§ NGRX Training Part 2 - Actions and Reducers Refined

Friday, August 14, 2020 6:52 PM

Introduction

`ngrx/store` provides framework support that helps refine how actions, reducers, and selectors are defined. In this tutorial, we'll explore the refinements for actions and reducers by updating the list management application from Part 1 of this tutorial.

ListApp Actions

Actions are what glue NgRx's state management approach together. They are the initiators of the one-way dataflow process for state management. Once an action is dispatched, the process for new state to be composed is begun - which is typically composed by the payload we sent through our dispatched action.

Let's recall the actions we defined in Part 1.

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;
```

Each of these actions has defined a unique string identifier that incorporates an action namespace; in this case, `[ICECREAM]`. They also associate a payload with the action.

These action identifiers are utilized in a reducer as follows.

```
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;
    }
}
```

There are shortcomings to defining actions in this manner. Namely, they are not strongly typed. Also, relying on strings does not allow us to find all the places the action is used throughout the codebase. For consumption, the string has to be used — and for dispatching, we need to use the class.

Better Actions

ngrx/store provides varieties of `createAction` functions that can be used to handle state transitions. They support creating actions that have no data, require payload data, or use an anonymous function.

Usages	Examples
Without data	`export const increment = createAction('[Counter] Increment');`

With data	`export const loginSuccess = createAction(`
	`'[Auth/API] Login Success',`
	`props<{ user: User }>()`
	`);`

With an anonymous function	`export const loginSuccess = createAction(`
	`'[Auth/API] Login Success',`
	`(response: Response) => response.user`
	`);`

From <https://ngrx.io/api/store/createAction>

We can utilize the "with data" form of `createAction` to update the actions related to `IceCream`.

ice-cream-action.ts

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

// Use createAction to define a function that will produce a desired Action
// with the data that is defined on the Action.

export const AddFlavor = createAction('[ICECREAM] Add', props< {iceCream: IceCream} >());
export const RemoveFlavor = createAction('[ICECREAM] Remove', props< {flavorId: number} >());
```

Note

What props<T>() does is to return an object with a predefined key(_as: 'props') and with a key of type T which is useful for type inference.

```
export function props<P extends object>(): Props<P> {
  return { _as: 'props', _p: undefined! };
}

export interface Props<T> {
  _as: 'props';
  _p: T;
}
```

ListApp Reducers

Reducers are pure functions that specify how state changes in response to an action. The following shows how we structured our `IceCream` reducers.

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;
    }
}
```

One shortcoming that we could identify here is the use of the action constant; it would be better if we could utilize `createAction` based definitions in a type-safe way. Another shortcoming is the `switch` statement itself.

Better Reducers

ngrx/store provides a `createReducer` function that can be used to update state.

```
createReducer<S, A extends Action = Action>(initialState: S, ...ons: On<S>[]):
ActionReducer<S, A>
```

Parameters

`initialState`	Provides a state value if the current state is `undefined`, as it is initially.
`ons`	Associations between actions and state changes.

From <https://ngrx.io/api/store/createReducer>

We can utilize `createReducer` as shown in the following update to the `ice-cream-reducer`.

ice-cream-reducer.ts

```
import { createReducer, on } from '@ngrx/store';
import { IceCream } from '../models/ice-cream';
import { AddFlavor, RemoveFlavor} from '../actions/ice-cream-actions';

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

export const iceCreamReducer = createReducer<IceCream[]>(
    [initialState],
    on(AddFlavor, (state, action) => {
        return [...state, action.iceCream];
    }),
    on(RemoveFlavor, (state, action) => {
        return state.filter((flavor, index , stateArray) => (index !== action.flavorId));
    })
);
```

Note the removal of the `switch` statement and the use of a string constant, and the syntax for initial state.

Conclusion

In this tutorial we learned how to utilize more of the NgRx framework by utilizing `createAction` and `createReducer`. These elements improved the type safety and readability of our code.