Notes - Data Stores - Firebase

- Dr Nick Hayward

A general intro and outline for using Firebase with JavaScript based client-side and mobile apps.

Contents

- intro
- what is Firebase?
  - authentication
  - cloud storage
  - real-time database
- Firebase - basic setup
- Firebase - create real-time database
- Firebase - import JSON data
- Firebase - permissions
- Add Firebase to JS based app
  - add to `index.html` - initial usage
- JS based app - initial usage for web and cross-platform
  - define DB connection
  - `ref()` method
- Write data
  - write data - set all data
  - write data - set data for a specific location
- Promises with Firebase
- Remove data
  - remove data - specify location
  - remove data - all data
  - remove data - `set()` with null
- Update data
  - update data - existing properties
  - update data - add new properties
  - update data - remove properties
  - update data - multiple properties at different locations
  - update data - Promise chain
- Read data
  - read data - all data, once
  - read data - single data, once
  - read data - listener for changes - subscribe
    - listener - why not use a Promise?
  - read data - listener for changes - unsubscribe
- Working with arrays
  - JSON import - JS array issues
  - Firebase `push()` method
  - work with arrays - Firebase `snapshot` methods
  - create array from Firebase data
- Add listeners for Firebase value changes
  - listener events for datastore updates
  - `child_removed` event
  - `child_changed` event
  - `child_added` event

Intro At its heart, Firebase offers a hosted NoSQL database. Particularly useful to us, this data store is JSON-based, offering quick, easy development from webview or UI logic to data store. It syncs our app's data across multiple connected devices in a matter of milliseconds, and is available for offline usage as well.

In effect, it provides an API for accessing and querying these JSON data stores in real-time for all connected users.

However, Firebase as a hosted option is more than just data stores and real-time API access. It has now grown, largely over the last year or so, to include additional features for analytics, cloud-based messaging, app authentication, file storage, test options for Android, notifications, adverts, and much more…

Due to its scope and options, Firebase has become an increasingly popular option for both web and mobile apps.

Many of Firebase's patterns and async query structures will also work with other data store options. Usually, it will only take few changes here or there to port your components.

What is Firebase? So, what is Firebase?

It's a hosted platform, acquired by Google, which provides various options for data starage, data access, and real-time database querying. For example, we often consider the *Authentication*, *Cloud Storage*, and *Real-time Database* for many JavaScript based apps, both web and mobile, we now build.

authentication **Authentication** with Firebase provides various backend services and SDKs to help developers manage authentication for an app. This service supports many different providers, including Facebook, Google, Twitter &c. using the industry standard **OAuth 2.0** and **OpenID Connect** protocols.

cloud storage **Cloud Storage** is customarily used for uploading, storing, and then downloading files for apps and their users. It also features a useful safety check if and when a user's connection is broken or lost. In effect, it monitors the connection and progress to help save bandwidth and time. Files are usually stored in a *Google Cloud Storage* bucket, which means that they should be accessible using either Firebase or Google Cloud. So, we may then consider using the Google Cloud platform for image filtering, processing, or perhaps even video encoding &c. Such modified files may then become available to Firebase again, and any connected apps.

For example, Google's Cloud Platform

real-time database The **Real-time Database** offers a hosted NoSQL data store with the ability to quickly and easily sync data. This data synchronisation is active across multiple devices, in real-time, as and when the data is updated in the cloud database.

Besides these banner services, Firebase also offers extras such as analytics, advertising services such as adwords, crash reporting, notifications, and various testing options.

All of these may be monitored and maintained from the Firebase console with a registered account.

Firebase - basic setup We can start using Firebase by creating an account with the service using a standard Google account,

- Firebase

After logging in to Firebase, you may then either view the *Get Started* material or simply navigate to the Firebase console. This is where we can monitor and set configurations for Firebase.

At the Console page, we can then get started by creating a new project. Simply click on the option to `Add project`, and enter the name of this new project, and select a region.

You'll then be redirected to the console dashboard page for the new project. This is primarily where we access the various options for each project, and generally control and maintain our project's structure and setup.

Documentation for the Firebase Real-Time database is as follows,

- https://firebase.google.com/docs/reference/js/firebase.database

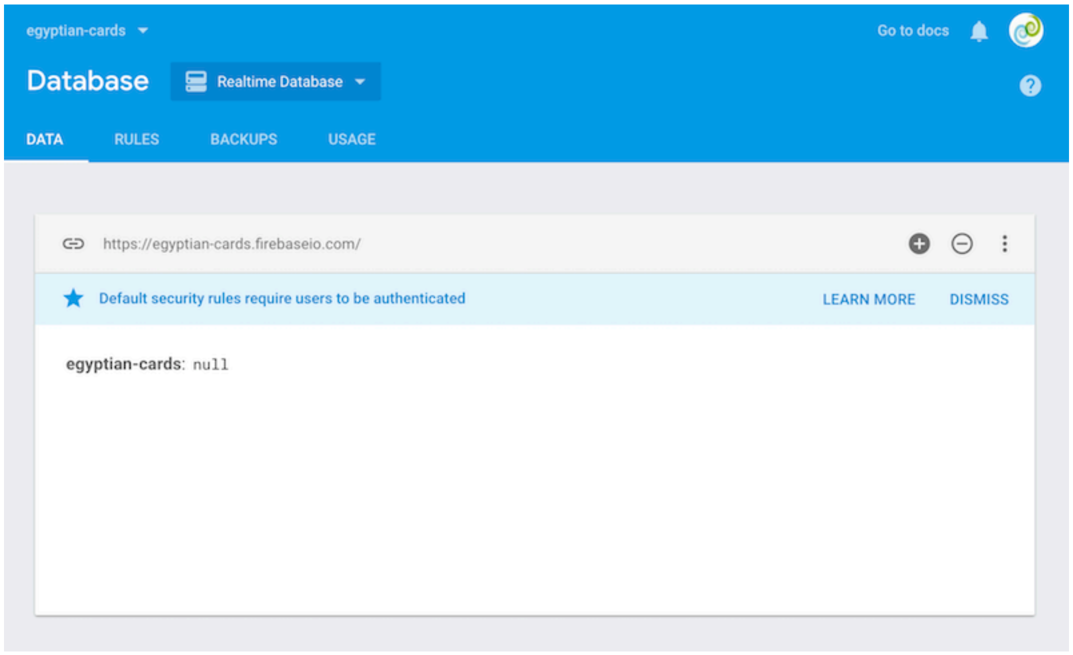
Firebase - create real-time database We can now setup a database with Firebase for a test JavaScript based app, including web and cross-platform mobile.

Start by selecting the *Database* option from the left sidebar on the Console Dashboard. It's available under the *DEVELOP* option. Then, simply select *Get Started* for the real-time database.

This will then present an empty database with an appropriate name to match the current project. This name is, as you might expect, derived from the project name and ID.

Data will be stored in a useful JSON format in the real-time database on Firebase, so it should be familiar for most developers. We also see similar patterns with other NoSQL data stores such as MongoDB.

In general, working with Firebase is simple, and for most examples it's also straightforward to use. It's also possible to get started quickly direct from the Firebase console, or simply import some existing JSON for your project's data.


egyptian-cards
Go to docs
Database
Realtime Database
DATA
RULES
BACKUPS
USAGE
https://egyptian-cards.firebaseio.com/
Default security rules require users to be authenticated
LEARN MORE
DISMISS

```
egyptian-cards: null
```

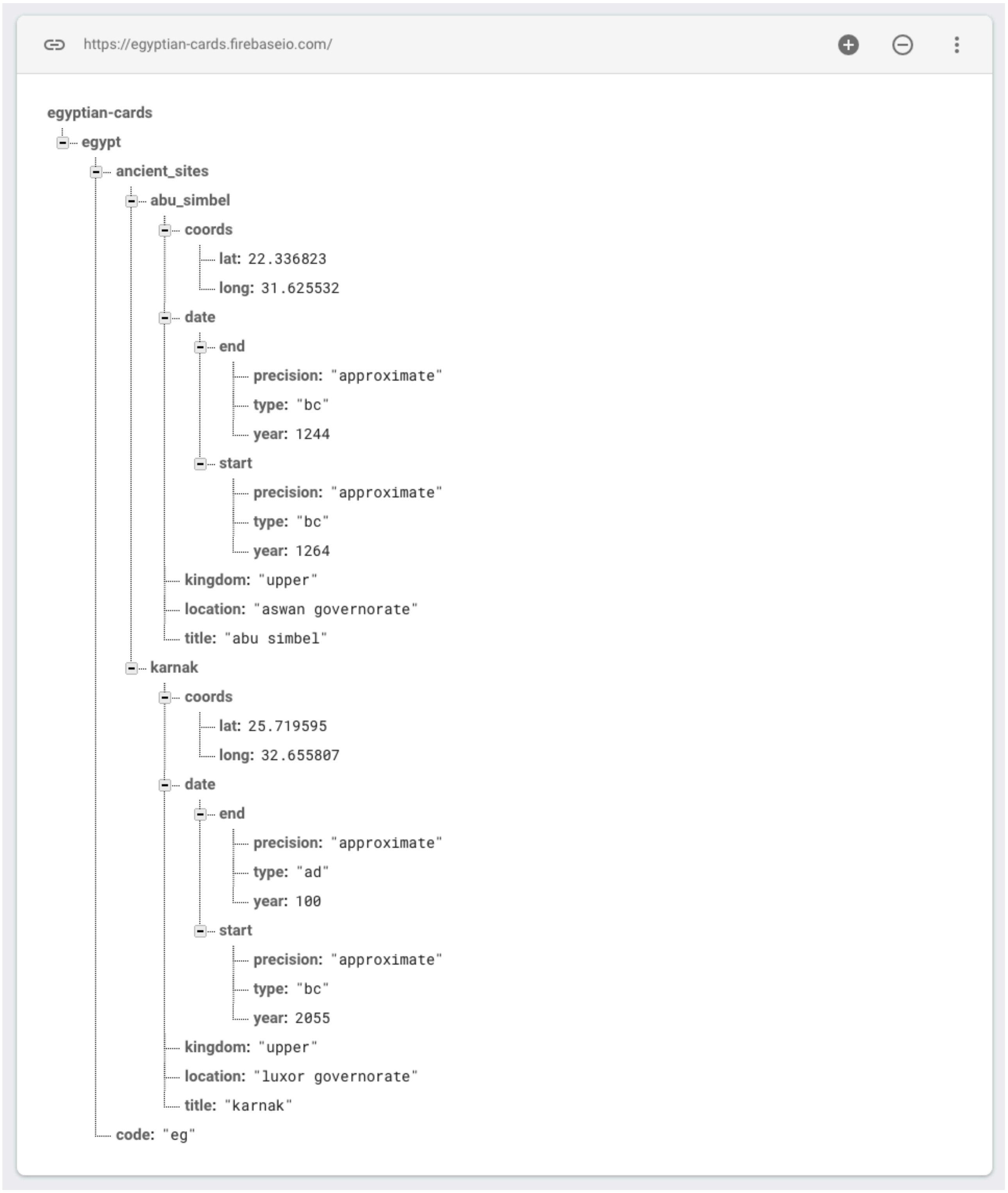
Figure 1: Firebase - create a database

Firebase - import JSON data So, we might start with some simple data to help test Firebase with a JavaScript based app. We can import the following into our test database, and then query the data &c. from the app.

For example,

```json
{
  "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
          }
        }
      }
    }
  }
}
```

In Firebase, this JSON will be stored as shown in the following image.



```
https://egyptian-cards.firebaseio.com/

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"
```

Figure 2: Firebase - import JSON file

Firebase - permissions One of the things you'll notice when you initially create a database with Firebase is a persistent notification concerning *Default security rules require users to be authenticated.* We have the option to either *LEARN MORE* or *DISMISS* this notification.

However, it's worth considering, for a moment, how permissions are managed with Firebase.

We can manage database access for our users as they login and authenticate with the Firebase service. There are many options available for permissions, e.g.

- Firebase - database rules

but a few we may consider to get us started. For example, if we click the *RULES* tab we can modify our rules to permit access to the data without authentication. This will, of course, help us initially test our app prior to adding any required authentication.

So, we modify the rules as follows.

From,

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

to

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

Add Firebase to JS based app We can now test our new Firebase database with a JavaScript based app.

We need to start by getting some useful information from Firebase, in particular by selecting the option from the console to *Add to app.*

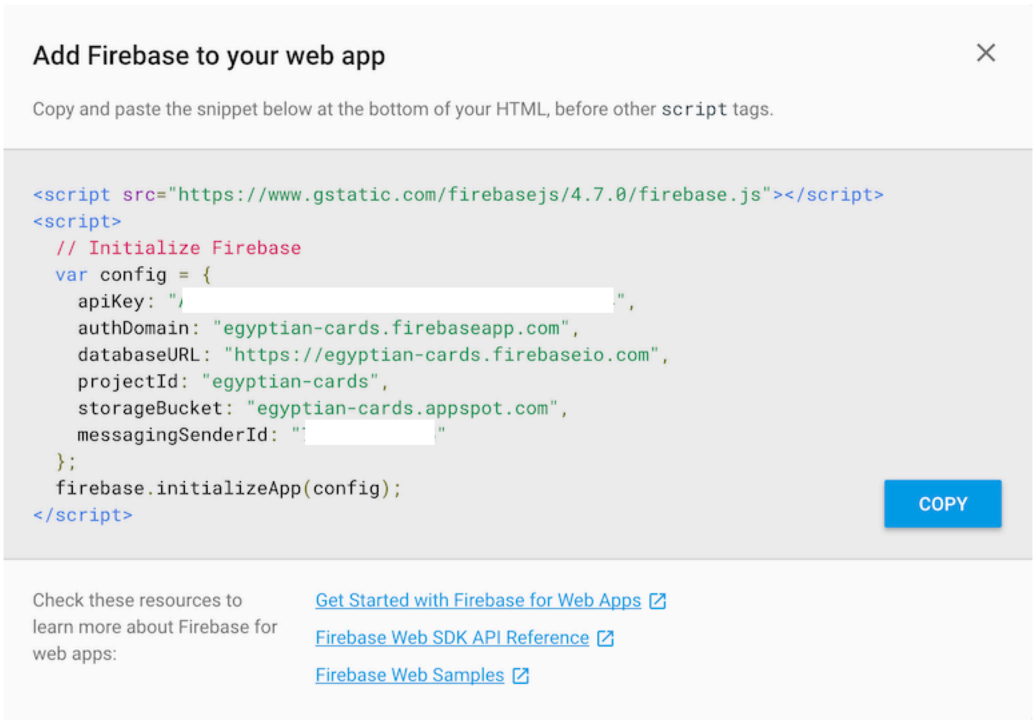
Select the *Project Overview* link in the left sidebar, and then click on the icon for *Add app.*

This will show options for *Android* and *iOS* native, plus JavaScript.

We're not limited to using Firebase with just a JS based web app, but we can take advantage of the provided JavaScript SDK with other options such as Apache Cordova, React Native, and web development &c.

The Firebase console will show us a modal with initialisation settings for adding Firebase usage to our app.

For example, we'll get the following sample config for our test database.


Add Firebase to your web app

Copy and paste the snippet below at the bottom of your HTML, before other `script` tags.

```
<script src="https://www.gstatic.com/firebasejs/4.7.0/firebase.js"></script>
<script>
  // Initialize Firebase
  var config = {
    apiKey: "[redacted]",
    authDomain: "egyptian-cards.firebaseapp.com",
    databaseURL: "https://egyptian-cards.firebaseio.com",
    projectId: "egyptian-cards",
    storageBucket: "egyptian-cards.appspot.com",
    messagingSenderId: "[redacted]"
  };
  firebase.initializeApp(config);
</script>
```

Check these resources to learn more about Firebase for web apps:

- Get Started with Firebase for Web Apps
- Firebase Web SDK API Reference
- Firebase Web Samples

Figure 3: Firebase - initialisation config settings

We can start by copying these config values for use with our app.

add to `index.html` - initial usage We'll start by testing this setup with the default config in our `index.html` file.

e.g.

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

This 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…

However, it's also possible to customise required components per app. This allows us to include only the features required for each app.

For example, 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>
```

Then, 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

```
<!-- 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, as noted in the above initial example, we define an object for the `config` of the required services and options,

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

JS based app - initial usage for web and cross-platform After defining the required config and initialisation, we can then start to add the required listeners and calls to a JS-based web app or cross-platform mobile app.

define DB connection We can establish a connection to our Firebase DB as follows,

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

We may then use this reference to connect and query our database.

`ref()` method Once we have the connection to the database, we may then call the `ref()` , or reference, method.

We 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. In effect, we'll be reading, writing &c. relative to the whole database.

We may also request a specific reference in the database by passing a location path, e.g.

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

This example will request a reference to the `title` property in the specified object relative to the datastore root. We may then use this reference, for example, to `set` a new value.

As we find with other databases, such as tables in SQL or collections in MongoDB &c., this allows us to create multiple parts of the Firebase database. Such parts might include multiple objects, properties, and values &c.

This becomes a quick and easy option for organising and distributing data within the database.

Write data We may 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
- ...

In effect, any values and data types we may customarily add to a JSON object. However, it's worth noting that Firebase may not maintain the native structure upon import. Arrays, for example, will be converted to plain JavaScript objects in Firebase.

write data - set all data We can `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'
  }
});
```

So, this example will now set the data for the whole database to the object passed as the argument to the `set()` method.

However, the data passed does not have to be an *object*. We may pass any of the above supported data types.

n.b. when we call `set()` relative to just the root, the existing data will be overwritten. In effect, the current database is deleted, and the passed object becomes the new database.

write data - set data for a specific data location As with reading specific data, we may also write data to a specific location in the database.

Again, we add an argument to the `ref()` method specifying the 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*, thereby allowing us to update anything from the whole database to a single property value.

Promises with Firebase Firebase includes native support for Promises and associated chains.

This means we do not need to create our own custom Promises. Instead, we may work with a return Promise object from Firebase using a standard chain.

For example, when we call the `set()` method, Firebase will return a Promise object for the method execution.

The `set()` method will not explicitly return anything except for success or error. This means 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) => {
    // catcg error from Firebase - error logged to console
    console.log('error returned', e);
  });
```

Remove data As expected, where we can add new data to the database, we may also delete and remove data from the connected database.

remove data - specify location As with setting data, 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) => {
    // catcg error from Firebase - error logged to console
    console.log('error returned', e);
  });
```

So, we need to get a reference to the specific location in the database, and then execute the `remove()` method for that data. This will delete the specific property and data value.

We may also chain a standard `then()` and `catch()` method to the return Promise object.

remove data - all data We may 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) => {
    // catcg error from Firebase - error logged to console
    console.log('error returned', e);
  });
```

As we can see in this example, we simply need to call the `ref()` method relative to the *root* of the connected database. This will effectively wipe all of the current data.

remove data - `set()` with null Another option specified in the Firebase docs for deleting data is by using the `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) => {
    // catcg error from Firebase - error logged to console
    console.log('error returned', e);
  });
```

(*n.b.* Whilst this option works, it seems a poor use of the `set()` method. It is also a poor use of the `null` data type, which is an object type in JavaScript. This call is, therefore, reliant on Firebase interpreting a `null` value and data type as a remove request to the database.)

We may, however, need to use this pattern with the following `update()` method.

Update data We may also combine setting and removing data in a single pattern by using the `update()` method call to the defined database reference.

The `update()` method is, therefore, meant to be used to update multiple items in the database in a single call. This means we must pass an object as the argument to the `update()` method.

update data - existing properties So, if we wanted to update multiple existing properties we might have the following example,

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

update data - add new properties We may also add a new property for the 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.'
});
```

So, we may now set new values for the two existing properties, `title` and `visible` , and add a new property and value for `data` .

A benefit of the `update()` method is that it will only update the specific properties, and not override everything at the reference location as with the `set()` method.

update data - remove properties We may also combine these updates with the 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.',
});
```

As with passing `null` to the `set()` method, we're effectively deleting the specified property from the reference location in the database.

Therefore, we're now able to combine creating a new property with updating and deleting any existing properties at the defined reference location.

update data - multiple properties at different locations We may also combine updating data in multiple objects at different locations relative to the 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 dabatase, we've now updated multiple `title` properties in different cards.

The property name, e.g. `egypt/ancient_sites/abu_simbel/visible` , is interpreted by Firebase as a location in the database relative to the passed root reference.

n.b. this pattern of update is **only** relative to the current path specified in the argument passed to the `ref()` method. Therefore, only child properties of the current location may be updated using this pattern. This is due to character restrictions on the property name. e.g. the name may not begin with `.` , `/` &c.

update data - Promise chain The `update()` method will also return a Promise object, which 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()` , the Promise object itself will simply return success or error for the method call.

Read data We can fetch data from the connected database in many different ways, including all of the data or a single specific part of the data.

We may also connect and retrieve data once, or setup a listener for polling the database for live updates.

read data - all data, once e.g. to 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);
  });
```

So, we can now return all of the data stored in the database at the time of this query. i.e. a snapshot of the current state of the database.

read data - single data, once Likewise, we may query the database once for a single specific value.

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

So, this will return the value for the card at location `egypt/ancient_sites/abu_simbel/` .

read data - listener for changes - subscribe We may also setup listeners for changes to the connected database, which will 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());
});
```

By using the `on()` method, we're able to setup a listener that polls the DB for any changes in `value` . As before, we may then get the current snapshot value for the data stored.

If we change any data in the online database, this listener will automatically execute the defined success callback function.

We may also add some initial error handling for the 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);
});
```

listener - why not use a Promise? As the listener is notified of updates to the online database, we need the callback function to be executed. In fact, we may need it to be executed multiple times for many updates to the stored data.

However, a Promise may only be resolved a single time with either `resolve` or `reject` . So, in effect, we would need to instantiate a new Promise for each update. This would not work as expected, which is why we use a standard callback function.

The callback may simply be executed each and every time there is an update to the DB.

read data - listener for changes - unsubscribe We may also need to *unsubscribe* from all or specific changes in the online database. For example, a user may navigate to a different screen or the application is paused in the background.

e.g.

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

This will remove *all* current subscriptions to the defined DB connection.

However, we may also unsubscribe a specific subscription by simply passing the callback we used for the original subscription.

So, we may abstract the callback function and pass it to both the `on()` and `off()` methods for the database `ref()` method.

e.g.

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

We may then simply pass this variable as the callback argument for both subscribe and unsubscribe events.

e.g.

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

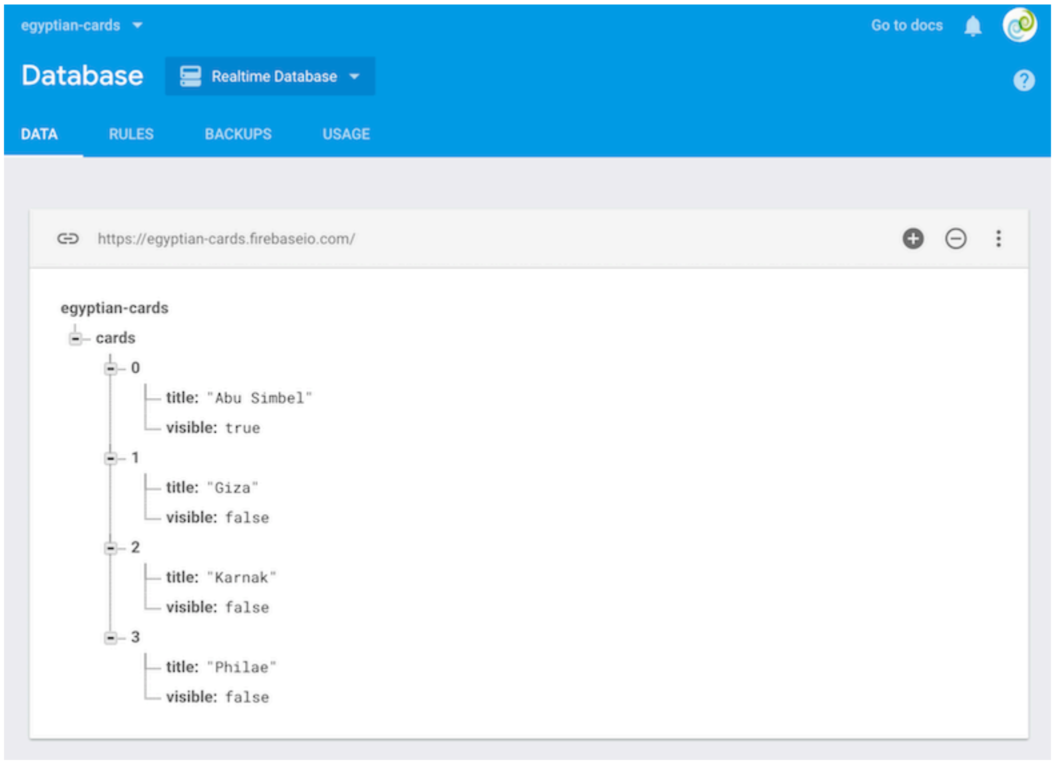
This will allow our app to maintain the DB connection, and unsubscribe a specific subscription.

Working with arrays Firebase does not explicitly support array data structures. Instead, it will convert array objects to plain JavaScript objects. This will negate the built-in iterator for an array object, thereby defining plain objects in the stored JSON data.

JSON import - JS array issues For example, if we import the following JSON

```
{
  "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"
    }
  ]
}
```

In Firebase, this JSON will be stored as follows,



https://egyptian-cards.firebaseio.com/

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

Figure 4: Firebase - import JSON file

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"
  }
}
```

So, we may still access each index value from the original array object but without easy access to pre-defined, known unique references.

For example, to access the title value of a given card, we would need to know its auto-generated property value in Firebase.

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

This reference will be the path to the required object, and we may then access a given property on the object.

So, even if we add a unique reference property to each card we may not use it without knowing the property value assigned to the card by Firebase.

Firebase `push()` method To add new content to an existing Firebase datastore, we may use the `push()` method to add this data.

As the data is added to the database, a unique property value will be auto-generated.

e.g.

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

This new data will be added to the connected DB, but it will be created with an auto-generated ID for the parent object, e.g.

```
LPcdS31H_u9N0dIn27_
```

This may be useful for dynamic content that is pushed to a datastore, e.g.

- notes, tasks, calendar dates &c.

In effect, we may use this auto-generated unique reference whilst iterating over the existing DB content.

So, as we iterate through the datastore and output the content, we may add a reference to the auto-generated ID for future reference and lookup. This reference may be stored in a local data structure, UI component or element, variable value, &c.

work with arrays - Firebase `snapshot` methods Data snapshot methods in Firebase documentation.

A commonly used method with `snapshot` is the `val()` method, which we've already used many times already.

However, there are many additional methods specified in the API documentation for *DataSnapshot*.

- `forEach()` - iterator for plain objects from Firebase

create array from Firebase data As we store data as plain objects in Firebase, we need to consider how we may work with array-like structures for technologies and patterns that require array data structures, such as Redux.

In effect, we may get the data from Firebase, and then prepare it for use as an array.

For example, to help us work with Firebase object data and arrays, we may call the `forEach()` method on the return `snapshot`. This will provide a required iterator for the plain objects stored in Firebase.

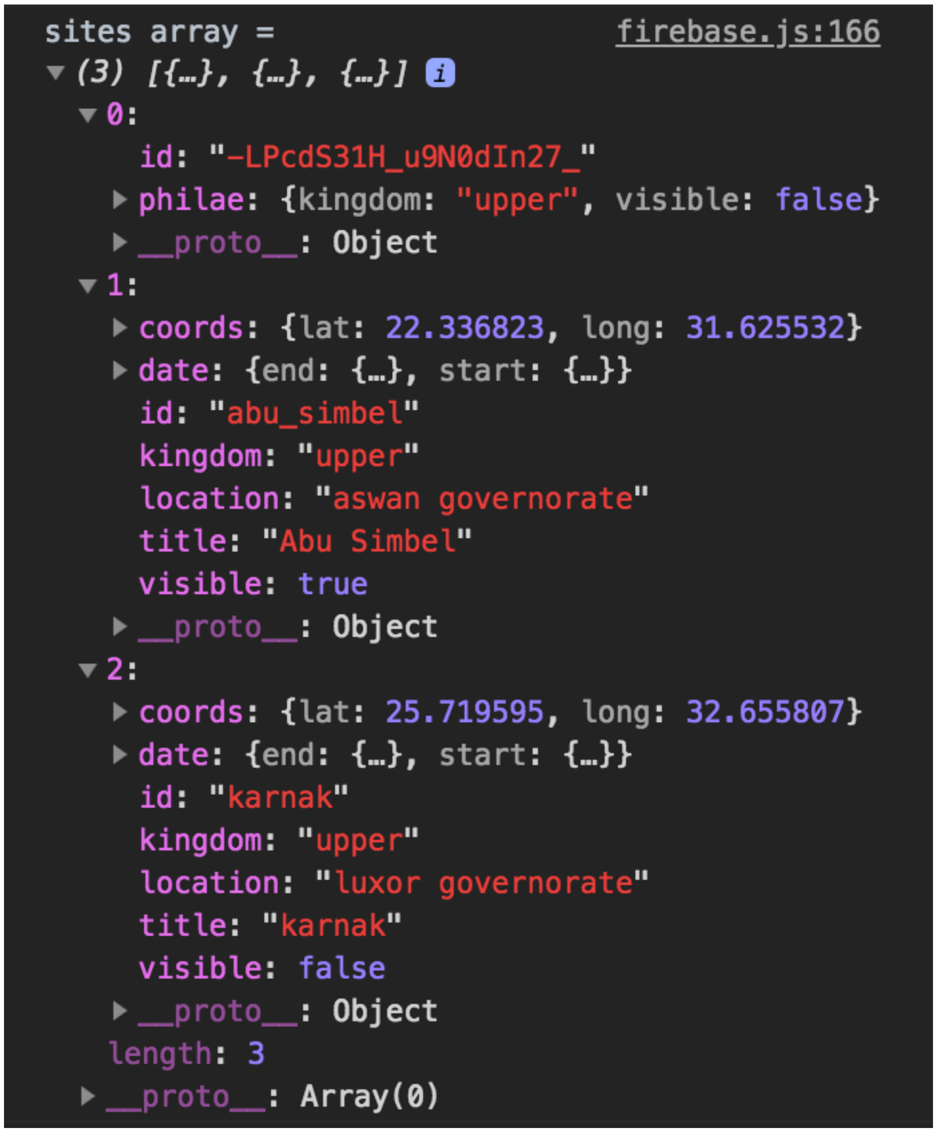
e.g.

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

So, we're able to iterate through the return snapshot value for the connected DB. We may then push selected values to either custom properties or use the existing properties in the DB.

In this example, we set a specific ID value in the array, and then use the spread operator to simply add the existing properties and values from the database to the local array data structure.

For example, we may return the following



```
sites array =                                     firebase.js:166
▼ (3) [{…}, {…}, {…}] i
  ▼ 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)
```

Figure 5: Firebase - snapshot forEach() - creating a local array

So, we now have a local array from the Firebase object data, which we may then use with options such as Redux.

Add listeners for Firebase value changes As we modify objects, properties, values &c. in Firebase, we may set listeners to return notifications for such updates.

So, there should be a notification for a given update relative to the datastore.

e.g. we may add a single listener for any update relative to the 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);
});
```

As noted earlier, the `on()` method does not return a Promise object so we need to define a callback for the return data.

listener events for datastore updates Firebase provides a few different events relative to subscriptions and Firebase updates.

For the `on()` method, we may initially consult the following documentation,

- `on()` events in Firebase docs = https://firebase.google.com/docs/reference/js/firebase.database.Reference#on

Then, we may test various listeners for datastore updates.

`child_removed` event We may add a subscription for event updates as a child object is removed from the data store.

The `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());
});
```

`child_changed` event Likewise, we may 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());
});
```

`child_added` **event** Another common event is adding a new child to the data store. For example, a user may create and add a new note or to-do item.

e.g.

```
// - 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());
});
```

References

- Firebase - https://firebase.google.com/
- Firebase JS Guide - https://firebase.google.com/docs/web/setup
- Firebase JS Reference - https://firebase.google.com/docs/reference/js/
- Firebase JS `on()` events - https://firebase.google.com/docs/reference/js/firebase.database.Reference#on