

1 Overview

React x Redux pattern is “more popular” with React, but we will cover in Angular.

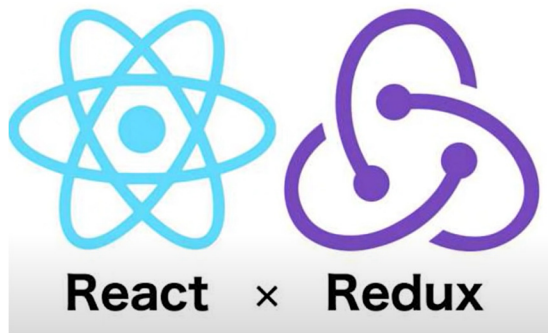


Figure 1 Redux closely linked to React development

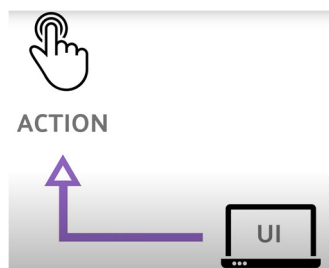
To use it in Angular we use an npm library called NgRx



NgRx uses redux principles with RxJs as the underlying library.

2 How it NgRx works

We have a UI displaying some data, user performs some action that will change the *state* of the data.

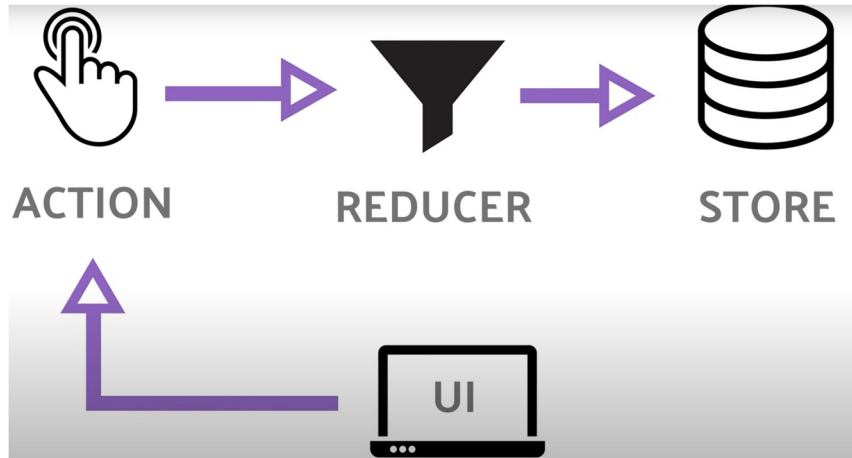


In **redux** the only way of *changing the state* is by dispatching some **action**.

The action will go to some reducer function, will copy the current state of the application along with any new changes to a new Javascript object.

State is immutable, it cannot be changed directly but rather has to be copied over to a brand new state. This produces a predictable state-tree allowing easier debugging of an App. This tree and traceability is the main benefit of **redux**.

Once the reducer creates this new state Javascript object it stores it to a client-side data store.



In NgRx data in this *store* is an observable. So it can be referred to by any component in the ng App.

It's like "global variables" if you will, but it will save a bunch of headache that we saw with attempting:

- a) Passing data back-n-forth betwixt components. Yuck!
- b) Using a service to store the state centrally.

Since this redux pattern, is baked into NgRx we assume it will be an easier methodology for component C to know things that component B set or even component A.

Ultimately this reduction of coupling of components allows a freer development of components and the store and accessing the observables becomes the *contract between components* and through that contract how the data passes without need for other mechanisms (like the ng service, or the idea of maintaining parameters between components or parent-child relationships for data flow.)

Sometimes globals are good 😊, globals with eventing & state wired (observable) in ... even better!

All of our components and services will be sharing the same data at any given point and time!
(Synchronicity ! subtle "Police" band reference)

3 Build an ngrx App

1. Create a new Angular App

```
> ng new ngrx-fire  
> cd ngrx-fire
```

2. Install ngrx store module

```
> npm install @ngrx/store --save
```

```
C:\git\ngrx\ngrx-fire>npm install @ngrx/store --save

added 1 package, removed 1 package, and audited 1012 packages in 2s

90 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

3. Edit the file: `.\src\app\app.module.ts` to import StoreModule as below:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { StoreModule } from '@ngrx/store';
import { simpleReducer } from './simple.reducer';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    StoreModule.forRoot({ salutation: simpleReducer })
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

a. Save the changes. Remember “salutation” as we will use that later.

4. Create the reducer code.

```
ngrx-fire > src > app > TS simple.reducer.ts > ...
1  import { Action } from '@ngrx/store';
2
3  export function simpleReducer(state: string = 'Hello World', action: Action) {
4    console.log(action.type, state)
5
6    switch (action.type) {
7      case 'SPANISH':
8        return state = 'Hola Mundo'
9      case 'FRENCH':
10       return state = "Bonjour le monde"
11      default:
12        return state;
13    }
14  }
```

5. Work on the Component Class (app.component.ts), we'll just use the main outer App component.

```
ngrx-fire > src > app > TS app.component.ts > AppComponent > spanishMessage
1  import { Component } from '@angular/core';
2  import { Store } from '@ngrx/store';
3  import { Observable } from 'rxjs';
4
5  interface AppState { // in the interface called AppState
6    | salutation: string; // define a simple message, a string
7  }
8
9  @Component({
10   selector: 'app-root',
11   templateUrl: './app.component.html',
12   styleUrls: ['./app.component.scss']
13 })
14 export class AppComponent {
15   title = 'ngrx-fire';
16   message$: Observable<string> // property convention: dollar-sign for Observable
17
18   // ctor
19   constructor(private store: Store<AppState>) { // inject defined AppState into our store
20     this.message$ = this.store.select('salutation') // link store : message to our Observable
21   }
22
23   spanishMessage() {
24     this.store.dispatch({type: 'SPANISH'}) // literal arg passed in
25   }
26   frenchMessage() {
27     this.store.dispatch({type: 'FRENCH'})
28   }
29 }
```

Here we define the interface, to the Store interface AppState area of code.

We create an observable `message$` to tie to the store (where we use 'salutation' to refer to the Store's holding of our data).

We create a couple functions to tie to buttons. `spanishMessage()` and `frenchMessage()`. They will dispatch the implicit action in the curly's `{ }` and pass in the action type == 'Spanish' to the reducer which will return our state.

6. Actions. There is a lot to this ngrx. Here is some information on [ngrx.io](https://ngrx.io/guide/store/reducers) <https://ngrx.io/guide/store/reducers> which also is showing Actions - which are 'unique events' being first class citizen, file descriptions.

login-page.actions.ts

```
import { createAction, props } from '@ngrx/store';

export const login = createAction(
  '[Login Page] Login',
  props<{ username: string; password: string }>()
);
```

The createAction function returns a function, that when called returns an object in the shape of the Action interface. The props method is used to define any additional metadata needed for the handling of the action. Action creators provide a consistent, type-safe way to construct an action that is being dispatched.

Use the action creator to return the Action when dispatching.

login-page.component.ts

```
onSubmit(username: string, password: string) {
  store.dispatch(login({ username: username, password: password }));
}
```

This means you would have Actions and Reducers in files, cooperating with each other. (or why we kept my example so simple, creating it in the app.module.ts itself! Some severe shorthand, below.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { StoreModule } from '@ngrx/store';
import { simpleReducer } from './simple.reducer';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    StoreModule.forRoot([salutation: simpleReducer])
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

7. Let's transform that to the longhand and create an action.
Actions are events, so we need one for languageChangeAction.
 - a. Create file
languageChangeAction.ts

Look at final code to see the new files and how we rename and create the languageChangeAction and rename simpleReducer to salutationReducer.ts.

This makes sense that we can initiate a languageChangeAction which returns that simple state, who's .type property contains the action.type string (which is : 'SPANISH', 'FRENCH', ...)

We can have several reducers then, that leverage that action.

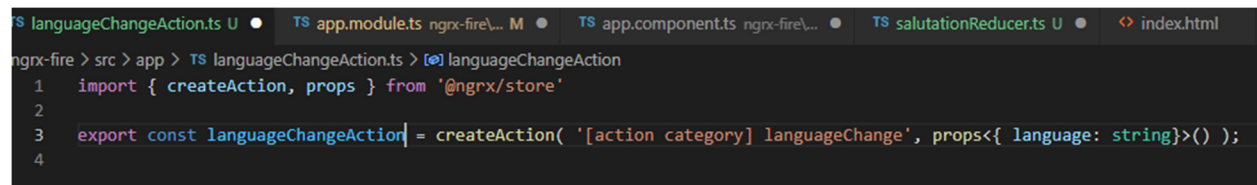
We have the salutationReducer.ts, which returns a greeting.

Maybe you have a currencyReducer.ts which returns a string that has the currency for that language's home country, IDK, I'm making this up as I go.

So the final code has the scaffold for setting up an ngrx Action and Reducer and kind of makes sense for the linkages, where the code goes, how it will be centralized in a Store and refer to the location of the data by some state.

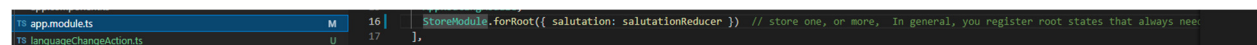
We have a simple state: a string : language

Set up in the properties of languageChangeAction.ts



```
1 import { createAction, props } from '@ngrx/store'
2
3 export const languageChangeAction = createAction( '[action category] languageChange', props<{ language: string }>() );
4
```

We tie the return value from the reducer, which we call 'salutation'



```
16 StoreModule.forRoot( { salutation: salutationReducer } ) // store one, or more, In general, you register root states that always need
17 ],
```

To the reducer in our app.module.ts

Finally,

```

grx-fire > src > app > TS salutationReducer.ts > salutationReducer
1  import { Action } from '@ngrx/store';
2  import { languageChangeAction } from './languageChangeAction';
3
4  export function salutationReducer(state: string = 'Hello World', action: Action) {
5
6      console.log(action.type, state)
7
8      switch (action.type) {
9          case 'SPANISH':
10             return state = 'Hola Mundo'
11          case 'FRENCH':
12             return state = "Bonjour le monde"
13          default:
14             return state;
15      }
16  }

```

The salutationReducer() returns the defined state ... whose interface we established

```

import { Component } from '@angular/core';
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';

interface AppState {    // in the interface called AppState
  salutation: string;    // define a simple message, a string
}

@Component({

```

Phew! That's pretty convoluted to follow.

Why all this work? In essence the reactive nature of Observables means we can centralize things that are:

- a) Repeated in several components
- b) Changeable via user interactions

That is worth the price of admission, as any other method of sharing these states will be heavy lifting. This is the problem NgRx was aimed at solving.

4 NgRx future study

There are many topics to study further, in the area of NgRx.

4.1 Effects

Effects are RxJS powered side effect model for Store. <https://ngrx.io/guide/effects>

Effects are where you handle tasks such as fetching data, long-running tasks that produce multiple events, and other external interactions where your components don't need explicit knowledge of these interactions.

4.2 Selectors

Selectors are pure functions used for obtaining slices of store state.

<https://ngrx.io/guide/store/selectors>

4.3 Meta-reducers

<https://ngrx.io/guide/store/metareducers>

`@ngrx/store` composes your map of reducers into a single reducer.

Developers can think of meta-reducers as hooks into the action->reducer pipeline. Meta-reducers allow developers to pre-process actions before *normal* reducers are invoked.

4.4 Feature Creators

A function (createFeature) that reduces repetitive code in selector files by generating a feature selector. <https://ngrx.io/guide/store/feature-creators>

4.5 Injecting Reducers

<https://ngrx.io/guide/store/recipes/injecting>

4.6 Testing

Using a Mock Store.

<https://ngrx.io/guide/store/testing>

4.7 Data

NgRx Data is an extension that offers a gentle introduction to NgRx by simplifying management of **entity data** while reducing the amount of explicitness.

<https://ngrx.io/guide/data>