# Connecting with firestore database

Follow this document for firestore database connectivity:

<https://firebase.google.com/docs/firestore>

For this you need to create an “expo” project

Create an online firebase firestore database.

Copy the configuration code (to be pasted in firestoreconfig.js)

Install firebase in project using: npm install firebase

Give command: npm run android

**firebaseconfig.js**

// Import the functions you need from the SDKs you need

import { initializeApp } from "firebase/app";

import {getFirestore} from  "firebase/firestore";

// TODO: Add SDKs for Firebase products that you want to use

// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration

const firebaseConfig = {

  apiKey: "AIzaSyDKSJfAZLNWA328ArcCdQXwtRMiIaRm7dA",

  authDomain: "dbproj-f3054.firebaseapp.com",

  projectId: "dbproj-f3054",

  storageBucket: "dbproj-f3054.appspot.com",

  messagingSenderId: "352306058995",

  appId: "1:352306058995:web:962d3872b952ef33478986"

};

// Initialize Firebase

const app = initializeApp(firebaseConfig);

const db = getFirestore(app);

export {db};

**App.js**

import React from 'react';

import { Button, View } from 'react-native';

import {collection, getDocs, getDoc, doc, setDoc } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

**// To create or overwrite a single document, use the following language-specific set() methods:**

async function writeData() {

  await setDoc(doc(db, "users", "kamal@gmail.com"), {

    name: "Kamal Ahmed",

    age: "22"

  });

}

/\*

If the document does not exist, it will be created. If the document does exist, its contents will be overwritten with the newly provided data, unless you specify that the data should be merged into the existing document, as follows:

\*/

function mergeData()

{

  const userRef = doc(db, 'users', 'ali@gmail.com');

setDoc(userRef, { age: 46 }, { merge: true });

}

async function readData() {

  const docRef = doc(db, "users", "ali@gmail.com");

  const docSnap = await getDoc(docRef);

  if (docSnap.exists()) {

    console.log("Document data:", docSnap.data());

  } else {

    // doc.data() will be undefined in this case

    console.log("No such document!");

  }

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Write Data'

   onPress={ () => writeData()}

    />

   <Button

   title='Read Data'

   onPress={ () => readData()}

    />

<Button

   title='Merge Data'

   onPress={ () => mergeData()}

    />

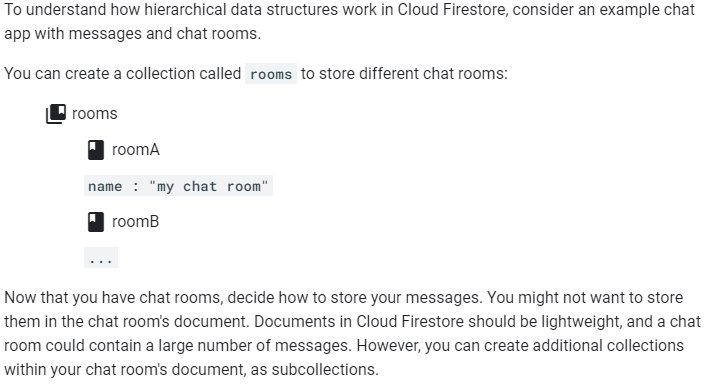
   </View>

   );

}

export default App;

Example Storing the Hierarchical Data



Graphical user interface, text, application, email

Description automatically generated

import React from 'react';

import { Button, View } from 'react-native';

import { doc, getDoc } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

async function readData() {

  const docRef = doc(db, "rooms", "roomA", "messages", "message1");

  const docSnap = await getDoc(docRef);

  if (docSnap.exists()) {

    console.log("Document data:", docSnap.data());

  } else {

    // doc.data() will be undefined in this case

    console.log("No such document!");

  }

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Read Data'

   onPress={ () => readData()}

    />

   </View>

   );

}

export default App;

# Data types in Firestore

Cloud Firestore lets you write a variety of data types inside a document, including strings, booleans, numbers, dates, null, and nested arrays and objects. Cloud Firestore always stores numbers as doubles, regardless of what type of number you use in your code.

import React from 'react';

import { Button, View } from 'react-native';

import { doc, setDoc, Timestamp } from "firebase/firestore";

import {db} from './firestoreconfig.js';

async function writeData() {

  const docData = {

    stringExample: "Hello world!",

    booleanExample: true,

    numberExample: 3.14159265,

    dateExample: Timestamp.fromDate(new Date("December 10, 1815")),

    arrayExample: [5, true, "hello"],

    nullExample: null,

    objectExample: {

        a: 5,

        b: {

            nested: "foo"

        }

    }

};

await setDoc(doc(db, "data", "one"), docData);

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Write Data'

   onPress={ () => writeData()}

    />

   </View>

   );

}

export default App;

# Custom objects

Using Map or Dictionary objects to represent your documents is often not very convenient, so Cloud Firestore supports writing documents with custom classes. Cloud Firestore converts the objects to supported data types.

Using custom classes, you could rewrite the initial example as shown:

import React from 'react';

import { Button, View } from 'react-native';

import { doc, setDoc } from "firebase/firestore";

import {db} from './firestoreconfig.js';

class City {

  constructor (name, state, country ) {

      this.name = name;

      this.state = state;

      this.country = country;

  }

  toString() {

      return this.name + ', ' + this.state + ', ' + this.country;

  }

}

// Firestore data converter

const cityConverter = {

  toFirestore: (city) => {

      return {

          name: city.name,

          state: city.state,

          country: city.country

          };

  },

  fromFirestore: (snapshot, options) => {

      const data = snapshot.data(options);

      return new City(data.name, data.state, data.country);

  }

};

async function writeData() {

  const ref = doc(db, "cities", "LA").withConverter(cityConverter);

  await setDoc(ref, new City("Los Angeles", "CA", "USA"));

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Write Data'

   onPress={ () => writeData()}

    />

   </View>

   );

}

export default App;

# Add a document with explicitly specified ID

When you use set() to create a document, you must specify an ID for the document to create. For example:

import React from 'react';

import { Button, View } from 'react-native';

import {collection, getDocs, getDoc, doc, setDoc } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

// To create or overwrite a single document, use the following language-specific set() methods:

async function writeData() {

    await setDoc(doc(db, "cities", "new-city-id"), {country: "Pakistan", name: "Sargodha", state: "Punjab"});

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Write Data'

   onPress={ () => writeData()}

    />

   <Button

   title='Read Data'

    />

<Button

   title='Merge Data'

    />

   </View>

   );

}

export default App;

# Add document with auto-generated ID

But sometimes there isn't a meaningful ID for the document, and it's more convenient to let Cloud Firestore auto-generate an ID for you. You can do this by calling the following language-specific add() methods:

import React from 'react';

import { Button, View } from 'react-native';

import {collection, getDocs, getDoc, doc, setDoc, addDoc } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

async function writeData() {

    // Add a new document with a generated id.

const docRef = await addDoc(collection(db, "cities"), {

    name: "Tokyo",

    country: "Japan",

    state: "North"

  });

  console.log("Document written with ID: ", docRef.id);

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Write Data'

   onPress={ () => writeData()}

    />

   <Button

   title='Read Data'

    />

<Button

   title='Merge Data'

    />

   </View>

   );

}

export default App;

# Create a document reference with auto-generated ID

In some cases, it can be useful to create a document reference with an auto-generated ID, then use the reference later. For this use case, you can call doc():

import React from 'react';

import { Button, View } from 'react-native';

import {collection, getDocs, getDoc, doc, setDoc, addDoc } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

async function writeData() {

    // Add a new document with a generated id

const newCityRef = doc(collection(db, "cities"));

// later...

await setDoc(newCityRef, {country: "Germany", name: "Berg", state: "South"});

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Write Data'

   onPress={ () => writeData()}

    />

   <Button

   title='Read Data'

    />

<Button

   title='Merge Data'

    />

   </View>

   );

}

export default App;

Behind the scenes, .add(...) and .doc().set(...) are completely equivalent, so you can use whichever is more convenient.

# Update a document

To update some fields of a document without overwriting the entire document, use the following language-specific update() methods:

import React from 'react';

import { Button, View } from 'react-native';

import {collection, getDocs, getDoc, doc, setDoc, addDoc, updateDoc } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

async function updateData() {

    const Ref = doc(db, "cities", "KHI");

// Set the "capital" field of the city 'DC'

await updateDoc(Ref, {state: "Punjab"});

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Update Data'

   onPress={ () => updateData()}

    />

   </View>

   );

}

export default App;

# Server Timestamp

You can set a field in your document to a server timestamp which tracks when the server receives the update.

import React from 'react';

import { Button, View } from 'react-native';

import {collection, getDocs, getDoc, doc, setDoc, addDoc, updateDoc, serverTimestamp  } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

async function writeData() {

    const docRef = doc(db, 'cities', 'KHI');

// Update the timestamp field with the value from the server

const updateTimestamp = await updateDoc(docRef, {

    timestamp: serverTimestamp()

});

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Update Data'

   onPress={ () => writeData()}

    />

   </View>

   );

}

export default App;

When updating multiple timestamp fields inside of a transaction, each field receives the same server timestamp value.

# Update fields in nested objects

If your document contains nested objects, you can use "dot notation" to reference nested fields within the document when you call update():

import React from 'react';

import { Button, View } from 'react-native';

import {collection, getDocs, getDoc, doc, setDoc, addDoc, updateDoc, serverTimestamp  } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

async function updateData() {

// Create an initial document to update.

const frankDocRef = doc(db, "users", "frank");

await setDoc(frankDocRef, {

    name: "Frank",

    favorites: { food: "Pizza", color: "Blue", subject: "recess" },

    age: 12

});

// To update age and favorite color:

await updateDoc(frankDocRef, {

    "age": 13,

    "favorites.color": "Red"

});

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Update Data'

   onPress={ () => updateData()}

    />

   </View>

   );

}

export default App;

Dot notation allows you to update a single nested field without overwriting other nested field. If you update a nested field without dot notation, you will overwrite the entire map field, for example:

import React from 'react';

import { Button, View } from 'react-native';

import {collection, getDocs, getDoc, doc, setDoc, updateDoc } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

// To create or overwrite a single document, use the following language-specific set() methods:

async function writeData() {

    const DocRef = doc(db, "users", "tommy");

    await setDoc(DocRef, {

        name: "Frank",

        favorites: { food: "Pizza", color: "Blue", subject: "recess" },

        age: 12

    });

    // To update age and favorite color:

    await updateDoc(DocRef, {

        "age": 13,

        "favorites.color": "Red"

    });

}

async function updateData() {

    // Update the doc without using dot notation.

// Notice the map value for favorites.

const DocRef = doc(db, "users", "tommy");

await updateDoc(DocRef, {

    favorites: {

      food: "Ice Cream"

    }

  });

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Write Data'

   onPress={ () => writeData()}

    />

   <Button

   title='Update Data'

   onPress={ () => updateData()}

    />

   </View>

   );

}

export default App;

# Update elements of an array

import React from 'react';

import { Button, View } from 'react-native';

import {collection, getDocs, getDoc, doc, setDoc, updateDoc, arrayUnion, arrayRemove } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

// To create or overwrite a single document, use the following language-specific set() methods:

async function removeData() {

    const docRef = doc(db, "cities", "KHI");

    // Atomically remove a region from the "regions" array field.

await updateDoc(docRef, {regions: arrayRemove("greater\_virginia")});

}

async function updateData() {

const Ref = doc(db, "cities", "KHI");

// Atomically add a new region to the "regions" array field.

await updateDoc(Ref, {

    regions: arrayUnion("greater\_virginia")

});

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Update Data'

   onPress={ () => updateData()}

    />

<Button

   title='Remove Data'

   onPress={ () => removeData()}

    />

   </View>

   );

}

export default App;

# Increment a numeric value

You can increment or decrement a numeric field value as shown in the following example. An increment operation increases or decreases the current value of a field by the given amount.

import React from 'react';

import { Button, View } from 'react-native';

import {collection, increment, getDocs, getDoc, doc, setDoc, updateDoc, arrayUnion, arrayRemove } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

async function updateData() {

    const washingtonRef = doc(db, "cities", "LA");

    //suppose we have a population field in cities for "DC" document

    // Atomically increment the population of the city by 50.

    await updateDoc(washingtonRef, {

        population: increment(50)

    });

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Update Data'

   onPress={ () => updateData()}

    />

<Button

   title='Remove Data'

   onPress={ () => removeData()}

    />

   </View>

   );

}

export default App;

# Transactions and batched writes

Cloud Firestore supports atomic operations for reading and writing data. In a set of atomic operations, either all of the operations succeed, or none of them are applied. There are two types of atomic operations in Cloud Firestore:

Transactions: a transaction is a set of read and write operations on one or more documents.

Batched Writes: a batched write is a set of write operations on one or more documents.

Each transaction or batch of writes can write to a maximum of 500 documents. For additional limits related to writes, see Quotas and Limits.

# Updating data with transactions

Using the Cloud Firestore client libraries, you can group multiple operations into a single transaction. Transactions are useful when you want to update a field's value based on its current value, or the value of some other field.

A transaction consists of any number of get() operations followed by any number of write operations such as set(), update(), or delete(). In the case of a concurrent edit, Cloud Firestore runs the entire transaction again. For example, if a transaction reads documents and another client modifies any of those documents, Cloud Firestore retries the transaction. This feature ensures that the transaction runs on up-to-date and consistent data.

Transactions never partially apply writes. All writes execute at the end of a successful transaction.

When using transactions, note that:

Read operations must come before write operations.

A function calling a transaction (transaction function) might run more than once if a concurrent edit affects a document that the transaction reads.

Transaction functions should not directly modify application state.

Transactions will fail when the client is offline.

The following example shows how to create and run a transaction:

import React from 'react';

import { Button, View } from 'react-native';

import {runTransaction, collection, increment, getDocs, getDoc, doc, setDoc, updateDoc, arrayUnion, arrayRemove } from 'firebase/firestore';

import {db} from './firestoreconfig.js';

async function updateData() {

  // a transaction to increase population of a city

    const sfDocRef = doc(db, "cities", "LA");

   try {

  await runTransaction(db, async (transaction) => {

    const sfDoc = await transaction.get(sfDocRef);

    if (!sfDoc.exists()) {

      throw "Document does not exist!";

    }

    const newPopulation = sfDoc.data().population + 1;

    transaction.update(sfDocRef, { population: newPopulation });

  });

  console.log("Transaction successfully committed!");

} catch (e) {

  console.log("Transaction failed: ", e);

}

}

const App = () => {

  return (

<View

style = {{marginTop: 50}}

>

   <Button

   title='Update Data'

   onPress={ () => updateData()}

    />

   </View>

   );

}

export default App;