

Comp 324/424 - Client-side Web Design

Spring Semester 2019 - Week 13

Dr Nick Hayward

Server-side considerations - data storage

MongoDB - intro

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

Server-side considerations - data storage

MongoDB - document oriented

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

Server-side considerations - data storage

MongoDB - BSON

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

Server-side considerations - data storage

MongoDB - general hierarchy of data

- in general, MongoDB has a three tiered data hierarchy

1. database

- *normally one database per app*
- *possible to have multiple per server*
- *same basic role as DB in SQL*

2. collection

- *a grouping of similar pieces of data*
- *documents in a collection*
- *name is usually a noun*
- *resembles in concept a table in SQL*
- *documents do not require the same schema*

3. document

- *a single item in the database*
- *data structure of field and value pairs*
- *similar to objects in JSON*
- *eg: an individual user record*

Server-side considerations - data storage

MongoDB - install and setup

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

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

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

Server-side considerations - data storage

MongoDB - a few shell commands

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

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

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

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

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

Server-side considerations - data storage

MongoDB - a few shell commands

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

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

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

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


Server-side considerations - data storage

MongoDB - test app

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

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

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

```
node server.js
```

Server-side considerations - data storage

MongoDB - Mongoose schema

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

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

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

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

- now start creating objects of this model type using JavaScript

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

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

Server-side considerations - data storage

MongoDB - test app

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

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

Server-side considerations - data storage

MongoDB - test app

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

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

Server-side considerations - data storage

MongoDB - test app

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

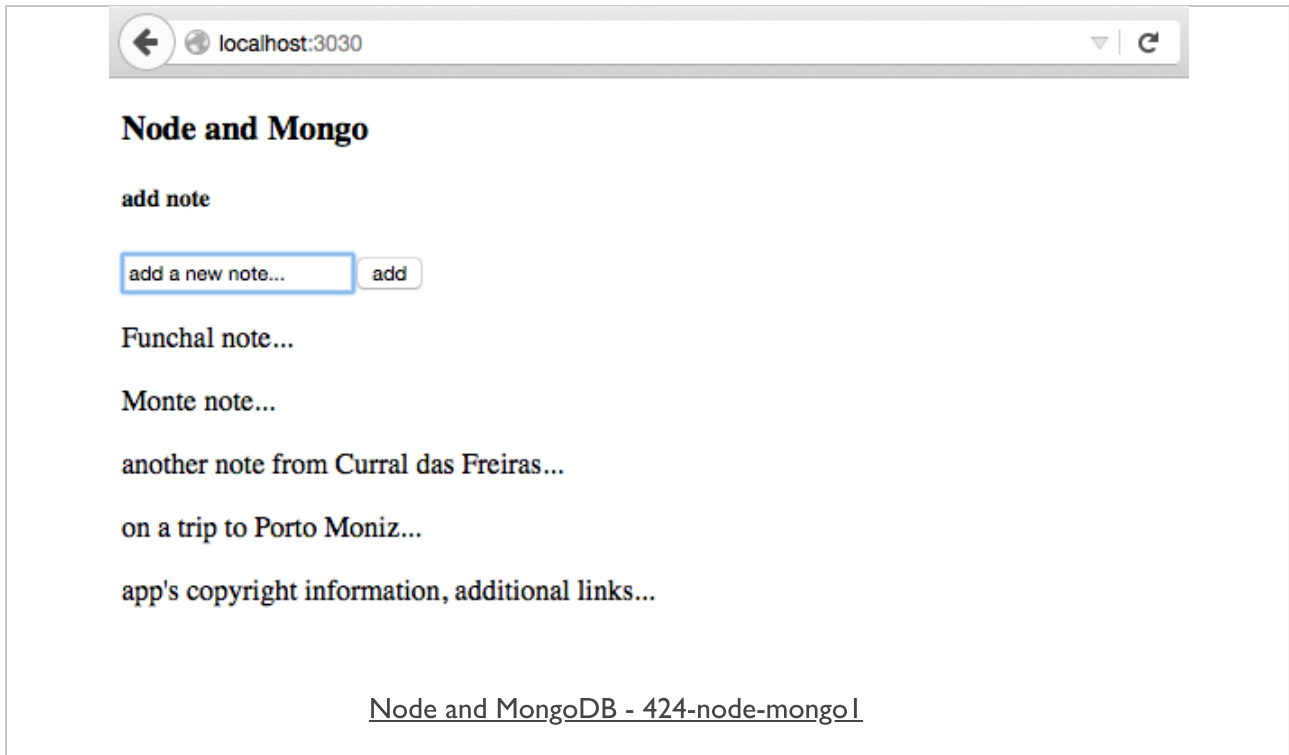
```
//json get route - update for mongo
jsonApp.get("/notes.json", function(req, res) {
  Note.find({}, function (error, notes) {
    //add some error checking...
    res.json(notes);
  });
});
```

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

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

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

Image - Client-side and server-side computing



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

extra notes

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

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 Firestore
 - *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>

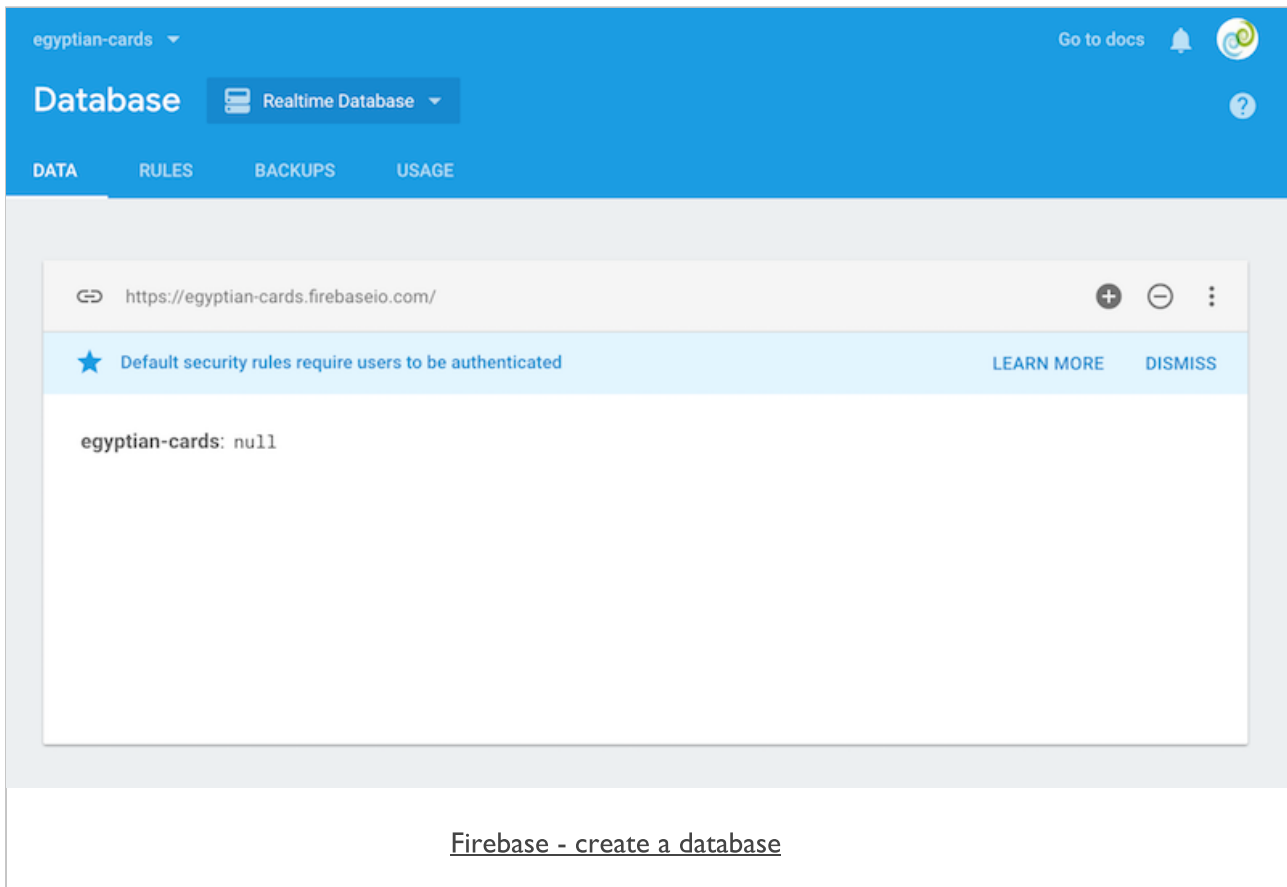
Client-side - Data - Firebase

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



Client-side - Data - Firebase

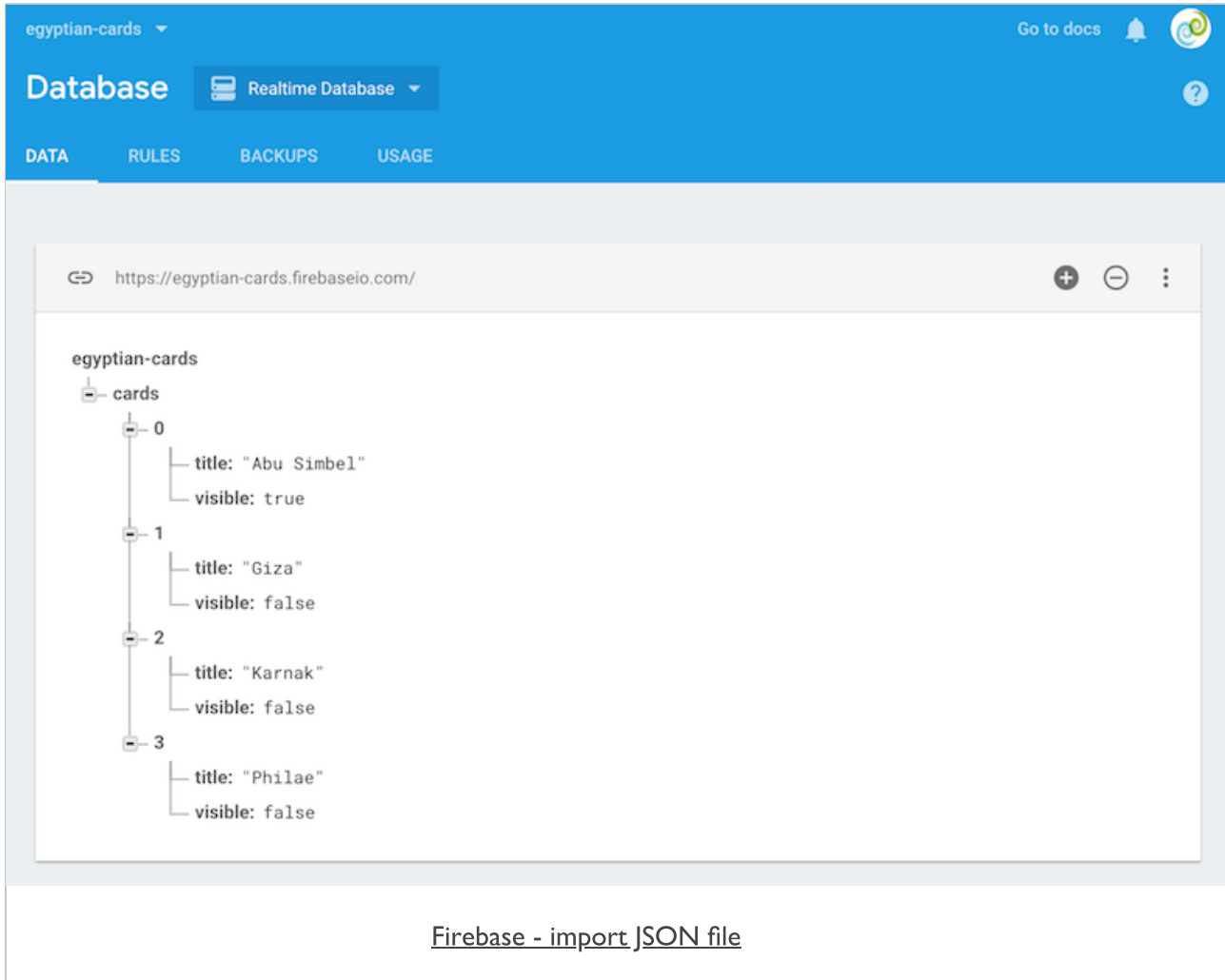
Firestore - import JSON data

- 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 console for a project named 'egyptian-cards'. The interface shows the 'Database' tab with a 'Realtime Database' dropdown. The 'DATA' tab is selected, showing a tree view of the database structure. The root node is 'egyptian-cards', which contains a child node 'cards'. The 'cards' node is expanded, revealing four indexed entries (0, 1, 2, 3) with the following data:

- Index 0: title: "Abu Simbel", visible: true
- Index 1: title: "Giza", visible: false
- Index 2: title: "Karnak", visible: false
- Index 3: title: "Philae", visible: false

Below the console view, the text 'Firebase - import JSON file' is displayed.

Client-side - Data - Firebase

Firestore - permissions

- initial notification in Firestore 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



The screenshot displays the Firebase console interface for a project named "egyptian-cards". The URL bar shows "https://egyptian-cards.firebaseio.com/". The main content area shows a hierarchical JSON structure representing a database of Egyptian cards. The structure is as follows:

- egyptian-cards
 - egypt
 - ancient_sites
 - abu_simbel
 - coords
 - lat: 22.336823
 - long: 31.625532
 - date
 - end
 - precision: "approximate"
 - type: "bc"
 - year: 1244
 - start
 - precision: "approximate"
 - type: "bc"
 - year: 1264
 - kingdom: "upper"
 - location: "aswan governorate"
 - title: "abu simbel"
 - karnak
 - coords
 - lat: 25.719595
 - long: 32.655807
 - date
 - end
 - precision: "approximate"
 - type: "ad"
 - year: 100
 - start
 - precision: "approximate"
 - type: "bc"
 - year: 2055
 - kingdom: "upper"
 - location: "luxor governorate"
 - title: "karnak"
 - code: "eg"

Below the console view, there is a text label: 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
 - *Firebase 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 Firebase - 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

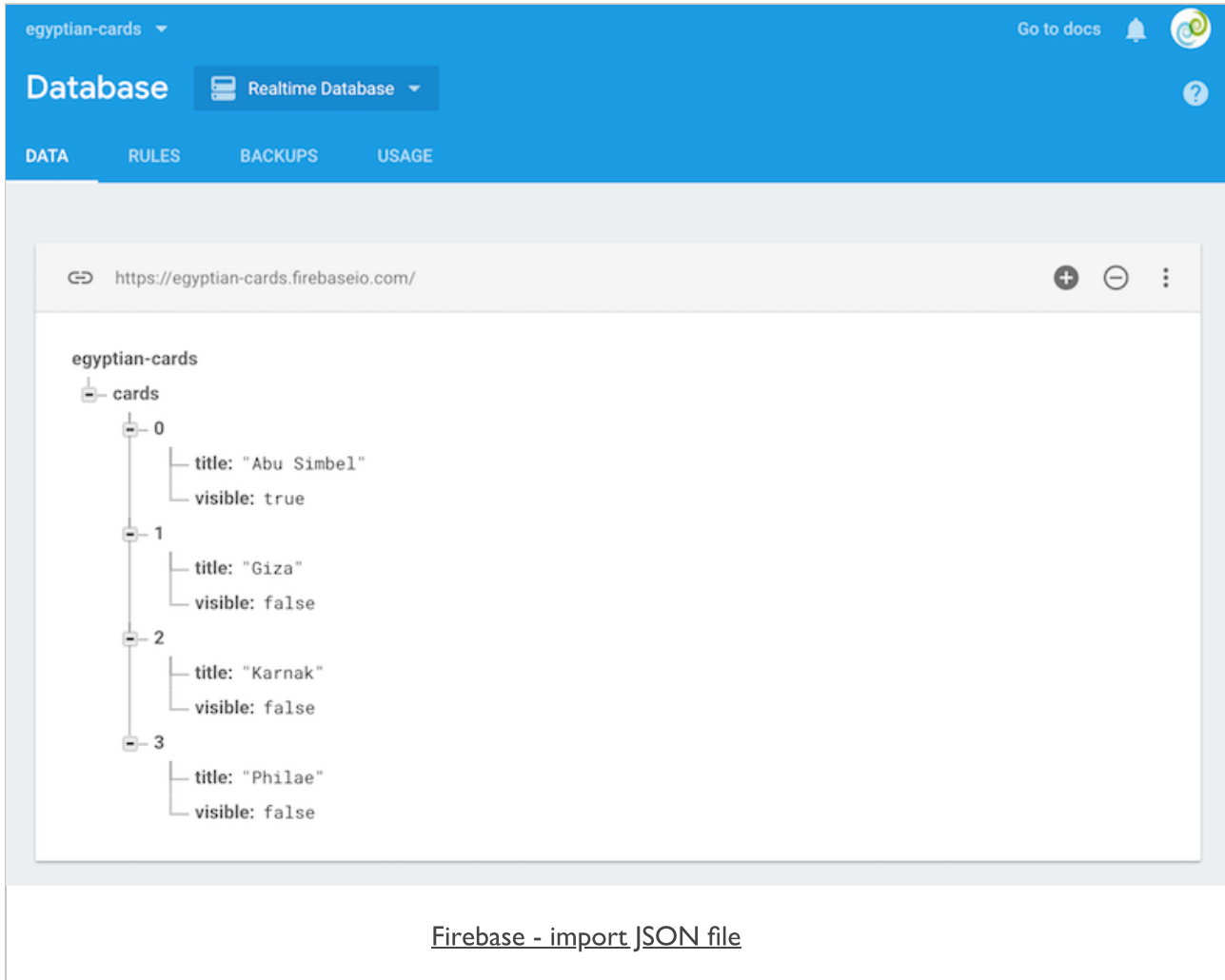
working with arrays

- 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



The screenshot displays the Firebase Realtime Database console for a project named 'egyptian-cards'. The interface shows the 'Database' tab with a 'Realtime Database' dropdown. The 'DATA' tab is selected, showing a tree view of the database structure. The root node is 'egyptian-cards', which contains a child node 'cards'. The 'cards' node is an array with four elements, indexed 0 through 3. Each element is an object with 'title' and 'visible' properties.

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

Below the console view, the text 'Firebase - import JSON file' is displayed.

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
 - *Firebase 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)
```

Firebase - local array.

- we now have a local array from the Firebase 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

Demos

- MongoDB
 - *424-node-mongo /*

Resources

- MongoDB
 - *MongoDB - For Giant Ideas*
 - *MongoDB - Getting Started (Node.js driver edition)*
 - *MongoDB - Getting Started (shell edition)*
- Mongoose
 - *MongooseJS Docs*
- Node.js
 - *Node.js home*
 - *Node.js - download*
 - *ExpressJS*
 - *ExpressJS body-parser*