Published in Better Programming

Estefanía García Gallardo Follow

Jul 21, 2022 · 14 min read · ✦ · Listen

Save

Angular 14 Firebase CRUD Tutorial With AngularFire 7

Learn how to implement a CRUD Pokedex with Angular and Firebase

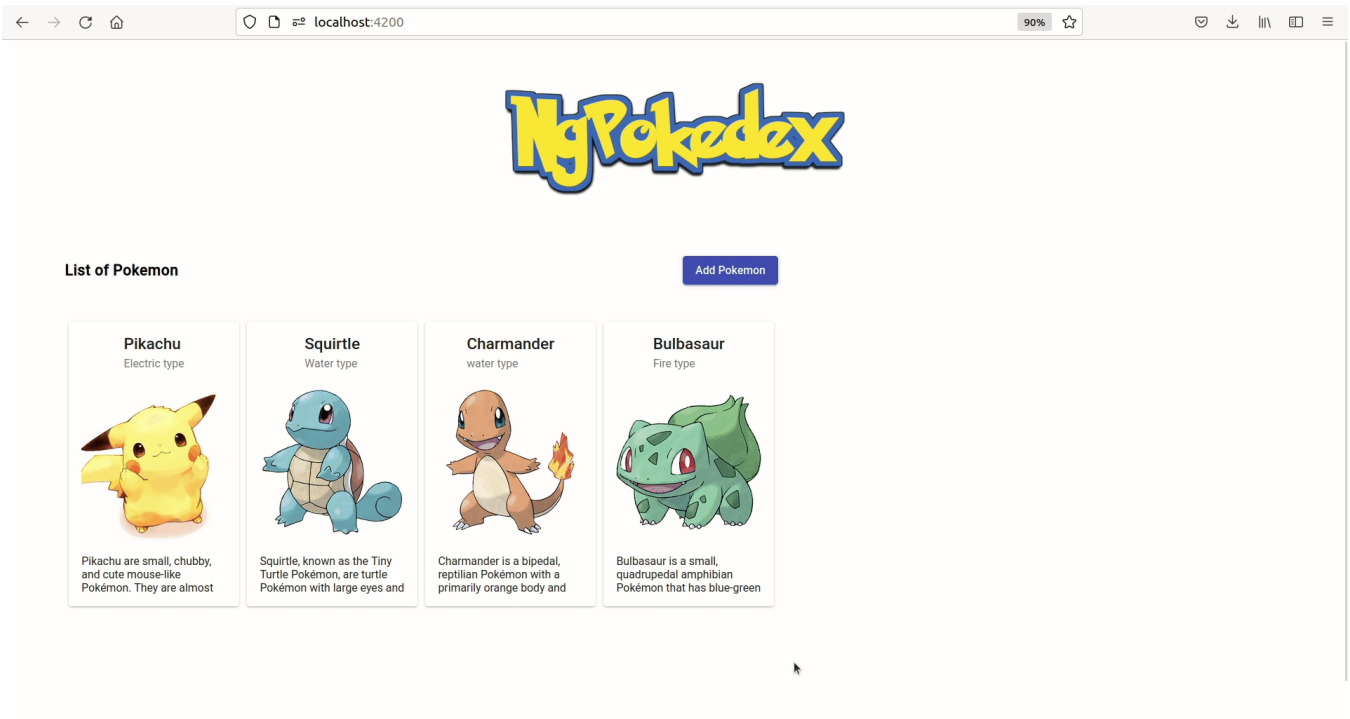
Photo by Thimo Pedersen on Unsplash

Introduction

In this tutorial, you will learn how to create a Pokedex CRUD application using Angular 14, Firebase's real-time NoSQL cloud database (Firestore), and the new tree-shakable AngularFire v.7.0 modular SDK, which allows us to take full advantage of the new tree-shakable Firebase JS SDK (v9).

> *Please note that, at the time of writing this tutorial, the Angular fire v.7.0 API is still in development, and isn't feature complete; and the docs aren't yet updated to the latest version. AngularFire provides a compatibility layer that will allow you to use the latest version of the library, while fully supporting the AngularFire v6.0 API. This said, if you're as curious as I am, and enjoy trying out the latest versions of all your libraries, go ahead and enjoy this tutorial.*

Here's a preview of the app we'll be building today:



App preview

You can find the full code here.

Index

- Setting up a Firebase project.
- Setting up an Angular project with AngularFire.
- Building the App.

- Creating the Firestore service
- Creating the Pokemon module

Setting up a Firebase project

The first thing that we need to do is create a new Firebase project, add a web application to our project, and set up the Firestore. If you've already done this, go ahead to the next section.

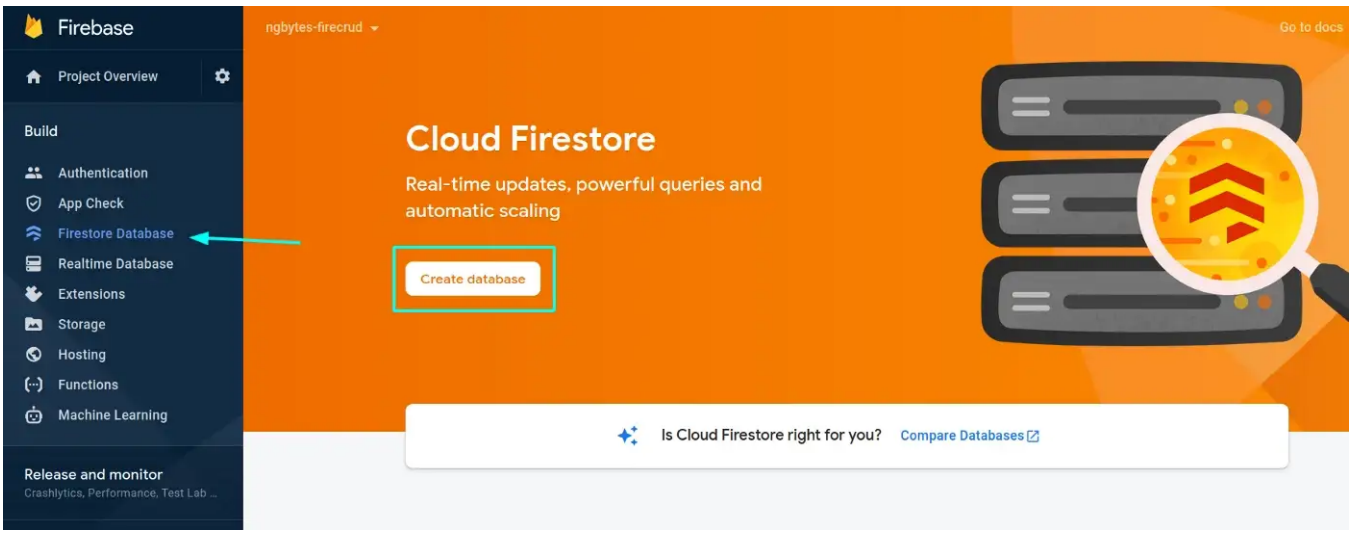
If you don't know how to do this, I recommend that you follow my How to Create and Configure a Firebase and Angular Project tutorial, and once your Firebase project is up and running, come back and continue reading.

Setting up the Firestore

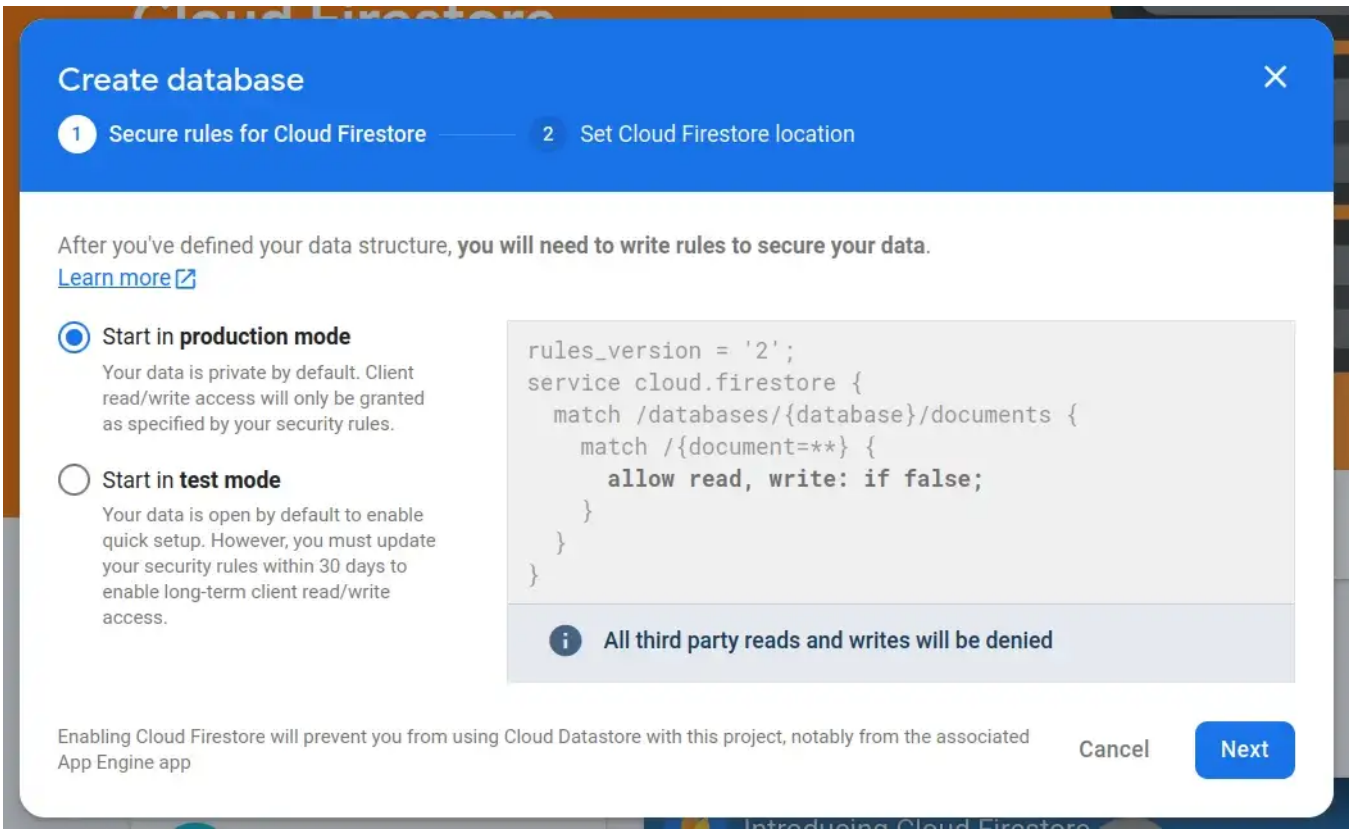
Note: In case you haven't yet added the Firestore to your Firebase project, we'll add it now. If you've already done so, feel free to skip this section.

It's time to add the Firestore to our project! The cloud Firestore is a database for mobile, web, and server development from Firebase and Google Cloud. Like Firebase Realtime Database, it keeps your data in sync across client apps through real-time listeners.

Head over to the Firestore Database page on your Firebase console, and click on the Create database button:



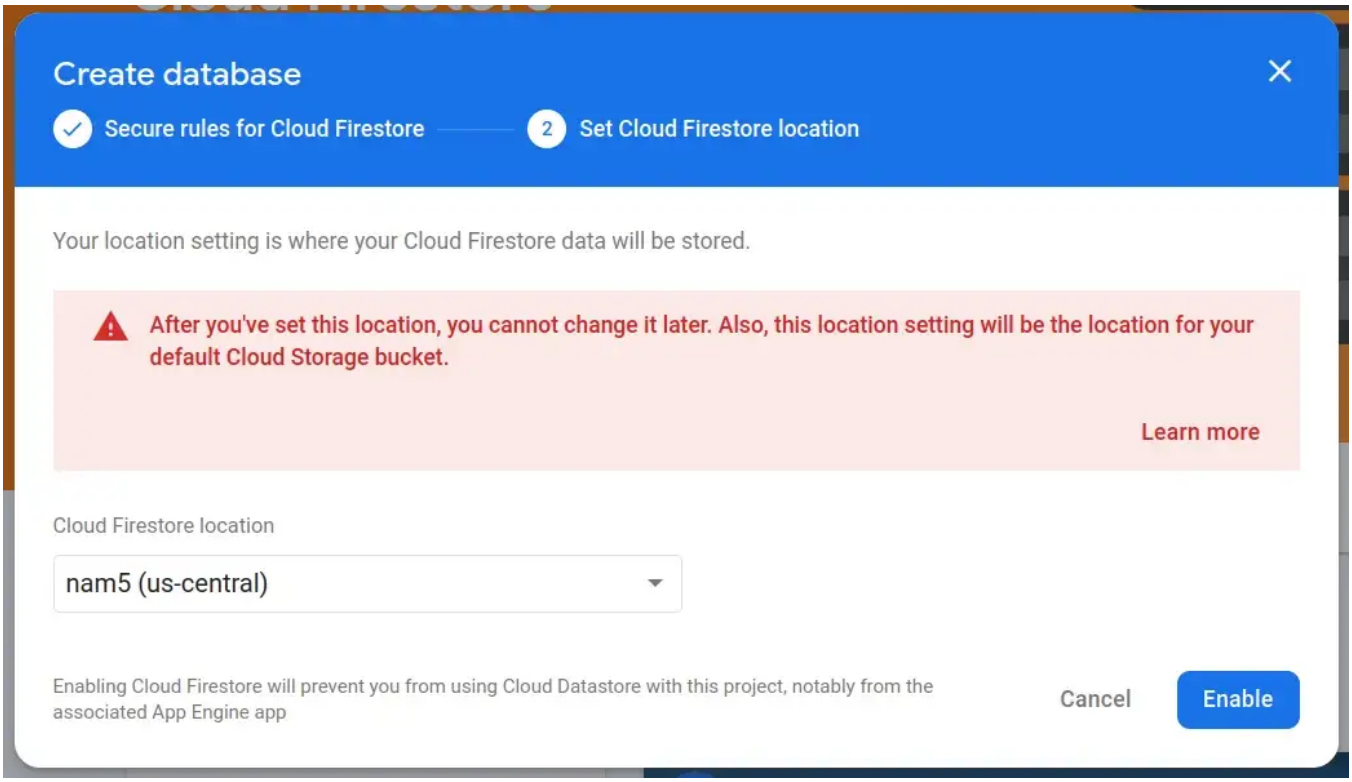
You'll have to choose whether to start in production or in test mode. Be very careful with what you choose, since starting in test mode will allow everyone to access your data. Only choose this option if you know what you're doing:



Firebase — choosing security rules

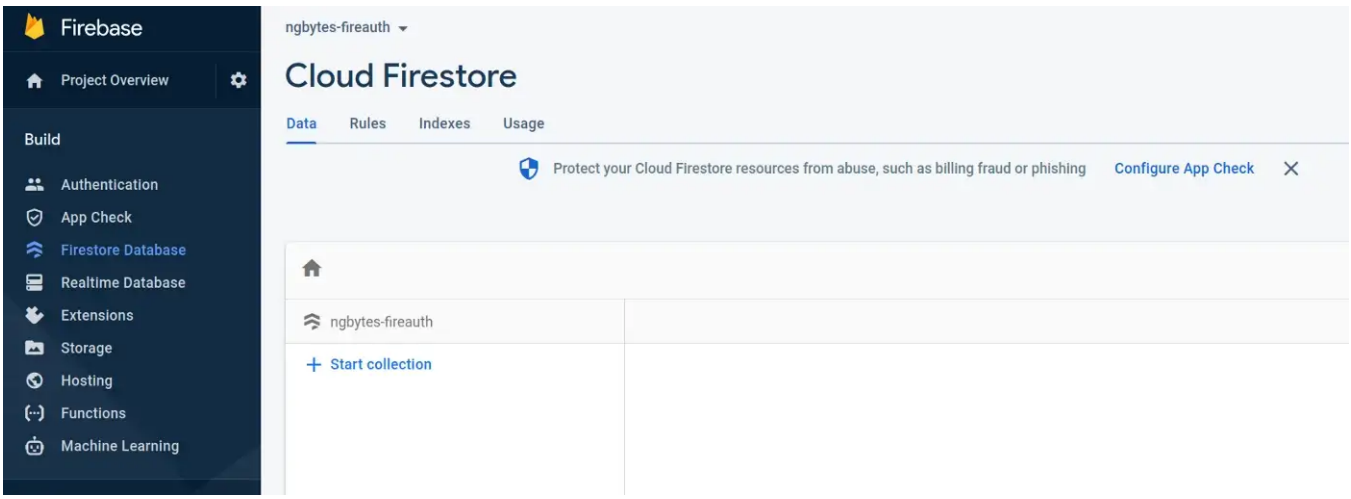
The last thing you will need to choose is the location of your Firestore data. As the warning message indicates, choose wisely, because you won't be able to change this location later.

Once you've chosen the location, click on the *enable* button, and your Firestore will be created:



Firebase — choosing the Firestore data location

Your Firestore Database page should now contain your newly created Cloud Firestore database:



Firebase — Newly created Cloud Firestore

Awesome! Our Firebase app is fully set up and ready. It's time to get started on our Angular app.

Setting up an Angular project

The first thing we're going to do is create a new project with the Angular CLI.

Tip: If you haven't installed the Angular CLI, you can do so by running the following command:

```
npm i -g @angular/cli
```

To create a new Angular project, we can run the following command:

```
ng new ngbytes-firepokedex
```

Note: Don't forget to answer yes when you're asked if you want to add routing to your new app!

Once the CLI has worked its magic, we can open the newly created project with our favorite IDE (I suggest VSCode, which is the one I normally use).

Adding Firebase and AngularFire

Let's add Firebase and Angularfire to our project. To do so, we'll use the AngularFire schematic, that will take care of setting everything up for us. Let's run the following command:

```
ng add @angular/fire
```

We'll be asked a series of questions, like which Firebase features we'd like to setup. For this tutorial, we only need to use the Firestore, so let's select that:

We'll then be asked about the Firebase account we'd like to use, and which project we want to setup. Select the project we created previously, and then select the app we also created earlier.

Once we've done all this, you will see that the schematic has taken care of all the Firebase configurations for us. Awesome!

Adding Angular Material

We'll also add Angular Material. Once again, we'll use a schematic:

Disabling strictPropertyInitialization

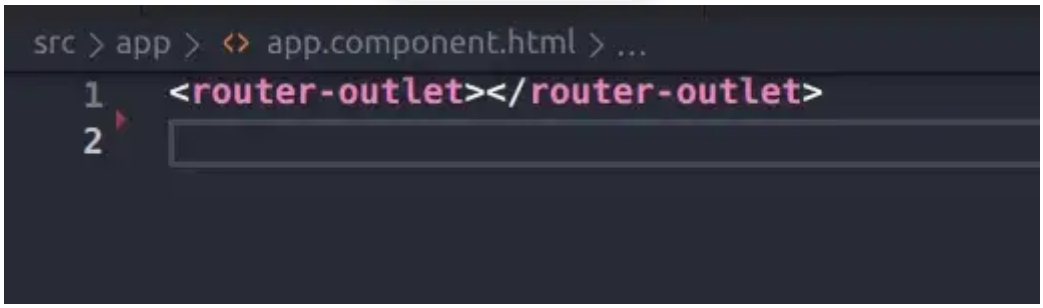
We'll need to set the `strictPropertyInitialization` property in the `tsconfig.json` file to false.

We're doing this because the strict mode is enabled by default in all new Angular apps starting from version 12, which means that TypeScript will complain if we declare any class properties without setting them in the constructor (a common practice in Angular). Here's what our `tsconfig.json` file should look like:

```
/* To learn more about this file see: https://angular.io/config/tsconfig. */
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "sourceMap": true,
    "strictPropertyInitialization": false,
    "declaration": false,
    "downlevelIteration": true,
    "experimentalDecorators": true,
    "moduleResolution": "node",
    "importHelpers": true,
    "target": "es2017",
    "module": "es2020",
    "lib": [
```

```
21        "es2018",
22        "dom"
23      ]
24    },
25    "angularCompilerOptions": {
26      "enableI18nLegacyMessageIdFormat": false,
27      "strictInjectionParameters": true,
28      "strictInputAccessModifiers": true,
29      "strictTemplates": true
30    }
31  }
```

tsconfig.json hosted with ❤️ by GitHub view raw



router-outlet tags in the app.component.html

Creating the firestore service

The first thing we're going to do is create the service that will interact with the Firestore, and provide us with the data that we'll display in our Pokedex. We'll use the traditional Angular modular structure, and create a `core` module with the Angular CLI to contain our service. In your terminal:

```
cd src/app
ng g m core
```

Once we've created our `core` module, we can create our service inside it, with the following command:

```
cd core
ng g s pokedex-firestore
```

Inside our service, the first thing that we need to do is inject the AngularFire `Firestore` instance in the constructor, which, as its name indicates, we can use to interact with the Firestore.

We'll also create a private `pokemonCollection` variable, which will contain a reference to the Firestore Pokemon collection instance. We'll use this collection later, in our queries. So far, our service should look like this:

The firestore service https://gist.github.com/NyaGarcia/5e7e372aa9ffa8812e3c7471ef4a4375

Now that we have the basic Firestore config, we can start on our CRUD functions. We'll have a total of 5 different functions:

- `getAll()` : Will return all of the Pokemon in the collection.
- `get(id)` : Will return the Pokemon that matches the id.
- `create(pokemon)` : Will add a new Pokemon to the collection.
- `update(pokemon)` : Will update a Pokemon in the collection.
- `delete(id)` : Will delete the Pokemon that matches the id.

Let's go ahead and implement these functions in our service:

Implementing the CRUD functions
https://gist.github.com/NyaGarcia/025c296fde0b93826eff6db6060cfb13

Awesome, we've finished our service! Now it's time to create the `pokemon` module, which will allow our users to interact with the service.

The Pokemon module

This feature module will contain all of the Pokemon components. The main routed component will be the Pokemon component, which will contain all of the necessary logic to interact with the Firestore service, and provide the other components with the data they need. We'll have a total of three presentational (purely visual) components:

- The Pokemon form, which we will reuse to both create and update Pokemon. We'll open the form component inside a `MatDialog`.
- The Pokemon list will display the list of Pokemon stored in the Firestore.
- The Pokemon detail will display the selected Pokemon's data and will have the update and delete buttons.

To create the module, we'll first create a `features` directory inside `app` and then use the CLI to automatically generate the module:

```
cd src/app
mkdir features
cd features
ng g m pokemon -m app --route pokemon
```

Tip: We're using the `--route` option, which creates a component in the new module, and adds the route to that component in the `Routes` array declared in the module provided in the `-m` option.

You'll see that the CLI has created a `pokemon` module inside the `features` directory, with its corresponding `pokemon-routing.module.ts` and component. It's also modified the `app-routing.module.ts`, and added a `pokemon` route, lazy loading the `PokemonModule`.

Since we want the Pokemon module to be shown as soon as we open the app (as opposed to being shown when we navigate to `/pokemon`), we need to modify the `app-routing.module.ts` file, and change the route path, like this:

Changing the pokemon route https://gist.github.com/NyaGarcia/ab68eb742d81b28fd835631622a72147

All done!

The Pokemon interface

Before we move on to creating the Pokemon form, we'll take a moment to refactor our `pokedex-firestore.service`. Wait, what? We just created the service and already we have to refactor?

Sorry folks, that's computer science for you! Remember the Pokemon interface we created inside the service? We should create an `interfaces` directory in the `pokemon.module`, and give it a new home:

```
cd features/pokemon
mkdir interfaces
cd interfaces
touch pokemon.interface.ts
```

Let's cut and paste the interface from the service into our new file:

The Pokemon service https://gist.github.com/NyaGarcia/0625bcb5c9bb215aee7dfbc5cb02483e

Awesome! Don't forget to import the interface in the service, like this:

Importing the Pokemon interface in the service
https://gist.github.com/NyaGarcia/fa8223633fb69f81de4fda33da24b10a

Congratulations, you've successfully refactored the Firestore service! We can now move on.

The Pokemon form

To be able to create and update Pokemon, we'll need to create a `form` component. First, we'll create a `components` folder inside the `pokemon` module, and then use the CLI to create our form:

```
cd features/pokemon
mkdir components
cd components
ng g c form
```

To create our form, we'll be using the Reactive Forms module. We'll also be using the Angular Material modules, to style our form. Since it isn't the goal of this tutorial, I won't go into any details about how to use reactive forms.

First, we'll import the `ReactiveFormsModule`, the `MatInputModule`, the `MatFormFieldModule`, the `MatButtonModule` and the `MatDialog` modules into our `PokemonModule`:

Importing modules https://gist.github.com/NyaGarcia/a8b4769c6fccbc1be1ea2eb743905081

Then, we'll create the form in our `form.component.ts` file:

Creating the Pokemon form https://gist.github.com/NyaGarcia/da0ed888f32cf2e58b2950247fb51ba4

As you can see, we've added basic validation for our form, making all of the form fields required. If you're going to use this code in production, you should probably add further validation, and, amongst other things, limit the length of all of the input fields.

We've also injected a `MatDialogRef`, which we can use to close the dialog in which our form is contained. When closing, we can provide an optional result value, which

we can then access in the component which opened the dialog in the first place.

The `@Inject(MAT_DIALOG_DATA)` injection token allows us to access the data that's passed to your dialog component. Why do we need this? Because we're going to reuse our form, both for creating a new Pokemon and for updating an existing one. If we want to create a new Pokemon, we won't pass any data to the dialog. If however, we want to update an existing Pokemon, we will pass a Pokemon to the dialog, and the form fields will be initialized with the Pokemon's values.

Finally, let's take a closer look at the `submit` function, more specifically, at the return value inside the `dialogRef.close()`:

```
{...this.pokemon, ...this.form.value}
```

Don't know what's going on in this line of code? Don't worry! We'll go through it together. We're basically merging the `pokemon` and the `form.value` objects into a single object, by using the spread operator (not familiar with it? Read this guide to become an expert!). We're doing this because we want to reuse the form both for creating and updating Pokemon. If we're using the form to create a pokemon, the `pokemon` object will be undefined and it won't affect the result. However, if we're using the form to update a Pokemon, the `pokemon` object will contain the `id` property, which isn't present in the `form.value` object, and it will be added to the result.

Let's implement the template in the `form.component.html` file:

Pokemon form template https://gist.github.com/NyaGarcia/cdb5d1c20c320e131ba600d4ccb22856

Last, but not least, a little CSS:

Form styling https://gist.github.com/NyaGarcia/3d685b7073340f231a57d3039c4271a8

The Pokemon list

This component will display a list of all the Pokemon in the collection. Let's create our `list` component with the Angular CLI (inside the `components` directory, don't forget!):

```
ng g c list
```

To style our list, we'll use `MatCard` components. Therefore, we'll need to import the `MatCardModule` in our `pokemon.module.ts` file:

Importing the MatCardModule https://gist.github.com/NyaGarcia/b0b5b6db81d3fc3e834e10b87c7d69e9

Now we can get started on our list component.

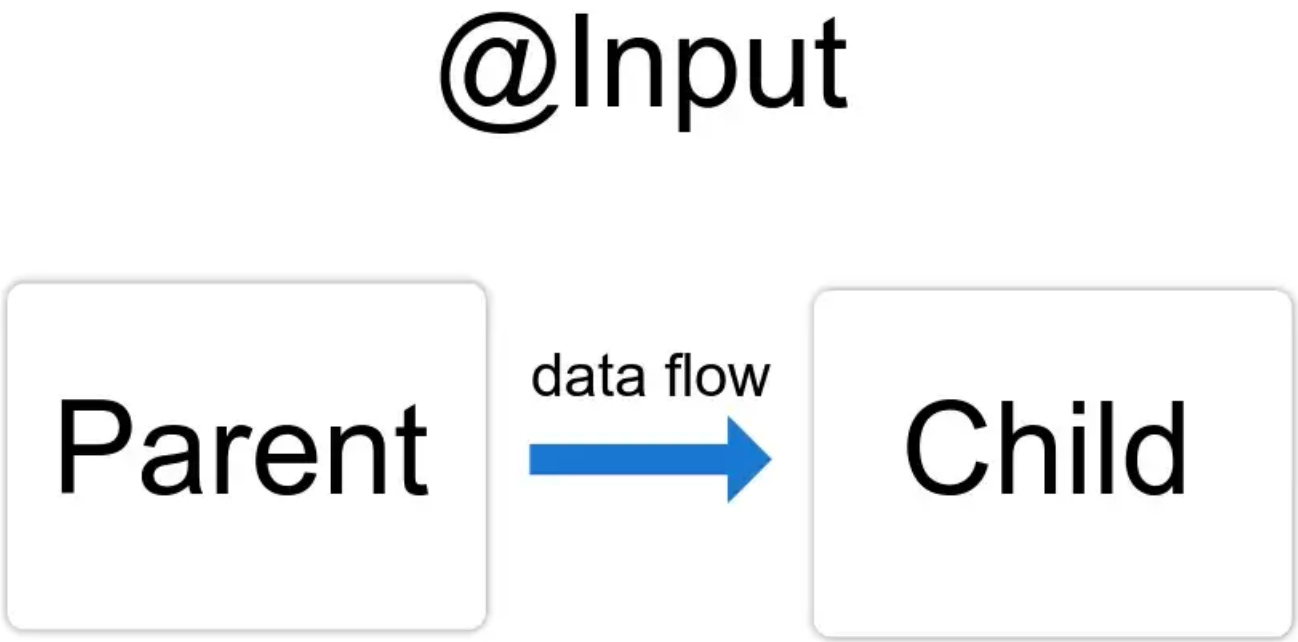
The list logic

Remember the `getAll` method in our firestore service? It returns an `Observable<Pokemon[]>`, an Observable of a Pokemon array, which contains all of the Pokemon stored in our Firestore collection. Our list component will receive this Observable, and use it to display all of the Pokemon. In our `list.component.ts`:

Receiving the pokemon$ Observable
https://gist.github.com/NyaGarcia/643312e34a9de8cba4b7510b02558a28

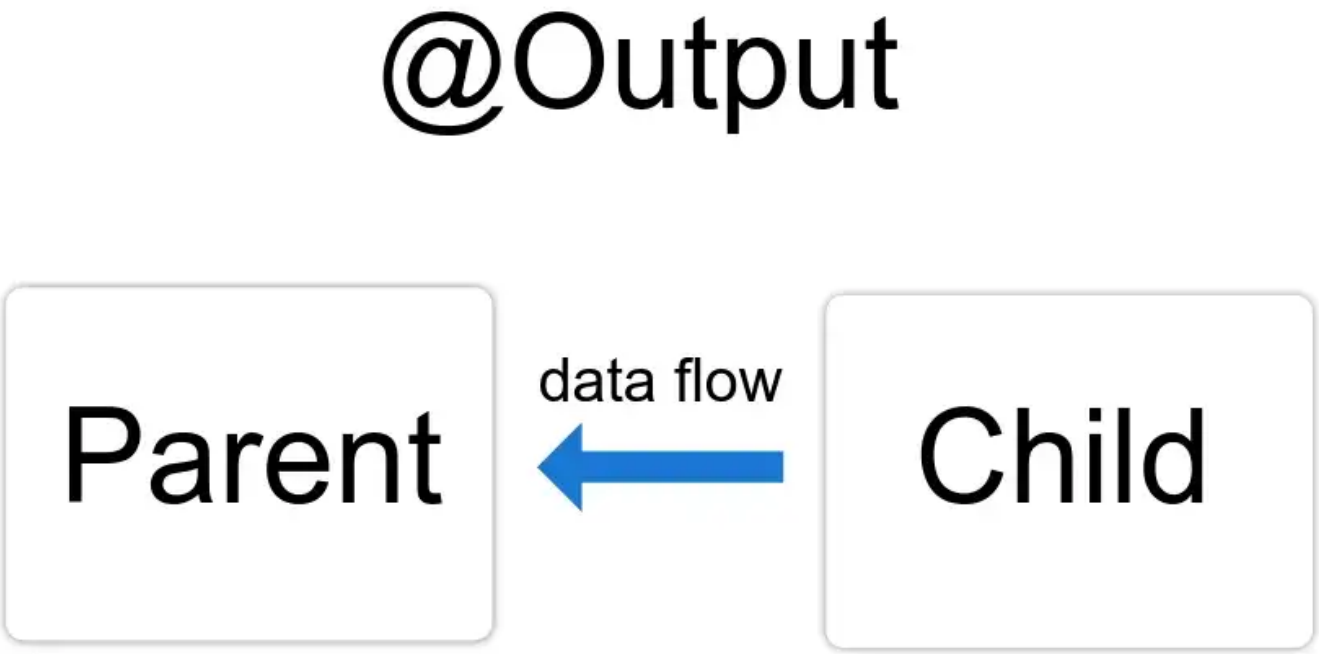
In case you aren't familiar with the `Input()` and `Output()` decorators, I'll explain them briefly. If you already know how they work, feel free to skip the following explanation:

- The `Input()` decorator means that the `pokemon$` property will receive its value from the parent component. We'll explain how we can send values from the parent to the child later on in the tutorial, when we create the Pokemon component (which is the parent component), so don't worry about that part for now.



Input flow diagram

- The `Output()` decorator allows us to send data to the parent component. It marks a property as a sort of doorway, through which we can send data to the parent. `Ouput()` properties **always** need to be of type `EventEmitter`. In this case we're using it to notify the parent component whenever we select a Pokemon from the list. We'll explain how we can receive the data in the parent later on, when we create the Pokemon component.



Output flow diagram

You can read more about inputs and outputs in the official Angular documentation.

The list template

Then, in our `list.component.html` component, we'll iterate through the `pokemon$` Observable, using an `ngFor` together with the `async` pipe.

Note: The `async` pipe automatically subscribes and unsubscribes from Observables, allowing us to easily iterate through them with `ngFor`.

We'll use `mat-card` components to display our Pokemon data, and make it look nice:

The list component template https://gist.github.com/NyaGarcia/1178db3ff0af5ec507c34d5e515b1a5d

Some CSS:

List styling https://gist.github.com/NyaGarcia/50a344a4eab10450ecb6fdc9bccb7718

Awesome! Our `list` component is ready.

The Pokemon detail

We'll also need a detail component, which will display the information of the Pokemon that we select. This component will also have the delete and update buttons. Let's create it inside our `components` directory:

```
ng g c detail
```

This component will receive the selected Pokemon from the `pokemon` component via `@Input`. Then, in the template, we'll display that Pokemon's data. In our `detail.component.html`:

The detail template https://gist.github.com/NyaGarcia/db904237991ef4348c92d072f41fc47d

As you can see, we've added the update and delete buttons. Clicking on the update button will call the `update` function, and clicking on the delete button will call the `delete` function. These functions will make the `updatePokemon` and `deletePokemon` event emitters emit a value, to let the parent component know which button we've

clicked. Then, the parent can act accordingly. Here's our component implementation:

Detail component logic https://gist.github.com/NyaGarcia/f7e46f4150f40fc5f1ed3309d020097d

Notice how we're sending `void` values in both event emitters? This is because we don't need to send any actual data to the parent component, we just want to know when the user clicks the update or the delete button in the child component, and react accordingly.

Let's add some styling:

Detail styling https://gist.github.com/NyaGarcia/a51ee9b2b35bf50aa4f255d24e12d3a2

That's all! Our detail component is ready. This was the last of our presentational components, so we're now ready to get started on our smart, logic-filled component: the Pokemon component.

The pokemon component

This component will contain all of the necessary logic to communicate with the Firestore service, and provide the other presentational components (form, list and detail) with the data that they'll need to display. It acts as a sort of controller, directing the flow of data of our application. Let's head over to our `pokemon.component.ts` file and get started.

The first thing that we need to do is inject the `PokemonFirestoreService` in the constructor, as well as the `MatDialog` for the form. We'll also create two class variables:

- An `allPokemon$` variable, which we'll initialize in the `ngOnInit` by calling the `pokemonService.getAll()` function.
- A `selectedPokemon` variable, which will contain the selected Pokemon.

Creating class variables and adding dependencies
https://gist.github.com/NyaGarcia/96dc6554ba14697982d9ecbfcda5a601

Awesome! Let's continue. Our Pokemon component will contain the following methods:

- `addPokemon` : This method will open the form dialog, and after the dialog has been closed, it will filter the stream getting rid of falsy values, and then make a call to the `pokemonService.create` function, to create a new Pokemon with the form data.
- `updatePokemon` : This method is similar to `addPokemon` . It also opens the form dialog, passing the `selectedPokemon` as data so we can update its values. Once the dialog has been closed, it also filters the stream and then calls the `pokemonService.update` function, to update the Pokemon. Lastly, it updates the `selectedPokemon` with the new Pokemon data.
- `selectPokemon` : This method receives a `Pokemon` , and updates the `selectedPokemon` class variable accordingly.
- `deletePokemon:` This method calls the `pokemonService.delete` function, with the `selectedPokemon.id` . It also sets the `selectedPokemon` to `undefined` .

Here's the implementation of these methods:

Implementing the logic https://gist.github.com/NyaGarcia/b69cd88602f3c348ddc340d8e9060bca

The last thing that we need to do is configure the template:

Pokemon component template https://gist.github.com/NyaGarcia/30249fdad5bd84e0a4c381e89a727ed0

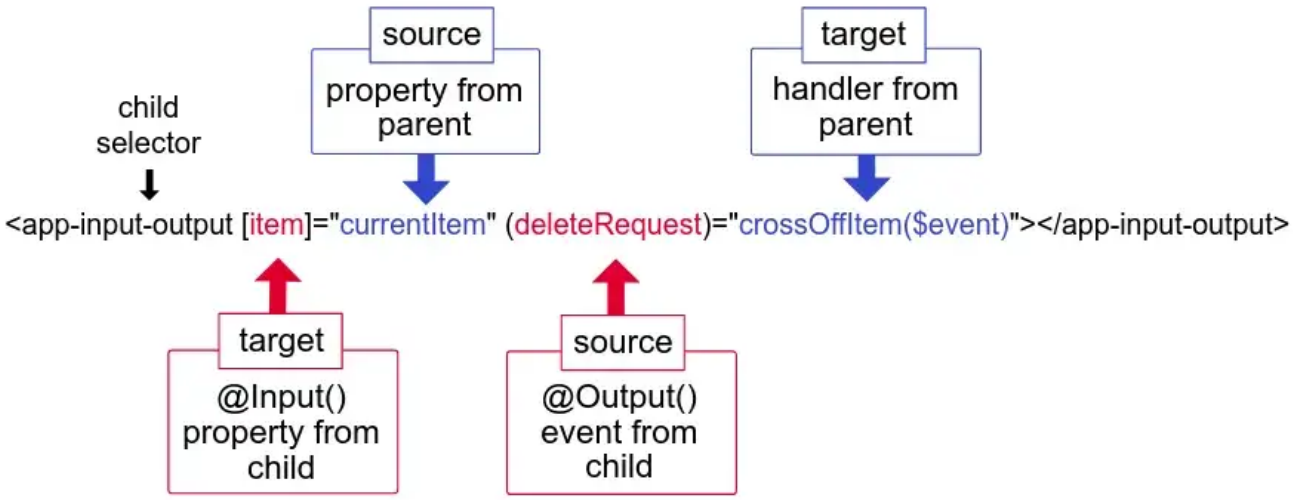
In case you aren't familiar with the binding syntax, I'll explain it briefly. Let's take a look at the `app-list`:

- `[pokemon$]="pokemon$"`: This is called property binding. Remember when we used the `Input()` decorator in the `app-list` component to create the `pokemon$` variable, which would receive its value from the parent component? With this property binding, we're binding the child's `pokemon$` variable to the parent's

`allPokemon$` variable. You can think of this as "sending" the `allPokemon$` variable to the child.

- `(pokemonEmitter)="selectPokemon($event)"` : This is called event binding. It connects the `pokemonEmitter` event that we created using the `Output()` decorator in the `app-list` component (child component) with the `selectPokemon` function that we implemented in this component (parent component). When the `PokemonEmitter` in the child component emits a value, the parent component receives it in the `$event` and passes it to the `selectPokemon` function.

Here is a diagram that explains how the `input` and `output` syntax works:



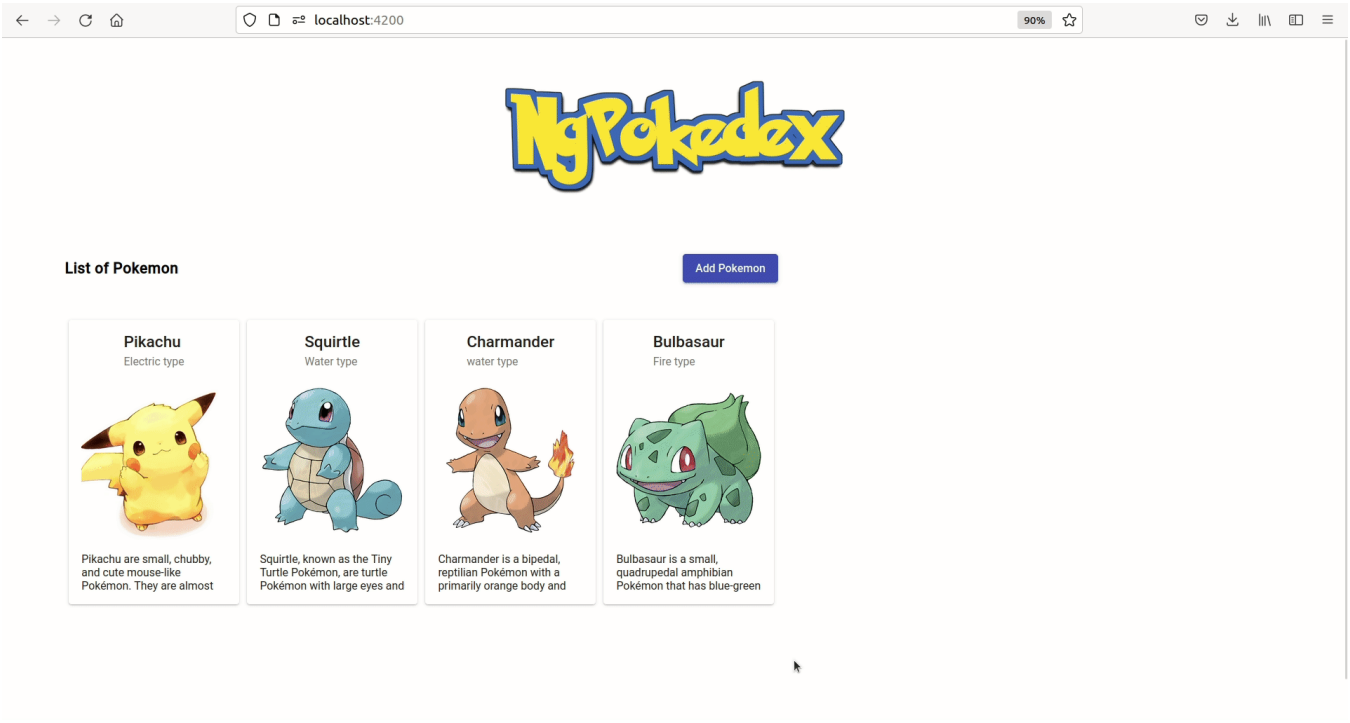
Input/output example diagram from the Angular documentation

You can read more about this in the official Angular documentation.

Finally, let's add a little CSS to make it easier on the eyes:

Adding styles to the Pokemon component
https://gist.github.com/NyaGarcia/4a190eac2f6dcb6abc45d855070e2938

We're done! Let's go ahead and test that everything works. Run `npm start` in your terminal if you haven't done so already, and head over to your browser:



Testing the app

Refactoring: Handling subscriptions

One of the dangers when using Observables is having memory leaks. Why is this? Because, once we subscribe to an Observable, it'll keep emitting values indefinitely until one of the following two conditions are met:

1. We manually unsubscribe from the Observable.
2. It completes.

As you can probably imagine, this can cause problems for us, which is why it's important to always make sure that subscriptions are properly handled.

When using Angular, the `async` pipe handles subscriptions for us, but, since we can only use it in the component templates, whenever we have to subscribe to an

Observable inside a component class (like we've done in the `pokemon.component.ts`), we're going to have to handle subscriptions ourselves. How?

Well, we can use a Subject, together with the `takeUntil()` operator, to force our Observables to complete when the component is destroyed. Let's implement it.

First, we'll create a `destroyed$` Subject in the `pokemon.component.ts` file:

```
destroyed$ = new Subject<void>();
```

Then, we'll use the `ngOnDestroy` hook, which is triggered when the component is destroyed, to make our Subject emit:

Making the destroyed$ subject emit
https://gist.github.com/NyaGarcia/77dc5c92858cb830b1d2bc634813bca9

Last, but not least, we'll use the aforementioned `takeUntil` operator to make our Observables complete when the `destroyed$` Subject emits a value:

Using takeUntil https://gist.github.com/NyaGarcia/1e843737e543cba44d42135f628df493

After all of the changes, our `pokemon.component.ts` file will look like this:

Handling subscriptions https://gist.github.com/NyaGarcia/25aae4e57cb15721de334f2c33822f23

Conclusion

That's all folks! Hopefully, you've learned how to create a CRUD application with Angular 14 and the latest AngularFire 7 modular SDK.

Please remember that the new API isn't complete, so if you decide to go ahead anyway and use it in production, do so at your own risk. Also, bear in mind that AngularFire 7 provides a compatibility layer so that you can continue using the previous AngularFire 6 API.

I hope you enjoyed this tutorial and found it useful. Thank you for reading!

Java Script Angular Coding Web Development Programming