Dismiss

Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master

Find file Copy path

angularfire / docs / rtdb / objects.md

jamesdaniels chore(): release @angular/fire 5.0 (#1854)

91ec37e on 7 Sep, 2018

7 contributors

Raw Blame History

180 lines (136 sloc) 5.62 KB

2. Retrieving data as objects

> The `AngularFireObject` is a service for manipulating and streaming object data.

The `AngularFireObject` service is not created by itself, but through the `AngularFireDatabase` service.

The guide below demonstrates how to retrieve, save, and remove data as objects.

Injecting the `AngularFireDatabase` service

Make sure you have bootstrapped your application for AngularFire. See the Installation guide for bootstrap setup.

`AngularFireDatabase` is a service which can be injected through the constructor of your Angular component or `@Injectable()` service.

If you've followed the earlier step "Installation and Setup" your `/src/app/app.component.ts` should look like below.

```
import { Component } from '@angular/core';
import { AngularFireDatabase } from '@angular/fire/database';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.css']
})
export class AppComponent {
  items: Observable<any[]>;
  constructor(db: AngularFireDatabase) {
    this.items = db.list('items').valueChanges();
  }
}
```

In this section, we're going to modify the `/src/app/app.component.ts` to retrieve data as object.

Create an object binding

```
const relative = db.object('item').valueChanges();
```

Retrieve data

To get the object in realtime, create an object binding as a property of your component or service.

Then in your template, you can use the `async` pipe to unwrap the binding.

```
import { Component } from '@angular/core';
import { AngularFireDatabase } from '@angular/fire/database';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  template: `
  <h1>{{ (item | async)?.name }}</h1>
  `,
})
export class AppComponent {
  item: Observable<any>;
  constructor(db: AngularFireDatabase) {
    this.item = db.object('item').valueChanges();
  }
}
```

Saving data

API Summary

The table below highlights some of the common methods on the `AngularFireObject`.

method	
`set(value: T)`	Replaces the current value in the database with the new value specified as the parameter. This is called a **destructive** update, because it deletes everything currently in place and saves the new value.
`update(value: T)`	Updates the current value with in the database with the new value specified as the parameter. This is called a **non-destructive** update, because it only updates the values specified.
`remove()`	Deletes all data present at that location. Same as calling `set(null)`.

Returning promises

Each data operation method in the table above returns a promise. However, you should rarely need to use the completion promise to indicate success, because the realtime database keeps the object in sync.

The promise can be useful to chain multiple operations, catching possible errors from security rules denials, or for debugging.

```
const promise = db.object('item').remove();
promise
  .then(_ => console.log('success'))
  .catch(err => console.log(err, 'You dont have access!'));
```

Saving data

Use the `set()` method for **destructive updates**.

```
const itemRef = db.object('item');
itemRef.set({ name: 'new name!'});
```

Updating data

Use the `update()` method for **non-destructive updates**.

```
const itemRef = db.object('item');
itemRef.update({ age: newAge });
```

Only objects are allowed for updates, not primitives. This is because using an update with a primitive is the exact same as doing a `.set()` with a primitive.

Deleting data

Use the `remove()` method to remove data at the object's location.

```
const itemRef = db.object('item');
itemRef.remove();
```

Example app:

```
import { Component } from '@angular/core';
import { AngularFireDatabase, AngularFireObject } from '@angular/fire/database';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  template: `
  <h1>{{ item | async | json }}</h1>
  <input type="text" #newname placeholder="Name" />
  <input type="text" #newsize placeholder="Size" />
  <br />
  <button (click)="save(newname.value)">Set Name</button>
  <button (click)="update(newsize.value)">Update Size</button>
  <button (click)="delete()">Delete</button>
  `,
})
export class AppComponent {
  itemRef: AngularFireObject<any>;
  item: Observable<any>;
  constructor(db: AngularFireDatabase) {
    this.itemRef = db.object('item');
    this.item = this.itemRef.valueChanges();
  }
  save(newName: string) {
    this.itemRef.set({ name: newName });
  }
  update(newSize: string) {
    this.itemRef.update({ size: newSize });
  }
  delete() {
    this.itemRef.remove();
  }
}
```

Retrieving the snapshot

AngularFire `valueChanges()` unwraps the Firebase DataSnapshot by default, but you can get the data as the original snapshot by using the `snapshotChanges()` option.

```
this.itemRef = db.object('item');
this.itemRef.snapshotChanges().subscribe(action => {
  console.log(action.type);
  console.log(action.key)
  console.log(action.payload.val())
});
```

Querying?

Because `AngularFireObject` synchronizes objects from the realtime database, sorting will have no effect for queries that are not also limited by a range. For example, when paginating you would provide a query with a sort and filter. Both the sort operation and the filter operation affect which subset of the data is returned by the query; however, because the resulting object is simply json, the sort order will not be preseved locally. Hence, for operations that require sorting, you are probably looking for a list

Next Step: Retrieving data as lists