NgRx: Action Creators redesigned

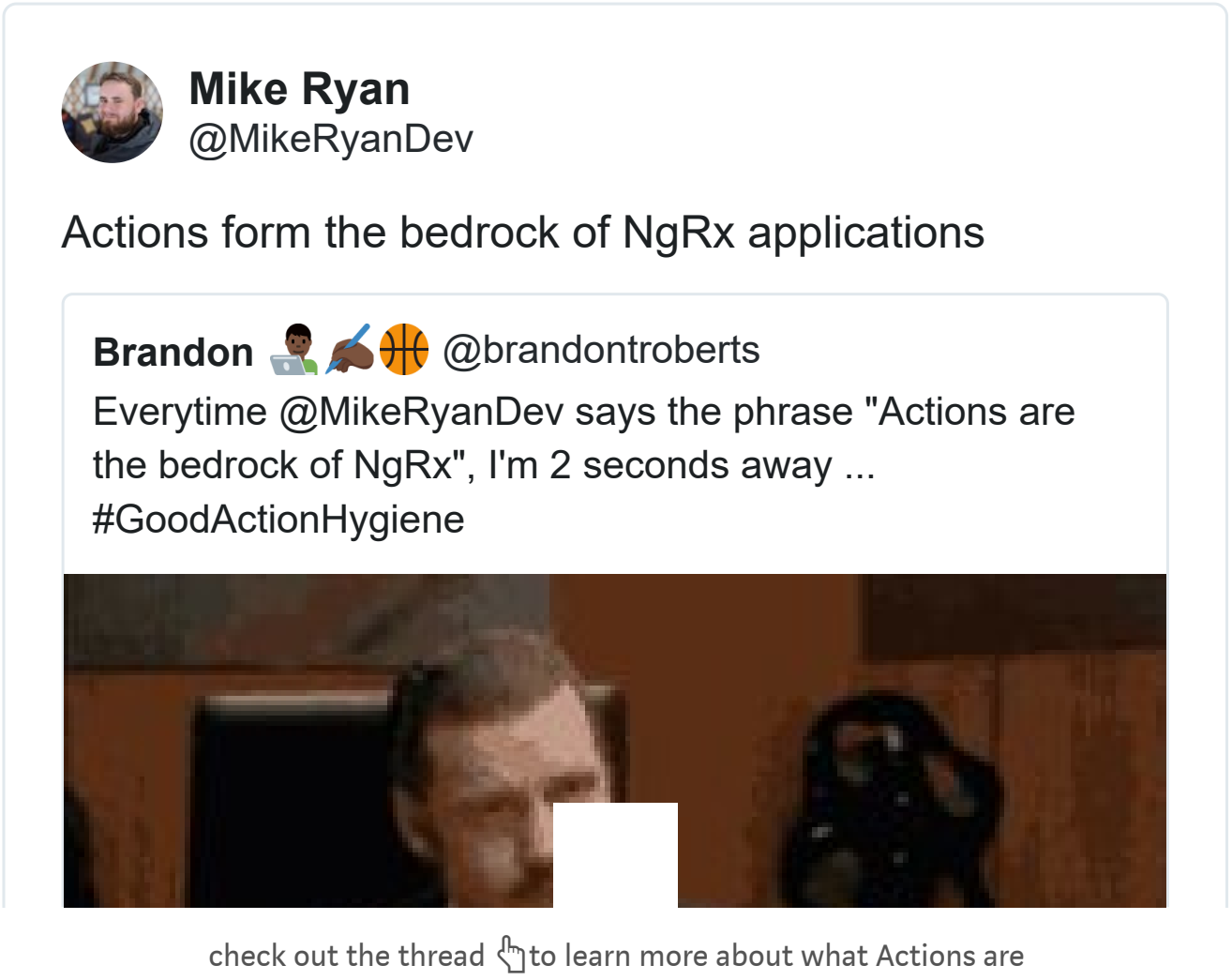
Alex Okrushko Follow
Apr 8 · 7 min read

Trimming Action Creators to the bare minimum that we actually need

In this article, we'll look into the limitations of the current Action Creators and some techniques that can help with them. Then I'll walk you through the new addition to the core NgRx— `createAction` function, where I'll highlight some of its advantages, how it can be used in `ofType` operator, and discuss Action Union.

Background

Actions are a core part of NgRx, or as some say



They are what glues the entire state management together.

However, over the years of maintaining NgRx at Google I've heard many times that Actions:

- Feel heavy and require quite a bit of boilerplate

- Are hard to track where they are **consumed from** and where they are **dispatched from**

What's so hard about tracking the Actions? Let's start by taking a look at the current Action Creators.

Action Creators

Action Creators are special functions or classes that help remove *some* of the boilerplate (Tim Deschryver described them very well in his article). They, however, also introduce a disconnect:

- Reducers and Effects rely on `type`, which is typically either an enum or a string constant:
  In a Reducer: `switch (action.type) { case ACTION_TYPE_STRING: {…} }`
  or in an Effect: `ofType(ActionEnum.ACTION_TYPE)`
- Components or results in Effects use the Action Creator itself, `new MyAction()` for a class-based approach or `myAction()` for a function-based.

Let's take a look at the Action Creator example from ngrx.io:

```
export enum ActionTypes {
  Login = '[Login Page] Login',
}

export class Login implements Action {
  readonly type = ActionTypes.Login;
```

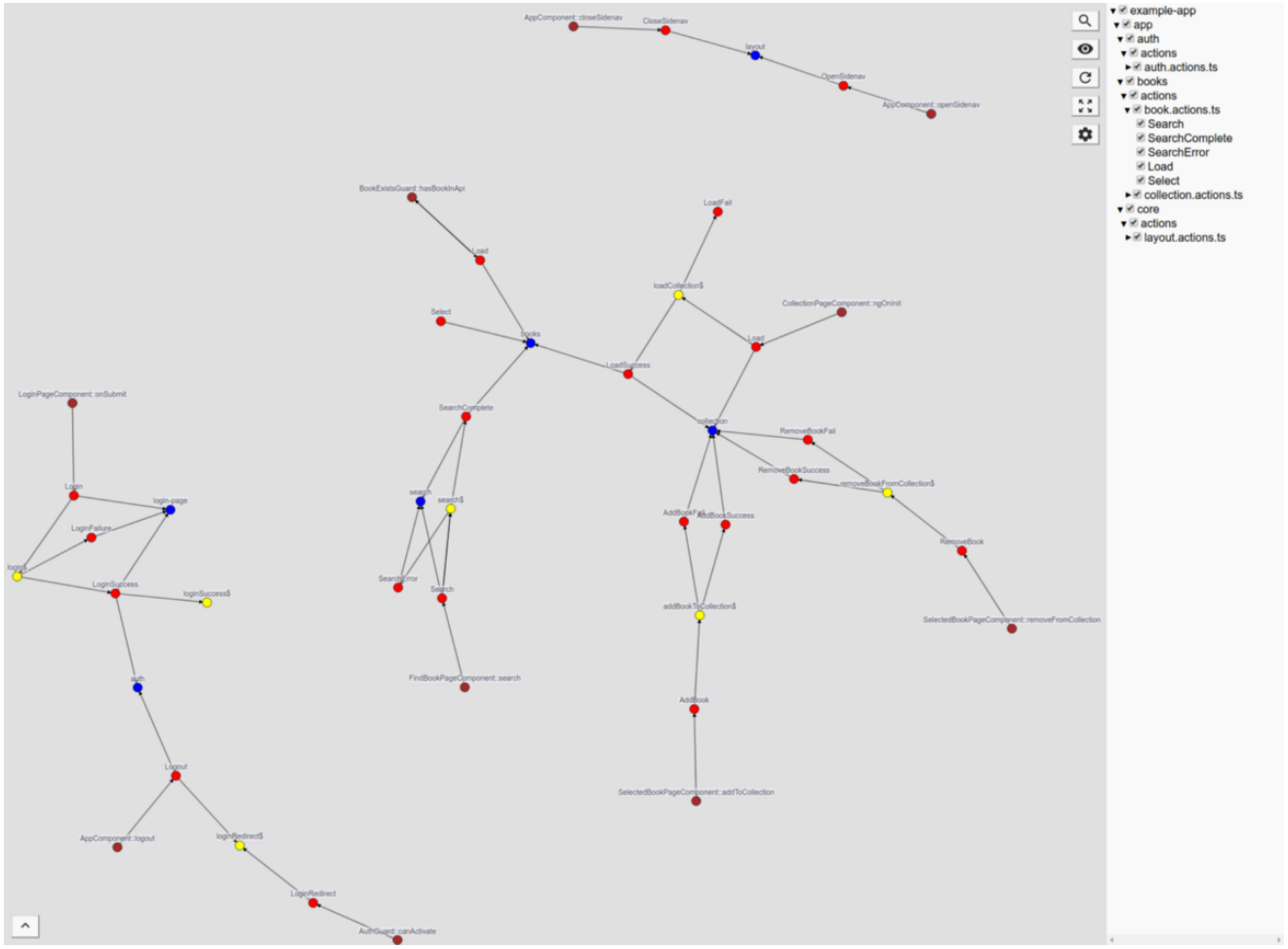
The Reducer will use `ActionTypes.Login` in the switch statement, which is an enum value in this case.
At Firebase Console (and mostly at all of Google) we settled on constant strings for types, and hence the `Login` Action would have the following shape:

```
/** see below - comment is required for all exported symbol
export const LOGIN = '[Login Page] Login',

interface LoginPayload {
  username: string;
  password: string;
}

/** Action to log in the User from the Login Page */
export class Login implements Action {
```

As you can see, that's a lot of code for something that is supposed to be unique and not reusable.

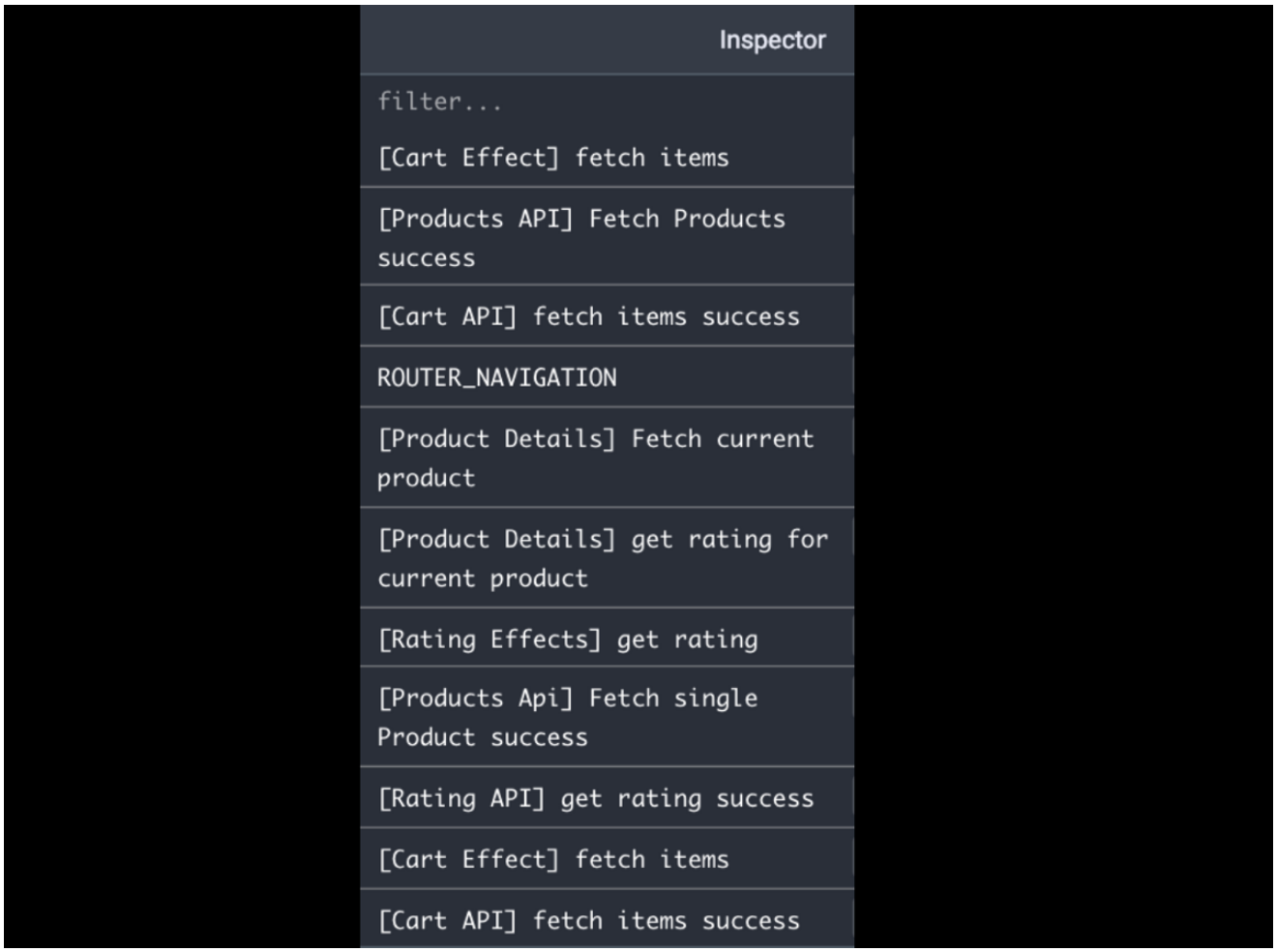
On top of that, enums or strings do not allow us to find **all the places the Action is used** throughout the codebase. For consumption, the enum/string has to be used—and for dispatching, we need to use the class.

It got to a point where our intern Idan Raiter created a tool that walks down AST, maps the Action dispatchers (Components and Effects) and consumers (Effects and Reducers) and **visualizes** their relationships in d3 graphs.



Visualization for example-app that is packaged with NgRx

Another approach that simplified tracking Actions is "Good Action Hygiene" (a term coined by Mike Ryan). It recommends using Actions strictly as unique events and adjusting Action types to include their sources explicitly, which makes the stream of Actions a lot more readable in DevTools.



Good Action Hygiene helps a lot, but only if the application is already running and it still doesn't address the problem of searching for Actions in the codebase.

Better NgRx

Internally at Firebase, Moshe Kolodny started a doc, where he outlined some of the improvements to NgRx that he thought could help make state management a bit less painful. One of such idea was to adjust the Action's class prototype and to include `type` as part of it. Users won't be forced to use enum/string for the type, and instead something like `Login.prototype.type` can be accepted instead.

I really liked the idea and some iterations later I managed to get to `Login.type` potential usage—it was addressing the searchability issue and reducing some boilerplate. This is how the initial proposal was drafted (that was eventually modified to its current state).

While I was working on the implementation, I stumbled upon ts-action—the Action Creators library by Nicholas Jamieson, and he further pointed me to the article here in Angular-in-Depth that covers this library very well (and that I somehow missed). It had everything that I drafted, and on top of that, it was solving the problem of properties/payload boilerplate, and basically addressing most of the complaints that I've heard about Actions.

After discussing it with the NgRx team, we decided to merge the `ts-action` Action Creator code into core NgRx, with some adjustments.

Improved Action Creators

The ergonomics of the new Action Creators are quite pleasant to work with. Starting from NgRx version `7.4.0` you can re-implement the `Login` Action from above:

```
/** Action to log in the User from the Login Page */
export const login = createAction(
    '[Login Page] Login',
    props<{username: string; password: string;}>(),
```

This Action feels a lot more light-weight and concise. The `createAction` function has some resemblance to `createSelector` and is quite easy to read.

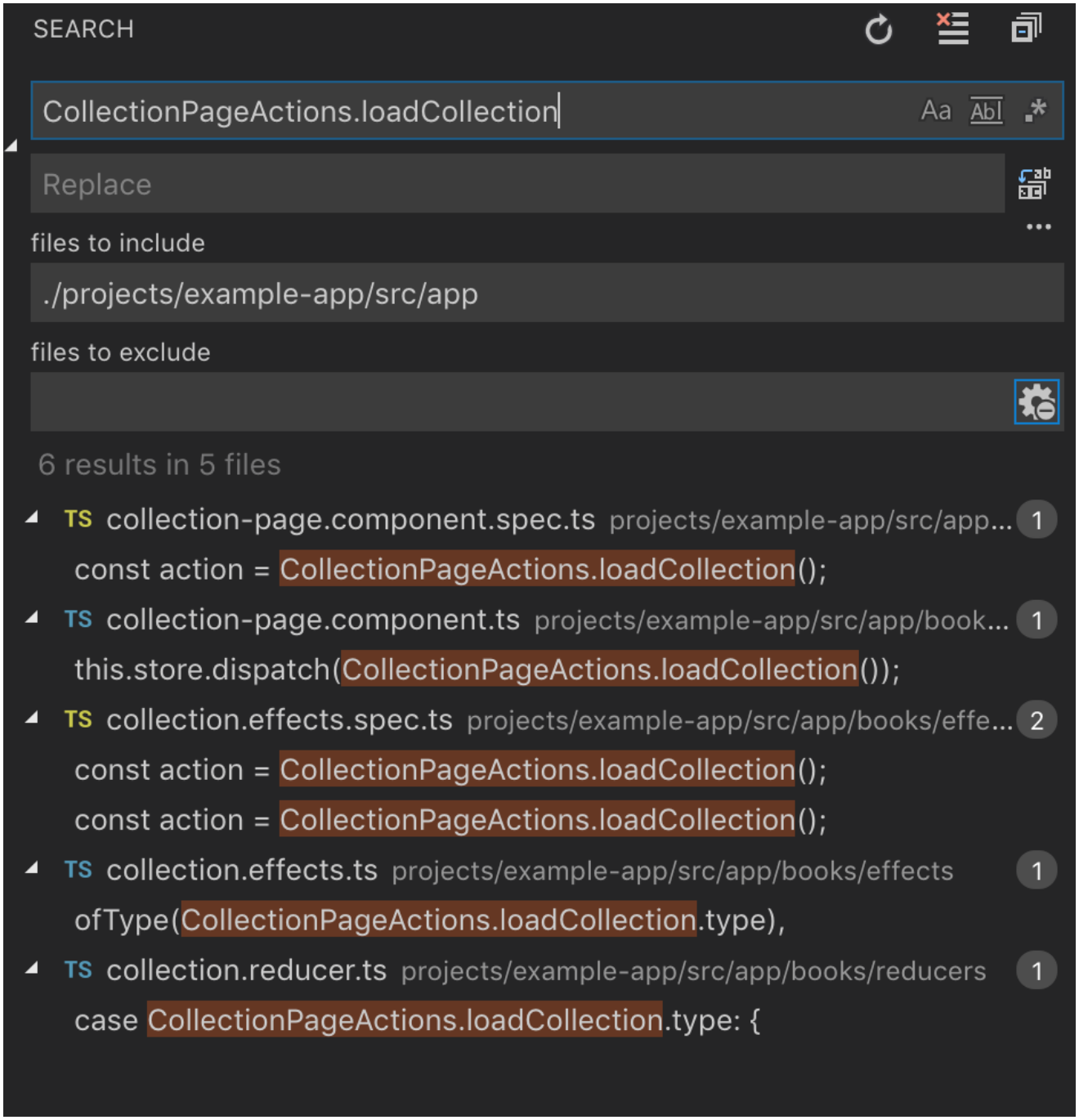
Here is how this Action Creator is used:

`login({username: 'Tim', password: 'NgRx'})`

Now, in the Reducers and Effects we will use the same `login` function. This function has the `type` property attached to it, so `login.type` is all we need:

```
function reducer(state, action) {
  switch(action.type) {
    case login.type: { // <----- usage in reducer
    ...
    }
  }
}


@Effect
```

Searching for all the usages is also a breeze, look at this example, where I'm looking for the `loadCollection` Action:



Obsolete Payload

When I write Actions I'd like to be explicit about which properties get argument values (when there is more than a single property), so I pass

named arguments:

```
new Login({username: 'Tim', password: 'NgRx'})
```

However, classes cannot destructure the passed object and assign it to properties in the constructor itself—this feature was requested a long time ago for Typescript. To bypass this limitation, the `payload` is typically used:

```
constructor(readonly payload: LoginPayload) {}
```

Wrapping properties in the extra `payload`, in turn, would require us to unwrap it where the Action is used:

```
service.loginUser(action.payload.username,
action.payload.password)
```

This was an inconvenience I was willing to go with to get the named arguments, but something that I never really enjoyed doing.

With the new Action Creator, payload could become a thing of the past—a `props()` function takes care of adding properties without payload!

```
service.loginUser(action.username, action.password)
```

Should you want to use the `payload` with the new Action Creators—you can still do that:

```
1
2  /** Action to log in the User from the Login Page */
3  export const login = createAction(
4      '[Login Page] Login',
5      // notice 👇 payload wrapping
```

> *What if I want to assign default values? Is it possible for me to pass the arguments without naming them?*

Yes, it is possible. The Action Creator can also take the `Creator` function as a second parameter:

```
1  /**
2   * Action to log in the User from the Login Page.
3   * Uses the most secure password by default.
4   */
5  export const login = createAction(
6      '[Login Page] Login',
```

Now we can create Actions with `login('Brandon')` or `login('Mike', 'lessSecurePassword')` .

ofType in Effects

When we specify which Actions need to be handled by the Effect, we use the custom `ofType` operator and pass in the `type` as a string.

However, we can take it even further and pass just the Action Creator itself:

```
ofType(login.type) // <-- still works
ofType(login) // <-- reads even better
```

Mixing strings in Action Creators are also possible, here is the example from the spec file:

```
ofType(divide, 'ADD', square, 'SUBTRACT', multiply)
```

When used with *strings*, in order to provide typing correctly, `ofType` relies on the Actions union to be provided for the Actions class in the constructor. Then it narrows it down to the specific Action:

```
export class MyEffects {
  constructor(private readonly actions$: Actions<ActionsUni

  @Effect
  login$ = this.actions$.pipe(
    // ensures type safety only when ActionsUnion is provid
    ofType(login.type),
    ...
  )

  @Effect
```

That's another advantage that the Action Creator brings—it doesn't need that generic.

```
export class MyEffects {
  constructor(private readonly actions$: Actions) {}

  @Effect
  login$ = this.actions$.pipe(
    ofType(login), // provide type safety *without* ActionsU
```

> `ofType` *that takes Action Creator should be released soon (either* `7.5.0` *or* `8` *).*

Reducer?

With Action Creator in place and `ofType` adjusted to handle it better, there are some discussions about implementing a `createReducer` function that would take a Map of types/Action Creators instead of going through a `switch` statement. But that's a topic for another article.

Actions Union

In the Redux pattern (that NgRx follows) **ALL** dispatched Actions go through **ALL** Reducers and Effects.

any dispatched action gets into Actions stream and goes through all reducers and all effects

Yet, it's impossible to combine the types of all of these Actions into one single type. That's why both `Actions<Action>` in Effects and `function reducer(state, action: Action)`, use the `Action` interface, that has `type: string`.

On the other hand, we want auto-completion and type checking. And for that to work, we need to pretend that we are working only with a limited subset of Actions—this is why we are creating the Actions Union type that we pass to our Reducers and Effects.

In the section **ofType in Effects** I explained how new Action Creators eliminate the need to provide a union to the `Actions` generic. **But we still need it for reducers** (at least until `createReducer` function is not part of NgRx—that would make Action Unions obsolete).

To create a union with *one* or *two* Actions, I recommend just combining them manually:

```
export type AuthApiActionsUnion = ReturnType<typeof loginSuccess | typeof loginFailure>;
```

When creating the union of 3+ Actions, there's a helper `union` function:

```
const all = union({login, loginSuccess, loginFailure, logout});
export type LoginActionsUnion = typeof all;
```

Unfortunately, the export cannot be combined into a single statement because TypeScript doesn't support it.

> *Note:* `union` *might change to take an array of Action Creators, so this could be a breaking change soon—we are still experimenting with it.*

Yes, this union is a bit crafty, so hopefully it will soon no longer be necessary.

Conclusion

The new Action Creator ticks all the boxes for me: it feels lighter, more concise, improves searchability and readability, helps to avoid payload and makes state management simpler.
NgRx's example-app has already been updated with the new Action Creators, so don't forget to check it out.