Angular State Managment — Don’t fear the boilerplate

Fear tight coupling!

Christian Janker Follow
Jun 2, 2018 · 7 min read

. . .

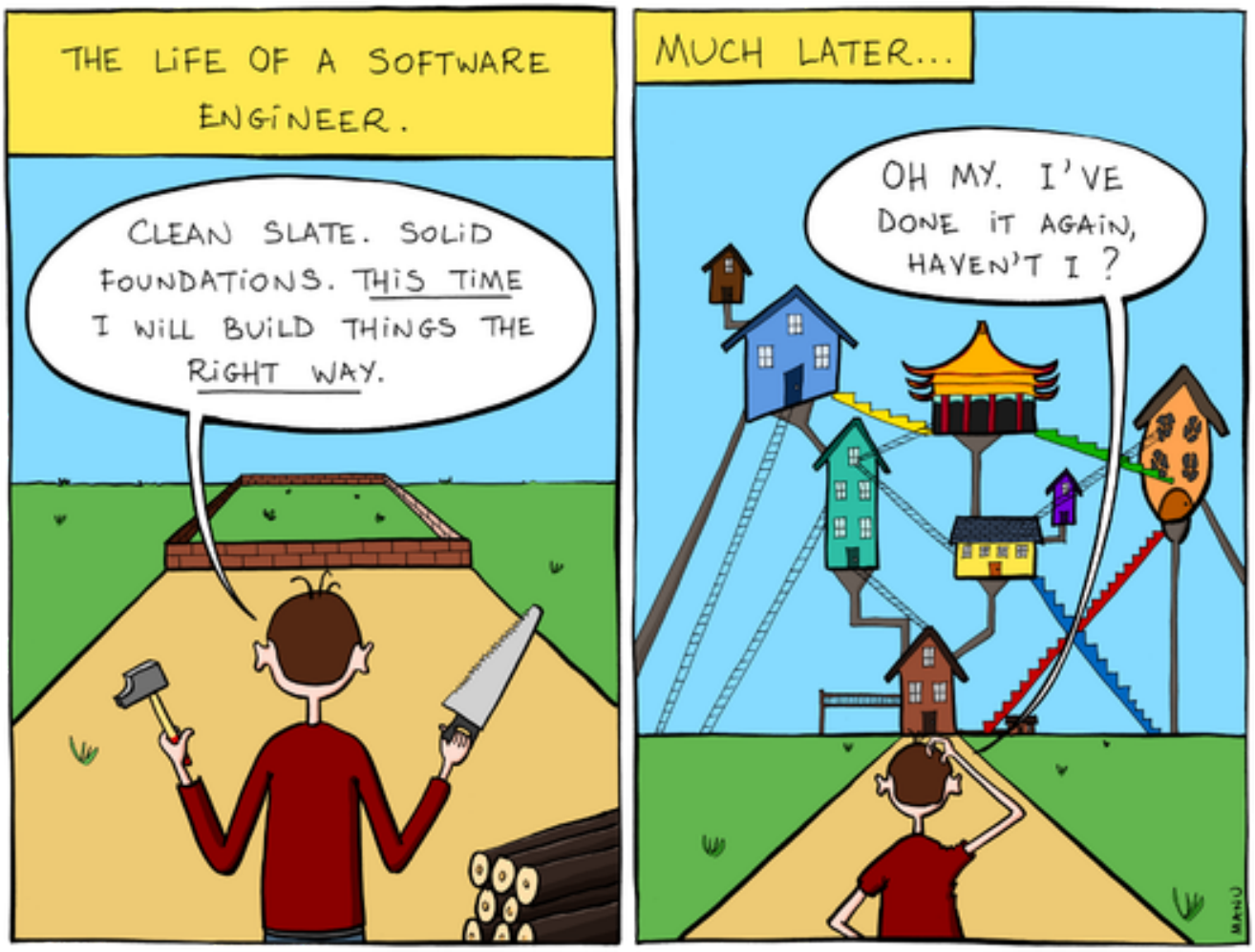
Many people rant about the boilerplate that is produced when introducing a redux based state management library in an Angular application. All I can respond to that is:

> *Don’t fear the boilerplate. Fear tight coupling!*

Though, I can understand those people, they are right in some way. State management can become hairy really fast. Especially if your are working in a bigger team with different levels of experience.

I think it’s not the boilerplate that causes the most pain. Sometimes it’s just the lack of clear separation of concerns. The absence of well defined smart- and dumb components. A state that is not well structured or simply too big to be handled by one container component. Complex subscriptions within the components that may combine multiple observables.

Add some side effects on the top. Add the routing information to the state. And suddenly:



Redux architecture gone wrong

The code is not maintainable anymore. The **Redux** architecture had promised to solve the complexity of not knowing where the state of the application is coming from, but now it is really hard to follow the flow of data through your app as well. You get angry. You blame your state management decision or the one that made it.

Please hold on for a moment :)

In this post I present an approach how you can “manage” your state management library. I will use **ngrx** in my examples, but it doesn’t really matter which redux based library you are using. The concept stays the same. Even plain old services could be used to hold your state.

1 Use Typescript

This is a no brainer for **Angular** developers. Once you have got used to it you don’t wanna miss it out any longer. We will use its power to define our typed actions and use them within reducers and effects. So coding errors occur during compile time as we develop and not during runtime. Let’s directly dive into our action types:

```
/*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/*                              TYPES                                 */
/*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
type Type = {
    readonly type: string;
}

type Payload<T> = {
    readonly payload: T;
```

Action Interfaces

This is a pretty straight forward definition of a generic `Action` type that is combined of the types `Type` and `Payload`. So effectively every object with the form of `{type, payload}` confirms to this type.

We could use this now to create a function that returns a specific action and dispatch it then via the **ngrx** store.dispatch:

```
// ActionCreator function
function LoadCommits(user: Pick<User, 'username'>): Action<
  return {
      type: '[Commits Page] Load Commits',
      payload: user
  };
}

const user = new User('vanych');
```

LoadCommits() ActionCreator function

Fair enough. This works well for one action creator function, but it’s not really satisfying. We didn’t take reducers and effects under consideration and when the application grows we would have to write a lot of boilerplate and duplicate code.

Let’s improve our design with some factory functions. They help us to cut the boilerplate down a little bit:

```
/*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/*                TYPES                                  */
/*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */

...

// A ActionCreator is a function that takes a payload and
// creates a dispatchable Action from it
type ActionCreator<T> = {
    (payload: T): Action<T>;
};

/*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/*                FACTORY FUNCTIONS                      */
/*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
function createType(type: string): Type {
    return { type };
}

function createPayload<T>(payload: T): Payload<T> {
    return { payload };
}
```

We can now use `createType` and `createActionCreator` to define our `CommitActions` :

```
export class CommitActions {
    /*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
    /*              TYPE DEFINITIONS                        */
    /*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
   static LOAD_COMMITS =           createType('[Commits Lis
   static LOAD_COMMITS_SUCCESS =   createType('[Commits Lis
   static LOAD_COMMITS_FAILURE =   createType('[Commits Lis
    /*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
    /*              ACTION CREATORS                         */
    /*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
```

This is an alternative to the class based action definition approach, which is widely adopted by **ngrx** users. Now we are ready to dispatch an action from a container component:

```
import { Store } from '@ngrx/store';
import { CommitActions } from '../store/actions';

@Component({
   selector: 'commits-container',
   templateUrl: './commits.container.html',
   styleUrls: ['./commits.container.scss'],
   changeDetection: ChangeDetectionStrategy.OnPush
})
export class CommitsContainer {
   commits: Observable<Commit[]>;
```

Dispatch action via ngrx store

The next step to do is to set up the reducer. The cool part about the reducers and effects is that they support type checking during compile time as well.

```
export interface CommitState {
    commits: Commit[];
}

export const initialState: CommitState = {
    commits: []
};

export function reducer(state = initialState, action: Actio

    if (isAction(action, CommitActions.LoadCommits)) {
        return state;
    }

    if (isAction(action, CommitActions.LoadCommitsSuccess))
        const commits = [...action.payload];
        return {
            ...state,
            commits
        };
    }
```

commits reducer

The `isAction` function does not only check if one of the expectedActions matches, it also allows us to use a typed payload within the if statements thanks to Typescript Type Guards.

```
function isAction<T>(action: Action<any>, ...expectedActionC
    return expectedActionCreators.some(expectedAction => exp
}
```

The type `TypedActionCreator` is a function that also keeps the action-type as a property on it. This feels a little bit hacky, but it allows us to do the `isAction` check without creating further boilerplate.

```
/*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/*                TYPES                                     */
/*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
type Type = {
    readonly type: string;
}

type Payload<T> = {
    readonly payload: T;
}

type Action<T>  = Type & Payload<T>;

type ActionCreator<T> = {
```

Check out the full source on github.

2 Think about Dependency Injection

Dependency Injection is one of the core features of Angular. It simplifies a lot for us developers, namely testing and nested dependencies. But nothing comes without a price.

At first sight it seems like a blessing to be able to inject everything everywhere else that easy. The problem I see is that developers stop thinking about the dependencies they are pulling into their components. Think about it:

> *Do I really wanna have a direct dependency on a 3rd party library in my component?*

This introduces tight coupling. Tight coupling is bad.

So we don’t want to have a direct dependency on the state management library in our container components.

No:

— shake your head —

```
import { Store } from '@ngrx/store'; // <-- u are a bad hab
//
// CommitsContainer is holding a list of git commits of a g
//
@Component({
    selector: 'commits-container',
    templateUrl: './commits.container.html',
    styleUrls: ['./commits.container.scss'],
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class CommitsContainer {
    commits: Observable<Commit[]>;
```

Unfortunately this is how it’s done in every other **ngrx** example online. Just because you’v been told that container components are the smart ones, does not necessarily mean that they have to pull in every third party library as a dependency. There is a better way.

Yes:

— nod in approval —

```
import { CommitActions } from '../domain/commit-actions';
@Component({
   selector: 'commits-container',
   templateUrl: './commits.container.html',
   styleUrls: ['./commits.container.scss'],
   changeDetection: ChangeDetectionStrategy.OnPush
})
export class CommitsContainer implements OnInit {
   commits: Observable<Commit[]>;

```

Now you can’t even tell that there is **ngrx** behind it or even that it is a redux pattern we are applying. Just the names of the variables and the name of the types let you guess that we are doing redux here. But I could rename those:

```
import { CommitCommands } from '../domain/commit-commands';

@Component({
   selector: 'commits-container',
   templateUrl: './commits.container.html',
   styleUrls: ['./commits.container.scss'],
   changeDetection: ChangeDetectionStrategy.OnPush
})
export class CommitsContainer implements OnInit {
   commits: Observable<Commit[]>;
```

I have to admit, the following is just a tiny detail and I still have to think about it. We could rename the variables to something more meaningful for our business domain. When we are talking in redux language and we are within the redux context the term ‘action’ is definitely valid. But when we are in our container components, talking the language of the business, then an action is more like a **command** (e.g *LOAD_COMMITS*). When the command is completed there is an **event** (e.g *LOAD_COMMITS_SUCCESS, LOAD_COMMITS_FAILED*) fired based on its outcome.

Pulling out dependencies also helps us increasing the test stability. Testing Rule #1 (at least for me) says: **Don’t mock what you don’t own**. Why? Because it is out of your control. If something in a third library changes, which we are not aware of, it eventually breaks all our tests. **The tests are breaking then for the wrong reason**. They should break if someone changes behaviour in our application logic and not if the state management tool got changed. We should have special contract tests to check the integration with your state management library. Those should fail if the library changes in an unexpected way.

What did we win?—Loose coupling—Clean Code—Maintainability—Wanna replace your redux implementation or current state management solution without going nuts? No problem now :)

But how is this even working? Where is the dispatch code? Let me introduce you to **Bound Actions**. This is not a new concept, it’s just not widely adopted by the angular community. But it’s definitely worth checking out.

```
/*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  */
/*                  FACTORY FUNCTIONS                         */
/*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  */
function createType(type: string): Type {
    return { type };
}

function createPayload<T>(payload: T): Payload<T> {
    return { payload };
}

function createAction<T>(type: Type, payload: Payload<T>):
    return {
        type: type.type,
        payload: payload.payload
    };
}

function createActionCreator<T>(type: Type): ActionCreator<
    return (payload: T) => createAction(type, createPayload
```

The `createBoundActionCreator` binds a given `ActionCreator` to a dispatch function. It returns a function which takes the payload, then creates the `Action` and directly dispatches it with the help of the given dispatch function. So we can hide the **ngrx** dependency from the action caller.

Lets define a bound action in our already existing `CommitActions` class. Therefor the `CommitActions` class has to become an Angular

service, which injects the **ngrx** store and then binds the dispatch function to the action function:

```
import { Store } from '@ngrx/store';
import { Injectable } from '@angular/core';

import { User } from './user';
import { Commit } from './commit';
import { createType, createActionCreator, createBoundAction


@Injectable()
export class CommitActions {
    /*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
    /*              TYPE DEFINITIONS                     */
    /*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
   static LOAD_COMMITS =           createType('[Commits Lis
   static LOAD_COMMITS_SUCCESS =   createType('[Commits Lis
   static LOAD_COMMITS_FAILURE =   createType('[Commits Lis
    /*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
    /*              ACTION CREATORS                      */
    /*  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
   static LoadCommits = createActionCreator<Pick<User, 'use
```

Now the container component does not have an direct dependency to **ngrx** anymore. This is what we have wanted to achieve.

```
import { CommitActions } from '../domain/commit-actions';
@Component({
    selector: 'commits-container',
    templateUrl: './commits.container.html',
    styleUrls: ['./commits.container.scss'],
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class CommitsContainer implements OnInit {
    commits: Observable<Commit[]>;

```

3 Use selectors

Until now we only have sent of a command to kick off a side effect (sending an HTTP call) and finally load the commits into the store. But we still have to show them to the user. Therefor we have to select the data from the Redux store from within our `CommitsContainer` component. This could be done directly if we inject the store service from **ngrx** into it.
Unfortunately with this approach, we get an unwanted dependency to our state management library again. Therefor we create a `CommitSelector` service, which contains our selectors and can be injected into the container component to effectively hide the **ngrx** dependency from it.

```
// Selectors - Pure Functions
export const commitsSelector = (state: AppState) => state.c

@Injectable()
export class CommitSelectors {
    constructor(private store: Store<any>) {}

    selectCommits(): Observable<Commit[]> {
```

Selector Service

In the CommitsContainer we can now use this service and select the data and pass it to our presentational component in the template:

```
@Component({
    selector: 'commits-container',
    templateUrl: './commits.container.html',
    styleUrls: ['./commits.container.scss'],
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class CommitsContainer implements OnInit, OnDestroy
    commits$: Observable<Commit[]>;

    constructor(private actions: CommitActions,
                private selectors: CommitSelectors) {
```

```
<commit-list [commits]="commits$ | async"></commit-list>
```

CommitsContainer Template

Another really handsome feature of selectors is the ability to combine them. So multiple small selectors can be combined to a complex one. **Ngrx** comes with custom `createSelector` functions. But you could also use the reselect library.

Let’s look at an example. In the code above we hardcoded the username. Now we would like to pass it dynamically via a route url param.

I have extended the example to use **ngrx/router-store** to sync the current router state to our **ngrx** store. This allows us to use the power of combined selectors, where the `routeSelector` fetches the router state from the state and the `routeParamSelector` reads the given route param from it.

```
export const routeSelector = createFeatureSelector('router')
export const routeParamSelector = (paramName: string) => (ro

export const usernameSelector = createSelector(
    routeSelector,
    routeParamSelector('username')
```

Combine selectors

Now we can use the selector in our `CommitSelectors` service:

```
@Injectable()
export class CommitSelectors {
    constructor(private store: Store<any>) {}

    selectUsername(): Observable<string> {
        return this.store.select(usernameSelector);
    }

    selectCommits(): Observable<Commit[]> {
```

The `CommitsContainer` describes to username observable and on every change it executes the loadCommits action:

```
@Component({
    selector: 'commits-container',
    templateUrl: './commits.container.html',
    styleUrls: ['./commits.container.scss'],
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class CommitsContainer implements OnInit, OnDestroy
    username$: Observable<string>;
    commits$: Observable<Commit[]>;

    constructor(private actions: CommitActions,
                private selectors: CommitSelectors) {

        this.commits$ = this.selectors.selectCommits();
        this.username$ = this.selectors.selectUsername();

        this.username$
            .pipe(
                distinctUntilChanged(),
                untilComponentDestroyed(this)
```

We achieved the development of a component that has no direct dependency to the router nor to a state management library. This is an important step for gaining maintainable, testable and cleaner code that expresses its business domain more clearly.

In the next post I will probably go a step further and introduce a `ConnectMixin` which will turn our container components effectively

into dumb ones, which just use `@Input()` and `@Output()` to connect to the state.

Check out the full source on github.

Follow me on Twitter :)

codeburst.io

> ✉ Subscribe to *CodeBurst’s* once-weekly **Email Blast**, 🔔 Follow *CodeBurst* on **Twitter**, view **The 2018 Web Developer Roadmap**, and **Learn Full Stack Web Development**.