State Management in Angular Using Firebase

JOAQUIN CID
Joaquin is a full-stack and hybrid mobile app developer with over 12 years of experience working for companies like WebMD and Getty Images.

State management is a very important piece of architecture to consider when developing a web app.

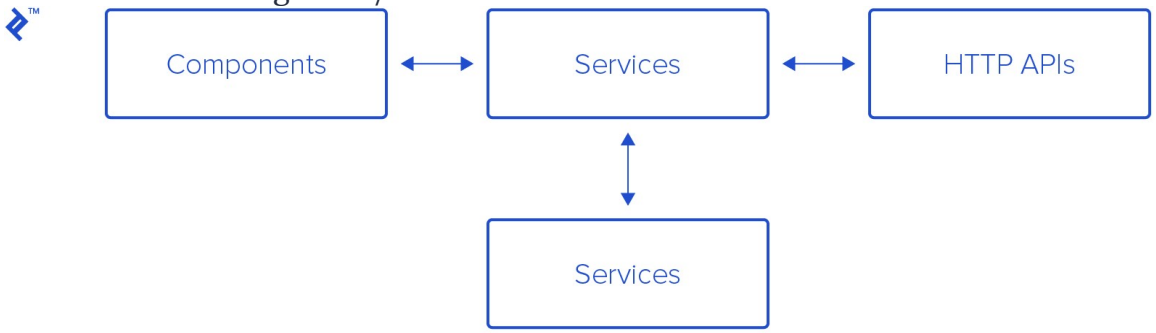
In this tutorial, we'll go over a simple approach to manage state in an Angular application that uses Firebase as its back end.

We'll go over some concepts such as state, stores, and services. Hopefully, this will help you get a better grasp of these terms and also better understand other state management libraries such as NgRx and NgXs.

We'll build an employee admin page in order to cover some different state management scenarios and the approaches that can handle them.

Components, Services, Firestore, and State Management in Angular

On a typical Angular application we have components and services. Usually, components will serve as the view template. Services will contain business logic and/or communicate with external APIs or other services to complete actions or retrieve data.



Components will usually display data and allow users to interact with the app to execute actions. While doing this, data may change and the app reflects those changes by updating the view.

Angular's change detection engine takes care of checking when a value in a component bound to the view has changed and updates the view accordingly.

As the app grows, we'll start having more and more components and services. Often understanding how data is changing and tracking where that happens can be tricky.

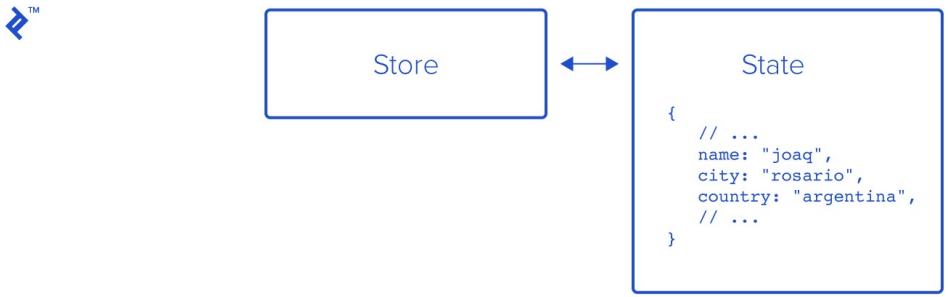
Angular and Firebase

When we use Firebase as our back end, we are provided with a really neat API that contains most of the operations and functionality we need to build a real-time application.
`@angular/fire` is the official Angular Firebase library. It's a layer on top of the Firebase JavaScript SDK library that simplifies the use of the Firebase SDK in an Angular app. It provides a nice fit with Angular good practices such as using Observables for getting and displaying data from Firebase to our components.

Stores and State

We can think of "state" as being the values displayed at any given point in time in the app. The store is simply the holder of that application state.

State can be modeled as a single plain object or a series of them, reflecting the values of the application.



Angular/Firebase Sample App

Let's build it: First, we'll create a basic app scaffold using Angular CLI, and connect it with a Firebase project.

```
$ npm install -g @angular/cli
$ ng new employees-admin`
```

```
Would you like to add Angular routing? Yes
Which stylesheet format would you like to use? SCSS
```

```
$ cd employees-admin/
$ npm install bootstrap # We'll add Bootstrap for the UI
```

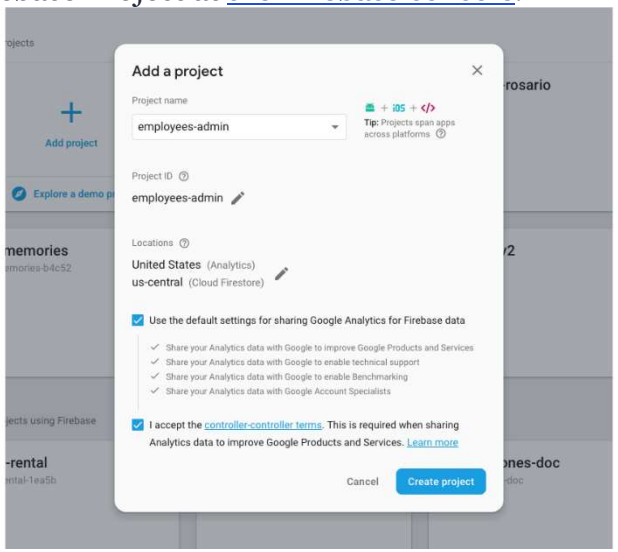
And, on `styles.scss`:

```
// ...
@import "~bootstrap/scss/bootstrap";
```

Next, we'll install `@angular/fire`:

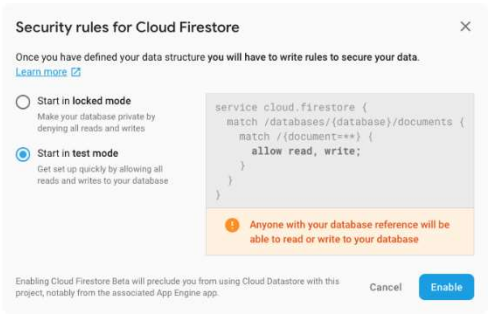
```
npm install firebase @angular/fire
```

Now, we'll create a Firebase Project at the Firebase console.

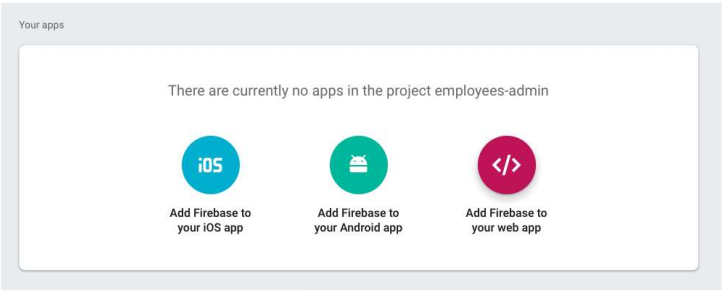


Then we're ready to create a Firestore database.

For this tutorial, I'll start in test mode. If you plan to release to production, you should enforce rules to forbid inappropriate access.



Go to Project Overview → Project Settings, and copy the Firebase web config to your local `environments/environment.ts`.




```
export const environment = {
    production: false,
        firebase: {
        apiKey: "<api-key>",
        authDomain: "<auth-domain>",
        databaseURL: "<database-url>",
        projectId: "<project-id>",
        storageBucket: "<storage-bucket>",
        messagingSenderId: "<messaging-sender-id>"
    }
};
```

At this point, we have the basic scaffold in place for our app. If we `ng serve`, we'll get:

Welcome to employees-admin!

Here are some links to help you start:

- Tour of Heroes
- CLI Documentation
- Angular blog

Firestore and Store Base Classes

We'll create two generic abstract classes, which we'll then type and extend from to build our services.

Generics let you write behavior without a bound type. This adds reusability and flexibility to your code.

Generic Firestore Service

In order to take advantage of TypeScript generics, what we'll do is create a base generic wrapper for the `@angular/fire` `firestore` service.

Let's create `app/core/services/firestore.service.ts`.

Here's the code:

```
import { Inject } from "@angular/core";
import { AngularFirestore, QueryFn } from "@angular/fire/firestore";
import { Observable } from "rxjs";
import { tap } from "rxjs/operators";
import { environment } from "src/environments/environment";

    export abstract class FirestoreService<T> {

    protected abstract basePath: string;

    constructor(
        @Inject(AngularFirestore) protected firestore: AngularFirestore,
    ) {

    }

    doc$(id: string): Observable<T> {
        return this.firestore.doc<T>(`${this.basePath}/${id}`).valueChanges().pipe(
            tap(r => {
                if (!environment.production) {
```

```
                    console.groupCollapsed(`Firestore Streaming [${this.basePath}] [doc$]
${id}`)
                    console.log(r)
                    console.groupEnd()
                }
            }),
        );
    }

    collection$(queryFn?: QueryFn): Observable<T[]> {
        return this.firestore.collection<T>(`${this.basePath}`, queryFn).valueChanges().pipe(
            tap(r => {
                if (!environment.production) {
                    console.groupCollapsed(`Firestore Streaming [${this.basePath}]
[collection$]`)
                    console.table(r)
                    console.groupEnd()
                }
            }),
        );
    }

    create(value: T) {
        const id = this.firestore.createId();
        return this.collection.doc(id).set(Object.assign({}, { id }, value)).then(_ => {
            if (!environment.production) {
```

```
                console.groupCollapsed(`Firestore Service [${this.basePath}] [create]`)
                console.log('[Id]', id, value)
                console.groupEnd()
            }
        })
    }

    delete(id: string) {
        return this.collection.doc(id).delete().then(_ => {
            if (!environment.production) {
                console.groupCollapsed(`Firestore Service [${this.basePath}] [delete]`)
                console.log('[Id]', id)
                console.groupEnd()
            }
        })
    }

    private get collection() {
        return this.firestore.collection(`${this.basePath}`);
    }
}
```

This `abstract class` will work as a generic wrapper for our Firestore services.

This should be the only place where we should inject `AngularFirestore`. This will minimize the impact when the `@angular/fire` library gets updated. Also, if at some point we want to change the library, we will only need to update this class.

I added `doc$`, `collection$`, `create`, and `delete`. They wrap `@angular/fire`'s methods and provide logging when Firebase streams data—this will become very handy for debugging—and after an object is created or deleted.

Generic Store Service

Our generic store service will be built using RxJS' `BehaviorSubject`. `BehaviorSubject` lets subscribers get the last emitted value as soon they subscribe. In our case, this is helpful because we'll be able to begin the store with an initial value for all our components when they subscribe to the store.
The store will have two methods, `patch` and `set`. (We'll create `get` methods later.)
Let's create `app/core/services/store.service.ts`:

```
import { BehaviorSubject, Observable } from 'rxjs';
import { environment } from 'src/environments/environment';

export abstract class StoreService<T> {

    protected bs: BehaviorSubject<T>;
    state$: Observable<T>;
    state: T;
    previous: T;

    protected abstract store: string;

    constructor(initialValue: Partial<T>) {
        this.bs = new BehaviorSubject<T>(initialValue as T);
        this.state$ = this.bs.asObservable();
```

```
        this.state = initialValue as T;
        this.state$.subscribe(s => {
            this.state = s
        })
    }

    patch(newValue: Partial<T>, event: string = "Not specified") {
        this.previous = this.state
        const newState = Object.assign({}, this.state, newValue);
        if (!environment.production) {
            console.groupCollapsed(`[${this.store} store] [patch] [event: ${event}]`)
            console.log("change", newValue)
            console.log("prev", this.previous)
            console.log("next", newState)
            console.groupEnd()
        }
        this.bs.next(newState)
    }

    set(newValue: Partial<T>, event: string = "Not specified") {
        this.previous = this.state
        const newState = Object.assign({}, newValue) as T;
        if (!environment.production) {
            console.groupCollapsed(`[${this.store} store] [set] [event: ${event}]`)
            console.log("change", newValue)
            console.log("prev", this.previous)
```

```
            console.log("next", newState)
            console.groupEnd()
        }
        this.bs.next(newState)
    }
}
```

As a generic class, we'll defer typing until it's properly extended.

The constructor will receive the initial value of type `Partial<T>`. This will allow us to only apply values to some properties of the state. The constructor will also subscribe to the internal `BehaviorSubject` emissions and keep the internal state updated after every change.
`patch()` will receive the `newValue` of type `Partial<T>` and will merge it with the current `this.state` value of the store. Finally, we `next()` the `newState` and emit the new state to all of the store subscribers.
`set()` works very similarly, only that instead of patching the state value, it will set it to the `newValue` it received.
We'll log the previous and next values of the state as changes occur, which will help us debug and easily track state changes.

Putting It All Together

Okay, let's see all this in action. What we'll do is create an employees page, which will contain a list of employees, plus a form to add new employees.

Let's update `app.component.html` to add a simple navigation bar:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light mb-3">
```

```
    <span class="navbar-brand mb-0 h1">Angular + Firebase + State Management</span>
    <ul class="navbar-nav mr-auto">
      <li class="nav-item" [routerLink]="['/employees']" routerLinkActive="active">
        <a class="nav-link">Employees</a>
      </li>
    </ul>
</nav>
<router-outlet></router-outlet>
```

Next, we'll create a Core module:

```
ng g m Core
```

In `core/core.module.ts`, we'll add the modules required for our app:

```
// ...
import { AngularFireModule } from '@angular/fire'
import { AngularFirestoreModule } from '@angular/fire/firestore'
import { environment } from 'src/environments/environment';
import { ReactiveFormsModule } from '@angular/forms'

@NgModule({
    // ...
  imports: [
    // ...
    AngularFireModule.initializeApp(environment.firebase),
    AngularFirestoreModule,
    ReactiveFormsModule,
  ],
  exports: [
```

```
        CommonModule,
        AngularFireModule,
        AngularFirestoreModule,
        ReactiveFormsModule
    ]
})
export class CoreModule { }
```

Now, let's create the employees page, starting with the Employees module:

```
ng g m Employees --routing
```

In `employees-routing.module.ts`, let's add the `employees` route:

```
// ...
import { EmployeesPageComponent } from './components/employees-page/employees-page.component';

// ...
const routes: Routes = [
    { path: 'employees', component: EmployeesPageComponent }
];
// ...
```

And in `employees.module.ts`, we'll import `ReactiveFormsModule`:

```
// ...
import { ReactiveFormsModule } from '@angular/forms';
// ...

@NgModule({
    // ...
```

```
    imports: [
        // ...
        ReactiveFormsModule
    ]
})
export class EmployeesModule { }
```

Now, let's add these two modules in the `app.module.ts` file:

```
// ...
import { EmployeesModule } from './employees/employees.module';
import { CoreModule } from './core/core.module';

imports: [
    // ...
    CoreModule,
    EmployeesModule
],
```

Finally, let's create the actual components of our employees page, plus the corresponding model, service, store, and state.

```
ng g c employees/components/EmployeesPage
ng g c employees/components/EmployeesList
ng g c employees/components/EmployeesForm
```

For our model, we'll need a file called `models/employee.ts`:

```
export interface Employee {
    id: string;
    name: string;
```

```
    location: string;
    hasDriverLicense: boolean;
}
```

Our service will live in a file called `employees/services/employee.firestore.ts`. This service will extend the generic `FirestoreService<T>` created before, and we'll just set the `basePath` of the Firestore collection:

```
import { Injectable } from '@angular/core';
import { FirestoreService } from 'src/app/core/services/firestore.service';
import { Employee } from '../models/employee';

@Injectable({
    providedIn: 'root'
})
export class EmployeeFirestore extends FirestoreService<Employee> {

    protected basePath: string = 'employees';

}
```

Then we'll create the file `employees/states/employees-page.ts`. This will serve as the state of the employees page:

```
import { Employee } from '../models/employee';
export interface EmployeesPage {

    loading: boolean;
    employees: Employee[];
    formStatus: string;

}
```

The state will have a `loading` value that determines whether to display a loading message on the page, the `employees` themselves, and a `formStatus` variable to handle the status of the form (e.g. `Saving` or `Saved`.)
We'll need a file at `employees/services/employees-page.store.ts`. Here we'll extend the `StoreService<T>` created before. We'll set the store name, which will be used to identify it when debugging.
This service will initialize and hold the state of the employees page. Note that the constructor calls `super()` with the initial state of the page. In this case, we'll initialize the state with `loading=true` and an empty array of employees.

```
import { EmployeesPage } from '../states/employees-page';
import { StoreService } from 'src/app/core/services/store.service';
import { Injectable } from '@angular/core';

@Injectable({
    providedIn: 'root'
})
export class EmployeesPageStore extends StoreService<EmployeesPage> {
    protected store: string = 'employees-page';

    constructor() {
        super({
            loading: true,
            employees: [],
        })
    }
}
```

Now let's create `EmployeesService` to integrate `EmployeeFirestore` and `EmployeesPageStore`:

```
ng g s employees/services/Employees
```

Note that we are injecting the `EmployeeFirestore` and `EmployeesPageStore` in this service. This means that the `EmployeesService` will contain and coordinate calls to Firestore and the store to update the state. This will help us create a single API for components to call.

```
import { EmployeesPageStore } from './employees-page.store';
import { EmployeeFirestore } from './employee.firestore';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Employee } from '../models/employee';
import { tap, map } from 'rxjs/operators';

@Injectable({
    providedIn: 'root'
})
export class EmployeesService {

    constructor(
        private firestore: EmployeeFirestore,
        private store: EmployeesPageStore
    ) {
        this.firestore.collection$().pipe(
            tap(employees => {
                this.store.patch({
                    loading: false,
                    employees,
                }, `employees collection subscription`)
            })
```

```
        ).subscribe()
    }

    get employees$(): Observable<Employee[]> {
        return this.store.state$.pipe(map(state => state.loading
            ? []
            : state.employees))
    }

    get loading$(): Observable<boolean> {
        return this.store.state$.pipe(map(state => state.loading))
    }

    get noResults$(): Observable<boolean> {
        return this.store.state$.pipe(
            map(state => {
                return !state.loading
                    && state.employees
                    && state.employees.length === 0
            })
        )
    }

    get formStatus$(): Observable<string> {
        return this.store.state$.pipe(map(state => state.formStatus))
    }
```

```
    create(employee: Employee) {
        this.store.patch({
            loading: true,
            employees: [],
            formStatus: 'Saving...'
        }, "employee create")
        return this.firestore.create(employee).then(_ => {
            this.store.patch({
                formStatus: 'Saved!'
            }, "employee create SUCCESS")
            setTimeout(() => this.store.patch({
                formStatus: ''
            }, "employee create timeout reset formStatus"), 2000)
        }).catch(err => {
            this.store.patch({
                loading: false,
                formStatus: 'An error ocurred'
            }, "employee create ERROR")
        })
    }

    delete(id: string): any {
        this.store.patch({ loading: true, employees: [] }, "employee delete")
        return this.firestore.delete(id).catch(err => {
            this.store.patch({
```

```
                loading: false,
                formStatus: 'An error ocurred'
            }, "employee delete ERROR")
        })
    }
}
```

Let's take a look at how the service will work.

In the constructor, we'll subscribe to the Firestore employees collection. As soon as Firestore emits data from the collection, we'll update the store, setting `loading=false` and `employees` with Firestore's returned collection. Since we have injected `EmployeeFirestore`, the objects returned from Firestore are typed to `Employee`, which enables more IntelliSense features.

This subscription will be alive while the app is active, listening for all changes and updating the store every time Firestore streams data.

```
this.firestore.collection$().pipe(
    tap(employees => {
        this.store.patch({
        loading: false,
        employees,
        }, `employees collection subscription`)
    })
).subscribe()
```

The `employees$()` and `loading$()` functions will select the piece of state we want to later use on the component. `employees$()` will return an empty array when the state is loading. This will allow us to display proper messaging on the view.

```
get employees$(): Observable<Employee[]> {
    return this.store.state$.pipe(map(state => state.loading ? [] : state.employees))
}

get loading$(): Observable<boolean> {
    return this.store.state$.pipe(map(state => state.loading))
}
```

Okay, so now we have all the services ready, and we can build our view components. But before we do that, a quick refresher might come in handy...

RxJs Observables and the `async` Pipe

Observables allow subscribers to receive emissions of data as a stream. This, in combination with the `async` pipe, can very powerful. The `async` pipe takes care of subscribing to an Observable and updating the view when new data is emitted. More importantly, it automatically unsubscribes when the component is destroyed, protecting us from memory leaks.
You can read more about Observables and RxJs library in general in the official docs.

Creating the View Components

In `employees/components/employees-page/employees-page.component.html`, we'll put this code:

```
<div class="container">
    <div class="row">
```

```
        <div class="col-12 mb-3">
            <h4>
                Employees
            </h4>
        </div>
    </div>
    <div class="row">
        <div class="col-6">
            <app-employees-list></app-employees-list>
        </div>
        <div class="col-6">
            <app-employees-form></app-employees-form>
        </div>
    </div>
</div>
```

Likewise, `employees/components/employees-list/employees-list.component.html` will have this, using the `async` pipe technique mentioned above:

```
<div *ngIf="loading$ | async">
    Loading...
</div>
<div *ngIf="noResults$ | async">
    No results
</div>
<div class="card bg-light mb-3" style="max-width: 18rem;" *ngFor="let employee of employees$ |
async">
    <div class="card-header">{{employee.location}}</div>
    <div class="card-body">
        <h5 class="card-title">{{employee.name}}</h5>
        <p class="card-text">{{employee.hasDriverLicense ? 'Can drive': ''}}</p>
        <button (click)="delete(employee)" class="btn btn-danger">Delete</button>
    </div>
```

```
</div>
```

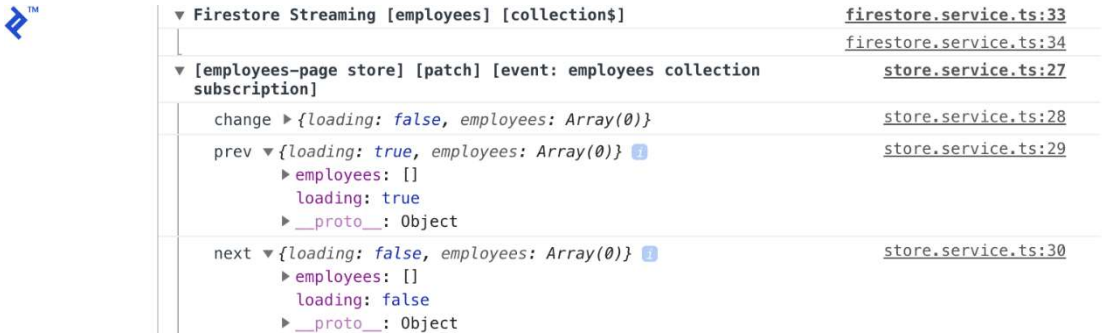
But in this case we'll need some TypeScript code for the component, too. The file `employees/components/employees-list/employees-list.component.ts` will need this:

```
import { Employee } from '../../models/employee';
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { EmployeesService } from '../../services/employees.service';

@Component({
    selector: 'app-employees-list',
    templateUrl: './employees-list.component.html',
    styleUrls: ['./employees-list.component.scss']
})
export class EmployeesListComponent implements OnInit {
    loading$: Observable<boolean>;
    employees$: Observable<Employee[]>;
    noResults$: Observable<boolean>;

    constructor(
        private employees: EmployeesService
    ) {}

    ngOnInit() {
        this.loading$ = this.employees.loading$;
        this.noResults$ = this.employees.noResults$;
        this.employees$ = this.employees.employees$;
```

```
    }

    delete(employee: Employee) {
        this.employees.delete(employee.id);
    }

}
```

So, going to the browser, what we'll have now is:

And the console will have the following output:



Looking at this, we can tell that Firestore streamed the `employees` collection with empty values, and the `employees-page` store was patched, setting `loading` from `true` to `false`.

OK, let's build the form to add new employees to Firestore:

The Employees Form

In `employees/components/employees-form/employees-form.component.html` we'll add this code:

```
<form [formGroup]="form" (ngSubmit)="submit()">
    <div class="form-group">
        <label for="name">Name</label>
        <input type="string" class="form-control" id="name"
         formControlName="name" [class.is-invalid]="isInvalid('name')">
        <div class="invalid-feedback">
            Please enter a Name.
        </div>
    </div>
    <div class="form-group">
        <select class="custom-select" formControlName="location"
         [class.is-invalid]="isInvalid('location')">
            <option value="" selected>Choose location</option>
            <option *ngFor="let loc of locations" [ngValue]="loc">{{loc}}</option>
        </select>
        <div class="invalid-feedback">
            Please select a Location.
        </div>
    </div>
    <div class="form-group form-check">
        <input type="checkbox" class="form-check-input" id="hasDriverLicense"
         formControlName="hasDriverLicense">
        <label class="form-check-label" for="hasDriverLicense">Has driver license</label>
    </div>
    <button [disabled]="form.invalid" type="submit" class="btn btn-primary d-inline">Add</button>
    <span class="ml-2">{{ status$ | async }}</span>
```

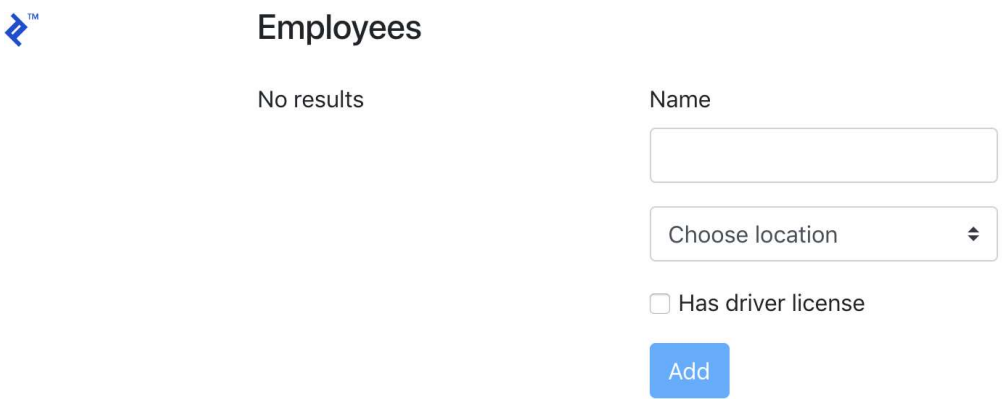
```
</form>
```

The corresponding TypeScript code will live in `employees/components/employees-form/employees-form.component.ts`:

```
import { EmployeesService } from './../../services/employees.service';
import { AngularFirestore } from '@angular/fire/firestore';
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
import { Observable } from 'rxjs';

@Component({
    selector: 'app-employees-form',
    templateUrl: './employees-form.component.html',
    styleUrls: ['./employees-form.component.scss']
})
export class EmployeesFormComponent implements OnInit {

    form: FormGroup = new FormGroup({
        name: new FormControl('', Validators.required),
        location: new FormControl('', Validators.required),
        hasDriverLicense: new FormControl(false)
    });

    locations = [
        'Rosario',
        'Buenos Aires',
        'Bariloche'
    ]
```

```
    status$: Observable < string > ;

    constructor (
        private employees: EmployeesService
    ) {}

    ngOnInit() {
        this.status$ = this.employees.formStatus$;
    }

    isInvalid(name) {
        return this.form.controls[name].invalid
            && (this.form.controls[name].dirty || this.form.controls[name].touched)
    }

    async submit() {
        this.form.disable()
        await this.employees.create({ ...this.form.value
        })
        this.form.reset()
        this.form.enable()
    }

}
```

The form will call the `create()` method of `EmployeesService`. Right now the page looks like this:



Let's take a look at what happens when we add a new employee.

Adding a New Employee

After adding a new employee, we'll see the following get output to the console:

These are all the events that get triggered when adding a new employee. Let's take a closer look.

When we call `create()` we'll execute the following code, setting `loading=true`, `formStatus='Saving...'` and the `employees` array to empty (`(1)` in the above image).

```
this.store.patch({
    loading: true,
    employees: [],
    formStatus: 'Saving...'
```

```
    }, "employee create")
    return this.firestore.create(employee).then(_ => {
      this.store.patch({
        formStatus: 'Saved!'
      }, "employee create SUCCESS")
      setTimeout(() => this.store.patch({
        formStatus: ''
      }, "employee create timeout reset formStatus"), 2000)
    }).catch(err => {
      this.store.patch({
        loading: false,
        formStatus: 'An error ocurred'
      }, "employee create ERROR")
    })
```

Next, we are calling the base Firestore service to create the employee, which logs `(4)`. On the promise callback, we set `formStatus='Saved!'` and log `(5)`. Finally, we set a timeout to set `formStatus` back to empty, logging `(6)`.

Log events `(2)` and `(3)` are the events triggered by the Firestore subscription to the employees collection. When the `EmployeesService` is instantiated, we subscribe to the collection and receive the collection upon every change that happens. This sets a new state to the store with `loading=false` by setting the `employees` array to the employees coming from Firestore.

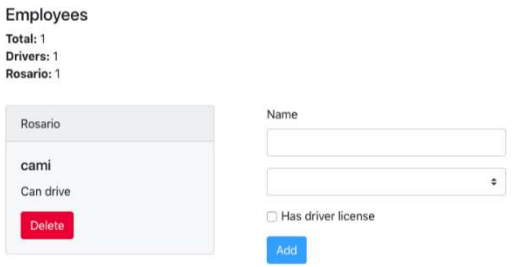
If we expand the log groups, we'll see detailed data of every event and update of the store, with the previous value and next, which is useful for debugging.

```
▼ [employees-page store] [patch] [event: employee create]                store.service.ts:27
    change                                                               store.service.ts:28
    ▶ {loading: true, employees: Array(0), formStatus: "Saving..."}
(1) prev ▶ {loading: false, employees: Array(0)}                         store.service.ts:29
    next                                                                 store.service.ts:30
    ▶ {loading: true, employees: Array(0), formStatus: "Saving..."}
▼ Firestore Streaming [employees] [collection$]                          firestore.service.ts:33
                                                                         firestore.service.ts:34
(2)
```

(index)	hasDriverLicense	id	location	name
0	true	"9AK24jknv90Oaa…	"Rosario"	"cami"

```
    ▶ Array(1)
▼ [employees-page store] [patch] [event: employees collection            store.service.ts:27
  subscription]
(3) change ▶ {loading: false, employees: Array(1)}                       store.service.ts:28
    prev                                                                 store.service.ts:29
    ▶ {loading: true, employees: Array(0), formStatus: "Saving..."}
    next                                                                 store.service.ts:30
    ▶ {loading: false, employees: Array(1), formStatus: "Saving..."}
▼ Firestore Service [employees] [create]                                 firestore.service.ts:45
(4) [Id] 9AK24jknv90OaaLaLQ1c                                            firestore.service.ts:46
    ▶ {name: "cami", location: "Rosario", hasDriverLicense: true}
▼ [employees-page store] [patch] [event: employee create SUCCESS]        store.service.ts:27
(5) change ▶ {formStatus: "Saved!"}                                      store.service.ts:28
    prev                                                                 store.service.ts:29
    ▶ {loading: false, employees: Array(1), formStatus: "Saving..."}
    next ▶ {loading: false, employees: Array(1), formStatus: "Saved!"}   store.service.ts:30
▼ [employees-page store] [patch] [event: employee create timeout reset   store.service.ts:27
  formStatus]
(6) change ▶ {formStatus: ""}                                            store.service.ts:28
    prev ▶ {loading: false, employees: Array(1), formStatus: "Saved!"}   store.service.ts:29
    next ▶ {loading: false, employees: Array(1), formStatus: ""}         store.service.ts:30
```

This is how the page looks like after adding a new employee:



Adding a Summary Component

Let's say we now want to display some summary data on our page. Let's say we want the total number of employees, how many are drivers, and how many are from Rosario.

We'll start by adding the new state properties to the page state model in `employees/states/employees-page.ts`:

```
// ...
export interface EmployeesPage {

    loading: boolean;
    employees: Employee[];
    formStatus: string;

    totalEmployees: number;
    totalDrivers: number;
    totalRosarioEmployees: number;

}
```

And we'll initialize them in the store in `employees/services/emplyees-page.store.ts`:

```
// ...
constructor() {
    super({
        loading: true,
        employees: [],
        totalDrivers: 0,
        totalEmployees: 0,
        totalRosarioEmployees: 0
    })
```

```
}
// ...
```

Next, we'll calculate the values for the new properties and add their respective selectors in the `EmployeesService`:

```
// ...

this.firestore.collection$().pipe(
    tap(employees => {
        this.store.patch({
            loading: false,
            employees,
            totalEmployees: employees.length,
            totalDrivers: employees.filter(employee => employee.hasDriverLicense).length,
            totalRosarioEmployees: employees.filter(employee => employee.location ===
'Rosario').length,
        }, `employees collection subscription`)
    })
).subscribe()

// ...

get totalEmployees$(): Observable<number> {
    return this.store.state$.pipe(map(state => state.totalEmployees))
}

get totalDrivers$(): Observable<number> {
    return this.store.state$.pipe(map(state => state.totalDrivers))
}
```

```
get totalRosarioEmployees$(): Observable < number > {
    return this.store.state$.pipe(map( state => state.totalRosarioEmployees))
}

// ...
```

Now, let's create the summary component:

```
ng g c employees/components/EmployeesSummary
```

We'll put this in `employees/components/employees-summary/employees-summary.html`:

```
<p>
    <span class="font-weight-bold">Total:</span> {{total$ | async}} <br>
    <span class="font-weight-bold">Drivers:</span> {{drivers$ | async}} <br>
    <span class="font-weight-bold">Rosario:</span> {{rosario$ | async}} <br>
</p>
```

And in `employees/components/employees-summary/employees-summary.ts`:

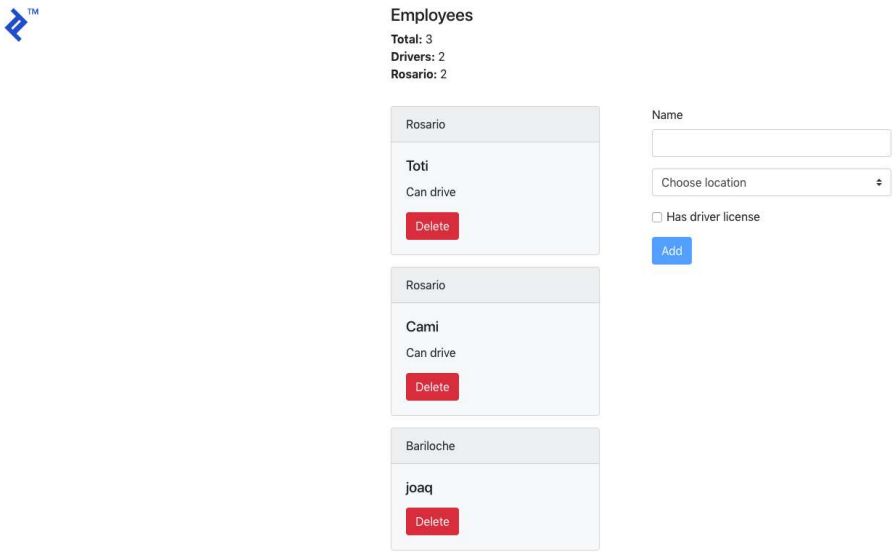
```
import { Component, OnInit } from '@angular/core';
import { EmployeesService } from '../../services/employees.service';
import { Observable } from 'rxjs';

@Component({
    selector: 'app-employees-summary',
    templateUrl: './employees-summary.component.html',
    styleUrls: ['./employees-summary.component.scss']
})
export class EmployeesSummaryComponent implements OnInit {
```

```
    total$: Observable < number > ;
    drivers$: Observable < number > ;
    rosario$: Observable < number > ;

    constructor (
        private employees: EmployeesService
    ) {}

    ngOnInit() {
        this.total$ = this.employees.totalEmployees$;
        this.drivers$ = this.employees.totalDrivers$;
        this.rosario$ = this.employees.totalRosarioEmployees$;
    }

}
```

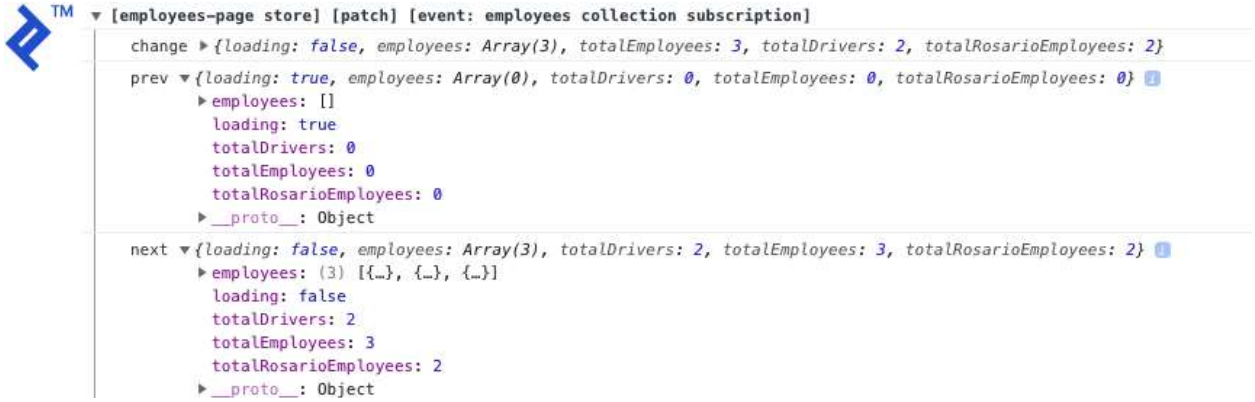
We'll then add the component to `employees/employees-page/employees-page.component.html`:

```
// ...
<div class="col-12 mb-3">
    <h4>
        Employees
    </h4>
    <app-employees-summary></app-employees-summary>
</div>
// ...
```

The result is the following:



In the console we have:



The employees service calculates the total `totalEmployees`, `totalDrivers`, and `totalRosarioEmployees` on each emission and updates the state.

The full code of this tutorial is available on GitHub, and there's also a live demo.

Managing Angular App State Using Observables… Check!

In this tutorial, we covered a simple approach for managing state in Angular apps using a Firebase back end.

This approach fits nicely with the Angular guidelines of using Observables. It also facilitates debugging by providing tracking for all updates to the app's state.

The generic store service can also be used to manage state of apps that don't use Firebase features, either to manage only the app's data or data coming from other APIs.

But before you go applying this indiscriminately, one thing to consider is that `EmployeesService` subscribes to Firestore on the constructor and keeps listening while the app is active. This might be useful if we use the employees list on multiple pages on the app, to avoid getting data from Firestore when navigating between pages.
But this might not be the best option in other scenarios like if you just need to pull initial values once and then manually trigger reloads of data from Firebase. The bottom line is, it's always important to understand your app's requirements in order to choose better methods of implementation.