

Understanding NgRx Effects and the Action Stream



Tanya Gray

Feb 11, 2018 · 3 min read

NgRx Store provides us a single stream of actions where we can either **dispatch** or **subscribe** any action across our whole app. This action stream is an **Observable**.

NgRx Effects allow us to listen for particular action types, and “do something” when that action happens. Any effect you write is also an Observable.

An effect is an Observable which uses the Action Stream as its source, and also as its destination. That is, an effect **subscribes** to the Action Stream, and it can also **publish** to the action stream.

. . .

This article provides example effects for the following cases:

- One input, one output
- Two inputs, one output
- One input, two outputs
- One input, no outputs
- Passing input payload to output

. . .

One input, one output

The most basic effect you could write would have one action in, and one action out. We can use the RxJs operator `mapTo` to achieve this.

```
@Effect()
public firstAction$: Observable<Action> = this.actions$.pipe(
  ofType( 'FIRST_ACTION' ),
  mapTo( new SecondAction() )
);
```

This effect says:

- Watch the Action Stream for any time `FirstAction` happens
- Dispatch a new `SecondAction` into the Action Stream

. . .

Two inputs, one output

Another common effect you might write is when multiple action types trigger the same result. The `ofType` operator allows for multiple types.

```
@Effect()
public dashboardLoadRequired$: Observable<Action> =
  this.actions$.pipe(
    ofType( 'SIGN_IN_SUCCESS', 'OPEN_DASHBOARD' ),
    mapTo( new LoadDashboardData() )
  );
```

This effect says:

- Watch the stream for a `SignInSuccess` or an `OpenDashboard`
- Dispatch a new `LoadDashboardData` action

. . .

One input, two outputs

You can also use an effect to trigger multiple side-effects when a single action occurs. To achieve this, use the RxJs `concatMapTo` operator.

```
@Effect()
public signInSuccess$: Observable<Action> = this.actions$.pipe(
```

```

ofType( 'SIGN_IN_SUCCESS' ),
concatMapTo([
  new LoadDashboardData(),
  new LoadChatHistory()
])
);

```

This effect says:

- Watch the stream for a `SignInSuccess` action
- Dispatch a `LoadDashboardData` action then a `LoadChatHistory` action

. . .

One input, no outputs

It is possible to write an effect which **does not** dispatch an action, but you need to be explicit about it.

If you do not dispatch an action, the input action will automatically be dispatched. This will crash your browser, because the effect you have written is both **subscribing to** and **dispatching** the exact same action, causing an infinite loop.

Common use cases for no-dispatch effects are when you want to just `console.log()` the action, or when you want to trigger router navigation.

For this use-case, add `{dispatch: false}` to the effect decorator.

```

@Effect({dispatch: false})
public signInSuccess$: Observable<Action> = this.actions$.pipe(
  ofType( 'SIGN_IN_SUCCESS' ),
  tap( () => this.router.navigate(['dashboard']) )
);

```

This effect says:

- Watch the stream for a `SignInSuccess` action
- Navigate to the dashboard
- Do not dispatch any further actions

. . .

Passing input payload to output

The above examples have used the operators `mapTo` and `concatMapTo`. These operators map to static values. Sometimes you want to map to dynamic values, such as using a value passed in via an action's `payload` property.

For dynamic values, use the matching operators `map` or `concatMap` which expect a function rather than a static value.

```
@Effect()
public signInSuccess$: Observable<Action> = this.actions$.pipe(
  ofType( 'SIGN_IN_SUCCESS' ),
  map( action => action.payload ),
  concatMap( payload => [
    new LoadDashboardData( payload.companyId ),
    new LoadChatData( payload.userId )
  ])
);
```

This effect says:

- Watch the stream for a `SignInSuccess` action
- Get the payload from the `SignInSuccess` action
- Dispatch `LoadDashboardData` with the user's `companyId`
- Dispatch `LoadChatData` with the user's `userId`

. . .

These examples are really just a start. They're not necessarily great examples naming-wise, and knowing what to do (or not do) in effects is tricky to figure out, and depends largely on your app.

If anything was unclear or you think there's another example which deserves to be included, leave a message in the comments :)

Was this article helpful? Clap so I'll know to write more!

[Angular](#) [NgRx](#) [Store](#)

[About](#) [Help](#) [Legal](#)