§ NGRX Training Part 1 - NgRx Basics

Wednesday, May 27, 2020 7:11 PM

Application State

State is a representation of an application at a certain point in time. State can be (not an exhaustive list):

- Server response data
  - A user initiates some POST or GET requests to retrieve data.
- Authentication state
  - A Jason Web Token for example.
- User information
  - A name or email address, roles for a user.
- User input
  - Data entered into forms or dialogs.
- UI state
  - A dropdown or a modal opened/closed, a loading state of a component.
- Router / location state
  - Contains all the necessary information to retrieve the state of your application if you hit the refresh button, or share the URL.

Managing application state is a critical element in web applications. This tutorial will investigate managing application state in Web applications based on Angular using NgRx.

Angular and Application State

NgRx is a group of Angular libraries for building reactive applications in Angular. Among this collection is ngrx/store, which provides state management. ngrx/store implements the Redux pattern for state management utilizing RxJS.

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables for simplifying asynchronous coding

Redux is an open-source JavaScript library for managing application state. It is commonly used with React or Angular.

NgRx packages

NgRx is implemented as six packages.

- ngrx/store
  - RxJS powered state management for Angular apps, inspired by Redux.
- ngrx/store-devtools
  - Instrumentation for @ngrx/store enabling time-travel debugging.
- ngrx/effects
  - Side-effect model for @ngrx/store.
  - Effects uses streams to provide new action sources from **external interactions**, such as, network requests, web socket messages and time-based events.
- ngrx/router-store
  - Bindings to connect the Angular Router to @ngrx/store.
- ngrx/entity
  - Entity State adapter for managing record collections.
- ngrx/schematics

- Scaffolding library for Angular applications using NgRx libraries.

NgRx and Redux

NgRx is based on three fundamental Redux principles for state management:

1. *Single source of truth*
2. *State is read-only (or immutable)*
3. *Pure functions update state*

These principle influence NgRx in the following way:

Single source of truth

- State is represented by one tree of objects and kept in a centralized **store**.
- Changes to state are done in a predictable and consistent way.
- Server-side rendering includes the initial app state in the initial load of the app.

State is read-only

- We can only access state. We can't overwrite or mutate it directly. We must dispatch an **action** to do so.
- We know when and where state changes occur, enabling simple, centralized change detection logic.
- Promotes simple testing approaches for state management via the store.

Pure functions update state

- A pure function is easy to test because, by definition, for the same input it will always return the same output. It doesn't mutate or access properties outside of its own scope.
- A **reducer** is a pure function that is triggered by a particular action type.

The NgRx implementation defines *store*, *action* and *reducer* elements, as well as others.

NgRx implementation elements

The core building blocks of NgRx state management that are used by Angular applications consist of

- Store
- Action
- Reducer
- Selector
- Effects

Store

With NgRx, the state of an application is kept as an object tree within a single Store. The Store has the responsibility of persisting the state and applying changes to it when told to do so (for example, when an action is dispatched). The store acts as the single source of truth for the state of an application.

Action

Actions describe unique events that are dispatched from components and services. They are implemented as classes that realize the NgRx Action interface. Actions are used to notify the Store that a particular change in state has occurred.

Reducer

Reducers are pure functions accepting two arguments, the previous state and an Action. When an Action is dispatched, NgRx goes through all the reducers passing as arguments the previous state and the Action, in the order that the reducers where created, until it finds a case for that action.

Selector

NgRx Store provides us the function `to obtain slices of our store. Selectors allow us to decouple any distinct slices of state data, and transformations of that data, from each component. NgRx `Store` provides a `select` function that accepts an argument a pure function, this pure function is our selector.

Effects
Effects allow us to deal with side-effects caused from dispatching an action outside angular components or the NgRx Store. Effects listen if any action is dispatched, then, similar to what reducers do, it checks if the action is one of the actions type for which it has a case. The side-effect is then performed, usually getting or sending data to an API.

ngrx/effects is middleware for handling side-effects in ngrx/store. It listens for dispatched actions in an observable stream, performs side-effects, and returns new actions either immediately or asynchronously. The returned actions get passed along to the reducer.

Let's investigate how to use NgRx by writing a simple application that will utilize Store, Action, and Reducer.

Tutorial application goal
Our application will manage simple lists. The first list we will create is a list of our favorite ice cream flavors. We want to list flavors of ice cream and who makes it.

The base application has been written and provided as ListApp1.zip. You can review its details here (ListApp1 Details).

Install NgRx
Next, we will install NgRx Store. You can use

```
npm install @ngrx/store
```

NOTE
An alternative to `npm`, is using `ng` as follows.

```
ng add @ngrx/store
```

where this command will

- Update `package.json`
- Run `npm install` on `ngrx/store`
- Create a `reducers` folder with a default reducer
- Update `app.module.ts`

For the purposes of this tutorial, use `npm`.

Our first state model
In order to track our favorite ice creams, we will permit users to enter the flavor and maker of their favorite treat. We will represent ice cream with the following state model. This definition is found in the folder named `src/app/models` as `ice-cream.ts`.

ice-cream.ts

```
export interface IceCream {
```

```
    flavor: string;
    maker: string;
}
```

Our application state will manage a collection (as an array) of IceCream.

AppState

NgRx applications have a single source for application state. We will define an interface for our AppState by creating the file app.module.ts in `src/app` and adding the following code.

app.module.ts

```
import { IceCream } from './models/ice-cream';

export interface AppState {
  readonly iceCream: IceCream[];
}
```

Note `readonly`. This follows the Rx principle that state cannot be mutated.

Selecting state

Our application is currently composed of a single "slice" of state, that is, a collection of ice cream flavors. What if we added another list to our application, classic muscle cars for example? We could define another "slice" for our `AppState`.

```
export interface AppState {
  readonly iceCream: IceCream[];
  readonly muscleCars: MuscleCar[];
}
```

We can then add a page that manages the UI of our favorite classic muscle cars list and access just the slice of state that is needed by that page.

NgRx `Store` leverages this idea of "slices of state" by utilizing **selectors**. The following example shows how to access the `muscleCars` state slice using the key '`muscleCars`' as define in the `AppState` example.

```
export class SomeComponent {
    muscleCars: Observable<MuscleCar[]>;

    constructor(private store: Store<AppState>) {
        this.muscleCars = store.select('muscleCars');
    }
}
```

Now that we've defined our model and application state, we'll move on to creating actions.

Adding action

NgRx Actions relate the unique events occurring throughout an Angular application to changes in application state. Every user interaction that triggers a state update should be described using an Action. An NgRx Action has the following definition

```
interface Action {
    type: string;
}
```

From <https://ngrx.io/guide/store/actions>

When utilized in our application, following convention, we will inherit our actions from this interface and add a constructor that will define the payload for the action.

The Action `type` is a read-only string that represents the type of action dispatched by the application.

The Action `payload` adds any related data required for completing the action.

In our application, two key events driven by the user are

- Adding a new ice cream flavor
- Removing an ice cream flavor from the list

Let's define the NgRx actions. Create this folder: `src/app/actions`. Add a new file named `ice-cream-actions.ts`.

ice-cream-actions.ts

```
import { Action } from '@ngrx/store';
import { IceCream } from '../models/ice-cream';

export const ADD_FLAVOR       = '[ICECREAM] Add';
export const REMOVE_FLAVOR    = '[ICECREAM] Remove';

export class AddFlavor implements Action {
    readonly type = ADD_FLAVOR;
    constructor(public payload: IceCream) {}
}

export class RemoveFlavor implements Action {
    readonly type = REMOVE_FLAVOR;
    constructor(public payload: number) {}
}

export type Actions = AddFlavor | RemoveFlavor;
```

Define unique string constants for each `Action`.
We can guard against duplicate action definitions by using an Action Namespace: [ICECREAM] *ActionName*

Create a class based on `Action` and define `type` and provide a constructor to define our `payload`.

Helper for importing our actions.

These actions will be utilized by the ice-cream-display component when it receives evets from its child components. Here are the changes to use the actions in our application.

ice-cream-display.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { Store, select } from '@ngrx/store';                    (1)
import { AppState } from '../app.state';
import { IceCream } from '../models/ice-cream';
import * as FlavorActions from '../actions/ice-cream-actions';  (2)

@Component({
  selector: 'app-ice-cream-display',
  templateUrl: './ice-cream-display.component.html',
  styleUrls: ['./ice-cream-display.component.scss']
})
export class IceCreamDisplayComponent implements OnInit {
  private iceCreams: Observable<IceCream[]>;

  constructor(private store: Store<AppState>) {
      this.iceCreams = this.store.pipe(select('iceCream'));     (5)
   }

  ngOnInit() { }

  onNewFlavor(newFlavor: IceCream) {
    this.store.dispatch(new FlavorActions.AddFlavor(newFlavor));   (3)
  }

  onDeleteFlavor(index: number) {
    this.store.dispatch(new FlavorActions.RemoveFlavor(index));    (4)
  }
}
```

1) Add an import statement for `Store` and `select`.
2) Import our actions as `FlavorActions`.

3) In the onNewFlavor event handler, use Store to dispatch a new flavor to the reducer associated with this action.
4) In the onDeleteFlavor event handler, use Store to dispatch a the index of the flavor to remove from our State collection.
5) Define the ice cream flavors that are going to be displayed by selecting the iceCream state from the Store. Since this is an observable, any changes will be to automatically forwarded for this component.

Focusing on the two event handlers (3 and 4 above), our component passes a new FlavorAction (either AddFlavor or RemoveFlavor, as appropriate) to the Store via its dispatch method.

Using the dispatch method will cause the payload from the event to be supplied to a related reducer where a new state is defined and accessible to the Store.

Let's add our reducers next.

Reducers

A reducer is the mechanism in NgRx that takes an action and decides what to do with it. The reducer will take the previous state and return a new state which is based on the given action. In ngrx/store, a reducer is a pure function, that accepts two arguments: the previous state, and an action with a type and payload associated with the event.

Create a new folder src/app/reducers. In this folder, create a new file named ice-cream-reducer.ts.

ice-cream-reducer.ts

```
import { IceCream } from '../models/ice-cream';
import * as FlavorActions from '../actions/ice-cream-actions';

const initialState: IceCream = {
    flavor: 'Vanilla',
    maker: 'Honey Hut'
};

export function iceCreamReducer(state: IceCream[] = [initialState], action: FlavorActions.Actions) {

    switch (action.type) {
        case FlavorActions.ADD_FLAVOR:
            return [...state, action.payload];
        case FlavorActions.REMOVE_FLAVOR:
            const id = action.payload;
            return state.filter((tutorial, index , stateArray) => (index !== id));
        default:
            return state;
    }
}
```

Reducers and Immutability

In keeping with Rx principles, when working with state, Reducers should not mutate state. They need to produce and replace.

Our implementation enforces the immutability of state by creating new instances of state, not updates to an existing instance.

Reducers and App.Module

We now need to tie the reducer to the slice of application state related to it. We do this by updating the src/app/app.module.ts file.

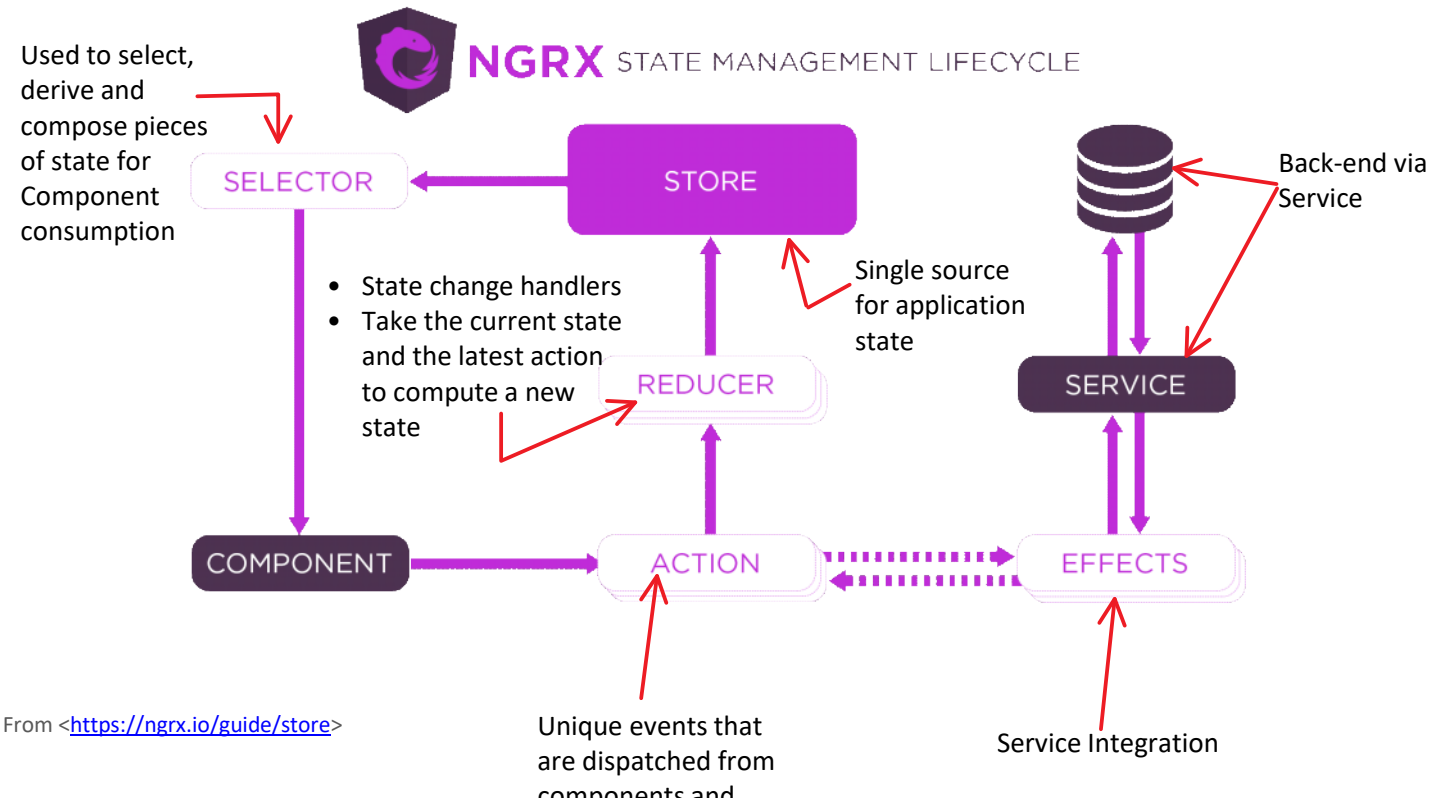
app.module.ts

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { IceCreamCreateComponent } from './ice-cream-create/ice-cream-create.component';
import { IceCreamReadComponent } from './ice-cream-read/ice-cream-read.component';
import { IceCreamDisplayComponent } from './ice-cream-display/ice-cream-display.component';
import { ListHomeComponent } from './list-home/list-home.component';
import { iceCreamReducer } from './reducers/ice-cream-reducer';

@NgModule({
  declarations: [
    AppComponent,
    IceCreamCreateComponent,
    IceCreamReadComponent,
    IceCreamDisplayComponent,
    ListHomeComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    StoreModule.forRoot({
      iceCream: iceCreamReducer
    })
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

In this update, `iceCreamReducer` is related to the selector key `iceCream`.

Part 1 Conclusion

The following diagram represents the general flow of application state in NgRx.



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

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

Unique events that are dispatched from components and services

Service Integration

Let's point out some benefits to using NgRx.

- Since we have a single source of truth and you can't directly change the state, applications are going to behave more consistently.
- Using the redux pattern helps simplify debugging.
- Applications become easier to test since we are introducing pure functions to handle changes in the state. Also, NgRx and RxJs have features for testing.
- When you are comfortable with using NgRx, understanding the flow of data in your application becomes incredibly easy and predictable.

We should fairly point out some cons as well.

- NgRx has a learning curve. It is significant enough that I think it requires some experience or deep understanding of some program patterns. Also, being fluent in RxJS (piping, operators, observables) is very important.
- Related to its learning curve, NgRx has a lot of moving parts. Every time you add some property to the state, you need to add the actions, the reducers, you may need to update or add the selectors, the effects if any, update the store
- NgRx is not part of the Angular core libraries and is not supported by Google, at least not directly (there are NgRx contributors that are part of the Angular team).

This completes our basic introduction to NgRx Store. In Part 2, we will refine our approaches to actions and reducers.