Notes - Web - Data - Firebase and Firestore

- Dr Nick Hayward

A basic introduction to integrating Firebase and Firestore with a web based application.

Contents

Intro

Firebase offers a hosted backend for web apps, with managed services including authentication, NoSQL databases, file storage, and cloud functions.

For data storage in web apps, we commonly use **Cloud Firestore**, a document-oriented, realtime database accessed over the network. Firestore runs in Google's infrastructure; your web app talks to it via the Firebase JS SDK.

In a typical HTML/CSS/JavaScript web app:

- include Firebase SDK modules in your build (via bundler or script tags)
- initialise Firebase with your config object (apiKey, authDomain, projectId, etc.)
- read and write documents in Firestore over the network
- subscribe to realtime listeners (`onSnapshot`) to reflect remote changes in the UI

The main benefits of Firebase usage for web apps include the following:

- fully managed backend with generous free tier
- Firebase Authentication with multiple identity providers
- Cloud Firestore as a scalable, realtime NoSQL database
- Cloud Functions for server-side logic
- simple client SDK for web that works with HTML/CSS/JavaScript

Offline-aware web apps

For web apps, we often need a graceful fallback for network issues while keeping a responsive UI.

Cloud Firestore's web SDK supports offline persistence (with some browser limitations). In addition, we can design an explicit offline-aware architecture:

- cache user actions in memory or `localStorage` / `IndexedDB` when offline
- write changes to Firestore when the network is available
- use realtime listeners to stream document changes back into the DOM when online

Offline-first behaviour comes from combining:

- browser storage (e.g. `localStorage` , `IndexedDB` , or Firestore persistence)
- Firestore's sync and conflict handling
- a UI that can display "pending" vs "synced" state

Firebase building blocks for web

Firebase offers several features and building blocks for web app development, including:

- Firebase Authentication with email/password and third-party OAuth providers
- Cloud Firestore and/or Realtime Database for data storage
- Cloud Functions for background/server-side logic
- Cloud Storage for files and media
- Firebase Security Rules to control access to collections/documents

For a plain JavaScript web app, these building blocks allow us to:

- protect routes and UI using auth state (e.g. show data only when signed in)
- load and update Firestore documents in response to user actions
- respond to realtime updates via `onSnapshot` and update the DOM

Create Firebase project

- In the Firebase console:
  - sign in at https://console.firebase.google.com
  - create a new project (name, analytics options)
- Add a **web app** to the project:
  - register the app and copy the Firebase config object
  - config includes keys such as `apiKey` , `authDomain` , `projectId` , etc.
- Enable required products for this example:
  - Authentication → choose sign-in methods (optional for simple demo)
  - Firestore Database → create a database in **native mode**
- You will paste the config object into your JavaScript to initialise Firebase.

Basic HTML/JS setup with Firebase For a simple web app (no bundler), we can use `<script type="module">` and add Firebase modular SDK scripts from a CDN.

Example `index.html` skeleton:

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Firebase Notes - Web</title>
        <link rel="stylesheet" href="styles.css" />
    </head>
    <body>
        <main>
            <h1>Firebase Notes (Web)</h1>

            <section id="note-input">
                <input id="note-title" type="text" placeholder="Note title" />
                <button id="add-note">Add Note</button>
            </section>

            <section id="notes-list">
                <h2>Notes</h2>
                <div id="notes-output"></div>
            </section>
        </main>

        <script type="module" src="app.js"></script>
```

```
    </body>
    </html>
```

Here `app.js` will initialise Firebase and wire up the DOM.

Initialise Firebase in JavaScript

In `app.js` , we import the Firebase modules from a CDN (or via bundler) and use the config from the Firebase console.

Example using CDN imports and modular SDK:

```
// app.js
import {initializeApp} from 'https://www.gstatic.com/firebasejs/11.0.0/firebase-app.js';
import {
    getFirestore,
    collection,
    addDoc,
    serverTimestamp,
    onSnapshot,
    orderBy,
    query,
} from 'https://www.gstatic.com/firebasejs/11.0.0/firebase-firestore.js';

const firebaseConfig = {
    apiKey: 'YOUR_API_KEY',
    authDomain: 'YOUR_AUTH_DOMAIN',
    projectId: 'YOUR_PROJECT_ID',
};

const app = initializeApp(firebaseConfig);
const db = getFirestore(app);
```

For production apps you would usually bundle this code (e.g. with Grunt, Vite, Webpack, etc.) and keep config values in a `.env` file during builds. In the browser, these values are still public: they identify your Firebase project but do not grant admin access.

Data models (notes collection)

For a simple notes app, we can define a `notes` collection in Firestore.

Conceptual structure:

- collection: `notes`
- each document fields:
  - `title: string`
  - `createdAt: Timestamp`
  - `status: string | null` (e.g. `local` / `synced`)

In plain JavaScript we typically work with objects of the form:

```
// Example note object shape loaded from Firestore
{
    id: 'abc123',          // document ID
    title: 'Visit Paris',    // string
    createdAt: Date,       // or Firestore Timestamp converted to Date
```

```
    status: 'local',      // or null
}
```

Wire up UI and Firestore (create note)

We can now connect the DOM elements to Firestore.

Continuing `app.js` :

```
// Get DOM elements
const noteInput = document.getElementById('note-title');
const addNoteButton = document.getElementById('add-note');
const notesOutput = document.getElementById('notes-output');

// Handle create note
addNoteButton.addEventListener('click', async () => {
    const title = noteInput.value.trim();
    if (!title) {
        return;
    }

    await addDoc(collection(db, 'notes'), {
        title,
        status: 'local',
        createdAt: serverTimestamp(),
    });

    noteInput.value = '';
});
```

Read and display data (realtime updates)

To display notes and keep the UI in sync, we can use `onSnapshot` on a query.

Continuing `app.js` :

```
// Listen to notes collection in realtime
const notesQuery = query(
    collection(db, 'notes'),
    orderBy('createdAt', 'desc')
);

onSnapshot(notesQuery, snapshot => {
    // Clear current output
    notesOutput.innerHTML = '';

    if (snapshot.empty) {
        notesOutput.textContent = 'No notes yet...';
        return;
    }

    const total = document.createElement('p');
    total.textContent = `Total notes: ${snapshot.size}`;
    notesOutput.appendChild(total);
```

```
    snapshot.docs.forEach(doc => {
        const data = doc.data();
        const item = document.createElement('p');
        item.textContent = data.title;
        notesOutput.appendChild(item);
    });
});
```

The `onSnapshot` listener keeps the HTML view in sync with Firestore in realtime.

Styling and UX considerations

For a small demo, basic CSS is enough to present the notes list clearly.

Example `styles.css` snippet:

```
body {
    font-family: system-ui, -apple-system, BlinkMacSystemFont, 'Segoe UI', sans-serif;
    margin: 0;
    padding: 2rem;
    background: #f4f4f4;
}

main {
    max-width: 640px;
    margin: 0 auto;
    background: #fff;
    padding: 1.5rem;
    border-radius: 8px;
    box-shadow: 0 2px 6px rgba(0, 0, 0, 0.08);
}

#note-input {
    display: flex;
    gap: 0.5rem;
    margin-bottom: 1rem;
}

#note-title {
    flex: 1;
    padding: 0.5rem;
}

#add-note {
    padding: 0.5rem 1rem;
}

#notes-output p {
    margin: 0.25rem 0;
}
```

You can extend this with loading indicators, error messages, and visual cues for offline/online state.

Firebase (web) vs Firebase (React Native) - quick comparison

Shared concepts

- Same Firebase project, Authentication, Firestore, and Security Rules
- Similar modular JS APIs (`initializeApp` , `getFirestore` , `collection` , `addDoc` , `onSnapshot`)
- Realtime updates and offline-aware patterns are conceptually the same

Web (HTML/CSS/JS)

- Use DOM APIs (`document.getElementById` , `addEventListener`) instead of React components
- Typically include Firebase JS SDK via script tags or a bundler
- Can use browser storage (`localStorage` , `IndexedDB`) alongside Firestore

React Native

- Use React components and hooks to manage UI and state
- Use packages like `react-native-firebase` or the JS SDK with additional setup
- Pair Firestore with mobile storage (AsyncStorage/SQLite/MMKV) for offline-aware flows

Both approaches follow the same underlying ideas: initialise Firebase with config, read/write documents in Firestore, and use realtime listeners to keep the UI in sync.