Angular에서 Firebase Realtime Database API를 활용하는 가장 일반적이고 권장되는 방법은 공식 라이브러리인 **AngularFire**를 사용하는 것입니다. AngularFire는 Firebase SDK를 Angular 환경에 맞게 최적화하고, RxJS를 활용하여 **실시간 데이터 동기화**를 **Observables** 형태로 쉽게 처리할 수 있게 해줍니다.

**1. 환경 설정 및 설치 🛠️**

Realtime Database를 사용하기 위한 기본적인 단계입니다.

**1.1. AngularFire 설치**

프로젝트 터미널에서 AngularFire를 설치합니다.

Bash

ng add @angular/fire

설치 과정에서 Firebase 프로젝트 설정 및 사용할 기능을 선택하라는 메시지가 나타납니다. 여기서 Realtime Database를 선택합니다.

**1.2. 환경 변수 설정**

Firebase 프로젝트 설정에서 받은 인증 정보(API Key, Auth Domain 등)가 environment.ts 파일에 설정되어 있는지 확인합니다.

TypeScript

// src/environments/environment.ts

export const environment = {

// ...

firebase: {

apiKey: "YOUR\_API\_KEY",

authDomain: "YOUR\_AUTH\_DOMAIN",

databaseURL: "YOUR\_DATABASE\_URL", // Realtime DB URL

projectId: "YOUR\_PROJECT\_ID",

storageBucket: "YOUR\_STORAGE\_BUCKET",

messagingSenderId: "YOUR\_MESSAGING\_SENDER\_ID",

appId: "YOUR\_APP\_ID"

}

};

**1.3. 모듈 임포트**

메인 모듈(일반적으로 app.module.ts)에서 필요한 AngularFire 모듈을 임포트합니다.

TypeScript

// app.module.ts

import { provideFirebaseApp, initializeApp } from '@angular/fire/app';

import { provideDatabase, getDatabase } from '@angular/fire/database';

import { environment } from '../environments/environment';

@NgModule({

imports: [

// ...

provideFirebaseApp(() => initializeApp(environment.firebase)),

provideDatabase(() => getDatabase()), // Realtime Database 모듈 제공

],

// ...

})

export class AppModule { }

**2. Realtime Database API 활용 (CRUD) 🚀**

서비스나 컴포넌트에서 AngularFireDatabase를 주입받아 사용합니다. AngularFire의 핵심은 데이터를 \*\*Observable\*\*로 가져와 실시간으로 반응한다는 것입니다.

**2.1. 데이터 읽기 (Read)**

object 또는 list 형태로 데이터를 읽어올 수 있습니다.

| **메서드** | **반환 타입** | **설명** |
| --- | --- | --- |
| object() | AngularFireObject<T> | 단일 객체 데이터 (경로의 전체 값)를 반환합니다. |
| list() | AngularFireList<T> | 배열 형태의 데이터 목록을 반환합니다. |

**예시: 서비스에서 데이터 가져오기**

TypeScript

// data.service.ts

import { Injectable } from '@angular/core';

import { AngularFireDatabase, AngularFireList, AngularFireObject } from '@angular/fire/database';

import { Observable } from 'rxjs';

interface Item {

name: string;

count: number;

}

@Injectable({

providedIn: 'root'

})

export class DataService {

itemsRef: AngularFireList<Item>;

singleItemRef: AngularFireObject<Item>;

constructor(private db: AngularFireDatabase) {

// '/items' 경로의 데이터 목록 참조

this.itemsRef = db.list<Item>('items');

// '/status/server' 경로의 단일 객체 참조

this.singleItemRef = db.object<Item>('status/server');

}

// 1. 객체 데이터 실시간 구독

getSingleItem(): Observable<Item | null> {

// valueChanges()는 객체의 값만 Observable로 반환합니다.

return this.singleItemRef.valueChanges();

}

// 2. 목록 데이터 실시간 구독

getItems(): Observable<Item[]> {

return this.itemsRef.valueChanges();

}

}

**2.2. 데이터 쓰기 (Create/Update)**

데이터를 쓰는 주요 메서드는 set(), update(), push()입니다.

| **메서드** | **용도** | **설명** |
| --- | --- | --- |
| set(value) | 덮어쓰기 (Write) | 지정된 경로의 **모든 데이터**를 새 값으로 **완전히 대체**합니다. |
| update(value) | 업데이트 (Update) | 지정된 경로의 **특정 필드**만 업데이트합니다 (병합). |
| push(value) | 생성 (Create) | 목록 경로에 고유한 **자동 생성 키**를 가진 새 데이터를 추가합니다. |

**예시: 데이터 수정/추가**

TypeScript

// DataService 내 메서드

// 새로운 항목 추가 (목록에 고유 키 생성)

addItem(item: Item): void {

this.itemsRef.push(item); // this.itemsRef는 db.list('items') 참조

}

// 특정 항목의 값 업데이트

updateItem(key: string, newCount: number): Promise<void> {

// list() 참조에서 key를 사용하여 특정 항목의 객체 참조를 가져옵니다.

return this.db.list('items').update(key, { count: newCount });

}

// 단일 객체 (e.g., 서버 상태) 전체 덮어쓰기

setServerStatus(status: Item): Promise<void> {

return this.singleItemRef.set(status); // this.singleItemRef는 db.object('status/server') 참조

}

**2.3. 데이터 삭제 (Delete)**

remove() 메서드를 사용하면 해당 경로의 데이터 전체를 삭제합니다.

TypeScript

// DataService 내 메서드

// 특정 항목 삭제

deleteItem(key: string): Promise<void> {

return this.db.list('items').remove(key);

}

// 전체 목록 삭제

deleteAllItems(): Promise<void> {

return this.itemsRef.remove();

}

**3. 고급 쿼리 (Advanced Querying) 🔍**

Realtime Database는 제한적인 쿼리 기능을 가지고 있지만, AngularFire를 통해 쉽게 적용할 수 있습니다. 쿼리 옵션은 db.list()를 호출할 때 두 번째 인수로 전달됩니다.

**예시: 항목 개수가 100보다 큰 데이터만 가져오기**

TypeScript

getHighCountItems(): Observable<Item[]> {

return this.db.list<Item>('items', ref =>

ref.orderByChild('count').startAt(100)

).valueChanges();

}

| **쿼리 옵션** | **설명** |
| --- | --- |
| orderByChild(key) | 지정된 자식 키를 기준으로 정렬합니다. |
| orderByKey() | 키 이름을 기준으로 정렬합니다. |
| limitToFirst(limit) | 목록의 처음 limit 개 항목을 반환합니다. |
| startAt(value) | 지정된 값 이상으로 시작하는 항목을 반환합니다. |
| equalTo(value) | 지정된 값과 정확히 일치하는 항목을 반환합니다. |

**주의:** Realtime Database는 하나의 쿼리에서 여러 필터를 조합할 수 없으며, 효율적인 쿼리를 위해서는 orderBy 조건을 반드시 사용해야 합니다.

realtime database crud설명해줘

**💾 Realtime Database CRUD 설명**

Firebase Realtime Database에서 **CRUD**($\text{Create, Read, Update, Delete}$) 작업은 데이터를 읽고 쓰는 핵심 기능입니다. 모든 작업은 데이터의 특정 위치를 나타내는 **Reference 객체**를 사용하여 수행됩니다.

**1. $\text{Create}$ (생성)**

새로운 데이터를 저장하는 작업입니다.

**A. 새 객체 생성 및 덮어쓰기: set()**

* 특정 경로에 데이터를 새로 만들거나, 기존 데이터를 완전히 **덮어씁니다**.
* **사용 시점:** 단일 객체나 데이터 셋을 저장하거나, 기존 데이터를 완전히 대체할 때 사용합니다.

JavaScript

// 'users/userID' 경로에 사용자 객체를 저장 (기존 데이터 덮어쓰기)

ref(db, 'users/' + userID).set({

username: 'Jane Doe',

email: 'jane@example.com'

});

**B. 목록에 새 항목 추가: push()**

* 목록(list) 형태의 데이터에 **고유한 시간 순서 키**를 자동으로 생성하여 그 아래에 새 데이터를 추가합니다.
* **사용 시점:** 댓글, 게시물, 채팅 메시지 등 **항목이 계속 추가되는 목록**을 만들 때 사용합니다.

JavaScript

// 'posts' 목록에 새 게시물을 추가하고 고유 키 생성

const newPostRef = push(ref(db, 'posts'));

set(newPostRef, {

author: 'Jane Doe',

title: '나의 첫 포스트',

timestamp: Date.now()

});

**2. $\text{Read}$ (읽기)**

저장된 데이터를 가져오는 작업입니다.

**A. 일회성 읽기: get()**

* 데이터베이스에서 현재 시점의 데이터를 **단 한 번** 가져옵니다.
* **사용 시점:** 화면을 로드할 때 한 번 필요한 설정 정보나 프로필 데이터 등을 가져올 때 사용합니다.

JavaScript

import { get, ref } from "firebase/database";

get(ref(db, 'users/' + userID)).then((snapshot) => {

if (snapshot.exists()) {

console.log("읽어온 데이터:", snapshot.val());

} else {

console.log("데이터가 존재하지 않습니다.");

}

}).catch((error) => {

console.error(error);

});

**B. 실시간으로 읽기: onValue()**

* 데이터를 한 번 가져오는 것 외에도, 해당 경로의 데이터가 **변경될 때마다** 자동으로 업데이트를 받아옵니다 (실시간 동기화).
* **사용 시점:** 채팅 앱, 실시간 대시보드 등 **데이터 변화를 즉시 반영**해야 할 때 사용합니다.

JavaScript

import { onValue, ref } from "firebase/database";

const starCountRef = ref(db, 'posts/' + postID + '/starCount');

onValue(starCountRef, (snapshot) => {

const starCount = snapshot.val();

// starCount가 변경될 때마다 이 코드가 실행됩니다.

updateUI(starCount);

});

**3. $\text{Update}$ (수정)**

기존 데이터의 **일부 필드만** 변경하는 작업입니다.

* **update()** 메소드는 기존 객체의 모든 필드를 덮어쓰지 않고, 제공된 객체 내의 필드만 변경하고 나머지 필드는 **그대로 유지**합니다.
* **사용 시점:** 사용자 이름, 상태 메시지, 카운터 값 등 **특정 속성만 수정**해야 할 때 사용합니다.

JavaScript

import { update, ref } from "firebase/database";

update(ref(db, 'users/' + userID), {

username: "Jane Smith", // username만 변경

// email 필드는 그대로 유지됨

});

**4. $\text{Delete}$ (삭제)**

데이터를 제거하는 작업입니다.

**A. 경로 삭제: remove()**

* 지정된 경로에 있는 데이터와 그 아래에 있는 **모든 하위 데이터**를 영구적으로 제거합니다.

JavaScript

import { remove, ref } from "firebase/database";

// 'posts/postID' 경로와 그 아래의 모든 데이터를 삭제

remove(ref(db, 'posts/' + postID));

**B. null로 설정하여 삭제: set() 또는 update()**

* 특정 경로를 null 값으로 설정하여 삭제할 수도 있습니다. update()를 사용하여 객체의 특정 필드만 null로 설정하면 해당 필드만 제거됩니다.

JavaScript

// 'users/userID/email' 필드만 삭제

update(ref(db, 'users/' + userID), {

email: null

});