...Read and Write Data on the Web

(Optional) Prototype and test with Firebase Local Emulator Suite

Before talking about how your app reads from and writes to Realtime Database, let's introduce a set of tools you can use to prototype and test Realtime Database functionality: Firebase Local Emulator Suite. If you're trying out different data models, optimizing your security rules, or working to find the most cost-effective way to interact with the back-end, being able to work locally without deploying live services can be a great idea.

A Realtime Database emulator is part of the Local Emulator Suite, which enables your app to interact with your emulated database content and config, as well as optionally your emulated project resources (functions, other databases, and security rules).

Using the Realtime Database emulator involves just a few steps:

1. Adding a line of code to your app's test config to connect to the emulator.
2. From the root of your local project directory, running `firebase emulators:start`.
3. Making calls from your app's prototype code using a Realtime Database platform SDK as usual, or using the Realtime Database REST API.

A detailed walkthrough involving Realtime Database and Cloud Functions (/docs/emulator-suite/connect_and_prototype?database=RTDB) is available. You should also have a look at the Local Emulator Suite introduction (/docs/emulator-suite).

Get a database reference

To read or write data from the database, you need an instance of `firebase.database.Reference`:

```
var database = firebase.database();
```

Write data

This document covers the basics of retrieving data and how to order and filter Firebase data.

Firebase data is retrieved by attaching an asynchronous listener to a `firebase.database.Reference`. The listener is triggered once for the initial state of the data and again anytime the data changes.

Note: By default, read and write access to your database is restricted so only authenticated users can read or write data. To get started without setting up Authentication (/docs/auth), you can configure your rules for public access (/docs/rules/basics#default_rules_locked_mode). This does make your database open to anyone, even people not using your app, so be sure to restrict your database again when you set up authentication.

Basic write operations

For basic write operations, you can use `set()` to save data to a specified reference, replacing any existing data at that path. For example a social blogging application might add a user with `set()` as follows:

Web modular API | Web namespaced API

```
function writeUserData(userId, name, email, imageUrl) {
  firebase.database().ref('users/' + userId).set({
    username: name,
    email: email,
    profile_picture : imageUrl
  });
}
```

Using `set()` overwrites data at the specified location, including any child nodes.

Read data

Listen for value events

To read data at a path and listen for changes, use `onValue()` to observe events. You can use this event to read static snapshots of the contents at a given path, as they existed at the time of the event. This method is triggered once when the listener is attached and again every time the data, including children, changes. The event callback is passed a snapshot containing all data at that location, including child data. If there is no data, the snapshot will return `false` when you call `exists()` and `null` when you call `val()` on it.

Important: `onValue()` is called every time data is changed at the specified database reference, including changes to children. To limit the size of your snapshots, attach only at the lowest level needed for watching changes. For example, attaching a listener to the root of your database is not recommended.

The following example demonstrates a social blogging application retrieving the star count of a post from the database:

Web modular API | Web namespaced API

```
var starCountRef = firebase.database().ref('posts/' + postId + '/starCount
starCountRef.on('value', (snapshot) => {
  const data = snapshot.val();
  updateStarCount(postElement, data);
});
```

The listener receives a `snapshot` that contains the data at the specified location in the database at the time of the event. You can retrieve the data in the `snapshot` with the `val()` method.

Read data once

Read data once with get()

The SDK is designed to manage interactions with database servers whether your app is online or offline.

Generally, you should use the value event techniques described above to read data to get notified of updates to the data from the backend. The listener techniques reduce your usage and billing, and are optimized to give your users the best experience as they go online and offline.

If you need the data only once, you can use `get()` to get a snapshot of the data from the database. If for any reason `get()` is unable to return the server value, the client will probe the local storage cache and return an error if the value is still not found.

Unnecessary use of `get()` can increase use of bandwidth and lead to loss of performance, which can be prevented by using a realtime listener as shown above.

Web modular API | Web namespaced API

```
const dbRef = firebase.database().ref();
dbRef.child("users").child(userId).get().then((snapshot) => {
  if (snapshot.exists()) {
    console.log(snapshot.val());
  } else {
    console.log("No data available");
  }
}).catch((error) => {
  console.error(error);
});
```

Read data once with an observer

In some cases you may want the value from the local cache to be returned immediately, instead of checking for an updated value on the server. In those cases you can use `once()` to get the data from the local disk cache immediately.

This is useful for data that only needs to be loaded once and isn't expected to change frequently or require active listening. For instance, the blogging app in the previous examples uses this method to load a user's profile when they begin authoring a new post:

Web modular API | Web namespaced API

```
var userId = firebase.auth().currentUser.uid;
return firebase.database().ref('/users/' + userId).once('value').then((snap
  var username = (snapshot.val() && snapshot.val().username) || 'Anonymous
  // ...
});
```

Updating or deleting data

Update specific fields

To simultaneously write to specific children of a node without overwriting other child nodes, use the `update()` method.

() When calling `update()`, you can update lower-level child values by specifying a path for the key. If data is stored in multiple locations to scale better, you can update all instances of that data using data fan-out (/docs/database/web/structure-data#fanout).

For example, a social blogging app might create a post and simultaneously update it to the recent activity feed and the posting user's activity feed using code like this:

Web modular API | Web namespaced API

```
function writeNewPost(uid, username, picture, title, body) {
  // A post entry.
  var postData = {
    author: username,
    uid: uid,
    body: body,
    title: title,
    starCount: 0,
    authorPic: picture
  };

  // Get a key for a new Post.
  var newPostKey = firebase.database().ref().child('posts').push().key;

  // Write the new post's data simultaneously in the posts list and the use
```

```
  var updates = {};
  updates['/posts/' + newPostKey] = postData;
  updates['/user-posts/' + uid + '/' + newPostKey] = postData;

  return firebase.database().ref().update(updates);
}
```

This example uses `push()` to create a post in the node containing posts for all users at `/posts/$postid` and simultaneously retrieve the key. The key can then be used to create a second entry in the user's posts at `/user-posts/$userid/$postid`.

Using these paths, you can perform simultaneous updates to multiple locations in the JSON tree with a single call to `update()`, such as how this example creates the new post in both locations. Simultaneous updates made this way are atomic: either all updates succeed or all updates fail.

Add a Completion Callback

If you want to know when your data has been committed, you can add a completion callback. Both `set()` and `update()` take an optional completion callback that is called when the write has been committed to the database. If the call was unsuccessful, the callback is passed an error object indicating why the failure occurred.

Web modular API | Web namespaced API

```
firebase.database().ref('users/' + userId).set({
  username: name,
  email: email,
  profile_picture : imageUrl
}, (error) => {
  if (error) {
    // The write failed...
  } else {
    // Data saved successfully!
  }
});
```

Delete data

The simplest way to delete data is to call `remove()` on a reference to the location of that data.

You can also delete by specifying `null` as the value for another write operation such as `set()` or `update()`. You can use this technique with `update()` to delete multiple children in a single API call.

Receive a `Promise`

To know when your data is committed to the Firebase Realtime Database server, you can use a `Promise` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise). Both `set()` and `update()` can return a `Promise` you can use to know when the write is committed to the database.

Detach listeners

Callbacks are removed by calling the `off()` method on your Firebase database reference.

You can remove a single listener by passing it as a parameter to `off()`. Calling `off()` on the location with no arguments removes all listeners at that location.

Calling `off()` on a parent listener does not automatically remove listeners registered on its child nodes; `off()` must also be called on any child listeners to remove the callback.

Save data as transactions

When working with data that could be corrupted by concurrent modifications, such as incremental counters, you can use a transaction operation (/docs/reference/js/database#runtransaction). You can give this operation an update function and an optional completion callback. The update function takes the current state of the data as an argument and returns the new desired state you would like to write. If another client writes to the location before your new value is successfully written, your update function is called again with the new current value, and the write is retried.

For instance, in the example social blogging app, you could allow users to star and unstar

posts and keep track of how many stars a post has received as follows:

Web modular API | Web namespaced API

```
function toggleStar(postRef, uid) {
  postRef.transaction((post) => {
    if (post) {
      if (post.stars && post.stars[uid]) {
        post.starCount--;
        post.stars[uid] = null;
      } else {
        post.starCount++;
        if (!post.stars) {
          post.stars = {};
        }
        post.stars[uid] = true;
      }
    }
    return post;
  });
}
```

Using a transaction prevents star counts from being incorrect if multiple users star the same post at the same time or the client had stale data. If the transaction is rejected, the server returns the current value to the client, which runs the transaction again with the updated value. This repeats until the transaction is accepted or you abort the transaction.

Note: Because your update function is called multiple times, it must be able to handle **`null`** data. Even if there is existing data in your remote database, it may not be locally cached when the transaction function is run, resulting in **`null`** for the initial value.

Atomic server-side increments

In the above use case we're writing two values to the database: the ID of the user who stars/unstars the post, and the incremented star count. If we already know that user is starring the post, we can use an atomic increment operation instead of a transaction.

Web

```
function addStar(uid, key) {
  const updates = {};
  updates[`posts/${key}/stars/${uid}`] = true;
  updates[`posts/${key}/starCount`] = firebase.database.ServerValue.increme
  updates[`user-posts/${key}/stars/${uid}`] = true;
  updates[`user-posts/${key}/starCount`] = firebase.database.ServerValue.ir
  firebase.database().ref().update(updates);
}
```

This code does not use a transaction operation, so it does not automatically get re-run if there is a conflicting update. However, since the increment operation happens directly on the database server, there is no chance of a conflict.

If you want to detect and reject application-specific conflicts, such as a user starring a post that they already starred before, you should write custom security rules for that use case.

Work with data offline

If a client loses its network connection, your app will continue functioning correctly.

Every client connected to a Firebase database maintains its own internal version of any active data. When data is written, it's written to this local version first. The Firebase client then synchronizes that data with the remote database servers and with other clients on a "best-effort" basis.

As a result, all writes to the database trigger local events immediately, before any data is written to the server. This means your app remains responsive regardless of network latency or connectivity.

Once connectivity is reestablished, your app receives the appropriate set of events so that the client syncs with the current server state, without having to write any custom code.