06

NgRx

What Is NgRx?

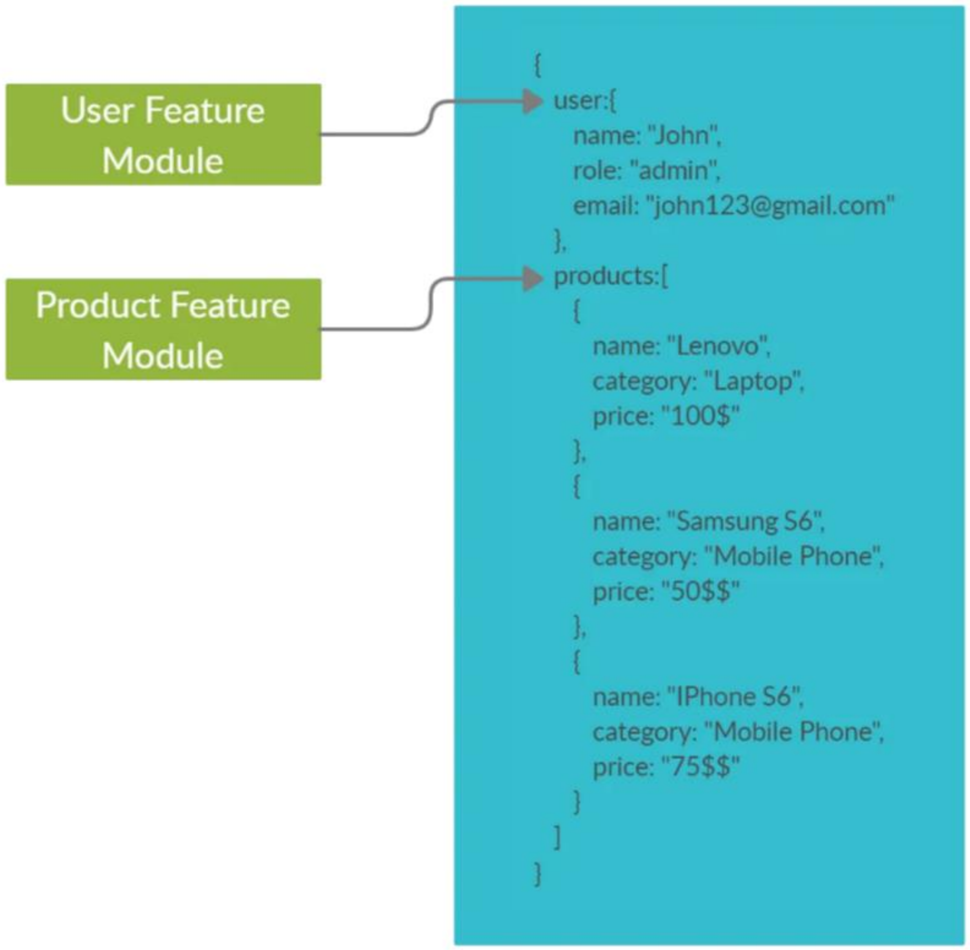
- NgRx stands for Angular Reactive Extensions.
- NgRx is a state management system that is based on the Redux pattern.

What is State?

- application state is the entire memory of the application.
- application state is composed of data received by API calls, user inputs, presentation UI state, application preferences, etc.

Structure of a state object tree

Suppose your application consists of two feature modules called User and Product. Each of these modules handles different parts of the overall state. Product information will always be maintained in the `products` section in the state. User information will always be maintained in the `user` section of the state. These sections are also called *slices*.



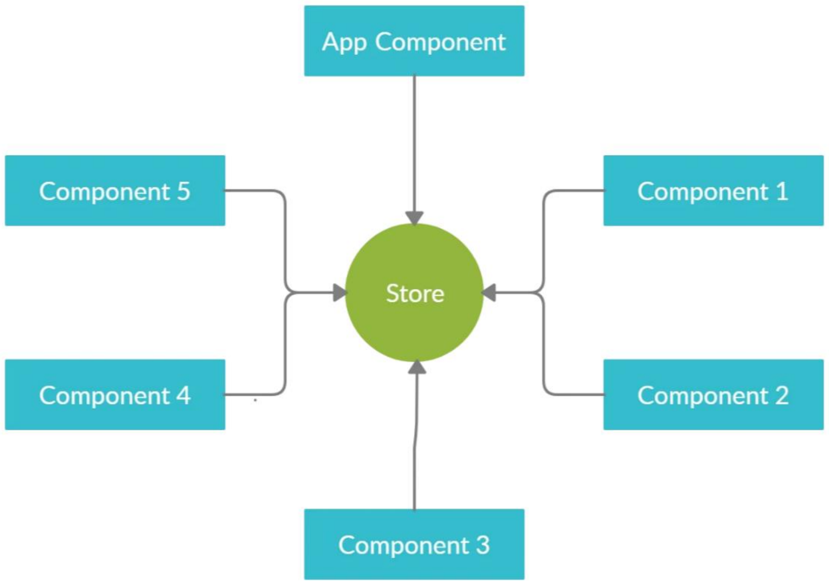
```
{
  user:{
    name: "John",
    role: "admin",
    email: "john123@gmail.com"
  },
  products:[
    {
      name: "Lenovo",
      category: "Laptop",
      price: "100$"
    },
    {
      name: "Samsung S6",
      category: "Mobile Phone",
      price: "50$$"
    },
    {
      name: "IPhone S6",
      category: "Mobile Phone",
      price: "75$$"
    }
  ]
}
```

What is Redux?

- Redux is a pattern that’s used to simplify the state management process in JavaScript applications (not just for Angular).
- Redux is primarily based on **three** main principles.

1. Single source of truth

   This means the state of your application is stored in an object tree within a single store. The store is responsible for storing the data and providing the data to components whenever requested.
   according to this architecture, data flows between the store and the components, instead of from component to component. The following figure illustrates this concept.



2. Read-only state

In other words, state is immutable. This doesn’t necessarily mean that state is always constant and cannot be changed. It only implies that you are not allowed to change the state directly. In order to make changes in the state, you have to dispatch *actions* (which we will discuss in detail later) from different parts of your application to the store.

3. State is modified with pure functions.

Dispatching actions will trigger a set of pure functions called *reducers*. Reducers are responsible for modifying the state in different ways based on the action received. A key thing to note here is that a reducer always returns a new state object with the modifications.

Fundamental Elements of NgRx:

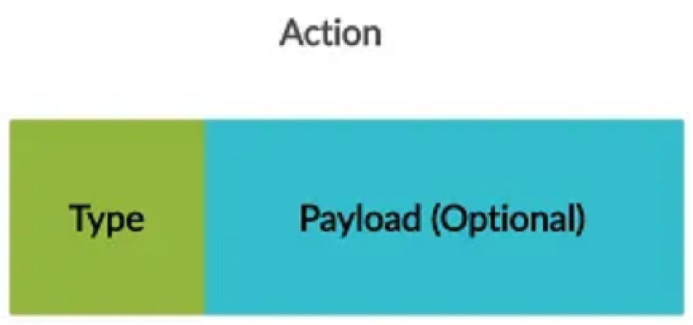
- Store

The store is the key element in the entire state management process. It holds the state and facilitates the interaction between the components and the state. You can obtain a reference to the store via Angular dependency injection, as shown below.

```
constructor(private store: Store<AppState>) {}
```

Actions

An action is an instruction that you dispatch to the store, optionally with some metadata (payload). Based on the action type, the store decides which operations to execute. In code, an action is represented by a plain old JavaScript object with two main attributes, namely `type` and `payload`. `payload` is an optional attribute that will be used by reducers to modify the state. The following code snippet and figure illustrate this concept.



```
{
  "type": "Login Action",
  "payload": {
    userProfile: user
  }
}
```

NgRx version 8 provides a utility function called `createAction` to define action creators (not actions, but action creators). Following is an example code for this.

```
export const login = createAction(
    "[Login Page] User Login",
    props<{user: User}>()
);
```

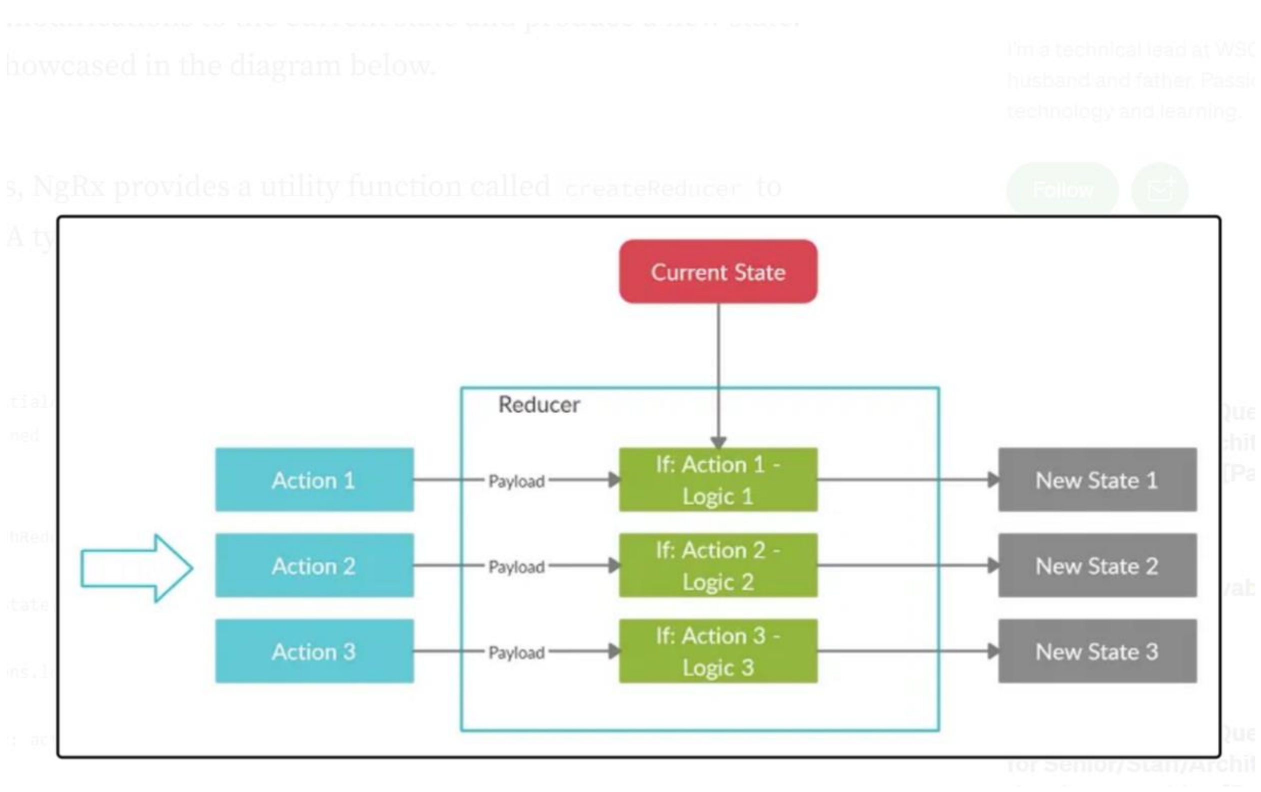
actions.ts hosted with ❤ by GitHub view raw

You can then use the `login` action creator (which is a function) to build actions and dispatch to them to the store as shown below. `user` is the `payload` object that you pass into the action.

```
this.store.dispatch(login({user}));
```

Reducers

Reducers are responsible for modifying the state and returning a new state object with the modifications. Reducers take in two parameters, the current state and the action. Based on the received action type, reducers will perform certain modifications to the current state and produce a new state. This concept is showcased in the diagram below.



Similar to actions, NgRx provides a utility function called `createReducer` to create reducers. A typical `createReducer` function call would like the following.

```
export const initialAuthState: AuthState = {
    user: undefined
};

export const authReducer = createReducer(

    initialAuthState,

    on(AuthActions.login, (state, action) => {
        return {
            user: action.user
        }
    }),

    on(AuthActions.logout, (state, action) => {
        return {
            user: undefined
        }
    })

);
```

As you can see, it takes in the initial state (the state at the application startup) and one-to-many state change functions that define how to react to different actions. Each of these state change functions receives the current state and the action as parameters, and returns a new state.

- Effects
    - Effects allow you to perform side effects when an action is dispatched to the store. Let’s try to understand this through an example.

NgRx provides a utility function called `createEffect` to create effects. A typical `createEffect` function call would look like the following.

```
login$ = createEffect(() =>
    this.actions$
        .pipe(
            ofType(AuthActions.login),
            tap(action => localStorage.setItem('user',
                    JSON.stringify(action.user))
            )
        )
,
{dispatch: false});
```

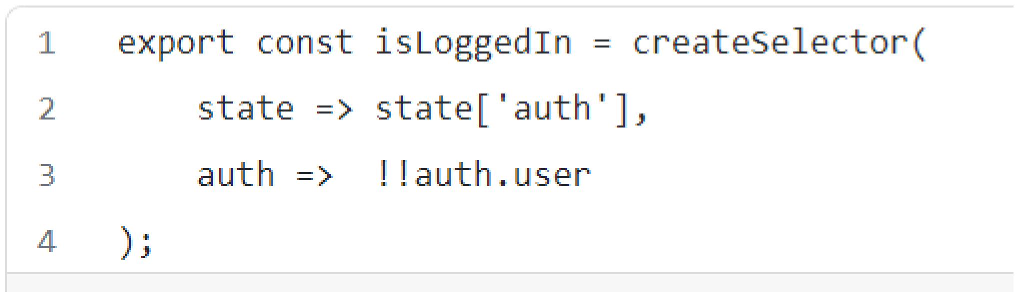
The `createEffect` method takes in a function that returns an observable and (optionally) a configuration object as parameters.

NgRx handles the subscription to the observable returned by the support function, and therefore you don’t have to manually subscribe or unsubscribe. In addition, if any error occurs in the operator chain, NgRx will create a new observable and resubscribe to ensure that the side effect always gets executed.

- Selectors

Selectors are pure functions used for obtaining slices of the store state. As shown below, you can query the state even without using selectors. But this approach again comes with a couple of major cons.

NgRx provides a utility function called `createSelector` to build selectors with memoization capability. Following is an example of the`createSelector` utility function.

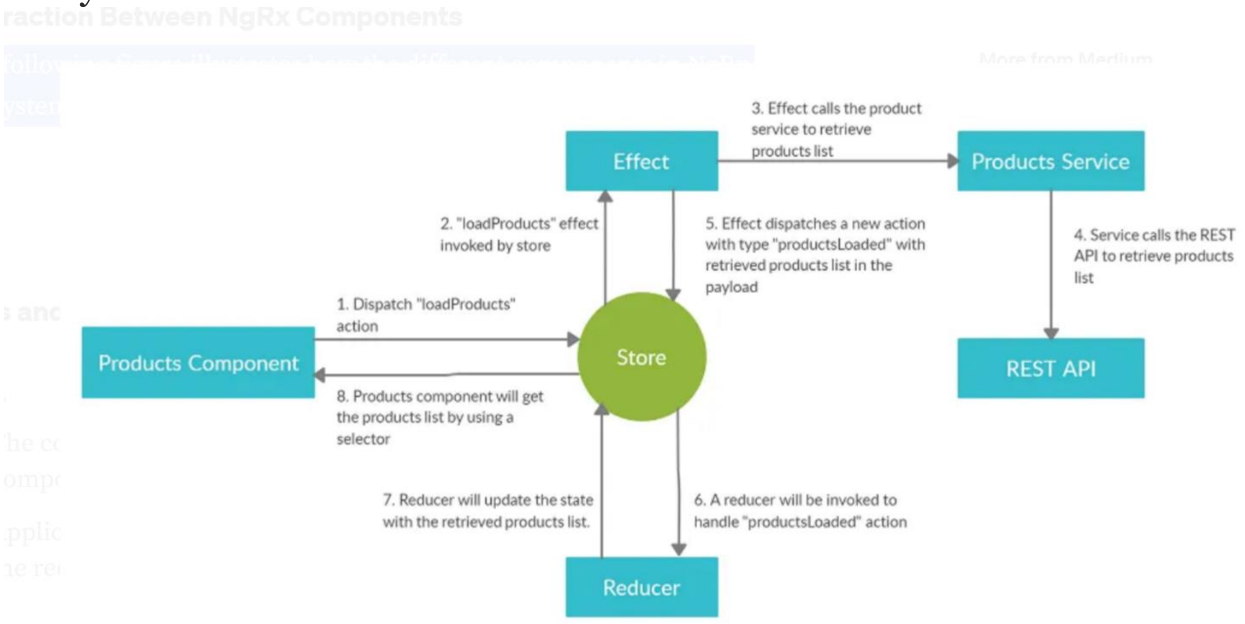


```
export const isLoggedIn = createSelector(
    state => state['auth'],
    auth =>  !!auth.user
);
```

The `createSelector` function takes in one-to-many mapping functions that give different slices of the state and a projector function that carries out the computation. The projector function will not be invoked if the state slices haven't changed from the last execution. In order to use the created selector function, you have to pass it as an argument to the `select` operator.

Interaction Between NgRx Components

The following figure illustrates how the different components in NgRx ecosystem interact with each other.


Effect
3. Effect calls the product service to retrieve products list
Products Service
2. "loadProducts" effect invoked by store
5. Effect dispatches a new action with type "productsLoaded" with retrieved products list in the payload
4. Service calls the REST API to retrieve products list
1. Dispatch "loadProducts" action
Products Component
Store
REST API
8. Products component will get the products list by using a selector
7. Reducer will update the state with the retrieved products list.
6. A reducer will be invoked to handle "productsLoaded" action
Reducer

Pros and Cons of NgRx

Pros

- The concept of a single source of the truth makes it easier for components to share information in an Angular application.
- Application state cannot be directly changed by the components. Only the reducers are able to change the state. That makes debugging easier.

Cons

    - There is a steep learning curve when you first start working with NgRx.

- Application will be a bit verbose as you have to introduce several new artifacts, such as reducers, selectors, effects, and so on.