How to Reduce Action Boilerplate

Nicholas Jamieson Follow
Nov 7, 2017 · 6 min read

Photo by Andrew Wulf on Unsplash

I use Redux for my application development and, to take advantage of RxJS, I use NgRx in Angular projects and redux-observable in React projects. I also use TypeScript.

Unfortunately, the amount of boilerplate required for TypeScript to be effective with Redux can be disheartening. In his article *Introducing*

@ngrx/entity, Mike Ryan shows how `@ngrx/entity` can be used to write CRUD reducers with little code. It's great. And much appreciated. However, it doesn't help with the TypeScript cruft in action declarations.

In the past, I've resorted to code generation—using doT—to avoid the usual repetition. More recently, I've investigated alternative approaches and I've found one that's terse and suits my needs.

Before I introduce the library I've written, let's look at how TypeScript works with Redux.

TypeScript and Redux

Redux is fundamentally about the dispatch and receipt of actions, and TypeScript has benefits for both.

When dispatching an action, the use of action creators—rather than object literals—is recommended. There are a number of reasons for using action creators—including brevity, encapsulation and testability —but TypeScript offers another: type safety. Strongly typed actions will prevent the omission of required properties, the inclusion of unnecessary properties, and the inclusion of properties that have the incorrect type. However, there's nothing complicated here: you just create an action using a TypeScript class or method and pass it to `dispatch`. It's where actions are received that problems arise.

In Redux, actions are simple, anonymous objects, so when an action is received, its `type` property is all that there is to work with. (If you create an action using a class, there is no guarantee that when it's received it will still be an instance of that class—it could be an action replayed by the Redux DevTools—so `instanceof` cannot be used.)

Typically, the base Redux action will be defined using an interface and will look like this:

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

When a reducer receives an action like this:

```
{
  "type": "ADD_TODO",
  "text": "Write another Medium article.",
  "id": 1
```

We want to work with it not as an `Action`, but as a type that also includes the `text` and `id` properties. In TypeScript, this is referred to as type narrowing and there are two mechanisms for performing narrowing: type guards; and discriminated unions.

Narrowing with type guards

TypeScript supports `typeof` and `instanceof` type guards, but for Redux actions, a user-defined type guard is what's required. A user-defined type guard is a function that performs a run-time check to evaluate its returned type predicate.

For example, if we have this interface:

```
interface AddTodo extends Action {
  id: number;
  text: string;
}
```

We can write a user-defined type guard to determine whether an `Action` is an `AddTodo` action. The user-defined type guard looks like this:

```
function isAddTodo(action: Action): action is AddTodo {
  return action && action.type === "ADD_TODO";
}
```

Of particular interest is the function signature's return type: `action is AddTodo`. This is the type predicate and it's what makes the function a type guard.

We can use our type guard to write a reducer like this (the example reducers in this article perform some basic CRUD actions by manipulating arrays; if you are using NgRx, I'd recommend using `@ngrx/entity` instead):

```
function todosReducer(state: State = [], action: Action): St
  if (isAddTodo(action)) {
    const { id, text } = action;
    return [ ...state, { id, text, completed: false }];
  }
```

TypeScript will recognise the use of the type guard and, inside the `if` statement, it will be aware that the `action` instance is of the type `AddTodo` and will have `text` and `id` properties.

Narrowing with discriminated unions

Discriminated unions are best explained by example, so lets create one using these interfaces:

```
interface AddTodo {
  type: "ADD_TODO";
  id: number;
  text: string;
}

interface RemoveTodo {
```

Both are Redux actions, so they each have `type` properties, but it's the types of those properties that are important. The `type` properties in the interfaces are not declared as `string`; instead, the are declared using distinct, string-literal types: `"ADD_TODO"`; and `"REMOVE_TODO"`.

A property with a string-literal type is not just a string; it's a string that can only have the specified value. So the `type` property in `AddTodo` can only have a value of `"ADD_TODO"` . TypeScript is able to narrow a union of types in which all of the types share a common, string-literal property.

With the interfaces, we can write a reducer like this:

```
function todosReducer(state: State = [], action: AddTodo |
  switch (action.type) {
    case "ADD_TODO": {
      const { id, text } = action;
      return [ ...state, { ...action, completed: false }];
    }
    case "REMOVE_TODO": {
      const { id } = action;
      return state.filter(t => t.id !== id);
```

Note the type of the reducer function's `action` parameter: `AddTodo | RemoveTodo` . It's a union type and it tells TypeScript that the action parameter will be either `AddTodo` or `RemoveTodo` . With that information—and the with the common, distinct, string-literal `type` properties in the interfaces—TypeScript will narrow the type of `action` within the `case` statements.

Now that we've looked at the two narrowing mechanisms, let's look at how they are used in two different implementations: NgRx and `typescript-fsa` .

Narrowing actions with NgRx

The approach NgRx takes—as illustrated Mike's article—uses a discriminated union for the narrowing. Classes are used as action creators and their declarations look like this:

```
import { Action } from "@ngrx/store";

export enum TodoActionTypes {
  ADD_TODO = "ADD_TODO",
  REMOVE_TODO = "REMOVE_TODO"
}

export class AddTodo implements Action {
  readonly type = TodoActionTypes.ADD_TODO;
  constructor(public id: number, public text: string) {}
}

```

The constants associated with the actions' `type` properties are declared in an `enum` and the `enum` is exported, so that its members can be used in reducers.

Actions are declared as classes and the `enum` constants are assigned to the classes' `type` properties. Because the `type` properties are `readonly`, TypeScript will infer a string-literal type for the property. This is a key point. If the type properties are not declared as `readonly`, TypeScript will widen the inferred type to `string`—as the properties could be re-assigned a string with an arbitrary value.

A union type—of all the action classes—is also exported for use in the reducer and the reducer looks something like this:

```
import { TodoActionTypes, TodoActions } from "./todo-action

function todoReducer(state: State = [], action: TodoActions
  switch (action.type) {
    case TodoActionTypes.ADD_TODO: {
      const { id, text } = action;
      return [ ...state, { ...action, completed: false }];
    }
    case TodoActionTypes.REMOVE_TODO: {
      const { id } = action;
```

Narrowing actions with typescript-fsa

`typescript-fsa` is a small action-creator library for Flux-standard actions—which have this shape:

```
interface Action<P> {
  type: string;
  payload: P;
  error?: boolean;
  meta?: Object;
```

It takes a different approach to NgRx and uses user-defined type guards to perform the narrowing. Instead of using classes as action creators, `typescript-fsa` uses functions that accept anonymous objects that have a specified shape. And those action creator functions are created using a factory, like this:

```
import { actionCreatorFactory } from "typescript-fsa";

const actionCreator = actionCreatorFactory();
export const addTodo = actionCreator<{ id: number, text: str
```

And, in the reducer, the narrowing is performed with the `isType` function—which is a user-defined type guard:

```
import { isType } from "typescript-fsa";
import { addTodo, removeTodo } from "./todo-actions";

function todoReducer(state: State = [], action: Action): St
  if (isType(action, addTodo)) {
    const { id, text } = action;
    return [ ...state, { ...action, completed: false }];
  }
  if (isType(action, removeTodo)) {
    const { id } = action;
```

Narrowing actions with ts-action

The approach taken by `typescript-fsa` involves less boilerplate than that taken by NgRx. However, there were a number of reasons why I was reluctant to adopt `typescript-fsa` :

- the actions I'd been using didn't have the shape of Flux-standard actions—it's common in Redux to add properties at the same level as the `type` property, rather than under a `payload` property; and
- all of my reducers used `switch` statements.

I just wanted to eliminate the code-generation in my projects, as it complicated the build process. I didn't want to have to change the action structure or re-write the reducers.

Instead, I wrote a library— `ts-action` —that takes a different approach and can narrow using either type guards or discriminated unions.

Like NgRx, `ts-action` uses classes as action creators. However, those classes are created through calls to its action method, so the action creator declarations look like this:

```
import { action, props } from "ts-action";

export const AddTodo = action("ADD_TODO", props<{ id: number
export const RemoveTodo = action("REMOVE_TODO", props<{ id:
```

The `props` method will place the specified properties at the same level as the `type` property. To place them inside a `payload` property, the `payload` method can be used instead.

`ts-action` also includes `base` method that allows for the base class to be specified inline, like this:

```
const AddTodo = action("ADD_TODO", base(class {
  constructor(public id: number, public text: string) {}
}));

const RemoveTodo = action("REMOVE_TODO", base(class {
```

Inline base classes offer flexibility for property defaults and initialization, etc.

The classes created by `ts-action` explicitly reset each action’s `prototype`, so that they are compatible with `reactjs/redux` —that is, so that each action is considered to be a plain object.

The action creators can be used in reducers that narrow using a discriminated union, like this:

```
import { union } from "ts-action";
import { AddTodo, RemoveTodo } from "./todo-actions";

const All = union(AddTodo, RemoveTodo);

function todoReducer(state: State = [], action: typeof All)
  switch (action.type) {
    case AddTodo.type: {
      const { id, text } = action;
      return [ ...state, { ...action, completed: false }];
    }
    case RemoveTodo.type: {
```

And they can be used in reducers that narrow using user-defined type guards, like this:

```
import { isType } from "ts-action";
import { AddTodo, RemoveTodo } from "./todo-actions";

function todoReducer(state: State = [], action: Action): St
  if (isType(action, AddTodo)) {
    const { id, text } = action;
    return [ ...state, { ...action, completed: false }];
  }
  if (isType(action, RemoveTodo)) {
    const { id } = action;
```

Narrowing observables

Being RxJS-based, NgRx also includes some methods that act like operators and can be used to filter and narrow an observable of actions. In particular, the `Actions` observable used with `@ngrx/effects` has an `ofType` property and it looks like this:

```
import { GetRepos, GitHubActions } from "./github-actions";

/* ... */
const effect = actions
  .ofType<GetRepos>(GitHubActions.GET_REPOS)
```

I wanted an operator that could be used with the action creators in `ts-action`, so I created `ts-action-operators`. It contains an `ofType` operator that accepts an action creator:

```
import { GetRepos } from "./github-actions";

/* ... */
const effect = actions
  .ofType(GetRepos)
```

With all of the information in the action creator, there is no need to specify a type parameter, as the `ofType` operator is implemented using the `isType` user-defined type guard in `ts-action.`

`ts-action-operators` is independent of NgRx, so it can be used with redux-observable epics, too.

Overall, I’m pleased with `ts-action`. With it, I've been able to remove a reasonable amount of boilerplate, and, building it, I’ve learned rather a lot about some of TypeScript’s less-often-used features. And what I’ve learned will likely be the subject of my next article.