script src="https://www.gstatic.com/firebasejs/5.5.3/firebase-messaging.js"></script>
<script src="https://www.gstatic.com/firebasejs/5.5.3/firebase-storage.js"></script>

<script src="https://www.gstatic.com/firebasejs/5.5.3/firebase-functions.js"></script>
```

- then define an object for the config of the required services and options,

```
var config = {
  // add API key, services &c.
};
firebase.initializeApp(config);
```

Client-side - Data - Firebase

initial app usage - DB connection

- after defining required config and initialisation
 - *start to add required listeners and calls to app's JS*

define DB connection

- we can establish a connection to our Firebase DB as follows,

```
const db = firebase.database();
```

- then use this reference to connect and query our database

Client-side - Data - Firebase

initial app usage - ref() method

- with the connection to the database
 - *we may then call the ref(), or reference, method*
 - *use this method to read, write &c. data in the database*
- by default, if we call ref() with no arguments
 - *our query will be relative to the root of the database*
 - *e.g. reading, writing &c. relative to the whole database*
- we may also request a specific reference in the database
 - *pass a location path, e.g.*

```
db.ref('egypt/ancient_sites/abu_simbel/title').set('Abydos');
```

- allows us to create multiple parts of the Firebase database
- such parts might include,
 - *multiple objects, properties, and values &c.*
- a quick and easy option for organising and distributing data

Client-side - Data - Firebase

write data - intro

- also write data to the connected database
 - *again from a JavaScript based application*
- Firebase supports many different JavaScript datatypes, including
 - *strings*
 - *numbers*
 - *booleans*
 - *objects*
 - *arrays*
 - *...*
- i.e. any values and data types we add to JSON
 - *n.b. Firebase may not maintain the native structure upon import*
 - *e.g. arrays will be converted to plain JavaScript objects in Firebase*

Client-side - Data - Firebase

write data - set all data

- set data for the whole database by calling the `ref()` method at the *root*
 - *e.g.*

```
db.ref().set({
  site: 'abu-simbel',
  title: 'Abu Simbel',
  date: 'c.1264 B.C.',
  visible: true,
  location: {
    country: 'Egypt',
    code: 'EG',
    address: 'aswan'
  }
  coords: {
    lat: '22.336823',
    long: '31.625532'
  }
});
```

Client-side - Data - Firebase

write data - set data for a specific data location

- also write data to a specific location in the database
- add an argument to the `ref()` method
 - *specifying required location in the database*
 - *e.g.*

```
db.ref('egypt/ancient_sites/abu_simbel/location').set('near aswan');
```

- `ref()` may be called relative to any depth in the database from the *root*
- allows us to update anything from whole DB to single property value

Client-side - Data - Firebase

Promises with Firebase

- Firebase includes native support for Promises and associated chains
 - *we do not need to create our own custom Promises*
- we may work with a return Promise object from Firebase
 - *using a standard chain, methods...*
- e.g. when we call the `set()` method
 - *Firestore will return a Promise object for the method execution*
- `set()` method will not explicitly return anything except for success or error
 - *we can simply check the return promise as follows,*

```
db.ref('egypt/ancient_sites/abu_simbel/title')
  .set('Abu Simbel')
  .then(() => {
    // Log data set success to console
    console.log('data set...');
  })
  .catch((e) => {
    // catch error from Firestore - error logged to console
    console.log('error returned', e);
  });
```

Client-side - Data - Firebase

remove data - intro

- we may also delete and remove data from the connected database
- various options for removing such data, including
 - *specific location*
 - *all data*
 - *set() with null*
 - *by updating data*
 - ...

Client-side - Data - Firebase

remove data - specify location

- we may also delete data at a specific location in the connected database
 - *e.g.*

```
db.ref('egypt/ancient_sites/abu_simbel/kingdom')
  .remove()
  .then(() => {
    // Log data removed success to console
    console.log('data removed...');
  })
  .catch((e) => {
    // catch error from Firebase - error logged to console
    console.log('error returned', e);
  });
```

Client-side - Data - Firebase

remove data - all data

- also remove all of the data in the connected database
 - *e.g.*

```
db.ref()
  .remove()
  .then(() => {
    // Log data removed success to console
    console.log('data removed...');
  })
  .catch((e) => {
    // catch error from Firebase - error logged to console
    console.log('error returned', e);
  });
```

Client-side - Data - Firebase

remove data - `set()` with null

- another option specified in the Firebase docs for deleting data
 - *by using `set()` method with a null value*
 - *e.g.*

```
db.ref('egypt/ancient_sites/abu_simbel/kingdom')
  .set(null)
  .then(() => {
    // Log data removed success to console
    console.log('data set to null...');
  })
  .catch((e) => {
    // catch error from Firebase - error logged to console
    console.log('error returned', e);
  });
```

Client-side - Data - Firebase

update data - intro

- also combine setting and removing data in a single pattern
 - *using the `update()` method call to the defined database reference*
- meant to be used to update multiple items in database in a single call
- we must pass an object as the argument to the `update()` method

Client-side - Data - Firebase

update data - existing properties

- to update multiple existing properties
 - *e.g.*

```
db.ref('egypt/ancient_sites/abu_simbel/').update({
  title: 'The temple of Abu Simbel',
  visible: false
});
```

Client-side - Data - Firebase

update data - add new properties

- also add a new property to a specific location in the database

```
db.ref('egypt/ancient_sites/abu_simbel/').update({
  title: 'The temple of Abu Simbel',
  visible: false,
  date: 'c.1264 B.C.'
});
```

- still set new values for the two existing properties
 - *title and visible*
- add a new property and value for data
- `update()` method will only update the specific properties
 - *does not override everything at the reference location*
 - *compare with the `set()` method...*

Client-side - Data - Firebase

update data - remove properties

- also combine these updates with option to remove an existing property
 - *e.g.*

```
db.ref('egypt/ancient_sites/abu_simbel/').update({
  card: null,
  title: 'The temple of Abu Simbel',
  visible: false,
  date: 'c.1264 B.C.',
});
```

- null used to delete specific property from reference location in DB
- at the reference location in the DB, we're able to combine
 - *creating new property*
 - *updating a property*
 - *deleting existing properties*

Client-side - Data - Firebase

update data - multiple properties at different locations

- also combine updating data in multiple objects at different locations
 - *locations relative to initial passed reference location*
 - *e.g.*

```
db.ref().update({
  'egypt/ancient_sites/abu_simbel/visible': true,
  'egypt/ancient_sites/karnak/visible': false
});
```

- relative to the root of the database
 - *now updated multiple title properties in different objects*
- *n.b.* update is only for child objects relative to specified ref location
 - *due to character restrictions on the property name*
 - *e.g. the name may not begin with . , / &c.*

Client-side - Data - Firebase

update data - Promise chain

- `update()` method will also return a Promise object
 - *allows us to chain the standard methods*
 - *e.g.*

```
db.ref().update({
  'egypt/ancient_sites/abu_simbel/visible': true,
  'egypt/ancient_sites/karnak/visible': false
}).then(() => {
  console.log('update success...');
}).catch((e) => {
  console.log('error = ', e);
});
```

- as with `set()` and `remove()`
 - *Promise object itself will return success or error for method call*

Client-side - Data - Firebase

read data - intro

- fetch data from the connected database in many different ways, e.g.
 - *all of the data*
 - *or a single specific part of the data*
- also connect and retrieve data once
- another option is to setup a listener
 - *used for polling the database for live updates...*

Client-side - Data - Firebase

read data - all data, once

- retrieve all data from the database a single time

```
// ALL DATA ONCE - request all data ONCE
// - returns Promise value
db.ref().once('value')
  .then((snapshot) => {
    // snapshot of the data - request the return value for the data at the time of query...
    const data = snapshot.val();
    console.log('data = ', data);
  })
  .catch((e) => {
    console.log('error returned - ', e);
  });
```

Client-side - Data - Firebase

read data - single data, once

- we may query the database once for a single specific value
 - *e.g.*

```
// SINGLE DATA - ONCE
db.ref('egypt/ancient_sites/abu_simbel/').once('value')
  .then((snapshot) => {
    // snapshot of the data - request the return value for the data at the time of query...
    const data = snapshot.val();
    console.log('single data = ', data);
  })
  .catch((e) => {
    console.log('error returned - ', e);
  });
```

- returns value for object at the specified location
 - *egypt/ancient_sites/abu_simbel/*

Client-side - Data - Firebase

read data - listener for changes - subscribe

- also setup listeners for changes to the connected database
 - *then continue to poll the DB for any subsequent changes*
 - *e.g.*

```
// LISTENER - poll DB for data changes
// - any changes in the data
db.ref().on('value', (snapshot) => {
  console.log('listener update = ', snapshot.val());
});
```

- `on()` method polls the DB for any changes in `value`
- then get the current snapshot value for the data stored
- any change in data in the online database
 - *listener will automatically execute defined success callback function*

Client-side - Data - Firebase

read data - listener for changes - subscribe - error handling

- also add some initial error handling for subscription callback
 - *e.g.*

```
// LISTENER - SUBSCRIBE
// - poll DB for data changes
// - any changes in the data
db.ref().on('value', (snapshot) => {
  console.log('listener update = ', snapshot.val());
}, (e) => {
  console.log('error reading db', e);
});
```

Client-side - Data - Firebase

read data - listener - why not use a Promise?

- as listener is notified of updates to the online database
 - *we need the callback function to be executed*
- callback may need to be executed multiple times
 - *e.g. for many updates to the stored data*
- a Promise may only be resolved a single time
 - *with either resolve or reject*
- to use a Promise in this context
 - *we would need to instantiate a new Promise for each update*
 - *would not work as expected*
 - *therefore, we use a standard callback function*
- a callback may be executed as needed
 - *each and every time there is an update to the DB*

Client-side - Data - Firebase

read data - listener for changes - unsubscribe

- need to *unsubscribe* from all or specific changes in online database
 - *e.g.*

```
db.ref().off();
```

- removes *all* current subscriptions to defined DB connection

Client-side - Data - Firebase

read data - listener for changes - unsubscribe

- also *unsubscribe* a specific subscription by passing callback
 - *callback as used for the original subscription*
- abstract the callback function
 - *pass it to both on() and off() methods for database ref() method*
 - *e.g.*

```
// abstract callback
const valChange = (snapshot) => {
  console.log('listener update = ', snapshot.val());
};
```

Client-side - Data - Firebase

read data - listener for changes - unsubscribe

- then pass this variable as callback argument
 - *for both subscribe and unsubscribe events*
 - *e.g.*

```
// subscribe
db.ref().on('value', valChange);
// unsubscribe
db.ref().off(valChange);
```

- allows our app to maintain the DB connection
 - *and unsubscribe a specific subscription*

Client-side - Data - Firebase

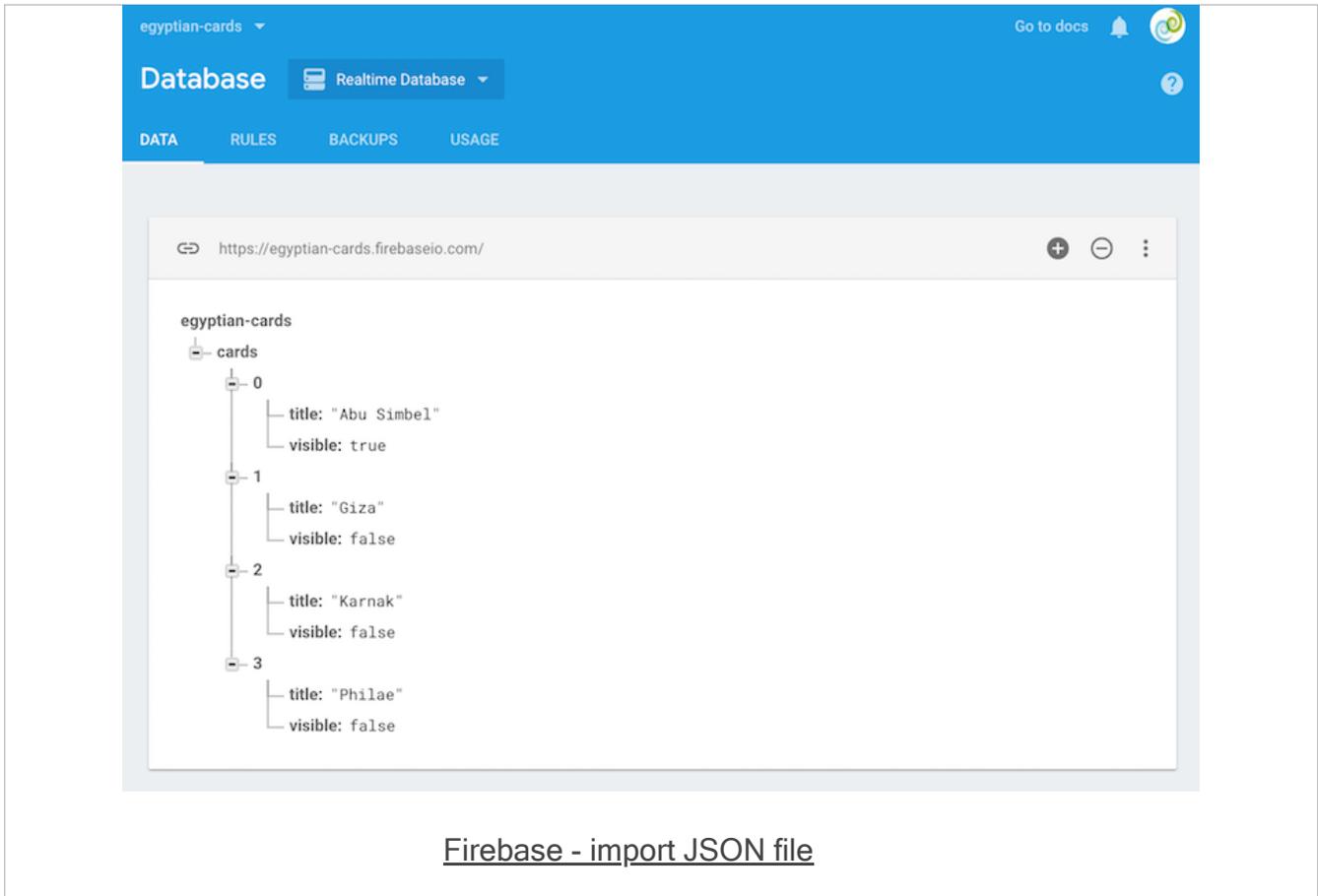
working with arrays

- Firebase does not explicitly support array data structures
 - *converts array objects to plain JavaScript objects*
- e.g. import the following JSON with an array

```
{
  "cards": [
    {
      "visible": true,
      "title": "Abu Simbel",
      "card": "temple complex built by Ramesses II"
    },
    {
      "visible": false,
      "title": "Amarna",
      "card": "capital city built by Akhenaten"
    },
    {
      "visible": false,
      "title": "Giza",
      "card": "Khufu's pyramid on the Giza plateau outside Cairo"
    },
    {
      "visible": false,
      "title": "Philae",
      "card": "temple complex built during the Ptolemaic period"
    }
  ]
}
```

Image - Firebase

JSON import with array



Client-side - Data - Firebase

working with arrays - index values

- each index value will now be stored as a plain object
 - *with an auto-increment value for the property*
 - *e.g.*

```
cards: {  
  0: {  
    card: "temple complex built by Ramesses II",  
    title: "Abu Simbel",  
    visible: "true"  
  }  
}
```

Client-side - Data - Firebase

working with arrays - access index values

- we may still access each index value from the original array object
 - *without easy access to pre-defined, known unique references*
- e.g. to access the title value of a given card
 - *need to know its auto-generated property value in Firebase*

```
db.ref('cards/0')
```

- reference will be the path to the required object
 - *then access a given property on the object*
- even if we add a unique reference property to each card
 - *still need to know assigned property value in Firebase*

Client-side - Data - Firebase

working with arrays - push() method

- add new content to an existing Firebase datastore
- we may use the push() method to add this data
- a unique property value will be auto-generated for pushed data
 - *e.g.*

```
// push new data to specific reference in db
db.ref('egypt/ancient_sites/').push({
  "philae": {
    "kingdom": "upper",
    "visible": false
  }
});
```

- new data created with auto-generated ID for parent object
 - *e.g.*

```
LPcdS31H_u9N0dIn27_
```

- may be useful for dynamic content pushed to a datastore
 - *e.g. notes, tasks, calendar dates &c.*

Client-side - Data - Firebase

working with arrays - Firebase snapshot methods

- various data snapshot methods in the Firebase documentation
- commonly used method with snapshot is the `val()` method
- many additional methods specified in API documentation for *DataSnapshot*
 - *e.g. forEach()* - iterator for plain objects from Firebase
 - *Firestore Docs - DataSnapshot*

Client-side - Data - Firebase

working with arrays - create array from Firebase data

- as we store data as plain objects in Firebase
 - *need to consider how we may work with array-like structures*
 - *i.e. for technologies and patterns that require array data structures*
 - *e.g. Redux*
- need to get data from Firebase, then prepare it for use as an array
- to help us work with Firebase object data and arrays
 - *we may call `forEach()` method on the return snapshot*
 - *provides required iterator for plain objects stored in Firebase*
 - *e.g.*

```
// get ref in db once
// call forEach() on return snapshot
// push values to local array
// unique id for each DB parent object is `key` property on snapshot
db.ref('egypt/ancient_sites')
  .once('value')
  .then((snapshot) => {
    const sites = [];
    snapshot.forEach((siteSnapshot) => {
      sites.push({
        id: siteSnapshot.key,
        ...siteSnapshot.val()
      });
    });
    console.log('sites array = ', sites);
  });
```

Image - Firebase

snapshot.forEach() - creating a local array

```
sites array = firebase.js:166
▼ (3) [{...}, {...}, {...}] ⓘ
  ▼ 0:
    id: "-LPcdS31H_u9N0dIn27_"
    ▶ philae: {kingdom: "upper", visible: false}
    ▶ __proto__: Object
  ▼ 1:
    ▶ coords: {lat: 22.336823, long: 31.625532}
    ▶ date: {end: {...}, start: {...}}
    id: "abu_simbel"
    kingdom: "upper"
    location: "aswan governorate"
    title: "Abu Simbel"
    visible: true
    ▶ __proto__: Object
  ▼ 2:
    ▶ coords: {lat: 25.719595, long: 32.655807}
    ▶ date: {end: {...}, start: {...}}
    id: "karnak"
    kingdom: "upper"
    location: "luxor governorate"
    title: "karnak"
    visible: false
    ▶ __proto__: Object
  length: 3
  ▶ __proto__: Array(0)
```

Firestore - local array

- we now have a local array from the Firestore object data
 - *use with options such as Redux...*

Client-side - Data - Firebase

add listeners for value changes

- as we modify objects, properties, values &c. in Firebase
 - *set listeners to return notifications for such updates*
 - *e.g. add a single listener for any update relative to full datastore*

```
// LISTENER - SUBSCRIBE - v.2
// - get all data & then push return data to local array...
db.ref('egypt').on('value', (snapshot) => {
  const sites = [];
  snapshot.forEach((siteSnapshot) => {
    sites.push({
      id: siteSnapshot.key,
      ...siteSnapshot.val()
    });
  });
  console.log('sites array after update = ', sites);
});
```

- the `on()` method does not return a Promise object
 - *we need to define a callback for the return data*

Client-side - Data - Firebase

listener events - intro

- for subscriptions and updates
 - *Firebase provides a few different events*
- for the `on()` method, we may initially consult the following documentation
 - *Firebase docs - `on()` events*
- need to test various listeners for datastore updates

Client-side - Data - Firebase

listener events - `child_removed` event

- add a subscription for event updates
 - *as a child object is removed from the data store.*
- `child_removed` event may be added as follows,

```
// - listen for child_removed event relative to current ref path in DB
db.ref('egypt/ancient_sites/').on('child_removed', (snapshot) => {
  console.log('child removed = ', snapshot.key, snapshot.val());
});
```

Client-side - Data - Firebase

listener events - `child_changed` event

- also listen for the `child_changed` event
 - *relative to the current path passed to `ref()`*
 - *e.g.*

```
// - Listen for child_changed event relative to current ref path in DB
db.ref('egypt/ancient_sites/').on('child_changed', (snapshot) => {
  console.log('child changed = ', snapshot.key, snapshot.val());
});
```

Client-side - Data - Firebase

listener events - `child_added` event

- another common event is adding a new child to the data store
 - *a user may create and add a new note or to-do item...*
 - *e.g. new child added to specified reference*

```
// - Listen for child_added event relative to current ref path in DB
db.ref('egypt/ancient_sites/').on('child_added', (snapshot) => {
  console.log('child added = ', snapshot.key, snapshot.val());
});
```

Client-side - Data - Firebase

extra notes

- Firebase - authentication
- Firebase - setup & usage

Data visualisation

intro - part 1

- data visualisation - study of how to visually communicate and analyse data
- covers many disparate aspects
 - *including infographics, exploratory tools, dashboards...*
- already some notable definitions of data visualisation
- one of the better known examples,

“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
 - *the underlying premise remains the same*
- simply, data visualisation is a visual representation of the underlying data
- visualisation aims to impart a better understanding of this data
 - *by association, its relevant context*

Data visualisation

intro - part 2

- an inherent flip-side to data visualisation
- without a correct understanding of its application
 - *it can simply impart a false perception, and understanding, on the dataset*
- run the risk of creating many examples of standard **areal unit** problem
 - *perception often based on creator's base standard and potential bias*
- inherently good at seeing what we want to see
- without due care and attention visualisations may provide false summations of the data

Data visualisation

types - part 1

- many different ways to visualise datasets
 - *many ways to customise a standard infographic*
 - some standard examples that allow us to consider the nature of visualisations
 - *infographics*
 - *exploratory visualisations*
 - *dashboards*
 - perceived that data visualisation is simply a variation between
 - *infographics, exploratory tools, charts, and some data art*
1. infographics * well suited for representing large datasets of contextual information * often used in projects more inclined to exploratory data analysis,
 - *tend to be more interactive for the user*
 - data science can perceive infographics as improper data visualisation because
 - *they are designed to guide a user through a story*
 - *the main facts are often already highlighted*
 - NB: such classifications often still only provide tangible reference points

Data visualisation

types - part 2

2. exploratory visualisations * more interested in the provision of tools to explore and interpret datasets * visualisations can be represented either static or interactive * from a user perspective these charts can be viewed
 - *either carefully*
 - *simply become interactive representations*
 - both perspectives help a user discover new and interesting concepts
 - interactivity may include
 - *option for the user to filter the dataset*
 - *interact with the visualisation via manipulation of the data*
 - *modify the resultant information represented from the data*
 - often perceived as more objective and data oriented than other forms
3. dashboards * dense displays of charts * represent and understand a given issue, domain...
 - *as quickly and effectively as possible*
 - examples of dashboards
 - *display of server logs, website users, business data...*

Data visualisation

Dashboards - intro

- dashboards are dense displays of charts
- allow us to represent and understand the key metrics of a given issue
 - *as quickly and effective as possible*
 - *eg: consider display of server logs, website users, and business data...*
- 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.

- dashboards are visual displays of information
 - *can contain text elements*
 - *primarily a visual display of data rendered as meaningful information*

Data visualisation

Dashboards - intro

- information needs to be consumed quickly
 - *often simply no available time to read long annotations or repeatedly click controls*
- information needs to be visible, and ready to be consumed
- dashboards are normally presented as a complementary environment
 - *an option to other tools and analytical/exploratory options*
- design issues presented by dashboards include effective distribution of available space
 - *compact charts that permit quick data retrieval are normally preferred*
- dashboards should be designed with a purpose in mind
 - *generalised information within a dashboard is rarely useful*
 - *display most important information necessary to achieve their defined purpose*
- a dashboard becomes a central view for collated data
 - *represented as meaningful information*

Data visualisation

Dashboards - good practices

- to help promote our information
 - *need to design the dashboard to fully exploit available screen space*
- need to use this space to help users absorb as much information as possible
- some visual elements more easily perceived and absorbed by users than others
- some naturally convey and communicate information more effectively than others
- such attributes are known as **pre-attentive attributes of visual perception**
- for example,
 - *colour*
 - *form*
 - *position*

Data visualisation

Dashboards - visual perception

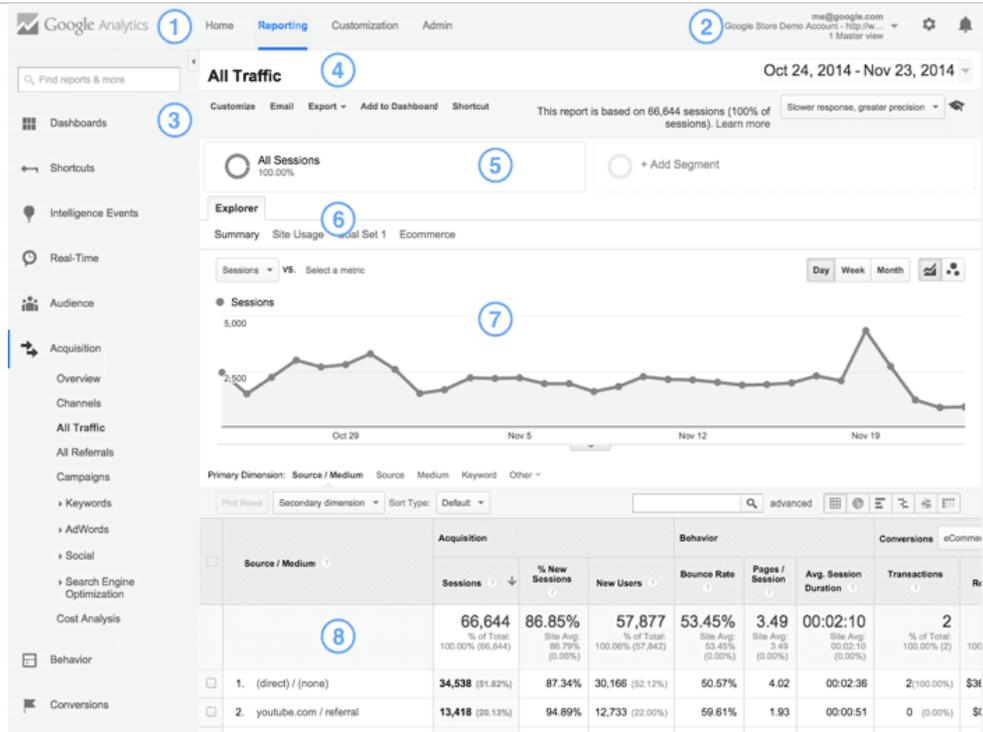
- pre-attentive attributes of visual perception
 1. *Colour* * many different colour models currently available * most useful relevant to dashboard design is the *HSL* model * this model describes colour in terms of three attributes
 - *hue*
 - *saturation*
 - *lightness* * *perception of colour often 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

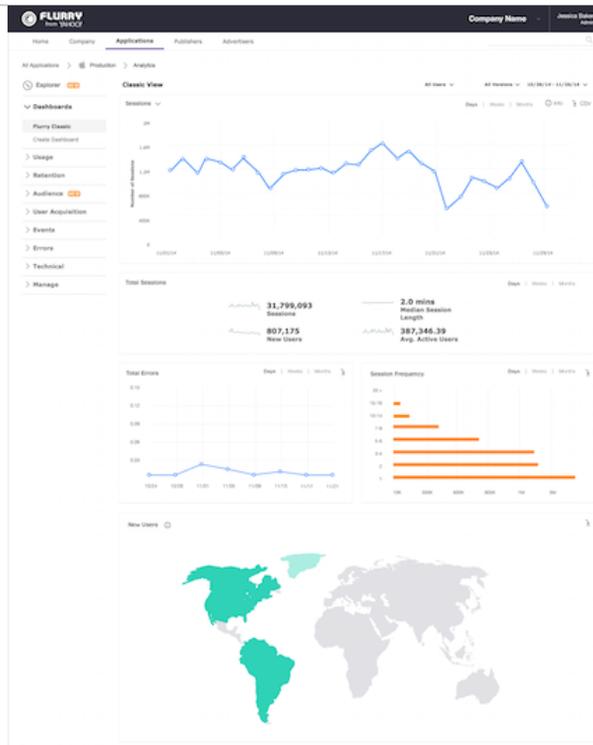
- need to clearly determine the questions that need to be answered
 - *given the information collated and presented within the dashboard*
- need to ensure that any problems can be detected on time
- be certain why we actually need a dashboard for the current dataset
- then begin to collect the requisite data to help us answer such questions
 - *data can be sourced from multiple, disparate datasets*
- chosen visualisations help us tell this story more effectively
- present it in a manner appealing to our users
- need to consider information visualisations familiar to our users
 - *helps reduce any potential user's cognitive overload*
- carefully consider organisation of data and information
- organise the data into logical units of information
 - *helps present dashboard information in a meaningful manner*
- dashboard sections should be organised
 - *to help highlight and detect any underlying or prevailing issues*
 - *then present them to the user*

Image - Google Analytics



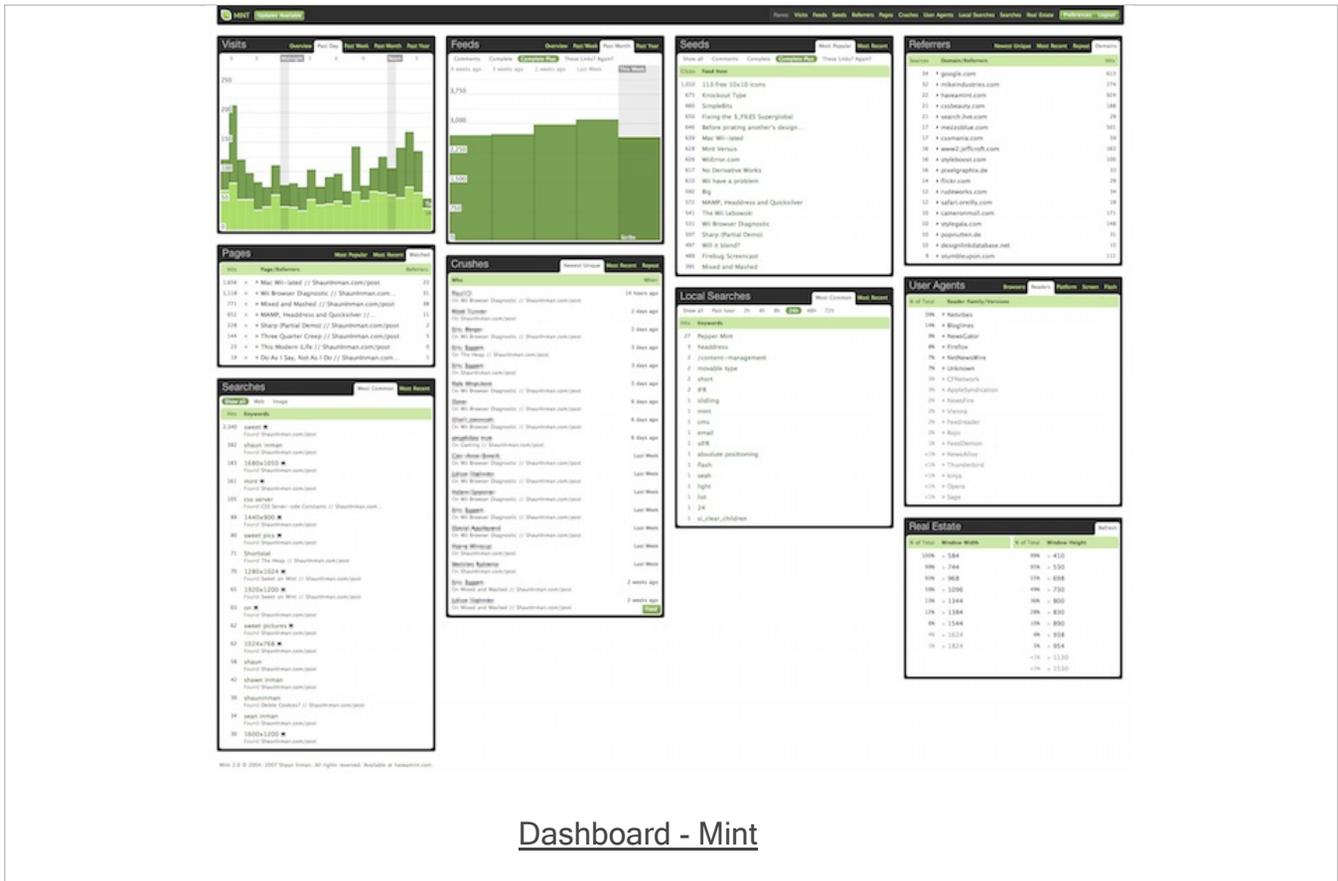
Dashboard - Google Analytics

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*
 - *uses a custom library with HTML, CSS, and SVG*
 - *creates graphically rich, informative documents for the presentation of data*
- D3 uses a data-driven approach to manipulate the DOM
- Setup and configuration of D3 is straightforward
 - *most involved aspect is the configuration of a web server*
- D3.js works with standard HTML files
 - *requires a web server capable of parsing and rendering HTML...*
- to parse D3 correctly we need
 - *UTF-8 encoding reference in a meta element in the head section of our file*
 - *reference D3 file, CDN in standard script element in HTML*

Data visualisation - D3

intro - part 2

- D3 Wiki describes the underlying functional concepts as follows,

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

D3 Wiki

- in JS, functions are objects
 - *as with other objects, a function is a collection of a name and value pair*
- real difference between a function object and a regular object
 - *a function can be invoked, and associated, with two hidden properties*
 - *include a function context and function code*
- variable resolution in D3 relies on variable searching being performed locally first
- if a variable declaration is not found
 - *search will continue to the parent object*
 - *continue recursively to the next static parent*
 - *until it reaches global variable definition*
 - *if not found, a reference error will be generated for this variable*
- 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
- consider data relative to D3
 - *need to know how data can be represented*
 - *both in programming constructs and its associated visual metaphor*
- 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

- a general concept of data and information
- consider them relative to visualisation, impart a richer interpretation
- information, in this context, is no longer
 - *the simple result of processed raw data or facts*
 - *it becomes a visual metaphor of the facts*
- same data set can generate any number of visualisations
 - *may lay equal claim in terms of its validity*
- visualisation is communicating creator's insight into data...

Data visualisation - D3

Data Intro - part 2

- relative to development for visualisation
 - *data will often be stored simply in a text or binary format*
- not simply textual data, can also include data representing
 - *images, audio, video, streams, archives, models...*
- for D3 this concept may often simply be restricted to
 - *textual data, or text-based data...*
 - *any data represented as a series of numbers and strings containing alpha numeric characters*
- suitable textual data for use with D3
 - *text stored as a comma-separated value file (.csv)*
 - *JSON document (.json)*
 - *plain text file (.txt)*
- data can then be *bound* to elements within the DOM of a page using D3
 - *inherent pattern for D3*

Data visualisation - D3

Data Intro - Enter-Update-Exit Pattern

- in D3, connection between data and its visual representation
 - *usually referred to as the enter-update-exit pattern*
- concept is starkly different from the standard imperative programming style
- pattern includes
 - *enter mode*
 - *update mode*
 - *exit mode*

Data visualisation - D3

Data Intro - Enter-Update-Exit Pattern

Enter mode

- `enter()` function returns all specified data that not yet represented in visual domain
- standard modifier function chained to a selection method
 - *create new visual elements representing given data elements*
 - *eg: keep updating an array, and outputting new data bound to elements*

Update mode

- `selection.data(data)` function on a given selection
 - *establishes connection between data domain and visual domain*
- returned result of intersection of data and visual will be a **data-bound selection**
- now invoke a modifier function on this newly created selection
 - *update all existing elements*
 - *this is what we mean by an update mode*

Exit mode

- invoke `selection.data(data).exit` function on a data-bound selection
 - *function computes new selection*
 - *contains all visual elements no longer associated with any valid data element*
- eg: create a bar chart with 25 data points
 - *then update it to 20, so we now have 5 left over*
 - *exit mode can now remove excess elements for 5 spare data points*

Data visualisation - D3

Data Intro - binding data - part 1

- consider standard patterns for working with data
- we can iterate through an array, and then bind the data to an element
 - *most common option in D3 is to use the enter-update-exit pattern*
- use same basic pattern for binding object literals as data
- to access our data we call the required attribute of the supplied data

```
var data = [  
  {height: 10, width: 20},  
  {height: 15, width: 25}  
];  
  
function (d) {  
  return (d.width) + "px";  
}
```

- then access the height attribute per object in the same manner
- we can also bind functions as data
 - *D3 allows functions to be treated as data...*

Data visualisation - D3

Data Intro - binding data - part 2

- D3 enables us to bind data to elements in the DOM
 - *associating data to specific elements*
 - *allows us to reference those values later*
 - *so that we can apply required mapping rules*
- use D3's `selection.data()` method to bind our data to DOM elements
 - *we obviously need some data to bind, and a selection of DOM elements*
- D3 is particularly flexible with data
 - *happily accepts various types*
- D3 also has a built-in function to handle loading JSON data

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

Data visualisation - D3

Data Intro - working with arrays - options

- 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));
```

- & many more...

Data visualisation - D3

Data Intro - working with arrays - nest

- D3's nest function used to build an algorithm
 - *transforms a flat array data structure into a hierarchical nested structure*
- function can be configured using the key function chained to nest
- nesting allows elements in an array to be grouped into a hierarchical tree structure
 - *similar in concept to the group by option in SQL*
 - *nest allows multiple levels of grouping*
 - *result is a tree rather than a flat table*
- levels in the tree are defined by the *key* function
- leaf nodes of the tree can be sorted by value
- internal nodes of the tree can be sorted by key

Data visualisation - D3

Selections - intro

- Selection is one of the key tasks required within D3 to manipulate and visualise our data
- 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
 - *supported by all of the modern web browsers*
 - *its limitations are particularly noticeable for work with visualising data*
- Selector API only provides support for selector and not selection
 - *able to select an element in the document*
 - *to manipulate or modify its data we need to implement a standard loop etc*
- D3 introduced its own selection API to address these issues and perceived shortcomings
 - *ability to select elements by ID or class, its attributes, set element IDs and class, and so on...*

Data visualisation - D3

Selections - single element

- select a single element within our page

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

- now select the first <p> element on the page, and then allow us to modify as necessary
 - *eg; we could simply add some text to this element*

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

- selection could be a generic element, such as <p>
 - *or a specific element defined by targeting its ID*
- use additional modifier functions, such as attr, to perform a given modification on the selected element

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

- 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; });
```

Data visualisation - D3

Selections - multiple elements

- also select all of the specified elements using D3

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

- use and implement multiple element selection
 - *same as single selection pattern*
- also use the same modifier functions
- allows us to modify each element's attributes, style, class...

Data visualisation - D3

Selections - iterating through a selection

- D3 provides us with a selection iteration API
 - *allows us to iterate through each selection*
 - *then modify each selection relative to its position*
 - *very similar to the way we normally loop through data*

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

- D3 selections are essentially like arrays with some enhancements
 - *use the iterative nature of Selection API*

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

Data visualisation - D3

Selections - performing sub-selection

- for selections - often necessary to perform specific scope requests
 - *eg: selecting all <p> elements for a given <div> element*

```
//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");
```

- both examples produce the same effect and output, but use very different selection techniques
 - *first example uses the CSS3, level-3, selectors*
 - *div > p is known as combinators in CSS syntax*

Data visualisation - D3

Selections - combinators

Example combinators..

1. descendant combinator

- uses the pattern of selector selector - describing loose parent-child relationship
- loose due to possible relationships - parent-child, parent-grandchild...

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

- select the <p> element as a child of the parent <div> element
 - *relationship can be generational*

2. child combinator

- uses same style of syntax, selector > selector
- able to describe a more restrictive parent-child relationship between two elements

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

- finds <p> element if it is a direct child to the <div> element

Data visualisation - D3

Selections - D3 sub-selection

- sub-selection using D3's built-in selection of child elements
- a simple option to select an element, then chain another selection to get the child element
- this type of chained selection defines a scoped selection within D3
 - *eg: selecting a <p> element nested within our selected <div> element*
 - *each selection is, effectively, independent*
- D3 API built around the inherent concept of function chaining
 - *can almost be considered a Domain Specific Language for dynamically building HTML/SVG elements*
- a benefit of chaining = easy to produce concise, readable code

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

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

Data visualisation - D3

Data Intro - page elements

- generation of new DOM elements normally fits
 - *either circles, rectangles, or some other visual form that represents the data*
- D3 can also create generic structural elements in HTML, such as a <p>
 - *eg: we can append a standard p element to our new page*

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

- used D3 to select body element, then append a new <p> element with text “new paragraph”
- D3 supports *chain syntax*
 - *allowed us to select, append, and add text in one statement*

Data visualisation - D3

Data Intro - page elements

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

- `d3`
 - *references the D3 object, access its built-in methods*
- `.select("body")`
 - *accepts a CSS selector, returns first instance of the matched selector in the document's DOM*
 - `.selectAll()`
 - NB: this method is a variant of the single `select()`
 - returns all of the matched CSS selectors in the DOM
- `.append("p")`
 - *creates specified new DOM element*
 - *appends it to the end of the defined select CSS selector*
- `.text("new paragraph")`
 - *takes defined string, "new paragraph"*
 - *adds it to the newly created <p> DOM element*

Data visualisation - D3

Binding data - making a selection

- choose a selector within our document
 - *eg: we could select all of the paragraphs in our document*

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

- if the element we require does not yet exist
 - *need to use the method enter()*

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

- we get new paragraphs that match total number of values currently available in the dataset
 - *akin to looping through an array*
 - *outputting a new paragraph for each value in the array*
- create new, data-bound elements using enter()
 - *method checks the current DOM selection, and the data being assigned to it*
- if more data values than matching DOM elements
 - *enter() creates a new placeholder element for the data value*
 - *then passes this placeholder on to the next step in the chain, eg: append()*
- data from dataset also assigned to new paragraphs
- NB: when D3 binds data to a DOM element, it does not exist in the DOM itself
 - *it does exist in the memory*

Data visualisation - D3

Binding data - using the data

- change our last code example as follows,

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

- then load our HTML, we'll now see dataset values output instead of fixed text
- anytime in the chain after calling the `data()` method
 - *we can then access the current data using `d`*
- also bind other things to elements with D3, eg: CSS selectors, styles...

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

- chain the above to the end of our existing code
 - *now bind an additional css style attribute to each `<p>` element*
 - *turning the font colour blue*
- extend code to include a conditional statement that checks the value of the data
 - *eg: simplistic striped colour option*

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

- DEMO - D3 basic elements

Image - D3 Basic Elements

Testing - D3

[Home](#) | [d3 basic element](#)

Basic - add text

some sample text...

Basic - add element

p element...

p element...

p element...

p element...

p element...

p element...

Basic - add array value to element (with colour)

0

1

2

3

4

5

Basic - add key & value to element

key = 0, value = 0

key = 1, value = 1

key = 2, value = 2

key = 3, value = 3

key = 4, value = 4

key = 5, value = 5

D3 - basic elements

Data visualisation - D3

Drawing - intro - part 1

1. drawing divs

- one of the easiest ways to draw a rectangle, for example, is with a HTML `<div>`
- an easy way to start drawing a bar chart for our stats
- start with standard HTML elements, then consider more powerful option of drawing with SVG
- semantically incorrect, we could use `<div>` to output bars for a bar chart
 - *use of an empty `<div>` for purely visual effect*
- using D3, add a class to an empty element using `selection.attr()` method

2. setting attributes

- `attr()` is used to set an HTML attribute and its value on an element
- After selecting the required element in the DOM
 - *assign an attributes as follows*

```
.attr("class", "barchart")
```

Data visualisation - D3

Drawing - intro - part 2

- use D3 to draw a set of bars in divs as follows

```
var dataset = [ 1, 2, 3, 4, 5 ];  
  
d3.select("body").selectAll("div")  
  .data(dataset)  
  .enter()  
  .append("div")  
  .attr("class", "bar");
```

- above sample outputs the values from our dataset with no space between them
 - *effectively as a bar chart of equal height*
- modify the height of each representative bar
 - *by setting height of each bar as a function of its corresponding data value*
 - *eg: append the following to our example chain*

```
.style("height", function(d) {  
  return d + "px";  
});
```

- make each bar in our chart more clearly defined by modifying style

```
.style("height", function(d) {  
  var barHeight = d * 3;  
  return barHeight + "px";  
});
```

Data visualisation - D3

Drawing - intro - part 3

1. drawing SVGs

- properties of SVG elements are specified as **attributes**
- represented as property/value pairs within each element tag

```
<element property="value">...</element>
```

- SVG elements exist in the DOM
 - *we can still use D3 methods `append()` and `attr()`*
 - *create new HTML elements and set their attributes*

2. create SVG

- need to create an element for our SVG
- allows us to draw and output all of our required shapes

```
d3.select("body").append("svg");
```

- variable effectively works as a reference
 - *points to the newly created SVG object*
 - *allows us to use this reference to access this element in the DOM*
- DEMO - Drawing with SVG

Image - D3 Basic Drawing

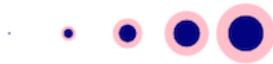
Testing - D3

[Home](#) | [d3 basic drawing](#)

Basic drawing - add text

genius is 1% inspiration, 99% perspiration

Basic drawing - add circles



Basic drawing - add rectangles



[D3 - basic drawing](#)

Data visualisation - D3

Drawing - SVG barchart - part 1

- create a new barchart using SVG, need to set the required size for our SVG output

```
//width & height  
var w = 750;  
var h = 200;
```

- then use D3 to create an empty SVG element, and add it to the DOM

```
var svg = d3.select("body")  
  .append("svg")  
  .attr("width", w)  
  .attr("height", h);
```

- instead of creating DIVs as before, we generate *rects* and add them to the *svg* element.

```
svg.selectAll("rect")  
  .data(dataset)  
  .enter()  
  .append("rect")  
  .attr("x", 0)  
  .attr("y", 0)  
  .attr("width", 10)  
  .attr("height", 50);
```

Data visualisation - D3

Drawing - SVG barchart - part 2

- this code selects all of the rect elements within svg
- initially none, D3 still needs to select them before creating them
- data() then checks the number of values in the specified dataset
 - *hands those values to the enter method for processing*
- enter method then creates a placeholder
 - *for each data value without a corresponding rect*
 - *also appends a rectangle to the DOM for each data value*
- then use attr method to set x, y, width, height values for each rectangle
- still only outputs a single bar due to an overlap issue
- need to amend our code to handle the width of each bar
 - *implement flexible, dynamic coordinates to fit available SVG width and height*
 - *visualisation scales appropriately with the supplied data*

```
.attr("x", function(d, i) {  
  return i * (w / dataset.length);  
})
```

Data visualisation - D3

Drawing - SVG barchart - part 3

- now linked the x value directly to the width of the SVG w
 - *and the number of values in the dataset, `dataset.length`*
 - *the bars will be evenly spaced regardless of the number of values*
- if we have a large number of data values
 - *bars still look like one horizontal bar*
 - *unless there is sufficient width for parent SVG and space between each bar*
- try to solve this as well by setting the bar width to be proportional
 - *narrower for more data, wider for less data*

```
var w = 750;  
var h = 200;  
var barPadding = 1;
```

- now set each bar's width
 - *as a fraction of the SVG width and number of data points, minus our padding value*

```
.attr("width", w / dataset.length - barPadding)
```

- our bar widths and x positions scale correctly regardless of data values

Data visualisation - D3

Drawing - SVG barchart - part 4

- encode our data as the *height* of each bar

```
.attr("height", function(d) {  
    return d * 4;  
});
```

- our bar chart will size correctly, albeit from the top down
 - *due to the nature of SVG*
 - *SVG adheres to a top left pattern for rendering shapes*
- to correct this issue
 - *need to calculate the top position of our bars relative to the SVG*
- top of each bar expressed as a relationship
 - *between the height of the SVG and the corresponding data value*

```
.attr("y", function(d) {  
    //height minus data value  
    return h - d;  
});
```

- bar chart will now display correctly from the bottom upwards
- DEMO - Drawing with SVG - barcharts

Image - D3 Barcharts

Testing - D3

[Home](#) | [d3 data drawing bar](#)

Bar chart 1 - no correction



Bar chart 2 - correction



D3 - drawing_barcharts

Data visualisation - D3

Drawing - SVG barchart - part 5

1. add some colour

- adding a colour per bar simply a matter of setting an attribute for the fill colour

```
.attr("fill", "blue");
```

- set many colours using the data itself to determine the colour

```
.attr("fill", function(d) {  
    return "rgb(0, 0, " + (d * 10) + ")";  
});
```

2. add text labels

- also set dynamic text labels per bar, which reflect the current dataset

```
svg.selectAll("text")  
  .data(dataset)  
  .enter()  
  .append("text")
```

- extend this further by positioning our text labels

```
.attr("x", function(d, i) {  
    return i * (w / dataset.length);  
})  
.attr("y", function(d, i) {  
    return h - (d * 4);  
});
```

- then position them relative to the applicable bars, add some styling, colours...

```
.attr("font-family", "sans-serif")  
.attr("font-size", "11px")  
.attr("fill", "white");
```

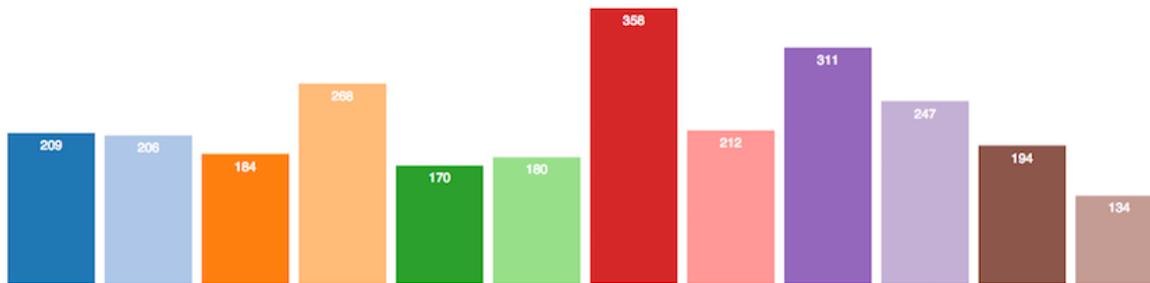
- DEMO - Drawing with SVG - barcharts, colour, and text labels

Image - D3 Barcharts

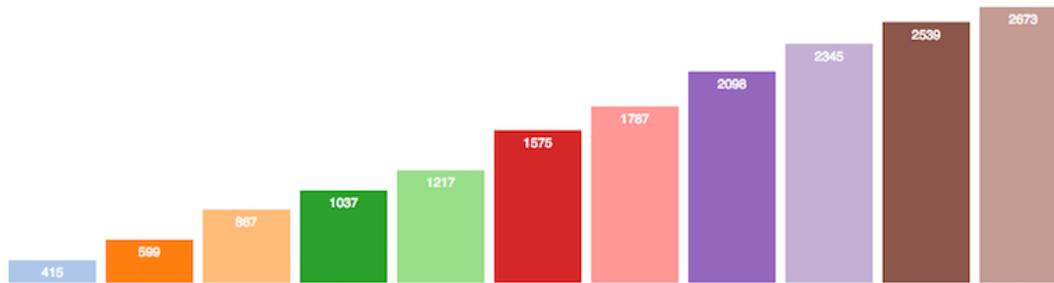
Testing - D3

[Home](#) | [d3 github commits barchart](#)

Total commits per month - calendar



Total commits per month - cumulative



D3 - drawing barcharts with colour and text

Data visualisation - D3

Drawing - add interaction - listeners

- event listeners apply to any DOM element for interaction
 - *from a button to a <p> with the body of a HTML page*

```
<p>this is a HTML paragraph...</p>
```

- add a listener to this DOM element

```
d3.select("p")  
  .on("click", function() {  
    //do something with the element...  
  });
```

- above sample code selects the <p> element
 - *then adds an event listener to that element*
- *event listener* is an anonymous function
 - *listens for .on event for a specific element or group of elements*
- in our example,
 - *on() function takes two arguments*

Data visualisation - D3

Drawing - add interaction - update visuals

- achieved by combining
 - *event listener*
 - *modification of the visuals relative to changes in data*

```
d3.select("p")
  .on("click", function() {

    dataset = [...];

    //update all of the rects
    svg.selectAll("rect")
      .data(dataset)
      .attr("y", function(d) {
        return h - yScale(d);
      });
      .attr("height", function(d) {
        return yScale(d);
      });
  });
```

- above code triggers a change to visuals for each call to the event listener
- eg: change the colours
 - *add call to fill() to update bar colours*

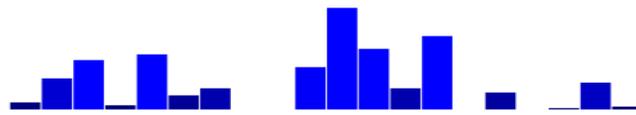
```
.attr("fill", function( d) {
  return "rgb( 0, 0, " + (d * 10) + ")";
});
```

- DEMO - update bar colours

Image - D3 Barcharts



Bar chart 3 - colours



D3 - drawing colour updates for barcharts

Data visualisation - D3

Drawing - add interaction - transitions

- adding a fun transition in D3 is as simple as adding the following,

```
.transition()
```

- add this to above code chain to get a fun and useful transition in the data
- animation reflects the change from the old to the new data
- add a call to the `duration()` function
 - *allows us to specify a time delay for the transition*
 - *quick, slow...we can specify each based upon time*
- chain the `duration()` function after `transition()`

```
.transition().duration(1000)
```

- if we want to specify a constant easing to the transition
 - *use `ease()` with a `Linear` parameter*

```
.ease(linear)
```

- other built-in options, including
 - *circle - gradual ease in and acceleration until elements snap into place*
 - *elastic - best described as springy*
 - *bounce - like a ball bouncing, and then coming to rest..*

Data visualisation - D3

Drawing - add interaction - transitions

- add a delay using the `delay()` function

```
.transition()  
.delay(1000)  
.duration(2000)
```

- also set the `delay()` function dynamically relative to the data,

```
.transition()  
.delay( function( d, i) {  
return i * 100;  
})  
.duration( 500)
```

- when passed an anonymous function
 - *datum bound to the current element is passed into d*
 - *index position of that element is passed into i*
- in the above code example, as D3 loops through each element
 - *delay for each element is set to $i * 100$*
 - *meaning each subsequent element will be delayed 100ms more than preceding element*
- DEMO - transitions - interactive sort

Data visualisation - D3

Drawing - add interaction - adding values and elements

- select all of the bars in our chart
 - *we can rebind the new data to those bars*
 - *and grab the new update as well*

```
var bars = svg.selectAll("rect")  
    .data(dataset);
```

- if more new elements, bars in our example, than original length
 - *use enter() to create references to those new elements that do not yet exist*
- with these reserved elements
 - *we can use append() to add those new elements to the DOM*
 - *now updates our bar chart as well*
- now made the new rect elements
 - *need to update all visual attributes for our rects*
 - *set x, and y position relative to new dataset length*
 - *set width and height based upon new xScale and yScale*
 - *calculated from new dataset length*

Data visualisation - D3

Drawing - add interaction - removing values and elements

- more DOM elements than provided data values
 - *D3's exit selection contains references to those elements without specified data*
 - *exit selection is simply accessed using the `exit()` function*
- grab the *exit* selection
- then transition exiting elements off the screen
 - *for example to the right*
- then finally remove it

```
bars.exit()  
  .transition()  
  .duration(500)  
  .attr("x", w)  
  .remove();
```

- `remove()` is a special transition method that awaits until transition is complete
- then deletes element from DOM forever
 - *to get it back, we'd need to rebuild it again*

Data visualisation - D3

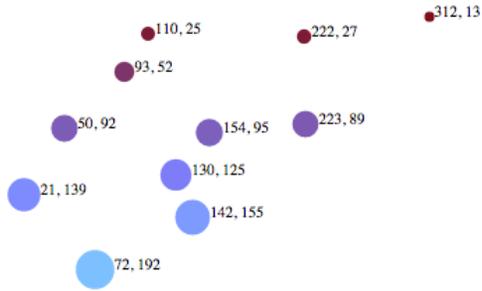
Drawing - SVG scatterplot - intro

- scatterplot allows us to visualise two sets of values on two different axes
 - *one set of data against another*
- plot one set of data on *x* axis, and the other on the *y* axis
- often create dimensions from our data
 - *helps us define patterns within our dataset*
 - *eg: date against age, or age against fitness...*
- dimensions will also be represented relative to *x* and *y* axes
- create our scatterplot using SVG
 - *add our SVG to a selected element*

Image - D3 Scatterplot

Testing - D3

[Home](#) | [d3 data drawing scatter](#)



[D3 - drawing a basic scatterplot](#)

Data visualisation - D3

Drawing - SVG scatterplot - data

- data for the scatterplot is normally stored as a multi-dimensional representation
 - *comparison x and y points*
- eg: we could store this data in a multi-dimensional array

```
var dataset = [  
  [10, 22], [33, 8], [76, 39], [4, 15]  
];
```

- in such a multi-dimensional array
 - *inner array stores the comparison data points for our scatterplot*
 - *each inner array stores x and y points for scatterplot diagram*
- we can also store such data in many different structures
 - *eg: JSON...*

Data visualisation - D3

Drawing - SVG scatterplot - create SVG

- need to create an element for our SVG
 - *allows us to draw and output all of our required shapes*

```
d3.select("body").append("svg");
```

- appends to the body an SVG element
 - *useful to encapsulate this new DOM element within a variable*

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

- variable effectively works as a reference
 - *points to the newly created SVG object*
 - *allows us to use this reference to access element in the DOM*

Data visualisation - D3

Drawing - SVG scatterplot - build scatterplot

- as with our barchart, we can set the width and height for our scatterplot,

```
//width & height  
var w = 750;  
var h = 200;
```

- we will need to create circles for use with scatterplot instead of rectangles

```
svg.selectAll('circle')  
  .data(dataset)  
  .enter()  
  .append('circle');
```

- corresponding to drawing circles
 - *set cx, the x position value of the centre of the circle*
 - *set cy, the y position value of the centre of the circle*
 - *set r, the radius of the circle*

Data visualisation - D3

Drawing - SVG scatterplot - adding circles

- draw circles for scatterplot

```
.attr('cx', function(d) {  
    return d[0]; //get first index value for inner array  
})  
.attr('cy', function(d) {  
    return d[1]; //get second index value for inner array  
})  
.attr('r', 5);
```

- outputs simple circle for each inner array within our supplied multi-dimensional dataset
- start to work with creating circle sizes relative to data quantities
- set a dynamic size for each circle
 - *representative of the data itself*
 - *modify the circle's area to correspond to its y value*
- as we create SVG circles, we cannot directly set the area
 - *so we need to calculate the radius r*
 - *then modify that for each circle*

Data visualisation - D3

Drawing - SVG scatterplot - calculate dynamic area

- assuming that `d[1]` is the original area value of our circles
 - *get the square root and set the radius for each circle*
- instead of setting each circle's radius as a static value
 - *now use the following*

```
.attr('r', function(d) {  
  return Math.sqrt(d[1]);  
});
```

- use the JavaScript `Math.sqrt()` function to help us with this calculation

Data visualisation - D3

Drawing - SVG scatterplot - add colour

- as with a barchart
- also set a dynamic colour relative to a circle's data

```
.attr('fill', function (d) {  
  return 'rgb(125,' + (d[1]) + ', ' + (d[1] * 2) + ')';  
});
```

Data visualisation - D3

Drawing - SVG scatterplot - add labels

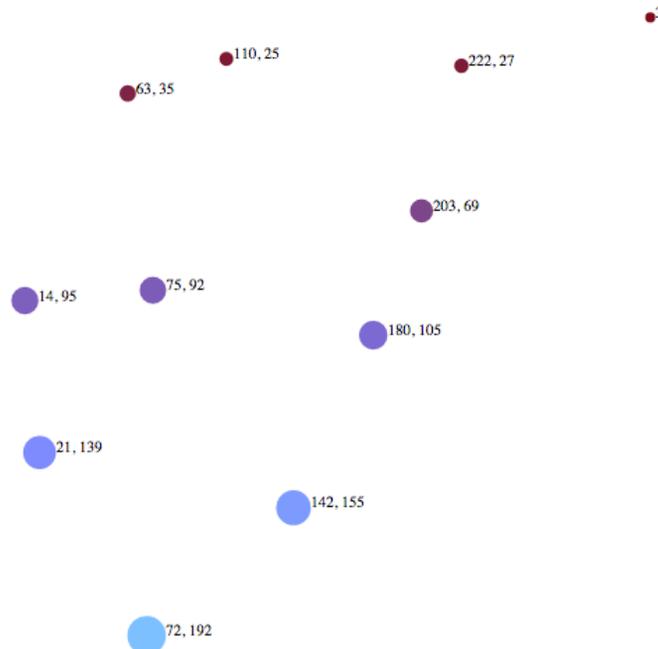
```
//add labels for each circle
svg.selectAll('text')
  .data(dataset)
  .enter()
  .append('text')
  .text(function(d) {
    return d[0] + ', ' + d[1]; //set each data point on the text label
  })
  .attr('x', function(d) {
    return d[0];
  })
  .attr('y', function(d) {
    return d[1];
  })
  .attr('font-family', 'serif')
  .attr('font-size', '12px')
  .attr('fill', 'navy');
```

- start by adding text labels for our data
 - *adding new text elements where they do not already exist*
- then set the text label itself for each circle
 - *using the data values stored in each inner array*
- make the label easier to read
 - *set x and y coordinates relative to data points for each circle*
- set some styles for the labels

Image - D3 Scatterplot

Testing - D3

[Home](#) | [d3 data drawing scales](#)



[D3 - drawing a basic scatterplot 2](#)

Data visualisation - D3

Drawing - SVG - scales

- in D3, *scales* are defined as follows,

“Scales are functions that map from an input domain to an output range”

Bostock, M.

- you can specify your own scale for the required dataset
 - *eg: to avoid massive data values that do not translate correctly to a visualisation*
 - *scale these values to look better within you graphic*
- to achieve this result, you simply use the following pattern.
 - *define the parameters for the scale function*
 - *call the scale function*
 - pass a data value to the function
 - *the scale function returns a scaled output value for rendering*
- also define and use as many scale functions as necessary for your visualisation
- important to realise that a scale has no direct relation to the visual output
 - *it is a mathematical relationship*
- need to consider scales and axes
 - *two separate, different concepts relative to visualisations*

Data visualisation - D3

Drawing - SVG - domains and ranges

- *input domain* for a scale is its possible range of input data values
 - *in effect, initial data values stored in your original dataset*
- *output range* is the possible range of output values
 - *normally use as the pixel representation of the data values*
 - *a personal consideration of the designer*
- normally set a minimum and maximum *output range* for our scaled data
- scale function then calculates the scaled output
 - *based upon original data and defined range for scaled output*
- many different types of scale available for use in D3
- three primary types
 - *quantitative*
 - *ordinal*
 - *time*
- *quantitative* scale types also include other built-in scale types
- many methods available for the scale types

Data visualisation - D3

Drawing - SVG - building a scale

- start building our scale in D3
 - use `d3.scale` with our preferred scale type

```
var scale = d3.scale.linear();
```

- to use the scale effectively, we now need to set our input domain

```
scale.domain([10, 350]);
```

- then we set the output range for the scale

```
scale.range([1, 100]);
```

- we can also chain these methods together

```
var scale = d3.scale.linear()  
  .domain([10, 350])  
  .range([1, 100]);
```

Data visualisation - D3

Drawing - SVG - adding dynamic scales

- we could pre-define values for our scale relative to a given dataset
- makes more sense to abstract these values relative to the defined dataset
- we can now use the D3 array functions to help us set these scale values
 - *eg; find highest number in array dataset*

```
d3.max(dataset, function(d) {  
  return d[0];  
});
```

- returns highest value from the supplied array
- getting minimum value in array works in the same manner
 - *with d3.min() being called instead*
- now create a scale function for x and y axes

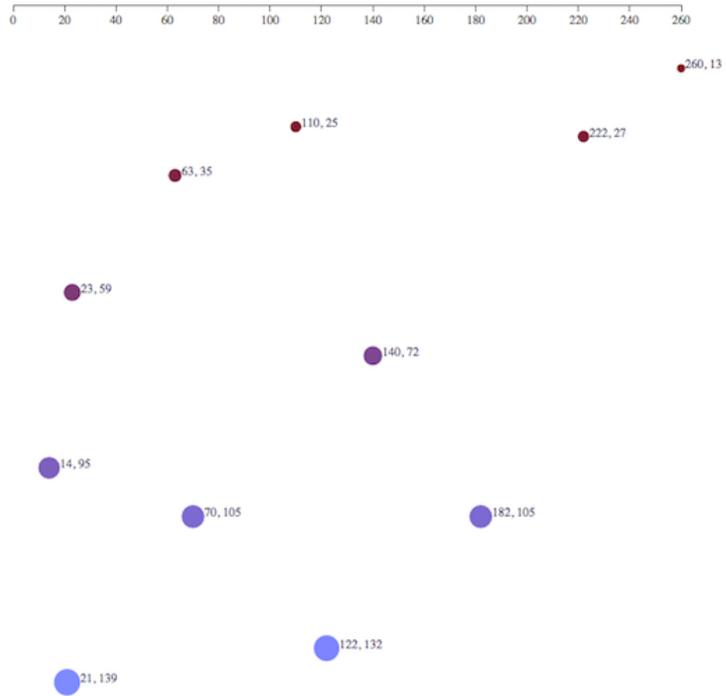
```
var scaleX = d3.scale.linear()  
  .domain([0, d3.max(dataset, function(d) { return d[0]; })])  
  .range([0, w]); //set output range from 0 to width of svg
```

- Y axis scale modifies above code relative to provided data, d[1]
 - *range uses height instead of width*
- for a scatterplot we can use these values to set cx and cy values

Image - D3 Scatterplot

Testing - D3

Home | d3 data drawing axes



D3 - add axis

Data visualisation - D3

Drawing - SVG - adding dynamic scales

- a few data visualisation examples
 - *Tests 1*
 - *Tests 2*

Data Visualisation

general examples

Sample dashboards and visualisations

- gaming dashboard
- schools and education
- students and grades
- D3 examples

Example datasets

- Chicago data portal

Article example

- dashboard designs
- replace jQuery with D3

Data Visualisation

projects examples

A few examples from recent projects,

- GitHub API tests
- check JSON return
- early test examples
- metrics test examples