API với firebase, authentication, cloud function

Firebase với flutter

<https://medium.com/flutterpub/realtime-database-in-flutter-bef0f29e3378>

<https://firebase.google.com/docs/reference/android/com/google/firebase/database/FirebaseDatabase>

<https://firebase.google.com/docs/reference/android/com/google/firebase/database/DatabaseReference.html>

FirebaseDatabase.instance.ref bla bla là gì??? -> giải thích??

FirebaseDatabase

The object for accessing a Firebase Database. You can get an instance by calling getInstance(). To access a location in the database and read or write data, use getReference().

Initialize & Reference

To get main reference, we need to do is get access to static field in FirebaseDatabase class. If we wanna access more specific child of our database, just use child() method:

final notesReference = FirebaseDatabase.instance.reference();

final notesReference = FirebaseDatabase.instance.reference().child('notes');

---

From the point of view of an Android app, a Firebase database is represented by an instance of the FirebaseDatabase class, a reference to which is obtained via a call to the getInstance() method of the class, for example:

FirebaseDatabase database = FirebaseDatabase.getInstance();

The FirebaseDatabase instance can then be used to get a database reference. This is a reference to a specific point within the database tree at which read and write operations may be performed. A database reference is represented by an instance of the DatabaseReference class and is obtained via a call to the getReference() method of the FirebaseDatabase instance:

DatabaseReference dbRef = database.getReference();

When the getReference() method is called without arguments, the path for the reference within the database is set to the root of the tree. Any write operations performed using this reference would, therefore, be performed relative to the tree root. In Figure 20 1 above, the address, name and phone tree nodes were all added as direct children of the root element.

If, on the other hand, a path is passed as an argument, all write operations will be performed relative to that path into the tree. In the following line of code, for example, the database reference is set to a specific path within the current database:

dbRef = database.getReference("/test/data/message1");

If the specified path does not already exist, it is created automatically within the tree as soon as data is written at that location.

Giải thích tham số trong các trường hợp dưới -> bảng tham số?

Create

*FirebaseDatabase.instance.reference().child('recent').child('id')*

*.set({*

*'title': 'Realtime db rocks',*

*'created\_at': time*

*});*

public static FirebaseDatabase getInstance (String url)

Gets a FirebaseDatabase instance for the specified URL

Parameters

url The URL to the Firebase Database instance you want to access.

Read -> giải thích dùng kèm hàm stream builder

*var recentJobsRef = recentJobsRef = FirebaseDatabase.instance*

*.reference()*

*.child('recent')*

*.orderByChild('created\_at') //order by creation time.*

*.limitToFirst(10); //limited to get only 10 documents.*

*//Now you can use stream builder to get the data.*

*StreamBuilder(*

*stream: recentJobsRef.onValue,*

*builder: (context, snap) {*

*if (snap.hasData && !snap.hasError && snap.data.snapshot.value!=null) {*

*//taking the data snapshot.*

*DataSnapshot snapshot = snap.data.snapshot;*

*List item=[];*

*List \_list=[];*

*//it gives all the documents in this list.*

*\_list=snapshot.value;*

*//Now we're just checking if document is not null then add it to another list called "item".*

*//I faced this problem it works fine without null check until you remove a document and then your stream reads data including the removed one with a null value(if you have some better approach let me know).*

*\_list.forEach((f){*

*if(f!=null){*

*item.add(f);*

*}*

*}*

*);*

*return snap.data.snapshot.value == null*

*//return sizedbox if there's nothing in database.*

*? SizedBox()*

*//otherwise return a list of widgets.*

*: ListView.builder(*

*scrollDirection: Axis.horizontal,*

*itemCount: item.length,*

*itemBuilder: (context, index) {*

*return \_containerForRecentJobs(*

*item[index]['title']*

*);*

*},*

*);*

*} else {*

*return Center(child: CircularProgressIndicator());*

*}*

*},*

),

public DatabaseReference getReference (String path)

Also: Google Play services

Gets a DatabaseReference for the provided path.

Parameters

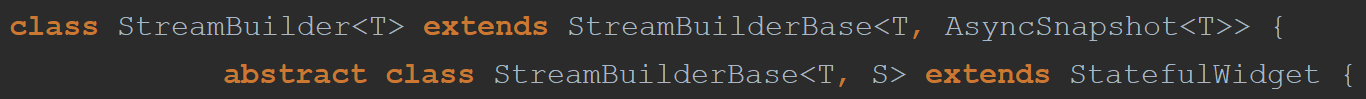
path Path to a location in your FirebaseDatabase.

Returns

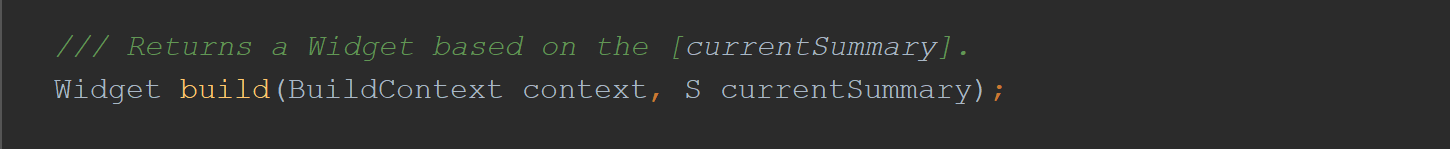
A DatabaseReference pointing to the specified path.

The Base Of The Stream

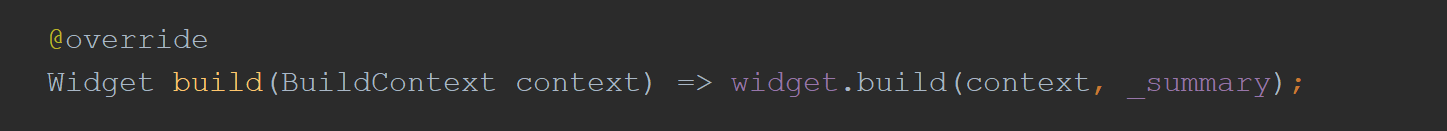
StreamBuilder is a StatefulWidget. It’s a StatefulWidget but not before extending another class that actually extends the StatefulWidget, StreamBuilderBase<T, AsyncSnapshot<T>>. The screenshot below depicts the class hierarchy involved:



The class, StreamBuilderBase<T, S> extends StatefulWidget and is an abstract class. It has a few abstract functions that must be implemented when the class is extended. One of which is it’s build() function.



Now this is curious since with most StatefulWidgets, it’s their associated State object that defines a build() function and not the StatefuleWidget itself. However, in the case of the StreamBuilderBase class, it’s associated State object refers back to this StatefulWidget’s own build() function.



Hoặc read bình thường:

To get data from database, we add listener to the reference. When getting the value, we simply add it to list (state variable) and call setState() method.

Update

Calling child(key) will return a reference to object we want to edit, then use set(newValue) method.

FirebaseDatabase.reference()

.child('recent')

.child('id')

.update({

'title':'sadab is amazing' //yes I know.

});

Delete -> them trường hợp set giá trị null = delete

//remove() is equivalent to calling set(null)

recentJobRef.child('id').remove();

Authentication

<https://developers.google.com/android/reference/com/google/firebase/auth/FirebaseAuth>

First, obtain an instance of this class by calling getInstance().

Then, sign up or sign in a user with one of the following methods:

* createUserWithEmailAndPassword(String, String)
* signInWithEmailAndPassword(String, String)
* signInWithCredential(AuthCredential)
* signInAnonymously()
* signInWithCustomToken(String)

Finally, call getCurrentUser() to get a FirebaseUser object, which contains information about the signed-in user.

public FirebaseUser getCurrentUser ()

Returns the currently signed-in FirebaseUser or null if there is none.

Use getCurrentUser() != null to check if a user is signed in.

Returns

the signed-in user or null

public Task<Void> sendPasswordResetEmail (String email, ActionCodeSettings actionCodeSettings)

Triggers the Firebase Authentication backend to send a password-reset email to the given email address, which must correspond to an existing user of your app. Takes in an ActionCodeSettings which allows linking back to your app from the sent email.

Exceptions

FirebaseAuthInvalidUserException thrown if there is no user corresponding to the given email address

Parameters

email the email of the account to which you wish to issue an account reset email

actionCodeSettings the settings used to allow your app to handle the link sent in the email on iOS, web, and Android.

public Task<AuthResult> signInWithEmailAndPassword (String email, String password)

Tries to sign in a user with the given email address and password.

Access the signed-in user with getCurrentUser().

Upon successful completion, this operation triggers an onIdTokenChanged(FirebaseAuth) event in all registered FirebaseAuth.IdTokenListeners and an onAuthStateChanged(FirebaseAuth) event in all registered FirebaseAuth.AuthStateListeners.

This is equivalent to calling signInWithCredential(AuthCredential) with an EmailAuthCredential generated by getCredential(String, String).

Important: you must enable Email & Password accounts in the Firebase console before being able to use this method.

Exceptions

FirebaseAuthInvalidUserException thrown if the user account corresponding to email does not exist or has been disabled

FirebaseAuthInvalidCredentialsException thrown if the password is wrong

See also:

createUserWithEmailAndPassword(String, String)

signInWithCredential(AuthCredential)

signInAnonymously()

signInWithCustomToken(String)

Returns

Public Methods

public void addAuthStateListener (FirebaseAuth.AuthStateListener listener)

Registers a listener to changes in the user authentication state. There can be more than one listener registered at the same time for one or more FirebaseAuth instances.

The listeners call back in the UI thread, on the following events:

* Right after the listener has been registered
* When a user signs in
* When the current user signs out
* When the current user changes

It is a recommended practice to always listen to sign-out events, as you may want to prompt the user to sign in again and maybe restrict the information or actions they have access to.

Cloud function

<https://firebase.google.com/docs/functions/database-events>

Handle event data

When handling a Realtime Database event, the data object returned is a DataSnapshot. For onWrite or onUpdate events, the first parameter is a Change object that contains two snapshots that represent the data state before and after the triggering event. For onCreate and onDelete events, the data object returned is a snapshot of the data created or deleted.

Set the event handler

Functions let you handle database events at two levels of specificity; you can listen for specifically for only creation, update, or deletion events, or you can listen for any change of any kind to a path. Cloud Functions supports these event handlers for Realtime Database:

* onWrite(), which triggers when data is created, updated, or deleted in the Realtime Database.
* onCreate(), which triggers when new data is created in the Realtime Database.
* onUpdate(), which triggers when data is updated in the Realtime Database.
* onDelete(), which triggers when data is deleted from the Realtime Database.

Specify the database instance and path

To control when and where your function should trigger, call ref(path) to specify a path, and optionally specify a database instance with instance('INSTANCE\_NAME'). If you do not specify an instance, the function deploys to the default database instance for the Firebase project For example:

* Default database instance: functions.database.ref('/foo/bar')
* Instance named "my-app-db-2": functions.database.instance('my-app-db-2').ref('/foo/bar')

These methods direct your function to handle writes at a certain path within the database instance. Path specifications match all writes that touch a path, including writes that happen anywhere below it. If you set the path for your function as /foo/bar, it matches events at both of these locations:

/foo/bar

/foo/bar/baz/really/deep/path

In either case, Firebase interprets that the event occurs at /foo/bar, and the event data includes the old and new data at /foo/bar. If the event data might be large, consider using multiple functions at deeper paths instead of a single function near the root of your database. For the best performance, only request data at the deepest level possible.

You can specify a path component as a wildcard by surrounding it with curly brackets; ref('foo/{bar}') matches any child of /foo. The values of these wildcard path components are available within the EventContext.params object of your function. In this example, the value is available as event.params.bar.

{

"foo": {

"hello": "world",

"firebase": "functions"

}

}

Paths with wildcards can match multiple events from a single write. An insert of

In this example, the function retrieves the snapshot for the specified path as snap, converts the string at that location to uppercase, and writes that modified string to the database:

// Listens for new messages added to /messages/:pushId/original and creates an

// uppercase version of the message to /messages/:pushId/uppercase

exports.makeUppercase = functions.database.ref('/messages/{pushId}/original')

.onCreate((snapshot, context) => {

// Grab the current value of what was written to the Realtime Database.

const original = snapshot.val();

console.log('Uppercasing', context.params.pushId, original);

const uppercase = original.toUpperCase();

// You must return a Promise when performing asynchronous tasks inside a Functions such as

// writing to the Firebase Realtime Database.

// Setting an "uppercase" sibling in the Realtime Database returns a Promise.

return snapshot.ref.parent.child('uppercase').set(uppercase);

});

Accessing user authentication information

From EventContext.auth and EventContext.authType, you can access the user information, including permissions, for the user that triggered a function. This can be useful for enforcing security rules, allowing your function to complete different operations based on the user's level of permissions:

const functions = require('firebase-functions');

const admin = require('firebase-admin');

exports.simpleDbFunction = functions.database.ref('/path')

.onCreate((snap, context) => {

if (context.authType === 'ADMIN') {

// do something

} else if (context.authType === 'USER') {

console.log(snap.val(), 'written by', context.auth.uid);

}

});

Also, you can leverage user authentication information to "impersonate" a user and perform write operations on the user's behalf. Make sure to delete the app instance as shown below in order to prevent concurrency issues:

exports.impersonateMakeUpperCase = functions.database.ref('/messages/{pushId}/original')

.onCreate((snap, context) => {

const appOptions = JSON.parse(process.env.FIREBASE\_CONFIG);

appOptions.databaseAuthVariableOverride = context.auth;

const app = admin.initializeApp(appOptions, 'app');

const uppercase = snap.val().toUpperCase();

const ref = snap.ref.parent.child('uppercase');

const deleteApp = () => app.delete().catch(() => null);

return app.database().ref(ref).set(uppercase).then(res => {

// Deleting the app is necessary for preventing concurrency leaks

return deleteApp().then(() => res);

}).catch(err => {

return deleteApp().then(() => Promise.reject(err));

});

});

Reading the previous value

The Change object has a before property that lets you inspect what was saved to the database before the event. The before property returns a DataSnapshot where all methods (for example, val() and exists()) refer to the previous value. You can read the new value again by either using the original DataSnapshot or reading the after property. This property on any Change is another DataSnapshot representing the state of the data after the event happened.

For example, the before property can be used to make sure the function only uppercases text when it is first created:

exports.makeUppercase = functions.database.ref('/messages/{pushId}/original')

.onWrite((change, context) => {

// Only edit data when it is first created.

if (change.before.exists()) {

return null;

}

// Exit when the data is deleted.

if (!change.after.exists()) {

return null;

}

// Grab the current value of what was written to the Realtime Database.

const original = change.after.val();

console.log('Uppercasing', context.params.pushId, original);

const uppercase = original.toUpperCase();

// You must return a Promise when performing asynchronous tasks inside a Functions such as

// writing to the Firebase Realtime Database.

// Setting an "uppercase" sibling in the Realtime Database returns a Promise.

return change.after.ref.parent.child('uppercase').set(uppercase);

});