

NGRX

Reactive State Management for Angular



Nils Mehlhorn

NgRx - Reactive State Management for Angular

Nils Mehlhorn

This version was published on 2020-11-24.

Copyright © 2020 Nils Mehlhorn. All Rights Reserved.

The official NgRx logo is used on the cover under [CC BY 4.0](#)

This book is for sale at gum.co/angular-ngrx-book

Contents

Preface	5
Share This Book	5
Feedback	5
Acknowledgements	5
About the Author	6
1 Introduction	7
1.1 Motivation	7
1.2 Concepts and Terminology	11
1.3 NgRx Is Not Just A Store	13
1.4 You Might Not Need NgRx	13
2 Example App	15
3 Installation	17
4 First Steps	19
5 Debugging	24
6 File Structure and Naming	28
7 State	32
7.1 Normalization	34
7.2 What Not to Put in the State	36
8 Actions	40
8.1 Action Creators	41
8.2 Action Hygiene	45
9 Reducers	46
9.1 Creating Reducers	47
9.2 Registering Reducers	49
9.3 Mutable APIs with immer.js	50
9.4 Meta-Reducers	52
9.5 Error Handling	55
10 Selectors	56
10.1 Computed Selectors	58
10.2 Parameterized Selectors	61
10.3 Pipeable Selectors	63
11 Fat vs. Thin Actions and Reducers	65
12 Feature Modules	68
12.1 Multiple Reducers per Module	73
12.2 Deciding between root and feature state	75

13 Effects	77
13.1 Installation	77
13.2 Creating Effects	80
13.3 Accessing the State	84
13.4 Error Handling	87
13.5 Optimistic vs. Pessimistic Updates	90
13.6 Initial Data and Effects	92
13.7 Non-Dispatching Effects	94
13.8 Other Effect Sources	96
14 Testing	99
14.1 Testing Reducers	99
14.2 Test Object Factories	102
14.3 Testing Action Creators	105
14.4 Testing Selectors	105
14.5 Testing Observables	107
14.6 Testing Effects	111
14.7 Testing Components and Services	114
15 Performance	118
15.1 OnPush Change Detection	119
15.2 Tracking List Elements	119
15.3 Efficient Handling of Remote Data	120
16 Patterns	121
16.1 Container and Presentational Components	121
16.2 Facades	125
16.3 Re-Hydration	128
17 Router Store	134
17.1 Installation	134
17.2 Selecting Router State	136
17.3 Reacting to Router Actions	138
18 Entity Abstraction	140
18.1 Installation	140
18.2 Entity State and Adapter	141
18.3 When to Use	144
19 Data Abstraction	145
Resources	151

Preface

Share This Book

Please help me by spreading the word about this book with your colleagues and on places like Twitter or LinkedIn. Here's something you can tweet:

I'm reading the [NgRx book for reactive state management in Angular](#) by [@n_mehlhorn](#) [#javascript](#)
[gum.co/angular-ngrx-book](#)

Also, it would mean the world to me if you left a five-star review on [Gumroad](#).

Feedback

If you have any feedback or questions, reach out to me on Twitter [@n_mehlhorn](#) or via email to contact@nils-mehlhorn.de.

Acknowledgements

Thanks to Simon Henke, Tim Deschryver, Gregor Woiwode, David Müllerchen and Alex Okrushko for reviewing this book. Moreover, I want to thank the NgRx, Angular and RxJS teams for their efforts. I'd also like to express my gratitude towards the whole Angular community for being welcoming and helping me learn and grow.

About the Author

I'm a freelance software engineer, trainer, speaker and author. While working on enterprise software, helping others to do the same, as well as building the online graphics tool startup [SceneLab](#), I became a big fan of the NgRx library. After writing multiple blog posts on advanced NgRx topics and building a [library for undo-redo](#), I've now put all my experience into this book to provide you with solid foundations and advanced patterns for approaching state management with NgRx in Angular.

You can follow me on [Twitter](#), connect with me on [LinkedIn](#), visit my [website](#) to read new articles and [work with me](#) to build user-focused solutions without sacrificing maintainability.



Chapter 1

Introduction

1.1 Motivation

NgRx (short for *Angular Reactive Extensions*) is a group of open-source libraries that's mainly concerned with state management in Angular applications. So, talking about NgRx mostly means talking about state. When starting out development with Angular you're probably not explicitly concerned with state or where it resides in your application. However, as your requirements grow, you may notice that some of the hardest tasks during development stem from updating and synchronizing states - and in modern web applications there's a couple of those and they're all over the place:

- view state: what's displayed?
- client state: where are we in the application? What's the data? What are the inputs and outputs?
- browser state: what's the URL? What's saved in the storage? Is the network online?
- server state: what's persisted in the database(s)?

You could break these apart further and probably mention additional ones - especially as platforms and tech in general progress more and more as time goes on. Basically, anything that can change within the context of your app may be called state.

The Angular framework, arguably even more than other ones, already has certain state management techniques built-in. Just consider one of the main building blocks: components. They act as a bridge between client and view state as they render and receive data via template bindings. At the same time, components are classes which can naturally encapsulate state through instance properties. This way, components aren't simply responsible for view synchronization, but also become state containers. That's totally fine, yet can get difficult when you need the same data in multiple components. In Angular you'd overcome such difficulties through `@Input()` and `@Output()` bindings between parent and child components. Consequently, your state will flow along the view hierarchy.

This gets tricky when some part of state has to be shared between components that are fairly distant in terms of the view hierarchy. You'd pull a lot of state into higher up components, if not into the root

component itself while other components might forward data that they're not really concerned with. We end up with these messy hybrids of state containers and view-state bridges only to serve the way in which the latter are organized.

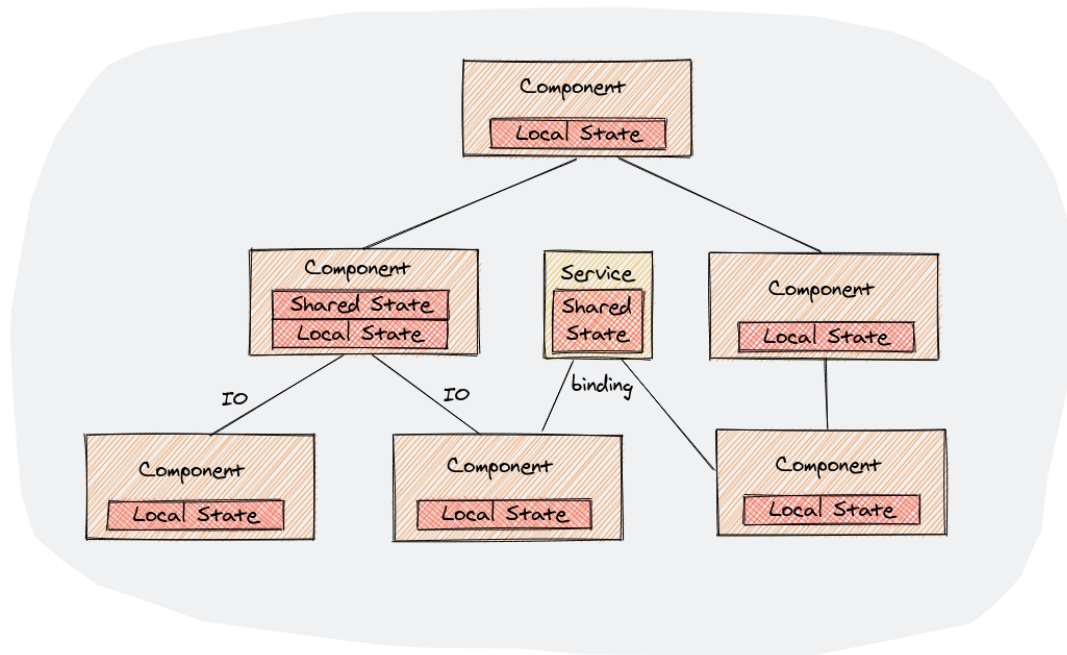


Figure 1.1: In Angular, state can be shared via inputs and outputs but also by binding to services outside of the view hierarchy. This way you don't need to funnel state across various parents when connecting otherwise distant components.

Again, Angular offers a solution: services. These class instances live outside of the view hierarchy and can be injected into any component - they're perfect for sharing state. If we'd be talking about other frameworks, this could've already been the point for proposing a solution in the vein of `NgRx`. But when we're working with Angular, services already enable us to build dedicated state containers while components can focus on rendering views and triggering state changes back in a service. The centerpiece of `NgRx` is in fact a state container service, but it's also a bit more than that.

Now, we've put our state into services, sharing data gets easier, but we still encounter problems with managing it in plain class properties. Particularly, it's hard to know when some state changes, especially when state objects can be mutated from all sorts of different places in our application. That's the point where Angular developers usually reach for observable streams and immutability. Components then listen to a stream of subsequent states while they send off commands via service methods. In practice, this approach is often based on an `RxJS` `BehaviorSubject`.

Essentially, we're introducing an indirection following the `Command Query Responsibility Segregation` (CQRS) pattern which gives us unidirectional data flow. There are now predefined ways in which state can change and we can be sure that all consumers will be notified of those changes (see Figure 1.2).

The thing is, while command and query are now technically separated, each component still has to know which command it has to send to which service in order to have its query resolved with the state

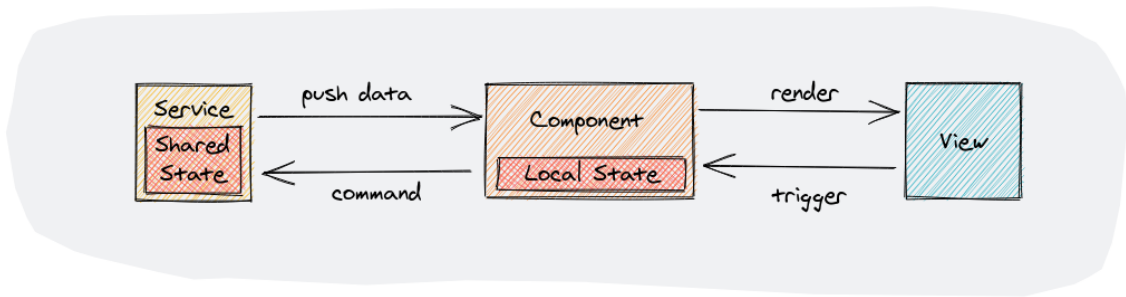


Figure 1.2: Separating commands and queries allows for unidirectional data flow and therefore optimized change detection and interception

it requires. As an application grows, the number of commands will do the same while some commands need to be propagated between different state containers. Throw asynchronism (e.g. HTTP requests) into the mix and it's easy to create an entangled mess of stateful services that's rife with race conditions (see Figure 1.3).

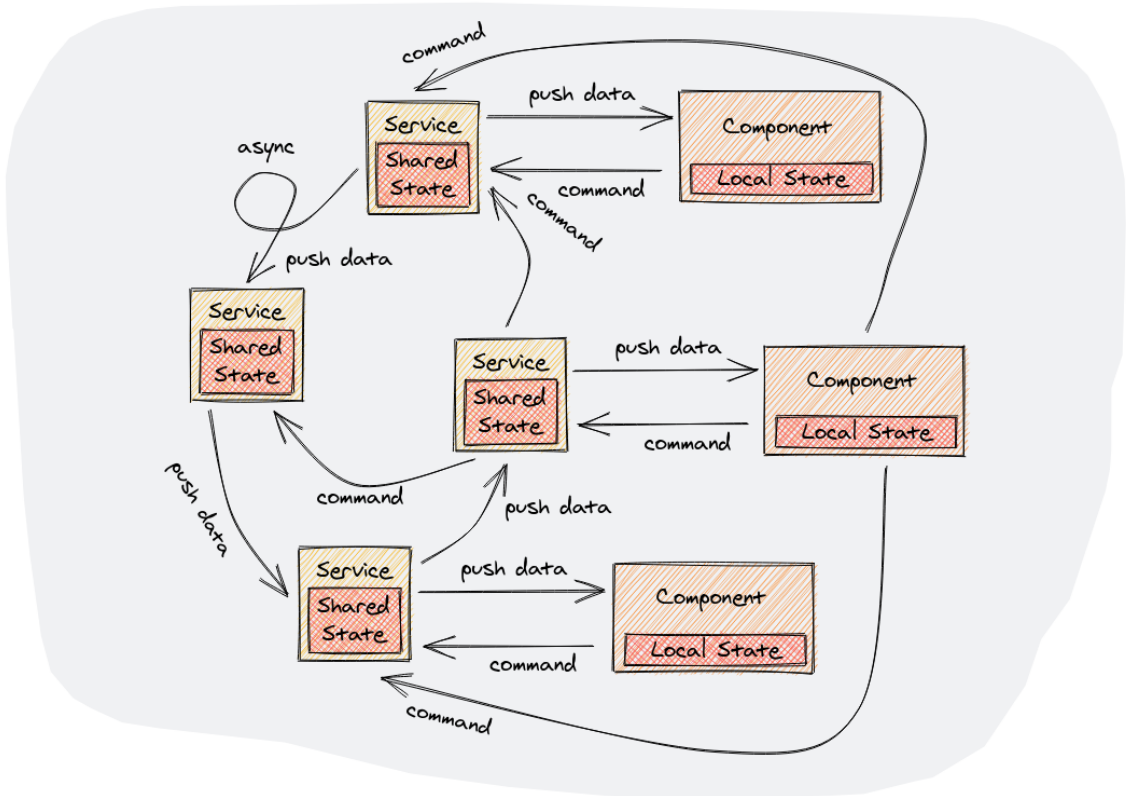


Figure 1.3: Synchronizing shared states can get messy, especially with a command-oriented architecture and when async tasks are involved

The NgRx solution: replace commands with events and introduce an event-bus. Instead of issuing commands to a specific service, we now broadcast events globally while each part of the state can react independently. Since we now no more know where the state is needed it has to reside on a global level. However, this way we also have a single-source of truth for shared state.

This second indirection is arguably even more scalable as we can just plug new receivers onto the

event-bus without modifying the sending side. Additionally, we can factor out asynchronism. Instead of having asynchronous command chains, tasks like an HTTP request can signal their completion through the event-bus. This way all considerations regarding state, while they still get complex as you're building complex applications, can remain comprehensible.

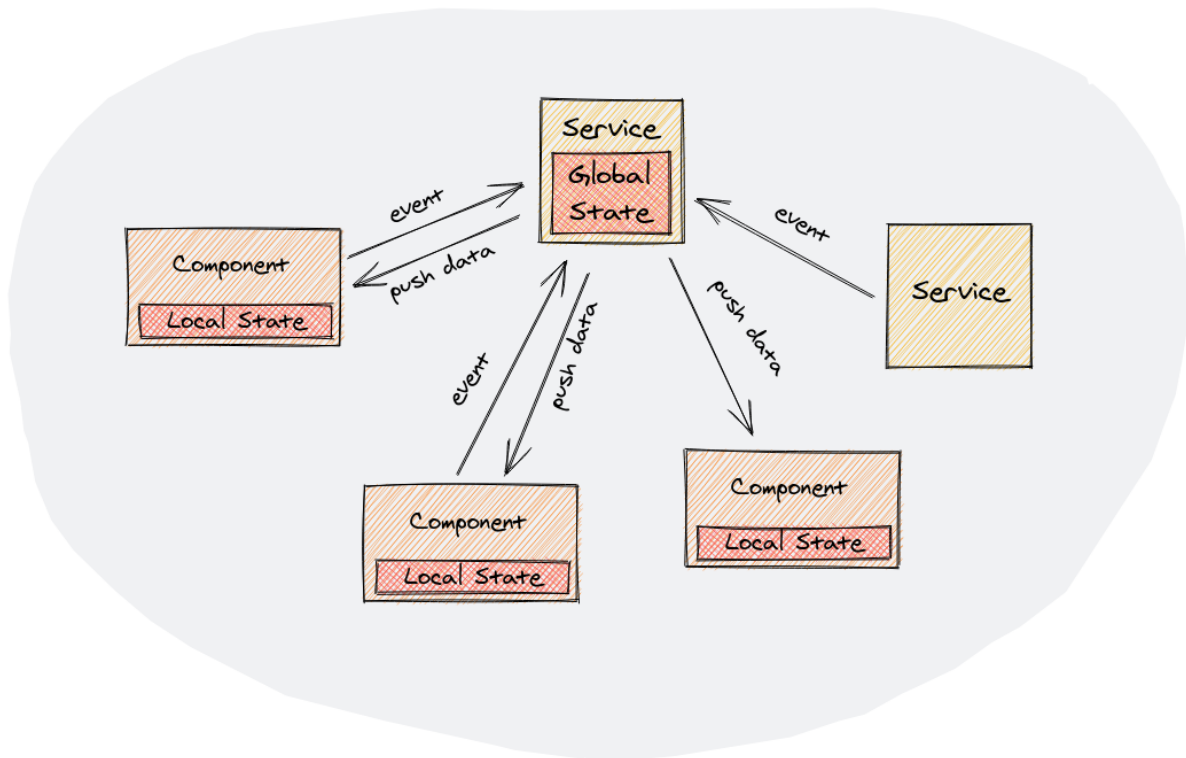


Figure 1.4: With NgRx, shared state is elevated to global state that updates based on events

Eventually, the reasoning behind NgRx is a combination of detaching state from the view hierarchy, separating commands and queries as well as benefitting from [event-based programming](#). That doesn't mean that any of the intermediate steps are wrong or that they exclude each other. It's just that these considerations seem to resurface over and over again for developers when they're working on fairly complex applications. That's what happened to [programmers working with Elm](#), then the people at Facebook formulated their [Flux pattern](#) leading to the [Redux](#) implementation. Later on, the Google engineers working on [Firebase](#) went on to express the same approach for the Angular world within NgRx. At the end of the day though, you're probably not getting paid for coming up with sophisticated state management solutions but rather for delivering working applications.

Leveraging a formalized solution like NgRx opens up a whole community where people speak the same state management language and have created convenient development tools and drop-in extensions. You'll be able to time-travel through your application, facilitate fast restarts based on cached data or easily implement features like undo-redo - all while providing maximum performance. Learning NgRx and its underlying principles won't solve all your problems, but it will put a battle-tested tool in your belt for approaching state management in modern software development.



Recommended Video

[Rethinking State in Angular Applications](#) by Alex Okrushko

1.2 Concepts and Terminology

That service containing the global state in Figure 1.4 is the NgRx store. It exposes the current state as an [RxJS observable](#) and accepts incoming events, so-called actions. Anytime such an action is dispatched, the store will compute the next state based on the incoming action and any attached information. Subsequently, the computed state is pushed to all subscribed consumers (e.g. components) through the store observable.

Each state is computed based on a recipe that you, the developer, can define in form a function. This function is called reducer and takes two arguments: the current state and the action that's currently dispatched. So, we basically *reduce* one state and one action to a single next state. We won't include any outside information because, as we've seen in Figure 1.3, that gets messy. At the very beginning of your application's lifecycle, the state is undefined. For that case it's necessary to include a default state in your recipe, something to start off with. In any other case, you'll want to look at the type of action you're dealing with and update the state accordingly. Lastly, because mutating existing state objects might have unintended consequences for consumers, we'll instead always create a new state.

Incidentally, the limitations we set on the reducer function, our recipe for computing the next state, are the definition of what's called a pure function in functional programming. Such a pure function operates exclusively on it's inputs and always returns the same result given the same arguments. The term in itself doesn't really matter, the consequences are what's important: state transitions become reliable and can be tested easily.

Any interaction with our state management that is not pure because it's async or involves external state, is represented by an effect. Effects are any impure operations that get triggered based on the store's event-bus while they in turn can also dispatch events themselves. This way we can explicitly define operations that would be off limits for a reducer like an asynchronous HTTP request or accessing some persistent storage. Meanwhile, the NgRx store and it's concepts remain unaltered. Our state management will stay comprehensible, but we don't loose anything in terms of functionality.



Recommended Read

Here's an article that demystifies NgRx by implementing a custom version of it:

[How NgRx Store & Effects Work: 20 LoC Re-Implementation](#)

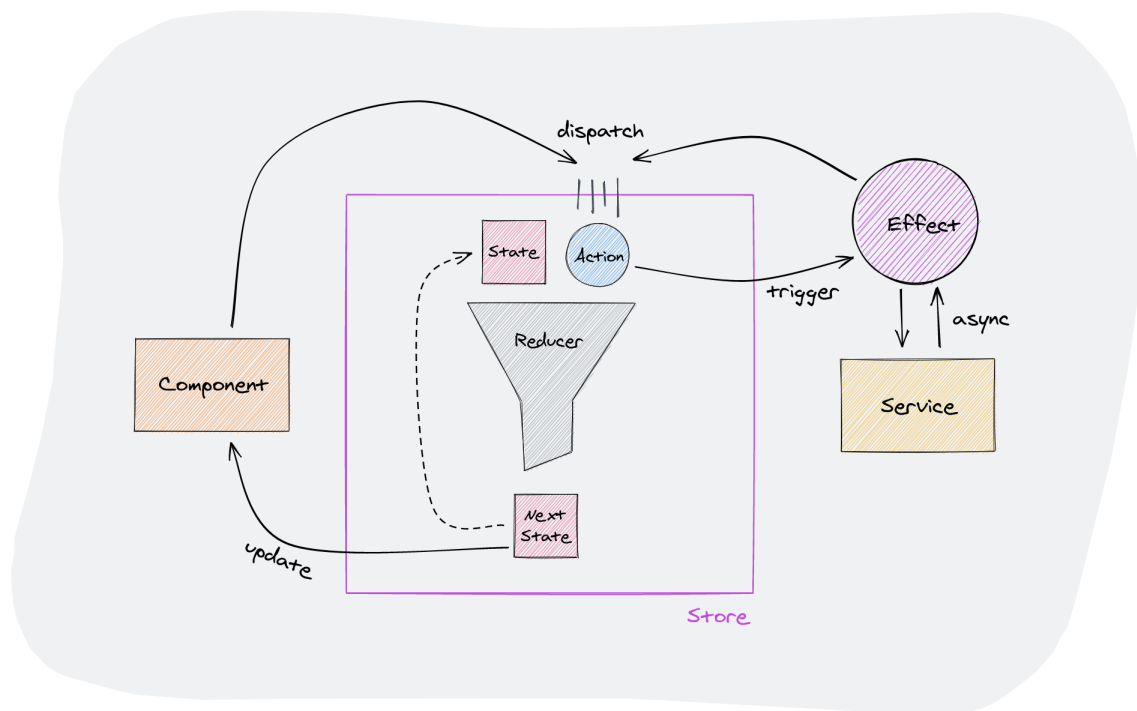


Figure 1.5: Components dispatch actions to the store, reducers compute the next state which updates the components. Effects are triggered by dispatched actions and will also dispatch actions themselves after performing async tasks.

NgRx Terminology at a Glance

State Refers to a single object that represents the global application state. Usually, you'll define an interface that specifies what the state looks like in order to reference it in reducers and consumers. You'll also need an initial state that starts off your application's state management with default values.

Action Represents a unique event occurring in your application that is relevant to the global state management and may lead reducers to producing a new state. It's an object distinguished by a type string and might contain certain metadata about the underlying event.

Reducer A pure function that takes the current state and an action and returns the next state.

Store The store manages the state by invoking all registered reducers when it receives an action. At the same time it makes the state accessible as a stream of values.

Selector A pure function that retrieves a certain part of the state from the store while optionally applying additional transformations. The re-computation of the transformation can be minimized through a process called memoization where cached results are returned as long as the inputs stay the same.

Feature A set of one or more reducers optionally plus corresponding actions, effects and selectors, together representing a unit of state-based functionality.

Effect Explicit definition of interaction between state management and other, possibly asynchronous, units (e.g. storage) or operations (e.g. HTTP request).

1.3 NgRx Is Not Just A Store

The term NgRx is often used synonymously with its store solution, but it's more than that. Nowadays, the NgRx project also provides other libraries which, while they work well in combination with the store, can be used independently. Right now, that would include [@ngrx/component](#) or [@ngrx/component-store](#).

Moreover, there's a [team](#) of clever and kind people behind the project who, like many others from the community, put their efforts into advancing the Angular landscape and helping developers - often in their free time. Consider [supporting NgRx through Open Collective](#), especially when you're using their open-source libraries to build commercial software. You may also [contribute](#) to the project in various ways as helping hands are always welcome.

1.4 You Might Not Need NgRx

Let's be honest, global state is a cop-out. No state is ever truly *global* - whatever that actually means. Granted, calling some state global is convenient and no software that's ever written is an accurate depiction of reality. The important thing here is recognizing the limitations of viewing state through the lense of solutions like NgRx.

You're building a real-time messaging app like [Google Messages](#)? NgRx will make your life easier. Creating a complex graphics editor like [SceneLab](#)? Trust me, NgRx is the way to go. Displaying some data and forms for an enterprise app? NgRx is probably overkill. That enterprise app has gotten more complex than expected and you're running into state management issues? Try integrating NgRx gradually where it makes sense.

One argument that's often mentioned against NgRx is boilerplate. Oh, just think of all the boilerplate, they'll say. All those actions, typings and effects you'll write before getting anything done. In fact, the NgRx APIs have improved tremendously with the introduction of [creator functions](#). Most of what's still called boilerplate is actually the cost of making your state management explicit. When there's a lot of state to manage, this cost pays off in the long run.

In order to guide your decision further, the creators of NgRx came up with the SHARI principle. When your state meets the attributes, NgRx might be for you:

Shared The same state is used by multiple components and/or services

Hydrated State is (de-)serialized from a persistent storage

Available State is supposed to outlive a component or route

Retrieved State is provided by a possibly expensive side-effect (e.g. HTTP request) which shouldn't run every time you need the state

Impacted State is impacted by events from various sources

There are also some alternatives to the NgRx store which you'll want to take a look at before deciding what's best for your project:

NGXS Based on the same building blocks as NgRx but focussed on reducing boilerplate and masking reactive implementation details.

Akita Built on top of RxJS with a focus on simplicity. It promotes similar concepts as NgRx, but doesn't integrate event-based programming - so there's one indirection less.

XState Finite state machines and state charts for JavaScript. If you find yourself using NgRx like a state machine, you might want to use this instead and take a look at the actor model.

Additionally, consider lower-level state management solutions like [@ngrx/component-store](#) or [@rx-angular/state](#). Maybe you're also fine without including another dependency and [RxJS](#) itself is already enough.



Recommended Video

[You Might Not Need NgRx](#) by Mike Ryan



Recommended Read

[Redux is Half of a Pattern](#) by David Khourshid

Chapter 2

Example App

In the course of this book we'll develop a basic issue tracker (see Figure 2.1). Users will be able to create issues with a corresponding description and priority. It should also be possible to mark an issue as resolved.

The issue tracker will be developed with Angular and NgRx 10 as well as Node.js 12 and npm 6. We'll start by creating a basic Angular application with the [Angular CLI](#) and its `ng new` command:

```
ng new ngrx-issue-tracker --routing=true
```

Here I'm also passing the `--routing` flag so that it'll setup a separate routing module inside `app-routing.module.ts`. This way we can implement multiple views: a list of all issues, a detail view and later on a view for configuring some imaginative settings.

The command will also ask you to select a styling language you want to use for the project. I'm using the CSS superset [SCSS](#), but you may use whatever you're most comfortable with since I won't cover styling the app in this book.

Other than that we'll be working with the Angular defaults in order to focus primarily on NgRx. This also means we'll use [Jasmine](#) and [Karma](#) for testing.

All code snippets shown in this book that refer to actual application code will contain the corresponding file name in it's first line. For readability reasons, I'll only include relevant module imports - e.g. when a file is shown twice I'll omit pre-existing import statements. I'll also omit component data and similar standard Angular code that isn't relevant to the respective topic of a section.




The complete source code for the example app is available on GitHub:

<https://github.com/nilsmehlhorn/ngrx-issue-tracker>

At the end of each section you'll find links to the corresponding changes, the resulting source code version and an online live demo.

Issue Tracker



Title

Description

Priority

☒ Low

☐ Medium

☐ High

Submit

Search ...

Setup HTTPS

High

Secure example.com with HTTPS encryption

Resolve

Configure Continuous Integration

Low

Improve the developer experience by building on every push

Resolve

Manage State with NgRx

Medium

Adopt Redux-like architecture for global state management

Resolved

Figure 2.1: Issue Tracker Mockup

Chapter 3

Installation

Up until now our app does not contain the slightest trace of NgRx - let's change that! The first way to do this is by manually adding the `@ngrx/store` dependency through the package manager. With npm you would execute the following command:

```
npm install @ngrx/store
```

When you're using yarn it would look like this:

```
yarn add @ngrx/store
```

Then we'll import the `StoreModule` into our `AppModule` using its static method `forRoot()` since this is the point where we register the root state of the application.

```
// app.module.ts
import { BrowserModule } from "@angular/platform-browser";
import { NgModule } from "@angular/core";

import { AppRoutingModule } from "./app-routing.module";
import { AppComponent } from "./app.component";
import { StoreModule } from "@ngrx/store";

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, AppRoutingModule, StoreModule.forRoot({})],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

However, you can also use the Angular schematic for NgRx to automate this process. The Angular CLI

provides a command called [add](#) which we can invoke as follows:

```
ng add @ngrx/store@latest
```

Additionally, you might want to install [@ngrx/schematics](#) for generating NgRx building blocks through [CLI schematics](#):

```
ng add @ngrx/schematics@latest
```

This command will also configure the NgRx schematic collection to be used as the default by Angular. Both of these steps can also be performed manually:

```
npm install @ngrx/schematics --save-dev
```

```
ng config cli.defaultCollection @ngrx/schematics
```



Installation

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 4

First Steps

Before we deep-dive into a specific part of NgRx, let's get a good overview of how everything fits together. As we've seen, state management with NgRx is a cycle without a single starting point. Therefore I think it's best to jump into a simple example: we'll implement a counter. Don't worry if anything isn't immediately clear to you, we'll explore each part in-depth after this little crash course.

The state of our application can be described by a plain interface with a number property for the counter:

```
// store.ts
export interface State {
  count: number;
}
```

For now we'll place all NgRx-related code in a file `store.ts` - we'll look at proper file structure and naming later.

In order to request the increment of our counter, we'll define an action creator - so a function we can call to create an event that indicates the intent to bump up the current count. Import the helper function `createAction()` from `@ngrx/store` and pass a type (basically a unique name) for the action:

```
// store.ts
import { createAction } from "@ngrx/store";

export const increment = createAction("[Counter] Increment");
```

The last thing we have to do is define how this action is handled. Since this is done by a reducer, we'll create one using the helper function `createReducer()` while passing an initial count of zero as the first argument.

For the second argument we pass the return value of another helper function `on()`. It creates a tiny reducer that will only handle the types of actions that we specify. We basically declare a state transition for a specific action. The reducer function we pass to `on()` defines how this transition will compute

the next state from the last one.

```
// store.ts
import { createAction, createReducer, on } from "@ngrx/store";

export const countReducer = createReducer(
  0,
  on(increment, (count) => {
    return count + 1;
  })
);
```

We reduce the next state by returning an updated count through a simple addition of 1. Now, every time an `increment` action occurs, we'll get a new state with an updated counter value - neat!

Let's connect our store setup with the actual application. For this we need to define which part of the application state should be handled by the `countReducer`. We do this by creating an object of the type `ActionReducerMap` with a generic parameter that represents our type of state. Such a map assigns a slice of state (left side) to a reducer (right side) as follows. Naturally, the `countReducer` is going to compute the `count` property of our `State` interface.

```
// store.ts
import { ActionReducerMap } from "@ngrx/store";

export const reducers: ActionReducerMap<State> = {
  count: countReducer,
};
```

Then we open `app.module.ts` back up and register the reducer mapping with the store by passing it to `forRoot()`:

```
// app.module.ts
import { reducers } from "./store";

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, AppRoutingModule, StoreModule.forRoot(reducers)],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

At this point our store setup is complete. The only thing left to do is build a component that can display

the current count and trigger an increment.

For this purpose you can use the `generate` command of the Angular CLI like this:

```
ng generate component components/counter
```

Make sure to either define a route for this component inside `app-routing.module.ts` or put its component selector `app-counter` inside `app.component.html` to have it rendered in your app.

Inside the component class we'll have a field for an observable number named `count$`. Note that the dollar sign in the name indicates a pluralization following the so-called [Finnish notation](#) introduced by Andre Staltz - a Finn.

The constructor declares a parameter property for injection of the NgRx store. The `Store` service is itself an observable emitting every time when a new application state is reduced. Calling `select()` allows us to only look at a specific slice of the whole state by passing a projecting function. It's basically the RxJS operators `map()` and `distinctUntilChanged()` combined, providing you with an observable of distinct state slices.

The store also provides a function for dispatching actions. We'll use it to provide a method for incrementing our counter where we invoke the action creator defined earlier.

```
// counter.component.ts
import { increment, State } from '../store';

@Component({ ... })
export class CounterComponent {
  count$: Observable<number>;

  constructor(private store: Store<State>) {
    this.count$ = this.store.select((state) => state.count);
  }

  increment(): void {
    this.store.dispatch(increment());
  }
}
```

The view template of the counter component then binds the current counter state to the view through the `AsyncPipe`. Meanwhile, we bind the click event of a button to the increment method.

```
<!-- counter.component.html -->
<p>Counter: {{ count$ | async }}</p>
<button (click)="increment()">Increment</button>
```

That's it! Running the application will give you a counter which you can happily increment. Now, you might argue that we had to write quite a lot of code just for incrementing a counter - something you can otherwise do in one or two lines in a component. I agree! However, structuring your application with NgRx pays off when its complexity grows. Having clearly defined events and deterministic ways in which the application state changes is a big plus when a lot is going on. Also, as with most things, the first step is always the hardest. Adding a second interaction now requires considerably less effort, I'll show you!

Let's create another action for multiplying the count with an arbitrary number. Again we're using the `createAction()`, but this time we call another helper function `props()` to setup a payload for the action. `props()` accepts a generic parameter that defines the shape of the action payload. The function itself actually doesn't do much besides typing.

```
// store.ts
export const multiply = createAction(
  "[Counter] Multiply",
  props<{ factor: number }>()
);
```

Inside our component the `multiply` action creator can be used just like `increment`, but now we can also pass a payload object for the resulting action:

```
// counter.component.ts
multiply(factor: string): void {
  this.store.dispatch(multiply({ factor: parseFloat(factor) }));
}
```

Let's bind this component method to another button and have an input field for the factor:

```
<!-- counter.component.html -->
<input #factorInput type="number" value="1" />
<button (click)="multiply(factorInput.value)">Multiply</button>
```

Then we only have to add another case to the existing reducer for handling `multiply` actions.

```
// store.ts
const reducer = createReducer(
  0,
  on(increment, (count) => count + 1),
  on(multiply, (count, { factor }) => count * factor)
);
```

Here you see that the reducer function passed to `on()` can actually accept the corresponding action as a second parameter. This way we can incorporate the action payload into the computation of our next

state. Note that I'm [destructuring](#) the action parameter to conveniently access the `factor` payload property.



First Steps

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 5

Debugging

The [@ngrx/store-devtools](#) library makes NgRx compatible with the [Redux DevTools](#). This extension is virtually available for all browsers allowing you to debug the application state during runtime. Add the extension to your browser and install the library with npm:

```
npm install @ngrx/store-devtools
```

The corresponding schematic will already take care of connecting the store to the extension:

```
ng add @ngrx/store-devtools@latest
```

In order to manually connect the DevTools to the NgRx store we need to import the [StoreDevtoolsModule](#) into our application module using its static method `instrument()`. Make sure to import the DevTools module after the actual store module. While doing so we can optionally pass options like `maxAge` to retain only a certain number of prior actions or `logOnly` in order to prevent someone from easily messing around with the state during production. Check the [documentation](#) for all available options.

```
// app.module.ts
import { StoreDevtoolsModule } from "@ngrx/store-devtools";

@NgModule({
  declarations: [AppComponent, CounterComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    StoreModule.forRoot(reducers),
    StoreDevtoolsModule.instrument({
      maxAge: 20,
      logOnly: environment.production,
    }),
  ],
})
```



```

],
providers: [],
bootstrap: [AppComponent],
}))
export class AppModule {}

```

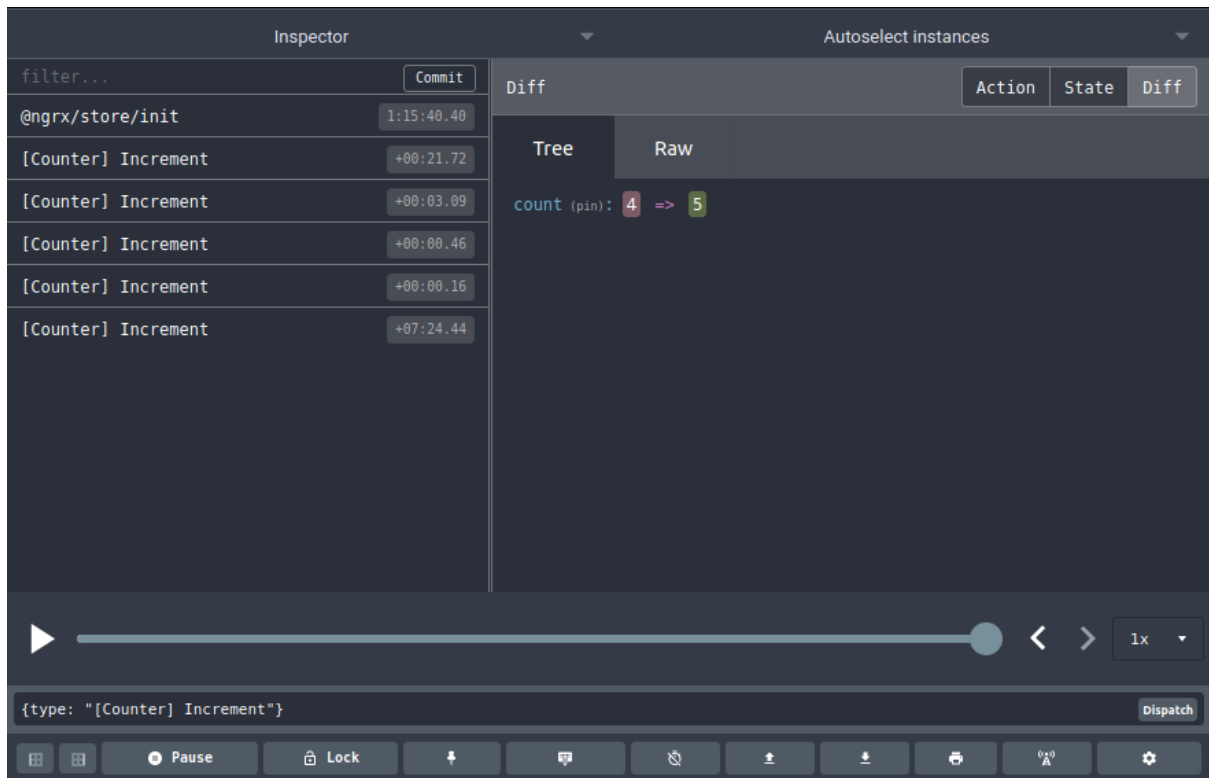


Figure 5.1: Redux DevTools instrumenting NgRx store

Now we have access to a powerful tool for developing with NgRx. It'll give you access to the following features:

Inspector View current and previous states as well as what actions produced which differences

Log monitor Disable and revert actions or commit the current state to have it used as the initial state.

Chart Display the state history as a chart.

Note that you can switch between inspector, log monitor and chart through the dropdown on the top-left.

Time-Traveling Flip back and forth through the state history while seeing your application travel through time with the slider towards the bottom.

Dispatcher Manually send actions to the store.

Export / Import Serialize the whole application state as JSON and reload it at a different time and place. This can also be really helpful when trying to reproduce bugs.

Lock / Persist Lock the state to prevent it from being altered or persist it between page reloads - the later is especially helpful when you're currently working on a feature that is based on numerous prior interactions, e.g. a checkout flow.

If you want to completely exclude the DevTools instrumentation during production, you can do so with a [file replacement](#). For this purpose we'll outsource environment-specific module imports into another file like `modules.ts`. Here we declare an array containing the DevTools module:

```
// modules.ts
import { StoreDevtoolsModule } from "@ngrx/store-devtools";

export const modules = [
  StoreDevtoolsModule.instrument({
    maxAge: 20,
  }),
];
```

Inside `app.module.ts` we'll add the `modules` array into the existing imports:

```
// app.module.ts
import { modules } from "./modules";

@NgModule({
  declarations: [AppComponent, CounterComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    StoreModule.forRoot(reducers),
    modules,
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Then we setup another file called `modules.prod.ts` which we'll use when building the app for production. There we just provide an empty set of modules:

```
// modules.prod.ts
export const modules = [];
```

Lastly, we'll edit `angular.json` and have `modules.ts` replaced with `modules.prod.ts` in a production configuration:

```
// angular.json
"architect": {
  ...
  "build": {
    "builder": "@angular-devkit/build-angular:browser",
    "options": {...},
    "configurations": {
      "production": {
        "fileReplacements": [
          ...
          {
            "replace": "src/app/modules.ts",
            "with": "src/app/modules.prod.ts"
          }
        ],
        ...
      }
    }
  }
  ...
}
```

Now, when running `ng build --prod`, Angular will pick the configuration called “production” and thus won’t instrument the Redux DevTools.



Debugging

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 6

File Structure and Naming

There's no pre-defined file structure for NgRx projects that you'd strictly have to follow. As long as you wire up everything correctly and don't have any collisions, it'll work. However, having some clear patterns for organizing state models, reducers and actions can prevent your codebase from becoming messy.

Here's a baseline structure you can follow. I'm not claiming it's the best way to manage your files, but it's probably a good starting point. I encourage you to adapt your setup to what works well for you and your team - the most important part here is keeping consistency.

Similar to how components and services are suffixed with `component` and `service` I'd recommend creating separate files for every part of your store and apply the same pattern. So, reducers, actions and state models each go in their own files ending in `.reducer.ts`, `.actions.ts` and `.state.ts` respectively. Later we'll also introduce `.selectors.ts` and `.effects.ts` files for selectors and effects.

The counter example from before didn't have a designated state model since the state was only represented by one number. For more complex use-cases you'll want to define a type for the state slice that is handled by your reducer - more on that in the next chapter.

Anything related to the store goes into a `store/` directory at the root level of your app. Feature-specific files are placed into corresponding sub-directories. Additionally, you may create a separate file `index.ts` where all state slices come together to define the root state of the application and how it's managed by individual reducers. Keep such configuration code out of your module declarations as much as possible - in most projects I've seen there's already going on enough in there. Some people also introduce separate Angular modules for encapsulating store imports. You might do so as well, personally though, in most cases I don't see the necessity as long as we have something like `index.ts`.

Starting out, the file and directory structure for managing the state of our issue tracker will look as follows:

```
src
|-- app
    |-- store
        |-- index.ts
        |-- issue
            |-- issue.actions.ts
            |-- issue.reducer.ts
            |-- issue.state.ts
```

I'd advice against grouping by type of code, e.g. putting all reducers in one directory, actions in another and so on. Instead always group by feature to co-locate code that works together, since it is likely that this code also changes together.

Inside `issue.actions.ts` we export individual actions like we've done before. You can then import them directly in your components ...

```
import { create } from "../../store/issue/actions";
```

... or use a named import to prevent collisions:

```
import * as IssueActions from "../../store/issue/actions";
```

The model for a state slice may be defined in an interface inside the corresponding `.state.ts` file. You'll also want to export an initial state that implements the interface from there. Here's `issue.state.ts`, we'll fill it in during the next chapter:

```
// issue.state.ts
export interface IssueState {}

export const initialState: IssueState = {};
```

Reducer files like `issue.reducer.ts` should always export a single reducer function, preferably prefixed with the state property it's meant for:

```
// issue.reducer.ts
import { createReducer } from "@ngrx/store";
import { initialState } from "../issue.state";

export const issueReducer = createReducer(initialState);
```



If you're not yet using Angular's Ivy engine, you'll have to wrap the reducer into a regular function before export.

Then everything is wired together inside `index.ts` by defining a type for the root state and exporting

a reducer mapping:

```
// index.ts
import { ActionReducerMap } from "@ngrx/store";
import { issueReducer } from "../issue.reducer";
import { IssueState } from "../issue.state";

export interface RootState {
  issue: IssueState;
}

export const reducers: ActionReducerMap<RootState> = {
  issue: issueReducer,
};
```

Lastly, after registering the reducer mapping through `StoreModule.forRoot()` in the app module, we can inject the store. I'm doing this in a parent component for the issue tracker while using `RootState` as its generic type parameter:

```
// issues.component.ts
import { Component, OnInit } from "@angular/core";
import { RootState } from "../../store/root";
import { Store } from "@ngrx/store";

@Component({
  selector: "app-issues",
  templateUrl: "../issues.component.html",
  styleUrls: ["../issues.component.scss"],
})
export class IssuesComponent {
  constructor(private store: Store<RootState>) {}
}
```

Sometimes you'll see people importing store files into a single variable prefixed with "from" like this:

```
import * as fromRoot from "../../store/root";
```

It's not necessary to do so and talking to various NgRx members I couldn't find clear reasoning for this. NgRx co-creator Mike Ryan pointed me to a [video](#) of Dan Abramov, co-creator of Redux, applying this approach. However, he only used it for importing selectors so let's put this off until we get to those. We'll also look at some more specific advice on structure and naming later when working with effects and feature modules.

One more thing though: when you find yourself in a position where an action should be handled by multiple reducers, you might exempt the action file from being named like a corresponding reducer. Instead you can name it after the page on which the actions occur and place it accordingly. The same applies for selectors that stretch over various parts of the state.



File Structure

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 7

State

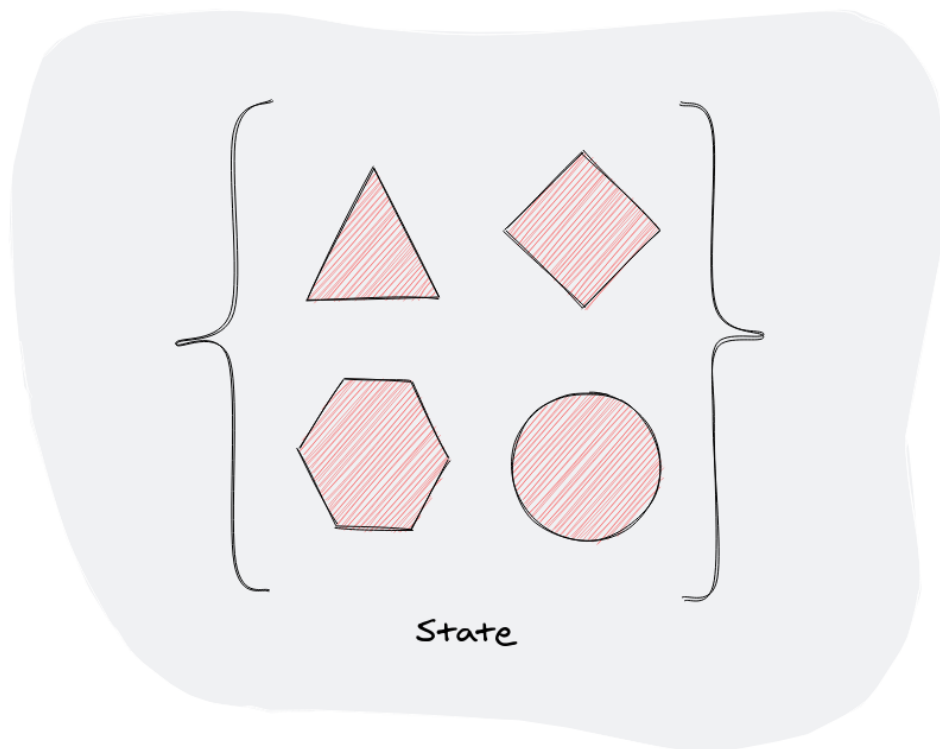


Figure 7.1: The application state is a plain object constrained by static typing. It can contain arbitrary information, but you should avoid redundancies and keep it serializable.

Thanks to TypeScript we are able to declare static types defining the application state. These will be used in reducers, components, services, effects and basically anywhere you're dealing with the state.

In TypeScript you can define data models either using interfaces or type declarations. So, when we want to describe the shape of a single issue we could do this with an interface ...

```
// issue.ts
export interface Issue {
  id: string;
```



```

    title: string;
    description: string;
    priority: "low" | "medium" | "high";
    resolved: boolean;
}

```

... but also using the `type` keyword:

```

// issue.ts
export type Issue = {
  id: string;
  title: string;
  description: string;
  priority: "low" | "medium" | "high";
  resolved: boolean;
};

```

Such [type alias declarations](#) are generally more powerful than interfaces since they can give a name to any type - including things like functions or primitives. I tend to use interfaces in most cases and sprinkle in some type aliases for tuples or unions - e.g. for extracting the `priority` type:

```

// priority.ts
export type Priority = "low" | "medium" | "high";

```

As mentioned, the base shape of your feature state should be outlined by an interface inside `<feature>.state.ts`. However, you can also incorporate models from other locations as long as they meet some conditions (see [What Not to Put in the State](#)). The issue model for example might also be placed in a file `src/app/models/issue.ts`.

Here's the first draft for our application state, containing a list of issue entities and the currently active search filter:

```

// issue.state.ts
export interface Filter {
  text: string;
}

export interface IssueState {
  entities: Issue[];
  filter: Filter;
}

```

An initial state for this type could look as follows where we provide an empty list and search text:

```
// issue.state.ts
export const initialState: IssueState = {
  entities: [],
  filter: {
    text: "",
  },
};
```

You should populate your initial state with defensive but valid values: strings and collections can be empty, booleans set to false and numbers start at zero. If you still decide to set some property to `undefined`, make sure that all consumers can deal with that. Of course you can provide other, more specific initial values if that makes more sense in your situation. It's also possible to retrieve the initial state from the `localStorage` or similar (see [Re-Hydration](#)), you'll still need this fallback though.

This state definition will work, however, now anytime we want to access a specific issue in a reducer or component you need to iterate through the `entities` list. If your model doesn't have an ID you might be using an array index here instead.

```
const specificIssue = entities.find((issue) => issue.id === id);
```

The impact on performance is probably negligible at first, but we can easily improve the developer experience with some normalization.

7.1 Normalization

Models with a unique identifier are best stored in an object where the identifier is used as a key while the model instance represents the corresponding value. We can specify such a setup as an interface with an index type:

```
// issue.state.ts
export interface Issues {
  [id: string]: Issue;
}
```

This declaration says that any object of type `Issues` is a dictionary having strings as keys and `Issue` objects as values. The type can then be used as a replacement for the list:

```
// issue.state.ts
export interface IssueState {
  entities: Issues;
  filter: Filter;
}
```

A corresponding initial state may initialize the issues with an empty dictionary object:

```
// issue.state.ts
export const initialState: IssueState = {
  entities: {},
  filter: {
    text: "",
  },
};
```

When accessing a specific issue we can now just use the bracket notation with an id:

```
const specificIssue = entities[id];
```

We can de-normalize a dictionary into a list that can be iterated with `*ngFor` by leveraging `Object.values()`. Also, `Object.keys()` will give you a list all existing keys while `Object.entries()` returns key-value tuples - both helpful from time to time.

```
const issueList: Issue[] = Object.values(entities);
```

Such de-normalizations are usually performed by selectors which we'll get to later on.

Other than that, state normalization also refers to having the same data item only in a single place. For example, imagine we'd implement a multi-select for our list of issues. In order to know which ones are selected we might be tempted to manage another dictionary in the store:

```
// issue.state.ts
export interface IssueState {
  entities: Issues;
  selected: Issues;
  filter: Filter;
}
```

However, when one issue is now updated we need to make any changes both inside `entities` as well as `selected`. Since the state is immutable and we can only create new objects with reducers, changes to an issue in one dictionary won't be reflected anywhere else - and that's a good thing. In order to work around this, we can simplify our state. It's actually pretty redundant to manage the same objects multiple times. All we actually need is to remember which issues are selected. So a reference to an existing issue is enough. Therefore we could replace the dictionary with a list of IDs that reference the actual issue inside the `entities` property:

```
// issue.state.ts
export interface IssueState {
  entities: Issues;
  selected: string[];
```

```
filter: Filter;  
}
```

We could de-normalize the list of references into actual issues with array mapping:

```
const selectedIssues = selected.map((id) => entities[id]);
```



If you need to normalize a lot of nested data - possibly as a result of API calls - take a look at Paul Armstrong's [normalizr](#) library.

7.2 What Not to Put in the State

There are certain things that don't belong in the NgRx store - some for conceptual reasons, others for technical reasons.

Derived State

Anything you can compute from the existing state doesn't need to be in the state - even if the computation is expensive. For example, I can compute the number of issues with the following code:

```
const numberOfIssues = Object.values(entities).length;
```

Therefore it would be redundant to also store this information. Instead, as we'll explore later, you can create a selector which will re-compute it when necessary. This also holds up for more complex computations though it's important to structure selectors properly in order to preserve performance (see [Selectors](#)).

If we placed derived state in our store, we would have to manage the same information in multiple places. This increases our work and the possibility of making mistakes.

Local State

This one is a bit controversial. Sometimes it's hard to know on which level state should reside (or should reside in the future). Some examples are pretty obvious. Like, you wouldn't track whether a drop-down is open or closed with NgRx. Meanwhile, you'll probably need information about the currently logged-in user in a lot of places - so that's something for the store.

Other times it's not that clear cut. Take Angular's reactive forms. They track their own state and often it's enough to only put submitted values into the store (as we're about to do for the issue tracker). However, in certain use-cases you might benefit from syncing forms with NgRx to retain their values between navigations and allow for easier debugging. There's even a whole library dedicated to this called [ngrx-forms](#) by Jonathan Ziller.

Eventually, you'll have to decide on a case-by-case basis if some state is global or local. As a rule of thumb:

if some state is only relevant to one component, manage it locally. You can always start with local state and lift it up to the store later when necessary.



When you're looking for a somewhat similar solution for managing local state, you might want to give [@ngrx/component-store](#) or [@rx-angular/state](#) a try.

Classes and Special Objects

Developers with a background in object-oriented programming intuitively reach for classes in TypeScript - I'm guilty of this myself. Admittedly, Angular makes heavy use of them for the application building blocks like components or services. However, they don't really fit the pattern when working with NgRx, because that means we're embracing immutability. Classes, however, are commonly used in a mutable fashion where we change their inner state by calling instance methods.

Reducers are supposed to compute a new state without modifying the last one. If we would work with mutable classes and re-use them for subsequent states, we would break this rule. This can then create all sorts of problems. For one thing, it's harder to update the view when state changes since NgRx generally relies upon simple reference checks for that. With plain objects where we always create a new state instead of modifying the last one, those checks work like a charm:

```
const lastState = {
  entities: {}
}

const nextState = {
  ...lastState,
  entities: {
    ...lastState.entities,
    ['issue-1']: {
      id: 'issue-1',
      title: 'Embrace Immutability',
      description: '...',
      priority: 'high',
      resolved: true
    }
  }
}

console.log(lastState === nextState) // prints 'false', NgRx will emit
```

When we're re-using class instance though, those checks stop working:

```
const lastState = new IssueState([]);

// modifies existing dictionary under the hood
lastState.addIssue({
  id: "issue-1",
  title: "Embrace Immutability",
  description: "...",
  priority: "high",
  resolved: false,
});

const nextState = lastState;

console.log(lastState === nextState); // prints 'true', NgRx won't emit
```

Technically, you could still create a new class instance every time you draft up the next state with `Object.assign()` like this:

```
Object.assign(new IssueState(), lastState);
```

Though, in that case you can't leverage most benefits of classes - so why use them in the first place?

Also, there's another pain point: serialization. Transforming a class instance to JSON with `JSON.stringify()` might work, but parsing the resulting JSON with `JSON.parse()` will just give you a plain object - the JavaScript prototype chain won't be recovered. This makes many use-cases more difficult, including debugging with the Redux DevTools or re-hydrating the store from local storage or the server.

The same arguments apply for built-in classes like `Map` or `Set`. These are optimized for mutability but also easily replaceable. Instead of `Map` you can leverage plain objects with index types like the `Issues` dictionary we defined earlier. A `Set` on the other hand is not identical to a plain array list since sets won't accept duplicate entries. If duplicates aren't preventable in other ways, you can still work with `Set` but convert back to array before placing it in the state:

```
const list = [1, 1, 2, 3, 3, 4];
const uniqueList = Array.from(new Set(list)); // [1, 2, 3, 4]
```

Despite classes you'll also want to avoid having any special objects like the following in your state as this can hinder garbage collection and lead to errors that are hard to debug:

- functions
- blobs
- dates (try `Date.toJson()` or similar instead)

- promises
- observables
- HTML elements
- `window` and similar

This list is non-exhaustive. So, anything that's similar probably doesn't belong in the state. Eventually, we're best off storing only plain, serializable objects in the state while still using interfaces or type aliases for type-safety.



State

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 8

Actions

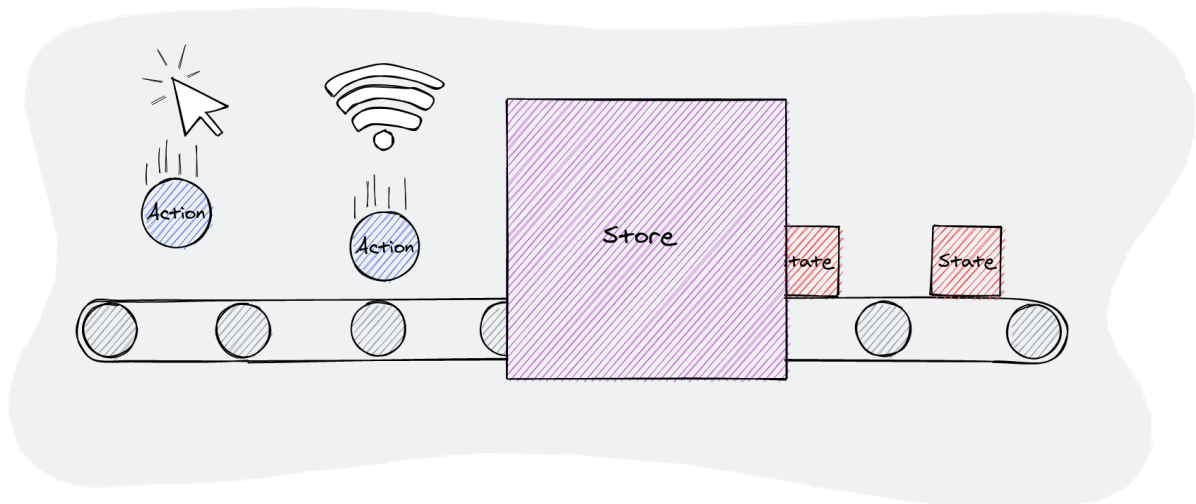


Figure 8.1: Events like a click or network response occurring in your application are sent to the store in form of actions. One-by-one they'll lead the store to produce a new state.

Actions represent unique events occurring in your application. These can originate from things like user-interactions, HTTP calls or WebSockets. Anytime something happens that might result in a change to the application state we send an action to the NgRx store.

While actions are an integral part of NgRx, they're actually pretty simple in their nature. This interface with a single property `type` is enough to describe them:

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

You don't even need to implement this interface specifically. Any plain object with the `type` property can count as an action. We're also allowed to add custom properties to actions in order to give context to an event. Here's how you could dispatch an action that indicates the submission of a new issue. Note that the NgRx store is usually injected into components or services and therefore available via `this`.


```
const submitAction = {
  type: "[Issue] Submit",
  issue: {
    id: randomId(),
    title: "Understanding Actions",
    description: "Actions represent events ..."
    priority: "high",
    resolved: false
  },
};

this.store.dispatch(submitAction);
```

NgRx doesn't care about names at all as long as they're unique per type of action. The type names are mainly for you and your co-developers to read. It's worth thinking twice about them in order to ease the maintenance of your app. Name your types so that when looking at the action log you can easily understand what happened in the application. Therefore, avoid types that sound like setters or ones that are too generic such as "Update State".



The `randomId()` function generates an arbitrary string to be used as an ID for our issue. Here's the implementation:

```
export const randomId = () => Math.random().toString(36).substr(2, 9);
```

In a real app we'd usually have this ID generated by a server. We'll learn how to do this later when we're working with asynchronous *effects*. For now, we can use this helper function to generate a random ID synchronously.

8.1 Action Creators

It's totally fine for the store to dispatch actions that you create inline. Though, to make our life easier we can leverage action creators. These are simply functions that create new actions upon invocation. Technically, we could write such an action creator ourselves:

```
function submitActionCreator(issue: Issue) {
  return {
    type: "[Issue] Submit",
    issue,
  };
}
```

```
const submitAction = createActionCreator({
  id: randomId(),
  title: "Understanding Actions",
  description: "Actions represent events ...",
  priority: "high",
  resolved: false,
});
```

However, there's also a handy utility function called `createAction()` that you can import from `@ngrx/store`. Together with the `props()` function it helps to reduce boilerplate when defining actions or rather action creators:

```
import { createAction, props } from "@ngrx/store";

export const submit = createAction(
  "[Issue] Submit",
  props<{issue: Issue}>
);
```

The generic type passed to `props()` defines any custom properties - also called payload or metadata. Both `createAction()` and `props()` actually create something very similar to our own action creator function - they just make clever use of TypeScript to enable the succinct code above.

Note that the variable `submit` now contains an action creator function. These are commonly named without suffixes like “actionCreator” since in most cases you won't really store the actual actions inside variables, but rather pass them directly to the store:

```
this.store.dispatch(
  submit({
    issue: {
      id: randomId(),
      title: "Understanding Actions",
      description: "Actions represent events ...",
      priority: "high",
    },
  })
);
```

It's crucial that you don't re-use (created) actions even when they don't contain a payload. This way every action is a unique object and NgRx as well as any tooling can work with those simple equality checks we already know from the [State](#) chapter. So, every time an event occurs you'll want to create and dispatch a new action to represent this specific event.

When you're deciding what to put in the metadata of your actions, the same rules that we use for the state apply (see [What Not to Put in the State](#)). Therefore only plain, serializable objects should be added to the payload. This ensures that actions can be persisted and don't cause unwanted side-effects.

Let's now create a component with a form for collecting the necessary metadata:

```
// new-issue.component.ts
import { Store } from "@ngrx/store"
import { RootState } from "../store";
import { FormGroup, FormBuilder } from "@angular/forms"
import * as IssueActions from "../store/issue/issue.actions.ts"

@Component({...})
export class NewIssueComponent {
  form: FormGroup;

  constructor(private store: Store<RootState>, private fb: FormBuilder) {
    this.form = this.fb.group({
      title: ["", Validators.required],
      description: ["", Validators.required],
      priority: ["low", Validators.required],
    });
  }

  submit(): void {
    const id = randomId();
    const issue = {...this.form.value, id};
    this.store.dispatch(IssueActions.submit({ issue }));
  }
}
```

Using the Angular's `FormBuilder` we can create a reactive form with two controls for an issue's text and priority. Don't forget to import the `ReactiveFormsModule` in your application module. Take a look at the [official guide for reactive forms](#) if any of this is new to you.

We also provide a `submit()` method that dispatches a corresponding action upon form submission while passing the form value as its payload.

Lastly, we'll setup a template and connect it to the reactive form as well as to the submission method:

```
<!-- new-issue.component.html -->
<form [formGroup]="form" (ngSubmit)="submit()">
  <label for="text">Title</label>
```

```

<input formControlName="title" type="text" id="title" />

<label for="description">Description</label>
<input formControlName="description" type="text" id="description" />

<label for="priority">Priority</label>
<select formControlName="priority" id="priority">
  <option value="low">Low</option>
  <option value="medium">Medium</option>
  <option value="high">High</option>
</select>

<button [disabled]="!form.valid" type="submit">Submit</button>
</form>

```

Now you can see our `submit` action being dispatched in the Redux DevTools upon form submission.

Action Creators with Additional Logic

Action creators can also embed some logic for pre-processing the payload. For example, we could outsource the ID generation instead of having this logic reside in the component:

```

// issue.actions.ts
export const submit = createAction("[Issue] Submit", (issue: Issue) => {
  return {
    issue: {
      ...issue,
      id: randomId(),
    },
  };
});

```

We'd get the same result, but the dispatch would then look like this:

```

// new-issue.component.ts
submit(): void {
  const issue = this.form.value;
  this.store.dispatch(IssueActions.submit(issue));
}

```

8.2 Action Hygiene

Actions are supposed to represent events that occur. Though it might be tempting, don't view them as commands. This decouples actions from their consequences and allows you to be more flexible when you need to change how they're handled.

The `submit` action above has the unique type “[Issue] Submit”. Action types are commonly prefixed with a category that you put in brackets. Generally, this category should reference the event source - e.g. a certain page or an API. However, this gets tricky when the same action can occur on multiple pages. One solution might be to duplicate the action creator. Though, remember to configure all corresponding reducers to listen for both action types (one `on()` can accept multiple action creators). Otherwise you might choose a more abstract category like “Issues” allowing you to use the action creator in multiple reducers and effects. This way you might avoid unnecessary duplication and keep the number of actions reasonable. However, you'd also be engaging in action re-use which loses you traceability. Eventually, actions are cheap. **If in doubt, define a new action.**



Recommended Video

[Good Action Hygiene](#) by Mike Ryan



Actions

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 9

Reducers

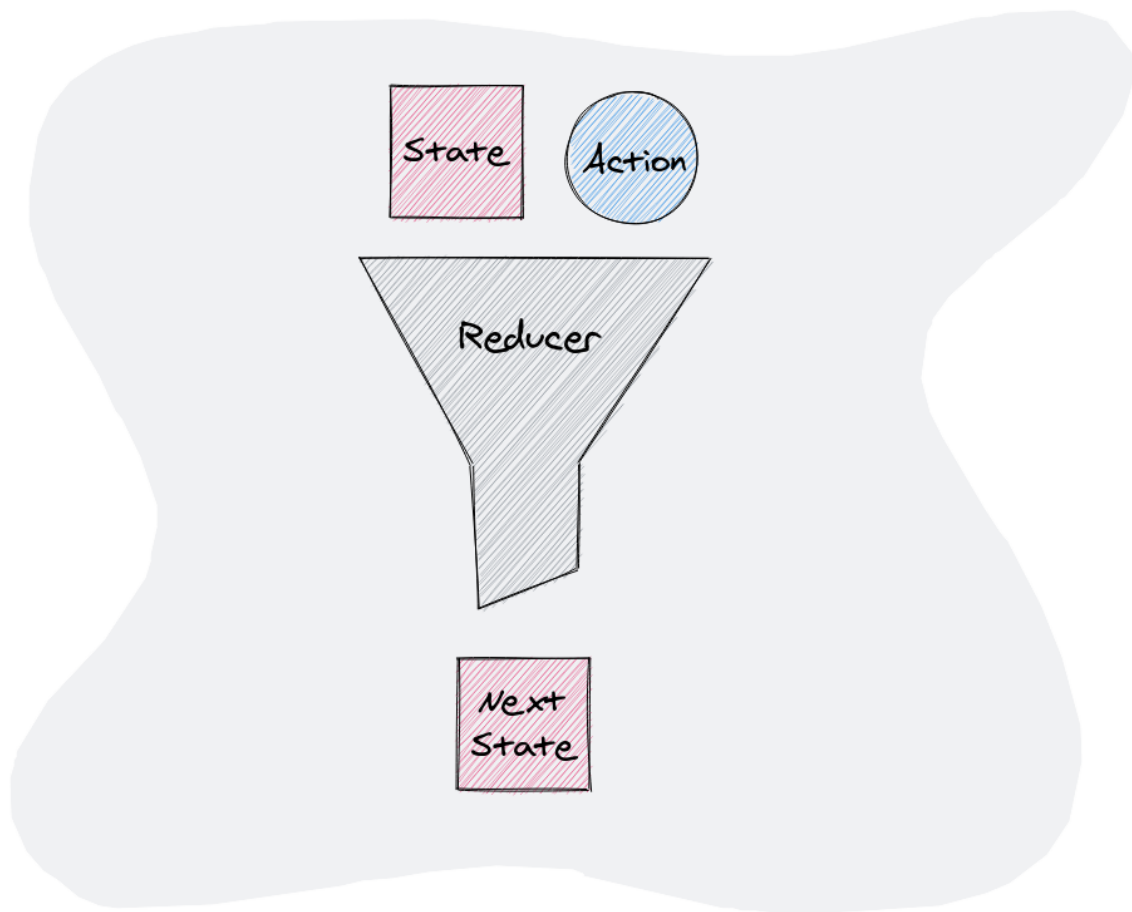


Figure 9.1: Reducers are like a funnel: the current state and an action are dropped in at the top, the next state comes out on the bottom - nothing else gets in or out

Actions can be dispatched even if there are no registered reducers. This way you could use them solely to trigger effects for example. Most of the time though you'll probably have at least one reducer react to an action type. Note that it's also possible to handle the same action type in multiple reducers since every reducer will receive every dispatched action.

A reducer produces the next state by returning a new object that implements the state interface. It's crucial to create a new object instead of modifying the existing state in order to keep the state immutable. As we've learned, this makes a lot of things easier, for example change detection. If every consecutive state is represented by a distinct object, we can just update our view when we receive a new instance. We don't have to perform any complicated checks where we traverse the state to see if there are any differences.

Nowadays, creating a new object based on an existing one is pretty straightforward using the [spread operator](#). Those three dots will copy all properties of our last state into the next state. When you're defining object literals in TypeScript (or JavaScript), the last declaration wins. So, although you're also spreading a property that already exists in the state, it's overridden with a new value by re-declaring it after the spreading. Therefore it's important to always spread first and update properties after that.

```
const issue = {
  title: "Understand Reducers",
  description: "Action and state go in, next state comes out",
  priority: "high",
  resolved: false,
};

const updatedIssue = {
  ...issue, // adds all properties with previous values
  resolved: true, // overrides property `resolved` in this new object
};
```

The spread operator only makes a shallow copy of an object, so parts of the current state will be integrated into the next one. Remember that it's forbidden to *mutate* the current state, yet, re-using everything that doesn't require change is actually encouraged since this increases performance. Additionally, you can mutate any newly created objects as you like *before* returning them as part of the next state from the reducer function.

9.1 Creating Reducers

We create a reducer by passing an initial state and several smaller state change functions to the utility method `createReducer()`. The individual state change functions defined with `on()` enable us to associate one or more types of actions with them. This way we can generate action-specific state transitions that benefit from strict type-checking. Note that you'll actually pass an action creator instead of its type, that's enough for NgRx to figure out the actual type.

```
// issue.reducer.ts
import { createReducer, on } from "@ngrx/store";
```

```

import { initialState } from "../issue.state";
import * as IssueActions from "../issue.actions";

export const issueReducer = createReducer(
  initialState,
  on(IssueActions.submit, (state, { issue }) => {
    return {
      ...state,
      entities: {
        ...state.entities,
        [issue.id]: {
          ...issue,
          resolved: false,
        },
      },
    };
  })
);

```

Once an action is dispatched, internally, NgRx will iterate all of your state change functions and execute the ones that match based on type. If there are multiple ones, they'll be executed in order. However, while you can use the same state change function for multiple action types you shouldn't handle the same action type with multiple state change functions - otherwise you'll have to look in multiple places in order to understand what an action is doing to a single slice of state.

Note that you can omit the braces and `return` keyword from the state change function by wrapping the returned state object literal into parenthesis:

```

// issue.reducer.ts
on(IssueActions.submit, (state, { issue }) => ({
  ...state,
  entities: {
    ...state.entities,
    [issue.id]: {
      ...issue,
      resolved: false,
    },
  },
})));

```




9.2 Registering Reducers

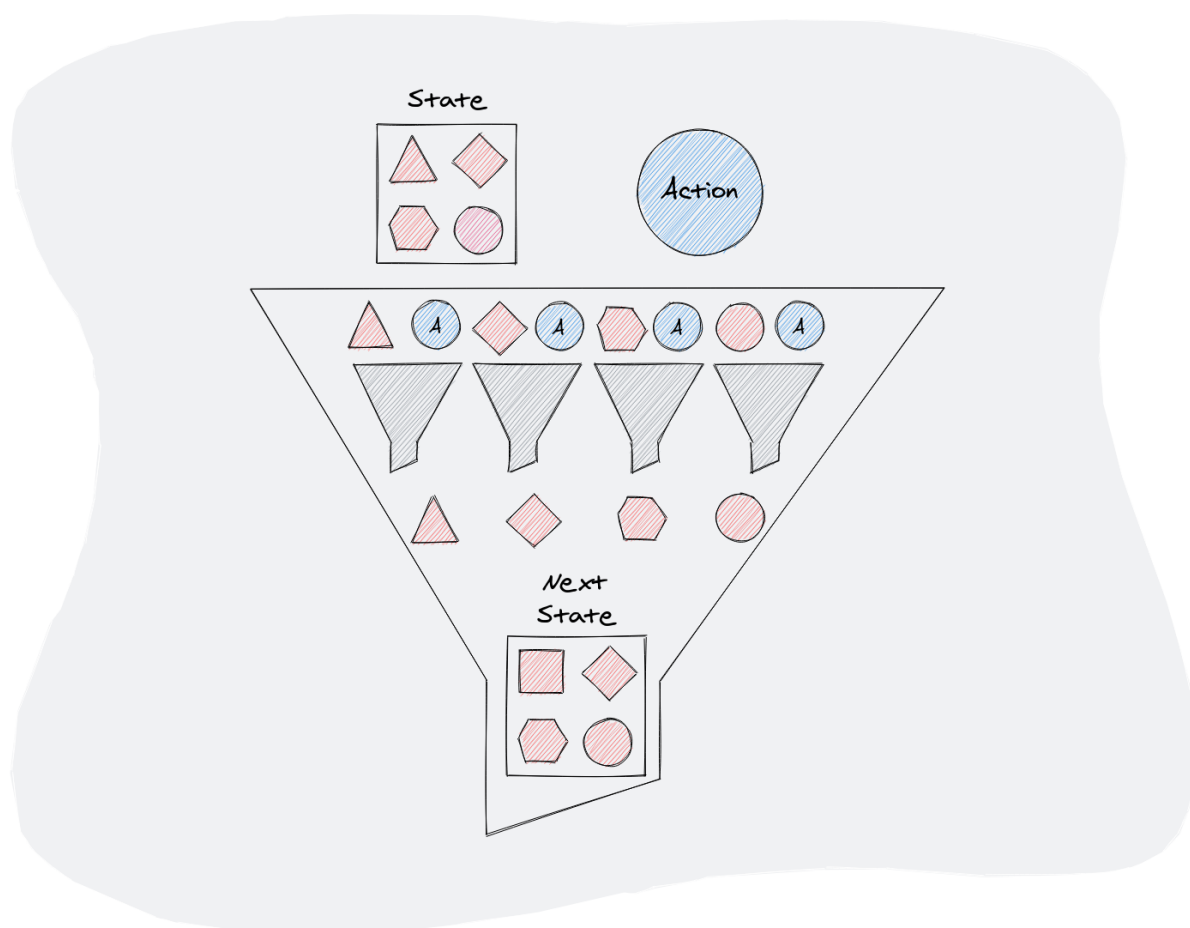


Figure 9.2: The next global state is produced by funneling state slices (red shapes) through their corresponding reducers (grey funnels). All reducers will be called irrespective of whether they actually handle the incoming action - if they don't, they return the state as it is.

The store manages the state in a single object while individual slices are processed by designated reducers. First we'll declare a type for the global state by assembling the types of all slices. So, we're basically describing the group of red shapes from Figure 9.2.

```
// index.ts
import { IssueState } from "../issue/issue.state";

export interface RootState {
  issue: IssueState;
}
```

Then we define a mapping from each state slice to a corresponding reducer. This mapping is represented

by an `ActionReducerMap` which we'll generically parametrize with the type of our application state. In relation to Figure 9.2, this map assigns each of the red state shapes to one of the grey reducer funnels. For our case there's one shape (or feature) defined by `IssueState` which will be managed by the reducer called `issueReducer` :

```
// index.ts
import { ActionReducerMap } from "@ngrx/store";
import { issueReducer } from "../issue/issue.reducer";

export const reducers: ActionReducerMap<RootState> = {
  issue: issueReducer,
};
```

Eventually, we'll pass the mapping to NgRx by importing the `StoreModule` through its `forRoot()` method:

```
// app.module.ts
import { StoreModule } from "@ngrx/store";
import { reducers } from "../store/root";

@NgModule({
  imports: [StoreModule.forRoot(reducers)],
})
export class AppModule {}
```

That's actually everything NgRx needs to kick-off state management for our application. Importing the `StoreModule` like this will also make the `Store` available for dependency injection. Additionally, `forRoot()` accepts a configuration as an optional second parameter. Among other things, this allows you to configure certain [runtime checks](#) notifying you about unintended violations of immutability, serializability or action type uniqueness.

Registering reducers in this way will initialize them directly at application startup. In Chapter 12 we'll be looking into registering additional reducers that can be lazy-loaded through feature modules.



Registering Reducers

[Changes](#) | [Source Code](#) | [Live Demo](#)

9.3 Mutable APIs with immer.js

While the spread operator definitely eases the creation of new states, you still might prefer a mutable approach for its conciseness. However, since we can't modify the last state, we would need some kind of workaround - the [immer.js](#) library by Michel Weststrate provides exactly this workaround. Instead of

modifying the state directly, it allows you to work on a draft of the next state. This draft is effectively a duplicate of the current state while any changes made to it will be copied onto an immutable object representing the next state. All of this happens when invoking the library's `produce()` function with an object and a callback that operates on the corresponding draft:

```
const issue = {
  title: "Using mutable APIs",
  description: "Write convenient reducers with immer",
  resolved: false,
};

const updatedIssue = produce(issue, (issueDraft) => {
  issueDraft.resolved = true;
});
```

As you can see, `produce()` will eventually return a new object that incorporates the changes made to the draft. This is especially helpful when working with arrays or dictionaries as it decreases the amount of code required for a state transition. Writing and reading code in a mutable fashion also eases the transition into working with NgRx. Here's how our reducer could look with immer:

```
// issue.reducer.ts
import produce from "immer";

export const issueReducer = createReducer(
  initialState,
  on(IssueActions.submit, (state, { issue }) =>
    produce(state, (draft) => {
      draft.entities[issue.id] = {
        ...issue,
        resolved: false,
      };
    })
  )
);
```

Note that we don't have to return anything from the drafting function while we return the result of `produce()` from the state change function.



If you don't like to invoke `produce()` in every state change function, take a look at [ngrx-etc](#) by NgRx team member Tim Deschryver. This library provides replacements for `createReducer()` and `on()` which make the state directly available as a mutable immer draft.



In addition to providing a convenient way of writing reducers, immer also [freezes](#) resulting objects during development in order to prevent unintended mutations at a later point. On top of that, immer is able to [generate patches](#) documenting any changes to the state. This is what the [ngrx-wieder](#) library uses to facilitate undo-redo.



Mutable APIs with immer.js

[Changes](#) | [Source Code](#) | [Live Demo](#)

9.4 Meta-Reducers

Meta-reducers are also called higher-order reducers and since reducers are just functions, meta-reducers are actually higher-order functions. “Higher-order” means operating on other functions in form of arguments or return values. Meta-reducers do both: they accept an existing reducer, wrap some logic around it and return a new reducer.

Sometimes people also refer to reducers factories as meta-reducers. Those are functions you invoke to create a reducers with certain parameters. While they are definitely useful as well, the definition we're using for meta-reducers implies that an existing reducers is wrapped with additional logic.

If this definition reminds you of Figure 9.2, you're on the right track! Put simply, there's only a single reducer in the NgRx store while all reducers that we register are rolled into one by a meta-reducer.

Usually, you don't need to write meta-reducers yourself very often. Still, their ability to extend domain-specific reducers with generic functionality can be pretty powerful. Right again, I'll mention [ngrx-wieder](#) as an example since its undo-redo functionality is actually implemented through a meta-reducer.

A small and commonly cited example would be a meta-reducers that logs the incoming action, the current and the next state. It might look like this:

```
// meta-reducers.ts
import { ActionReducer } from "@ngrx/store";

export const loggingMetaReducer = (
  reducer: ActionReducer<any>
): ActionReducer<any> => {
```

```

return (state, action) => {
  console.log("current state", state);
  console.log("action", action);
  // execute the actual reducer
  const nextState = reducer(state, action);
  console.log("next state", nextState);
  return nextState;
};
};

```

You can then either wrap one specific reducer that you created with `createReducer()` ...

```

// issue.reducer.ts

// pass initial state and action handling
const reducer = createReducer();

// wrap into meta-reducer
export const issueReducer = loggingMetaReducer(reducer);

```

... or configure the meta-reducer during reducer registration to apply it on top of the root state:

```

// app.module.ts
StoreModule.forRoot(reducers, { metaReducers: [loggingMetaReducer] });

```

Keep in mind though, that static calls like `console.log()` are technically side-effects and therefore make your reducers impure. So, you're generally better off [debugging your store with the Redux DevTools](#) than rolling such a replacement.

A better suited example might be a meta-reducer that is able to reset the whole state. This could be something you'd use in multiple applications, e.g. when the user logs out. For this purpose, we'll leverage the fact that a reducer returns its initial state when called with an undefined state (that's also what happens when the application starts-up). This way we can write a meta-reducer that effectively replaces the current state with the initial one when a special `reset` action occurs:

```

// meta-reducers.ts
import { ActionReducer, createAction } from "@ngrx/store";

export const reset = createAction("Reset");

export const resettingMetaReducer = (
  reducer: ActionReducer<any>
): ActionReducer<any> => {

```

```

return (state, action) => {
  if (action.type === reset.type) {
    return reducer(undefined, action);
  }
  return reducer(state, action);
};
};

```

After registering this meta-reducer, we can add a button to our application that triggers the dispatch of the `reset` action:

```

<!-- app.component.html -->
<header>
  <h1>Issue Tracker</h1>
  <a routerLink="/">Issues</a>
  <button (click)="reset()">Reset</button>
</header>
<main>
  <router-outlet></router-outlet>
</main>

```

```

// app.component.ts
import { Component } from "@angular/core";
import { reset } from "../store/meta-reducers";
import { Store } from "@ngrx/store";

@Component({ ... })
export class AppComponent {
  constructor(private store: Store) {}

  reset() {
    this.store.dispatch(reset());
  }
}

```

Note that you could also wrap the meta-reducer into a factory function so that you can pass a pre-defined action type (e.g. a logout action).



Meta-Reducers

[Changes](#) | [Source Code](#) | [Live Demo](#)

9.5 Error Handling

By definition, pure functions aren't really allowed to error - at least not in a way that's recoverable. But what if they still do? While you shouldn't explicitly throw errors inside a reducer, there's no guarantee that your code won't produce them anyway. You might be accessing an undefined property or getting a type error while parsing a string. In those cases, the error bubbles up to Angular's [global error handler](#) while NgRx keeps the current state and carries on like nothing happened.

As a general rule you should keep anything that might error out of the reducer. An operation like parsing that is expected to error for some inputs is a side-effect. Either run it before dispatching an action or define a corresponding [effect](#).

However, if you'd like to enhance the stack trace in some way or recover from an unexpected error by placing a flag into the state, you might do so by wrapping your reducer in a try-catch block:

```
// issue.reducer.ts
const reducer = createReducer(
  initialState
  /* action handling */
);

export const issueReducer = (state: IssueState, action: Action): IssueState => {
  try {
    return reducer(state, action);
  } catch (error) {
    console.error(error);
    return state;
  }
};
```



Recommended Read

[Handling Error States with NgRx](#) by Brandon Roberts



Error Handling

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 10

Selectors

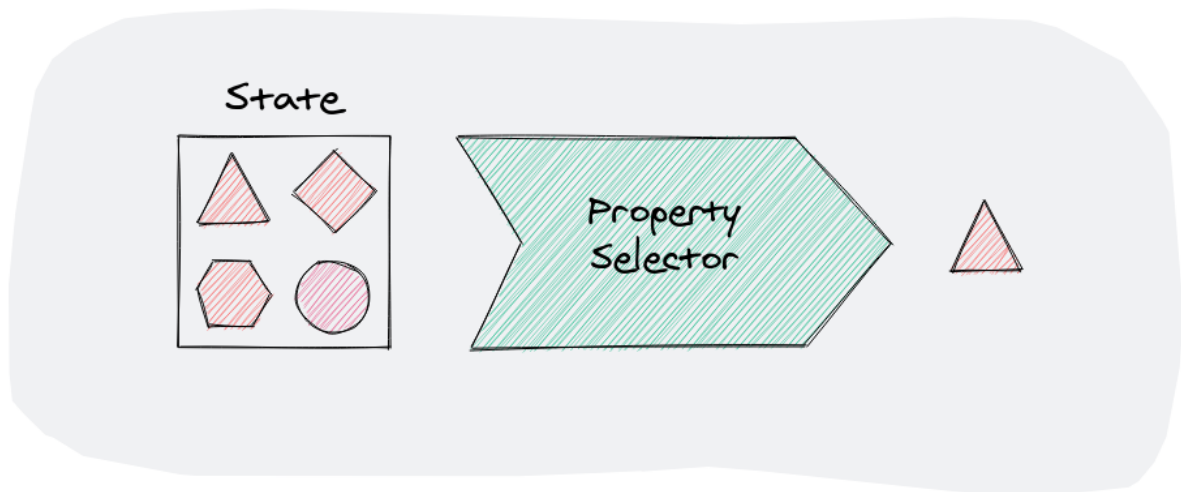


Figure 10.1: Selectors allow us to retrieve properties from the state.

We use selectors in components and services to access the state as an RxJS observable. Up until now we've only done this by invoking `select()` on the store while passing a mapping function to the slice of state that we need. This mapping function is called a selector. Let's use one to display all issues in a newly created list component:

```
// issue-list.component.ts
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';
import { Issue } from '../../models/issue';
import { RootState } from '../../store';

@Component({ ... })
export class IssueListComponent {
  issues$: Observable<Issue[]>;
```



```

constructor(private store: Store<RootState>) {
  this.issues$ = this.store.select((state) =>
    Object.values(state.issue.entities)
  );
}
}

```

With `Object.values()` the values of the issues dictionary can be accessed as an array. This way we can display the issues with `*ngFor` and the `AsyncPipe`.

```

<!-- issue-list.component.html -->
<ul>
  <li *ngFor="let issue of issues$ | async">
    <h2>
      {{ issue.title }}
      <small>{{ issue.priority }}</small>
    </h2>
    <p>{{ issue.description }}</p>
  </li>
</ul>

```

If we're using the same mapping functions or selectors in multiple places, it makes sense to refactor them into a separate file for re-usability:

```

// issue.selectors.ts
export const selectAll = (state: RootState) =>
  Object.values(state.issue.entities);

```

Then we can import the selector and pass it to `select()` like this:

```

// issue-list.component.ts
import * as fromIssue from "../../store/issues.selectors";

this.issues$ = this.store.select(fromIssue.selectAll);

```

I'm using named import prefixed with "from" for the selectors module as this is the only case where it technically makes sense: we're selecting *from* the issue state. Note that this is not required. You're free to use individual imports or a different naming pattern instead.



You can omit the generic type while injecting the store into your components when you're using extracted selectors - the type will be inferred from the selector function.

An observable produced by `store.select()` will only emit distinct values. This way the view will only be updated when there are changes to the slice of the state that is relevant. Therefore it makes sense to restrict the return value of selectors to values that you actually need in your template.

Under the hood, NgRx is leveraging the RxJS operator `distinctUntilChanged()` which relies on simple equality checks to filter duplicate values from an observable. This goes to show how beneficial immutability can be not only for maintainability but for performance as well.



Selectors

[Changes](#) | [Source Code](#) | [Live Demo](#)

10.1 Computed Selectors

When selectors get more complex or need to be composed from rather different parts of the state, we can leverage the `createSelector()` function to create a computed selector. It accepts up to 8 existing selectors and a projector for combining their results. This is helpful for composing selectors. Usually, you'll start by defining a selector that retrieves the feature state in order to then base all other selectors off of that:

```
// issue.selectors.ts
import { createSelector } from "@ngrx/store";
import { RootState } from "..";
import { Issue } from "../../models/issue";
import { Filter } from "../issue.state";

export const selectFeature = (state: RootState) => state.issue;

export const selectEntities = createSelector(
  selectFeature,
  ({ entities }) => entities
);

export const selectAll = createSelector(selectEntities, (entities) =>
  Object.values(entities)
);
```

Composition is not the only advantage of computed selectors. They can also contain logic for deriving view models from the underlying state. A good use-case would be the creation of a selector for retrieving all issues that match our filter. Here we can use the already existing issues selector and combine it with another one that selects the current filter state. After passing these both to `createSelector()` we define a projector that performs the actual filtering:

```
// issue.selectors.ts
export const selectFilter = createSelector(
  selectFeature,
  ({ filter }) => filter
);

export const selectFiltered = createSelector(
  selectAll,
  selectFilter,
  (issues: Issue[], { text }: Filter) => {
    if (text) {
      const lowercased = text.toLowerCase();
      return issues.filter(
        ({ title, description }) =>
          title.toLowerCase().includes(lowercased) ||
          description.toLowerCase().includes(lowercased)
      );
    } else {
      return issues;
    }
  }
);
```

The resulting selector exported as `selectFiltered` can then be passed to `store.select()` like we've seen before. However, computed selectors created with `createSelector()` have an added benefit: **memoization**. This means the projector function won't be called as long as `selectAll` and `selectFilter` return the same values (based on strict equality checks). Instead, NgRx will remember the result from the last invocation and simply return it again. The selector, therefore the projector function, is memoized. Consequently, our search logic will only be run when necessary. On top of that, you can use a computed selector in multiple places and its resulting values will be cached and shared between all subscriptions.

Note that I'm also setting up a text input in the issue list component that dispatches a `search` action upon change. A corresponding reducer then updates the filter text. Check the sources linked at the end of this section to see the corresponding code.

Memoization is another process that relies on simple equality checks and thus immutability. So, in order to have them work correctly always keep in mind **What Not to Put in The State** and only create new objects in reducers instead of modifying existing ones.

Also note that only the projector of selectors created with `createSelector()` is memoized. Therefore

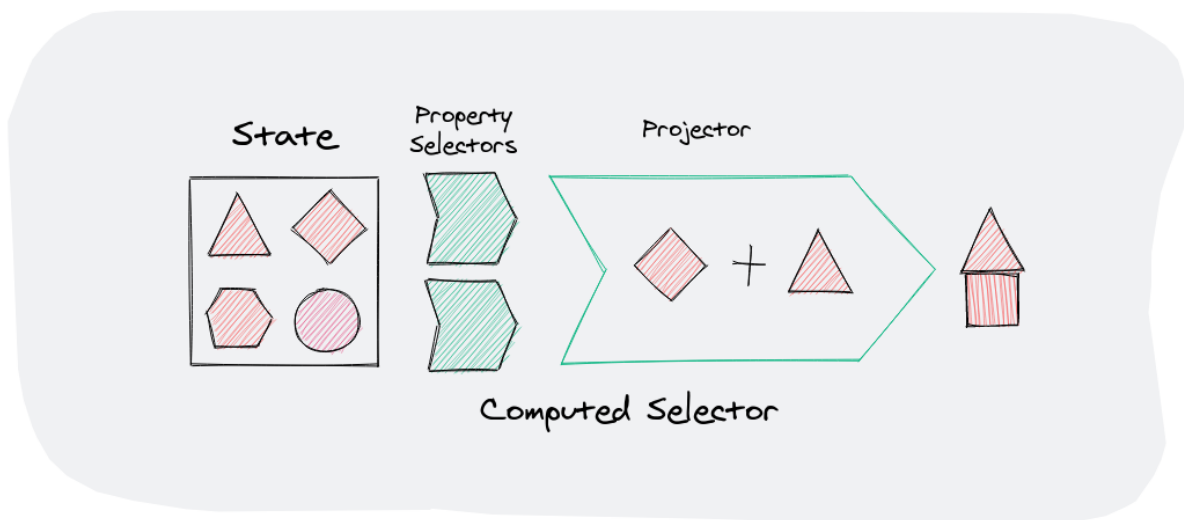


Figure 10.2: Computed selectors transform the state into view models. The state can stay normalized while memoization of the projector ensures performance.

you might move more complex selection logic into a projector even if you're just selecting a single state slice. A good example would be the creation of a view model that offers some statistics on our issues like how many there are in total and how many have already been resolved:

```
// issue.selectors.ts
export interface IssueStats {
  total: number;
  resolved: number;
}

export const selectStats = createSelector(
  selectAll,
  (issues): IssueStats => {
    const resolved = issues.filter((issue) => issue.resolved);
    return {
      total: issues.length,
      resolved: resolved.length,
    };
  }
);
```

We could then display these statistics always up-to-date in the application header:

```
// app.component.ts
@Component({ ... })
export class AppComponent {
```

```

stats$: Observable<fromIssue.IssueStats>;

constructor(private store: Store) {
  this.stats$ = this.store.select(fromIssue.selectStats);
}
}

<!-- app.component.html -->
<header>
  <h1>Issue Tracker</h1>
  <a routerLink="/issues">Issues</a>
  <div *ngIf="stats$ | async as stats">
    <span>{{ stats.resolved }} / {{ stats.total }} resolved</span>
    <button (click)="reset()">Reset</button>
  </div>
</header>
<main>
  <router-outlet></router-outlet>
</main>

```

Lastly, if you ever find yourself in a situation where a selector might be caching a large dataset that isn't used anymore, you can reset its memoization:

```
selectIssuesByPriority.release();
```

In the end, memoization trades space for speed by caching computed results. Therefore it makes sense to release obsolete caches once their values are no longer required.



Recommended Video

[NgRx: Selectors Are More Powerful than You Think](#) by Alex Okrushko



Computed Selectors

[Changes](#) | [Source Code](#) | [Live Demo](#)

10.2 Parameterized Selectors

Sometimes you need outside information for retrieve the right slice of state. For example when selecting a specific issue we'd need to provide its ID. Even for those cases we can leverage memoized selectors - simply specify additional parameters for the projector:

```
// issue.selectors.ts
export const selectOne = createSelector(
  selectEntities,
  (entities: Issues, id: string) => entities[id]
);
```

You'd then use this selector by passing the parameter properties as a second argument to `store.select()` . We might do so in an issue detail-view to retrieve a specific issue based on a route parameter:

```
// issue-detail.component.ts
import { Issue } from "../../store/issues.state";
import * as fromIssue from "../../store/issues.selectors";
import { ActivatedRoute } from "@angular/router";

@Component({ ... })
export class IssueDetailComponent {
  issue$: Observable<Issue>;

  constructor(private route: ActivatedRoute, private store: Store) {
    this.issue$ = this.route.params.pipe(
      switchMap((params) => this.store.select(fromIssue.selectOne, params.id))
    );
  }
}
```

Parameterized selectors are incorporating the parameter properties in their memoization. So, if you're reusing the same selector while alternating properties, its result will be computed again each time. Instead you might want to create distinct selectors with a factory function:

```
// issue.selectors.ts
export const createSelectOne = () =>
  createSelector(
    selectEntities,
    (entities: Issues, id: string) => entities[id]
  );
```

```
// issue-detail.component.ts
this.store.select(fromIssue.createSelectOne(), params.id);
```

However, this mostly makes sense when you're applying the same selector with different properties multiple times on one page. Our detail view will only ever display a single issue at a time. Consequently,

it wouldn't matter much that the selector memoization is reset when switching between different detail views.



Parameterized Selectors

[Changes](#) | [Source Code](#) | [Live Demo](#)

10.3 Pipeable Selectors

Later on we'll be using effects to load issues from a server upon application start. Now, since we're always defining an initial state for every reducer, it's hard to know in a component whether there are no existing issues or if they're just not loaded yet. In order to solve this dilemma we'll introduce a `loaded` flag into the state:

```
// issue.state.ts
export interface State {
  issues: Issue[];
  filter: Filter;
  loaded: boolean;
}
```

Here's a corresponding selector:

```
// issue.selectors.ts
export const selectLoaded = createSelector(
  selectFeature,
  ({ loaded }) => loaded
);
```

Now, instead of selecting from the store we can also pipe it like a regular observable. This way we're able to leverage `RxJS operators` like `skipWhile()` to wait for the flag being switched.

```
// issue-list.component.ts
import { skipWhile } from "rxjs/operators";
import { select } from "@ngrx/store";
import * as fromIssue from "../../store/issues.selectors";

this.issues$ = this.store.pipe(
  skipWhile((state: RootState) => !fromIssue.selectLoaded(state)),
  select(fromIssue.selectAll)
);
```

Note that the `select()` method defined on the store is also available in [form of an RxJS operator](#). Therefore, we're able to use selectors even after applying other operators.

If we need our operator sequence in several places, we can extract it into a custom RxJS operator with the standalone `pipe()` function.

```
// issue.selectors.ts
import { pipe } from "rxjs";
import { skipWhile } from "rxjs/operators";
import { select } from "@ngrx/store";

export const selectAllLoaded = () =>
  pipe(
    skipWhile((state: RootState) => !selectLoaded(state)),
    select(selectAll)
  );
```

Then we can use this pipeable selector by applying it as an operator to the store observable:

```
// issue-list.component.ts
this.issues$ = this.store.pipe(fromIssue.selectAllLoaded());
```

Pipeable selectors like this always come in handy when you want to view the state in a time-dependent way. However, often times you're probably fine by sticking to regular selections and chaining a couple component-specific operators after that.



Instead of having a bunch of boolean flags in your state you might want to introduce some kind of `status` property based on an enum that explicitly pre-defines valid states (similar to a state machine). Otherwise you'll have lots of code dedicated to switching flags which likely ends up with invalid combinations.



Pipeable Selectors

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 11

Fat vs. Thin Actions and Reducers

Let's talk about a common question: how much of your logic should be run when creating actions versus when its corresponding reducer is executed? In other words: should you prefer fat actions and thin reducers with a lot of information payload information or thin actions and fat reducers where there's only the bare minimum in the payload?

As with many things in software development, it depends, but in general you should favor thin actions and fat reducers for two main reasons:

1. Reducers are just pure functions that return the new state. It requires little effort to test their logic by passing them a state plus an action and assert that the resulting state is correct. Testing with fat actions usually involves more moving parts.
2. Having logic in reducers means you know where to look for it. When the logic is tied to the action creation it might be in a lot of different places. Note that you can still put reducer logic in a separate file (e.g. `issue.logic.ts`) in order to keep the actual reducer more readable. Remember though that the functions you write there are still executed inside the reducer and should therefore be effectively pure.

Additionally, you're probably more in an *immutable mindset* while writing reducers and therefore less likely to mutate state as you'd be when creating actions somewhere else. To show you what I mean, let's implement the functionality for resolving existing issues. We could work with a fat action allowing us to simply re-assign the issue inside the reducer:

```
// issue.actions.ts
export const resolve = createAction(
  "[Issue] Resolve",
  props<{ issue: Issue }>()
);
```

```
// issue.reducer.ts
on(IssueActions.resolve, (state, { issue }) => ({
  ...state,
  entities: {
    ...state.entities,
    [issue.id]: issue,
  },
}));
```

However, this might tempt us to perform a mutation during dispatch with a component method like follows where `issue` would be part of the last state:

```
// issue-list.component.ts
resolve(issue: Issue): void {
  issue.resolved = true;
  this.store.dispatch(IssueActions.resolve({ issue }));
}
```

What you'd actually need to do here is copy the issue during action creation:

```
// issue-list.component.ts
resolve(issue: Issue): void {
  this.store.dispatch(
    IssueActions.resolve({ issue: { ...issue, resolved: true } })
  );
}
```



Fat Action

[Changes](#) | [Source Code](#) | [Live Demo](#)

Eventually, due to the reasons just mentioned, we might be better off with a thin action that only contains the ID of a corresponding issue:

```
// issue.actions.ts
export const resolve = createAction(
  "[Issue] Resolve",
  props<{ issueId: string }>()
);

// issue.reducer.ts
on(IssueActions.resolve, (state, { issueId }) => {
  const issue = state.entities[issueId];
```

```
return {
  ...state,
  entities: {
    ...state.entities,
    [issueId]: {
      ...issue,
      resolved: true,
    },
  },
};
});
```

Such a thin action also complies better with viewing actions as events whereas the fat action seems more like a setter. Other than that, thin actions might also ease debugging since it's easier to dispatch thin actions through the Redux DevTools.

An exception of this rule are side-effects like the ID generation that we talked about for the issue submission. Keep these out of reducers even if they add a little more weight to your actions. As mentioned in the previous chapter, the same applies for operations that might error.



Thin Action

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 12

Feature Modules

Reducers that are registered with `StoreModule.forRoot()` in the app module will be loaded eagerly upon startup. Yet, in large Angular apps you're probably working with several, possibly lazy-loaded feature modules. At first sight, this doesn't seem to fit well with the NgRx approach of managing the app state in one global object. But, don't worry, there's a solution: we can also import additional reducers at a later point in time when a module is loaded with `StoreModule.forFeature()`. Note that feature reducers extend your existing store dynamically as you'll always have only one store per application.

Let's implement a feature module that allows the user to manage some settings. We'll use the Angular CLI to generate a new lazy-loaded module:

```
ng generate module settings --routing true --route settings --module app
```

Then we'll create files for actions, selectors, reducer and state definition. You may want to use the [NgRx feature schematic](#) here as follows:

```
ng generate @ngrx/schematics:feature settings/store/settings  
  --module settings/settings.module --creators --flat
```

After running this command you might still have to do some manual tweaking to end up with the following file structure:

```
src/  
  |-- app/  
    |-- settings/  
      |-- store/  
        |-- settings.actions.ts  
        |-- settings.reducer.ts  
        |-- settings.selectors.ts  
        |-- settings.state.ts  
      |-- settings.module.ts
```

The actions file should be similar to the one used for the issues - nothing really new here.

```
// settings.actions.ts
import { createAction, props } from "@ngrx/store";
import { Priority } from "../../models/priority";

export const changeNotificationPriority = createAction(
  "[Settings] Change Notification Priority",
  props<{ notificationPriority: Priority }>()
);
```

Inside `settings.state.ts`, however, we'll have to provide some more typings. Besides defining a feature state interface called `SettingsState` and a corresponding initial state, we'll also export an extended root state. The latter isn't strictly required but generally makes it easier to reason about the state. That's because when the settings module is loaded, the root state will contain the `SettingsState` under a new property `"settings"`. By extracting the property key into a constant `settingsFeatureKey` we are able to re-use it in some other places without typos. The `SettingsRootState` is then defined with this key through the bracket notation:

```
// settings.state.ts
import { Priority } from "../../models/priority";
import { RootState } from "../../store";

export interface SettingsState {
  notificationPriority: Priority;
}

export const initialState: SettingsState = {
  notificationPriority: "low",
};

export const settingsFeatureKey = "settings";

export interface SettingsRootState extends RootState {
  [settingsFeatureKey]: SettingsState;
}
```

Take a look at Figure 12.1 to get a better understanding of how these new state typings fit into the global state. Note that the feature state for the settings module is called `SettingsState` while its feature root state is defined through `SettingsRootState`. Meanwhile, the issues state might be represented by one of the other red shapes inside the root state.

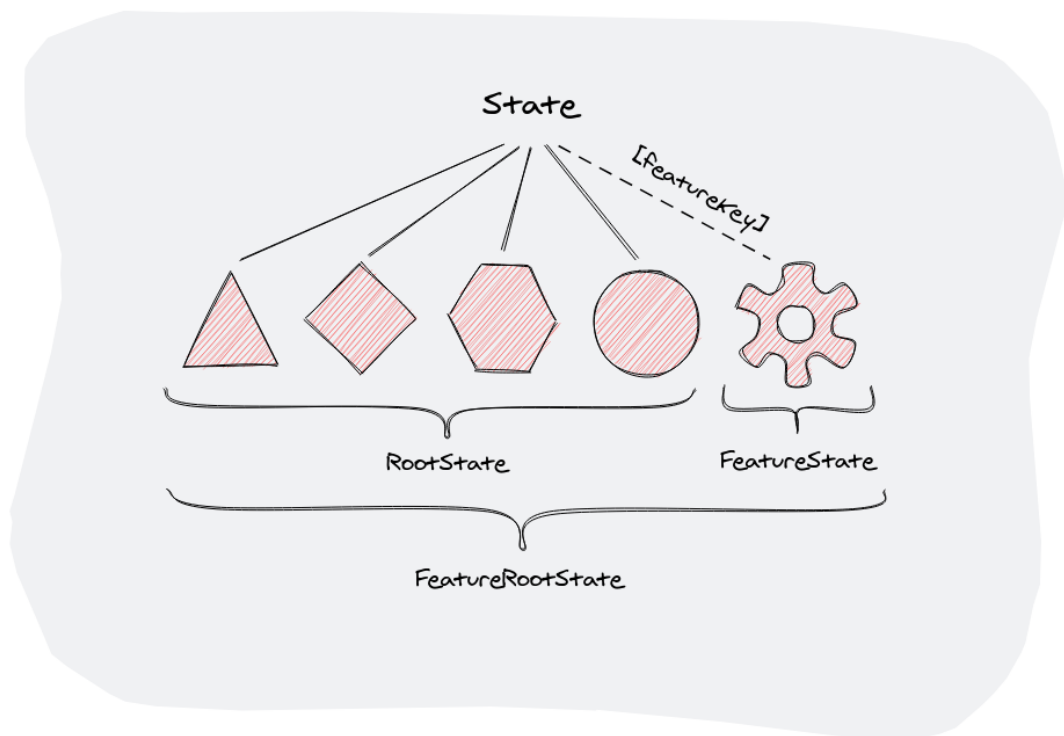


Figure 12.1: NgRx manages all state slices in one state object that can be viewed as a tree. All reducers registered at startup make up the root state. Feature states can be appended to the state tree later on. In order to reference the whole state inside feature modules we define a specific feature root state.

The feature reducer is again really similar to what we already know. Here we only have one state change function for reacting to the change of some priority level at which a user would receive notifications:

```
// settings.reducer.ts
import { createReducer, on } from "@ngrx/store";
import { initialState } from "../settings.state";
import * as SettingsActions from "../settings.actions";

export const settingsReducer = createReducer(
  initialState,
  on(
    SettingsActions.changeNotificationPriority,
    (state, { notificationPriority }) => ({
      ...state,
      notificationPriority,
    })
  )
);
```

Finally, we register the reducer in the module using our feature key with `StoreModule.forFeature()`. When registering a single reducer this method accepts a string key for placing the feature state into the

global state as well as the reducer itself:

```
// settings.module.ts
import { StoreModule } from "@ngrx/store";
import { settingsReducer } from "../store/settings.reducer";
import { settingsFeatureKey } from "../store/settings.state";

@NgModule({
  imports: [
    CommonModule,
    StoreModule.forFeature(settingsFeatureKey, settingsReducer),
  ],
})
export class SettingsModule {}
```

In order to access the feature state you'll want to first define a feature selector. NgRx provides a utility method `createFeatureSelector()` for this to which we'll pass our feature key. Subsequent selectors can then be based off of this feature selector.

```
// settings.selectors.ts
import { createFeatureSelector, createSelector } from "@ngrx/store";
import {
  SettingsRootState,
  SettingsState,
  settingsFeatureKey,
} from "../settings.state";

export const selectFeature = createFeatureSelector<
  SettingsRootState,
  SettingsState
>(settingsFeatureKey);

export const selectNotificationPriority = createSelector(
  selectFeature,
  (settings) => settings.notificationPriority
);
```

Here's how a corresponding component for managing our fictional settings can look:

```
// settings.component.ts
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
```

```

import { Observable } from 'rxjs';
import { Priority } from '../models/priority';
import * as fromSettings from '../store/settings.selectors';
import * as SettingsActions from '../store/settings.actions';

@Component({ ... })
export class SettingsComponent {
  notificationPriority$: Observable<Priority>;

  constructor(private store: Store) {
    this.notificationPriority$ = this.store.select(
      fromSettings.selectNotificationPriority
    );
  }

  changeNotificationPriority(notificationPriority: Priority): void {
    this.store.dispatch(
      SettingsActions.changeNotificationPriority({ notificationPriority })
    );
  }
}

```

```

<!-- settings.component.html -->
<h2>Settings</h2>
<label for="priority">Notification Priority</label>
<select
  FormControlName="priority"
  id="priority"
  #prioritySelect
  [value]="notificationPriority$ | async"
  (change)="changeNotificationPriority(prioritySelect.value)"
>
  <option value="low">Low</option>
  <option value="medium">Medium</option>
  <option value="high">High</option>
</select>

```




I've deliberately excluded effects from this chapter since there's almost no difference between feature and root effects. The architectural considerations for feature modules mostly relate to state and reducer definitions. Don't worry, I'll be mentioning feature effects in the [effects chapter](#).



Feature Modules

[Changes](#) | [Source Code](#) | [Live Demo](#)

12.1 Multiple Reducers per Module

Sometimes a feature gets too complex for a single reducer and we would like to divide it into separate slices. Imagine that the settings page grows into several sub-categories: one for managing notification settings, another one for configuring your profile and so on. It would make sense to have a designated reducer for managing each settings category. Luckily, it's also possible to register multiple reducers within a feature module just like we did for the root state. For this purpose I'd encourage the following file structure where a module-specific `index.ts` is introduced similar to the one for the store root:

```
src/
|-- app/
|   |-- settings/
|   |   |-- store/
|   |   |   |-- notification/
|   |   |   |   |-- notification.actions.ts
|   |   |   |   |-- notification.reducer.ts
|   |   |   |   |-- notification.selectors.ts
|   |   |   |   |-- notification.state.ts
|   |   |   |-- profile/
|   |   |   |   |-- profile.actions.ts
|   |   |   |   |-- profile.reducer.ts
|   |   |   |   |-- profile.selectors.ts
|   |   |   |   |-- profile.state.ts
|   |   |-- index.ts
|-- store/
|   |-- issue/
|   |   |-- issue.actions.ts
|   |   |-- issue.reducer.ts
|   |   |-- issue.selectors.ts
|   |   |-- issue.state.ts
|-- index.ts
```

This way the files of directories like `issue/` or `notification/` don't have to care whether they provide root or feature reducers. Let's move the reducer from `settings.reducer.ts` to `notification.reducer.ts` and rename it to `notificationReducer`; same thing for the state interface from `settings.state.ts` to `notification.state.ts`. Next, we can introduce corresponding files for profile settings in a `profile/` sub-directory. Then we define the combined module state inside the new `index.ts`:

```
// index.ts
import { ActionReducerMap, createFeatureSelector } from "@ngrx/store";
import { RootState } from "../../store";
import { notificationReducer } from "../notification/notification.reducer";
import { NotificationState } from "../notification/notification.state";
import { profileReducer } from "../profile/profile.reducer";
import { ProfileState } from "../profile/profile.state";

export interface SettingsState {
  notification: NotificationState;
  profile: ProfileState;
}

export const settingsReducers: ActionReducerMap<SettingsState> = {
  notification: notificationReducer,
  profile: profileReducer,
};

export const settingsFeatureKey = "settings";

export interface SettingsRootState extends RootState {
  [settingsFeatureKey]: SettingsState;
}

export const selectFeature = createFeatureSelector<
  SettingsRootState,
  SettingsState
>(settingsFeatureKey);
```

Note that I've pulled the selector as well as the interfaces `SettingsState` and `SettingsRootState` up while embedding `NotificationState` and `ProfileState` accordingly. Also, I defined a reducer mapping that we can pass to `StoreModule.forFeature()` instead of a single reducer:

```
// settings.module.ts
import { StoreModule } from "@ngrx/store";
import { settingsFeatureKey, settingsReducers } from "../store";

@NgModule({
  imports: [StoreModule.forFeature(settingsFeatureKey, settingsReducers)],
})
export class SettingsModule {}
```

Eventually, we need to create new selectors for the sub-feature states based on `selectFeature` :

```
// notification.selectors.ts
import { createSelector } from "@ngrx/store";
import * as fromSettings from "../..";

export const selectFeature = createSelector(
  fromSettings.selectFeature,
  ({ notification }) => notification
);

export const selectPriority = createSelector(
  selectFeature,
  (settings) => settings.priority
);
```



Multiple Reducers per Module

[Changes](#) | [Source Code](#) | [Live Demo](#)

12.2 Deciding between root and feature state

Any state that is managed by a root reducer will be accessible from every module of your application. The state of feature reducers, in turn, is only available when the corresponding module is loaded. Despite the fact that all of these state slices will be held in a single object by NgRx, you can only guarantee that root reducers are loaded everywhere while feature modules might get initialized later when they're lazy-loaded. Therefore you need to structure your state and modules accordingly.

Generally, you can apply the same rules you might already be using for organizing other parts of your app. For example, when deciding where to register a component you'll probably place it in the most specific place possible. So, if we're talking about a component that is only used by a certain module, it'll reside with the module. If the component is instead re-used in many modules or serves a core purpose in your app (e.g. navigation) it's commonly registered in the root or a shared module.

Replacing the word “component” with “reducer” gives us a good starting point for organizing our store. Module-specific state should be managed per module while shared or core state belongs into the store root. When you get to a point where feature state is suddenly required by multiple modules you’ve got two options:

1. Re-think your module separation
2. Pull feature state up into the root

Both options are equally valid and you should decide on a case-by-case basis. Although pulling the state up probably seems easier most of the time, there might be worth in revisiting your architecture. Either way it makes sense to keep state local as long as possible and only pull it up when you need to.

Chapter 13

Effects

A side-effect is a function that does anything else than deterministically compute a value based on its parameters. Most of the time this is narrowed to changing external state or performing asynchronous tasks. So, when you're mutating a class attribute, that's a side-effect. Making a network request to update a database? Definitely a side-effect. Persisting something to local storage? You guessed it!

Now you may ask: well, what's so bad about side-effects? I'll give you the answer that I was missing for a long time when getting in functional programming concepts: there's nothing bad about side-effects. In fact, they're an essential part of any non-trivial application. Imagine we could never save any user inputs because it's a side-effect - that would be pretty boring.

By referring to code that is not pure as having side-effects or being effectful we make things explicit. If effects are explicit, they're easier to know about and trace. Effects we don't know about, in contrast, probably cause bugs more often than ones we register explicitly. In Angular, side-effects are usually scattered throughout the application. Components can trigger HTTP requests via services which might populate class attributes with the response values and so on. With NgRx, however, we're able to isolate effects to keep them in check.

Up until now the only place to execute logic with NgRx is inside reducers. But reducers are - and should remain - pure functions in order to keep state transitions easily comprehensible. Luckily, NgRx has a package that provides remedy: `@ngrx/effects`. It allows us to define side-effects in form of long-running observables that are based on the event-bus, dispatch events themselves or both. While you could still define side-effects inside of components or plain services, the effects package offers valuable foundations for managing the creation, lifecycle and error handling of effects.

13.1 Installation

You can install the most recent version of `@ngrx/effects` with npm or yarn respectively like this:

```
npm install @ngrx/effects
```

```
yarn add @ngrx/effects
```

Again, there's also a schematic for adding the library which creates and registers an effects class by default. When passing `--minimal` it omits this setup which is probably better in our case:

```
ng add @ngrx/effects@latest --minimal
```

This way you should end up importing `EffectsModule.forRoot()` into the app module with an empty array after the `StoreModule` :

```
// app.module.ts
import { StoreModule } from "@ngrx/store";
import { reducers } from "../store/root";
import { EffectsModule } from "@ngrx/effects";

@NgModule({
  imports: [StoreModule.forRoot(reducers), EffectsModule.forRoot([])],
})
export class AppModule {}
```



Installation

[Changes](#) | [Source Code](#) | [Live Demo](#)

Simulating a REST API

I'm using Angular's [In-memory Web API](#) to simulate a backend server. Note that its source has moved into the official Angular monorepo [here](#). Either way, you can install the module as follows:

```
npm install angular-in-memory-web-api
```

After that you may implement an Angular service that acts as a backend database:

```
// database.service.ts
import { Injectable } from "@angular/core";
import { InMemoryDbService } from "angular-in-memory-web-api";
import { Issue } from "../models/issue";
import { randomId } from "../util";

interface Database {
  issues: Issue[];
}
```

```

@Injectable()
export class DatabaseService implements InMemoryDbService {
  createDb(): Database {
    return {
      issues: [
        {
          id: this.genId(),
          title: "Example Issue",
          description: "This is a pre-existing issue",
          priority: "medium",
          resolved: false,
        },
      ],
    };
  }

  genId(): string {
    return randomId();
  }
}

```

Finally, spin this in-memory database up by importing the `InMemoryWebApiModule` into your root module while passing the service to its `forRoot()` method:

```

import { InMemoryWebApiModule } from "angular-in-memory-web-api";

@NgModule({
  imports: [InMemoryWebApiModule.forRoot(DatabaseService)],
})
export class AppModule {}

```

This module will now intercept all requests made under `/api` by the HTTP client while we can work as if there was a real backend server.

You might also want to try an Angular-external solution like the [json-server](#) module. In any case, it should go without saying that you'll want to use an actual server for a real application.



Simulating a REST API

[Changes](#) | [Source Code](#) | [Live Demo](#)

13.2 Creating Effects

Let's start with an effect for saving issues to a server. This effect will be encapsulated in an injectable service class called `IssuesEffects`. You can either create this class yourself inside `issue.effects.ts` and register it through the array passed to `EffectsModule.forRoot()` - or run the following schematic:

```
ng generate @ngrx/schematics:effect store/issue --module app --root true
```

If the command offers to generate any additional code, you may decline for now so we can build the class up step-by-step. It should look as follows:

```
// store/issues/effects.ts
import { Injectable } from "@angular/core";
import { Actions, Effect } from "@ngrx/effects";

@Injectable()
export class IssuesEffects {
  constructor(private action$: Actions) {}
}
```

Note that the `@Injectable()` decorator should not contain the `providedIn` metadata option since the class will be managed by NgRx. The `Actions` service is an observable provided by NgRx that emits any dispatched action after it went through your reducers (i.e. the state has been updated). It's injected as `action$` following the Finnish notation. This observable will be the basis for most effects.

We'll want to save issues to the server as soon as the user submits it and preferably before it's stored in the state. However, listening to `action$` will emit the `submit` action after the `issueReducer` has run. Therefore we need to describe this process with two actions instead of one. A `submit` action should trigger our effect while a newly defined `submitSuccess` action should make the reducer store the issue into the state.

```
// issue.actions.ts
export const submit = createAction("[Issue] Submit", props<{ issue: Issue }>());

export const submitSuccess = createAction(
  "[Issue] Submit Success",
  props<{ issue: Issue }>()
);
```

Actions created through `submitSuccess` now also contain a fully-constructed `Issue` that'll be returned from the server. This issue then already contains a server-generated ID, so there's no need for our ID generator anymore. Consequently, the `submit` method of our component as well as the

reducer have to be refactored as follows:

```
// new-issue.component.ts
submit(): void {
  const issue = this.form.value;
  this.store.dispatch(IssueActions.submit({ issue }));
}

// issue.reducer.ts
on(IssueActions.submitSuccess, (state, { issue }) => {
  return {
    ...state,
    entities: {
      ...state.entities,
      [issue.id]: issue,
    },
  };
});
```

We create an individual effect using the `createEffect()` function. It accepts a factory function that returns a stream producing actions. For saving submitted issues this stream is created by piping the `Actions` observable through some RxJS operators. This way we can expose an observable that effectively dispatches additional actions to the store after performing an asynchronous operation.

```
// issue.effects.ts
import { HttpClient } from "@angular/common/http";
import { Injectable } from "@angular/core";
import { Actions, createEffect, ofType } from "@ngrx/effects";
import { map, mergeMap } from "rxjs/operators";
import { Issue } from "../../models/issue";
import * as IssueActions from "../issue.actions";

@Injectable()
export class IssueEffects {
  submit$ = createEffect(() =>
    this.action$.pipe(
      ofType(IssueActions.submit),
      mergeMap((action) => this.http.post<Issue>(`/api/issues`, action.issue)),
      map((issue) => IssueActions.submitSuccess({ issue }))
    )
  );
};
```

```

    constructor(private action$: Actions, private http: HttpClient) {}
}

```

Let's break those pipeable operators down one-by-one:

1. `ofType()` : This is a custom operator provided by NgRx. It's basically like RxJS' `filter()` , but specifically for actions. We can simply pass an action creator and the operator will check the type of all actions coming through while letting only the ones we specified pass. After applying it we've got an observable that will only emit dispatched actions created with the `submit` action creator - so any action of type `"[Issues] Submit"` . You're also able to pass multiple types or creators to `ofType()` for cases where you'd like to listen for more than one kind of action.
2. `mergeMap()` : This operator does two things: it maps emitted values to another observable through a project function and also merges the values that in turn are emitted from all resulting observables. By destructuring the action parameter we're able to construct a server request for saving an issue. Now we've got an observable that will map each `submit` action to an HTTP POST request and subsequently merge all responses. Merging is the right choice here, because a user might create a second issue before the first one is saved. However, we want to save all issues no matter when they are submitted or in what order. Other so-called "flattening" operators like `switchMap()` or `concatMap()` wouldn't allow this. With `switchMap()` an issue would only be saved successfully when it's not interrupted by a more recent submission. In turn, `concatMap()` would retain the order of submissions, possibly making us wait longer than necessary.
3. `map()` : Lastly, we need to *map* each HTTP response back to an action that can be dispatched to the store. The observable created by the HTTP client is unwrapped to an `Issue` through `mergeMap()` at this point. Therefore we only need to pass this server-generated issue to our new `submitSuccess` action creator. We end up with an observable that will emit actions of the type `"[Issues] Submit Success"` .

The resulting effect is stored in a public property of the effect class - again using Finnish notation. The naming doesn't really matter, storing it this way just makes the effect available for subscription by NgRx.

In order to kick off our effect we need to register its class with NgRx through `EffectsModule.forRoot()` . NgRx will then subscribe to any observable created with `createEffect()` and dispatch resulting actions to the store. Don't forget to also import the `HttpClientModule` so that the `HttpClient` becomes available to our effect class.

```

// app.module.ts
import { HttpClientModule } from "@angular/common/http";
import { EffectsModule } from "@ngrx/effects";
import { IssuesEffects } from "../store/issue/issue.effects";

```

```
@NgModule({
  imports: [HttpClientModule, EffectsModule.forRoot([IssuesEffects])],
})
export class AppModule {}
```



Effects from feature modules can be defined in the same way, however, they're registered by passing all effect classes as an array to `EffectsModule.forFeature()` in their corresponding feature module. Note that these kinds of effects will only be started once a respective feature module is loaded.

As you can see, `@ngrx/effects` allows us to explicitly define side-effects like HTTP calls without interfering with any of the store concepts that we learned before. We just get a place where we can trigger tasks based on actions and eventually return other actions once these tasks are done.

That said, it's advisable to use effects only for such orchestration while extracting other details. For our example, that would mean moving the HTTP request into a regular Angular service:

```
// issue.service.ts
@Injectable({ providedIn: "root" })
export class IssueService {
  constructor(private http: HttpClient) {}

  save(issue: Issue): Observable<Issue> {
    return this.http.post<Issue>(`/api/issues`, issue);
  }
}
```

```
// issue.effects.ts
import { IssueService } from "../../services/issue.service";

@Injectable()
export class IssueEffects {
  submit$ = createEffect(() =>
    this.action$.pipe(
      ofType(IssueActions.submit),
      mergeMap((action) => this.issues.save(action.issue)),
      map((issue) => IssueActions.submitSuccess({ issue }))
    )
  );
};
```

```
constructor(private action$: Actions, private issues: IssueService) {}  
}
```



You'll get better type hints on effect observables when you pass an arrow function with a body to `createEffect()`, see [here](#) for more details.



Creating Effects

[Changes](#) | [Source Code](#) | [Live Demo](#)

13.3 Accessing the State

There will be times where you can't or don't want to attach all required metadata to an action. Instead, you'd rather access the state from your effect. Let's say we want to prevent submission of issues having the same title as an already existing one. Consequently, we need to filter our existing effect based on what's in the store. We can do so using the RxJS operator `withLatestFrom()`. It'll hold the latest value from one or multiple other observables ready and gives us an observable that emits our underlying action and such additional values combined. In our case, there's one other observable: a selection of all issues from the store:

```
// issue.effects.ts  
import { filter, map, mergeMap, withLatestFrom } from "rxjs/operators";  
import { Store } from "@ngrx/store";  
import * as fromIssue from "../issue.selectors";  
  
@Injectable()  
export class IssueEffects {  
  submit$ = createEffect(() =>  
    this.action$.pipe(  
      ofType(IssueActions.submit),  
      withLatestFrom(this.store.select(fromIssue.selectAll)),  
      filter(([action, issues]) =>  
        issues.every(({ title }) => title !== action.issue.title)  
      ),  
      mergeMap(([action, issues]) => this.issues.save(action.issue)),  
      map((issue) => IssueActions.submitSuccess({ issue })))  
  )  
};
```

```

constructor(
  private action$: Actions,
  private issues: IssueService,
  private store: Store
) {}
}

```

Inside the subsequent `filter()` operator we're then able to check that the title of the issue-to-be is different from every existing issue. If the predicate function defined inside `filter()` returns `false`, the underlying value is excluded from the stream - otherwise it passes through unaltered. That's why the projection inside `mergeMap()` now also works with the tuple of action and issues array.



Always remember that **effects are run after reducers**. So, the action you're basing your effect on may already have changed the state. If this doesn't fit your use-case you might want to split up your action like we did with `submit` and `submitSuccess`.

While `withLatestFrom()` is definitely a neat way to incorporate the state into effects, it also has a catch: applying it directly on the stream will run the selector upon effect registration - so, basically upon app startup. This might degrade performance when your application grows and you're registering a lot of effects like this. As a workaround, we can combine our stream lazily with the state selection by using the `concatMap()` operator. It maps stream values to observables which are unwrapped in order once they complete. Therefore it's similar to `mergeMap()`, but while `mergeMap()` just merges the values of resulting observables, `concatMap()` also maintains their order.

```

// issue.effects.ts
import { map, mergeMap, withLatestFrom, concatMap } from "rxjs/operators";
import { of } from "rxjs";

createEffect(() =>
  this.action$.pipe(
    ofType(IssueActions.submit),
    concatMap((action) =>
      of(action).pipe(withLatestFrom(this.store.select(fromIssue.selectAll)))
    ),
    filter(([action, issues]) =>
      issues.every(({ title }) => title !== action.issue.title)
    ),
    mergeMap(([action, issues]) => this.issues.save(action.issue)),
    map((issue) => IssueActions.submitSuccess({ issue }))
  )
)

```

```
)  
);
```

`of()` is a function that creates an observable which will emit the function argument(s) and complete right after that. By creating the underlying observable this way and applying `withLatestFrom()` we basically get the same stream as before, but the selector will only be run once the first action passes through the effect.



I know that all this RxJS operator mumbo jumbo can be hard to grasp - I've been there, I feel you. Don't get discouraged, there's nothing wrong with you! At some point it'll click - actually there probably will be several clicks each time you get a deeper understanding of reactive concepts. What helps me is imagining observables like [pneumatic tubes](#) where packages rush through. Operators are a way to transform or filter these packages as well as combine multiple tubes in different ways. That doesn't mean you need to use the same mental concept. Many people seem to respond well to visual representations - draw your own, take a look at [marble diagrams](#) or experiment with an [interactive visualizer](#). See what works for you, ask questions and keep going. I believe in you!

If you find yourself accessing the state this way in multiple effects, you can also create a custom RxJS operator to re-use this operator sequence. It might look like this ...

```
export const withLatestFromDeferred = <A, B>(other: Observable<B>) =>  
  pipe(concatMap((value: A) => of(value).pipe(withLatestFrom(other))));
```

... and could be used like that:

```
// issue.effects.ts  
createEffect(() =>  
  this.action$.pipe(  
    ofType(IssueActions.submit),  
    withLatestFromDeferred(this.store.select(fromIssue.selectAll)),  
    filter(([action, issues]) =>  
      issues.every(({ title }) => title !== action.issue.title)  
    ),  
    mergeMap(([action, issues]) => this.issues.save(action.issue)),  
    map((issue) => IssueActions.submitSuccess({ issue })))  
);
```

In general your actions should already provide enough metadata to run corresponding effects. This makes state transitions more predictable. So, don't blindly grab anything from the state, but also rethink

how and where you're dispatching actions from time to time.



Take care not to mutate the state or action payloads inside effects. The rules of immutability still apply here. If you're tempted, rather use the spread operator or leverage Immer's `produce()` function.



Accessing the State

[Changes](#) | [Source Code](#) | [Live Demo](#)

13.4 Error Handling

Effects are basically just long-running observables. So, once an error occurs, they're done and won't emit any more values. Luckily, NgRx has already some error handling built-in: effects will be restarted up to 10 times when they produce errors. This is meant to provide some baseline recoverability while preventing your application from producing errors in an endless loop.



You can disable the default error handler where NgRx resubscribes by passing a configuration object with the property `useEffectsErrorHandler` set to `false` as a second argument to `createEffect()`. However, you don't need to do this even when you're handling some errors yourself - you can have both. Moreover, you're able to provide your own default error handler through the `EFFECTS_ERROR_HANDLER` injection token. This way you could for example alter the number of times that effects are restarted.

When you're expecting errors though, you're better off defining explicit error handling per effect. Maybe you already noticed that filtering duplicate issue titles on the client-side might be ill-advised. Multiple clients could run into race conditions, therefore the solution from the previous chapter should probably be moved to the server. So, let's pretend that the HTTP request triggered by the effect can produce an error-response due to server validation and consequently remove the filtering. With NgRx's default error handling, a user could then submit 10 issues with an already existing title until the effect breaks - after that they wouldn't even be able to submit an issue with a valid title. We can fix this with custom error handling using the `catchError()` operator. It allows us to handle individual errors and return a fallback observable.

```
// issue.effects.ts
import { EMPTY } from "rxjs";
import { catchError, map, mergeMap } from "rxjs/operators";

createEffect(() =>
  this.action$.pipe(
```

```

    ofType(IssueActions.submit),
    mergeMap((action) =>
      this.issues.save(action.issue).pipe(catchError(() => EMPTY))
    ),
    map((issue) => IssueActions.submitSuccess({ issue }))
  )
);

```

Notice that I've applied the error handling to the inner observable, the one returned from the HTTP call underneath `this.issues.save()`. This way we're providing a fallback for the request observable. It's important to handle errors inside flattening operators or rather on inner observables. If we instead apply `catchError()` on the outer piping, we'd be replacing the whole effect observable upon error.

Moreover, the `EMPTY` constant from RxJS represents an observable that completes immediately without emitting any values. By returning such an empty observable from our error handling we're basically letting the HTTP call fail silently. You might want to get more fine-grained here, maybe even move some specific error cases into the respective service. Eventually though, we end up with a similar behavior to what we had before while using the `filter()` operator: invalid issue submissions will be ignored - only now the validation can take place server-side.

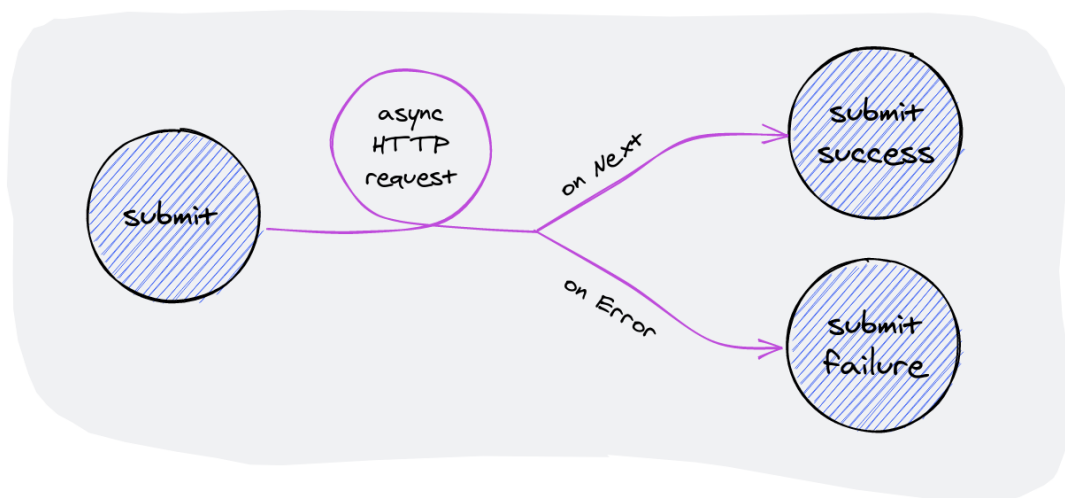


Figure 13.1: In most cases effects perform async tasks based on dispatched actions and communicate their result back to the store in form of new actions.

Sometimes it's necessary to let our store know about the failure of effects. Let's introduce a `loading` flag into our issue state to demonstrate this:

```

// issue.state.ts
export interface State {
  issues: Issue[];
  filter: Filter;
  loaded: boolean;
}

```



```
loading: boolean;
}
```

You might want to display some kind of loading spinner while this flag is true. We'd flip it when the user submits the issue and reset it once the effect is done:

```
// issue.reducer.ts
createReducer(
  initialState,
  on(IssueActions.submit, (state) => {
    return {
      ...state,
      loading: true,
    },
  ),
  on(IssueActions.submitSuccess, (state, { issue }) => {
    return {
      ...state,
      entities: {
        ...state.entities,
        [issue.id]: issue,
      },
      loading: false,
    };
  })
);
```

Now, what happens when the submission fails? Nothing, since we're letting it fail silently. In those cases you can instead recover from an underlying error with an action that informs the store about the failure - remember to create this action inside `issues.actions.ts`.

```
// issue.actions.ts
export const submitFailure = createAction("[Issue] Submit Failure");
```

```
// issue.effects.ts
createEffect(() =>
  this.action$.pipe(
    ofType(IssueActions.submit),
    mergeMap((action) =>
      this.issues.save(action.issue).pipe(
        map((issue) => IssueActions.submitSuccess({ issue })),
        catchError(() => of(IssueActions.submitFailure()))
      )
    )
)
```

```

    )
  )
)
);

```

Note that we'll now also need to move the action mapping into the inner piping so that `mergeMap()` can always unwrap an observable containing an action. Then we can also reset the `loading` flag when the submission has failed:

```

// issue.reducer.ts
on(IssueActions.submitFailure, (state) => ({
  ...state,
  loading: false,
})));

```

Of course, you're free to implement the error handling that best fits your use-case. Other options might include custom retry logic with `retry()` / `retryWhen()` or querying another API as a fallback.



Error Handling

[Changes](#) | [Source Code](#) | [Live Demo](#)

13.5 Optimistic vs. Pessimistic Updates

There are two points in time where we can update client state that's reflecting data stored on a server:

1. *Before* the changes are made on the server. That would be called **optimistic** since we can't be sure that a corresponding server request is successful - worst case we'll have to rollback the local state.
2. *After* the changes are made on the server. We'd call that **pessimistic** since we don't want to reflect a state locally that didn't already persist to a database.

Currently, our issue submission is implemented pessimistically because we're only putting issues into the state once `submitSuccess` is dispatched. So, let's implement issue resolving in an optimistic way to see both approaches and their consequences for the user experience. First, we'll define an action for triggering the resolve process. It'll just contain the ID of an issue that should be resolved. Additionally, we define an action that indicates effect failure which will allow us to rollback client-side changes when they've been too optimistic.

```

// issue.actions.ts
export const resolve = createAction(
  "[Issue] Resolve",
  props<{ issueId: string }>()
);

```

```
export const resolveFailure = createAction(
  "[Issue] Resolve Failure",
  props<{ issueId: string }>()
);

export const resolveSuccess = createAction("[Issue] Resolve Success");
```

You can dispatch actions created with `resolve` through a little button on each issue in your view. Since we're optimistic about this, we'll keep updating the state directly after this in our reducer. However, when the reducer gets a failure action it'll rollback the resolve:

```
// issue.reducer.ts
on(IssueActions.resolveFailure, (state, { issueId }) => {
  const issue = state.entities[issueId];
  return {
    ...state,
    entities: {
      ...state.entities,
      [issueId]: {
        ...issue,
        resolved: false,
      },
    },
  };
});
```

We still need an effect to propagate this change to the server. Keep in mind that it'll run after a corresponding `resolve` action has been reduced. Therefore we'll fallback to a `resolveFailure` action upon error in order to trigger a rollback:

```
createEffect(() =>
  this.action$.pipe(
    ofType(IssueActions.resolve),
    mergeMap(({ issueId }) =>
      this.issues.resolve(issueId).pipe(
        map(() => IssueActions.resolveSuccess()),
        catchError(() => of(IssueActions.resolveFailure({ issueId })))
      )
    )
  )
)
```

```
);
```

Operations that add data (like our issue submission) are a bit more difficult to implement in an optimistic way. When a resource already exists (as it's the case for the resolving) we can reference it by its server-generated ID for the rollback. However, when you need an optimistic behavior while you're just creating a resource you'd have to use an intermediate ID on the client-side. You'd subsequently replace this client-generated ID once the resource was created successfully on the server - or have the server persist the client-side ID.

Moreover, you might want to limit optimistic updates to non-crucial parts of your application. This way the user won't ever see wrong states where it matters. Additionally, you should let optimistic updates timeout after a short duration so that no data is rolled back too late for the user to notice.



Optimistic vs. Pessimistic Updates

[Changes](#) | [Source Code](#) | [Live Demo](#)

13.6 Initial Data and Effects

Remember the `loaded` flag that we introduced in the example for [pipeable selectors](#)? Let's put it to use by loading issues from the server in an effect.

First, we'll define an action for initiating the load as well as another one for notifying the store about success. You might also define a failure action here, if you need it.

```
// issue.actions.ts
export const load = createAction("[Issue] Load");

export const loadSuccess = createAction(
  "[Issue] Load Success",
  props<{ issues: Issue[] }>()
);
```

Then we can write a corresponding effect where `this.issues.getAll()` returns an HTTP-based observable containing an issue array:

```
// issue.effects.ts
createEffect(() =>
  this.action$.pipe(
    ofType(IssueActions.load),
    switchMap(() => this.issues.getAll()),
    map((issues) => IssueActions.loadSuccess({ issues }))
  )
)
```

```
);
```

Lastly, our reducer could put the retrieved issues into the store and flip the `loaded` flag accordingly:

```
// issue.reducer.ts
on(IssueActions.loadSuccess, (state, { issues }) => {
  const entities: Issues = {};
  issues.forEach((issue) => (entities[issue.id] = issue));
  return {
    ...state,
    entities,
    loaded: true,
  };
});
```

The only question is: where are we going to dispatch the `load` action? You'll find several suggestions out there, but let's explore two solid options:

1. Upon initialization of a container component
2. Upon effect initialization

The first option might also be the most obvious one: we'd request the issues where we need them and only there. This could take place in the component which displays them in a list or its parent. We could simply dispatch from inside `ngOnInit()` :

```
// issues.component.ts
ngOnInit(): void {
  this.store.dispatch(IssueActions.load())
}
```

This approach comes with some drawbacks though - depending on what you're trying to achieve. On one side, we'll be loading the data each time the container component is added to the view. You might work around this behavior by checking the `loaded` flag in the corresponding effect by **accessing the state**. On the other side, we need to make sure that the issues are loaded in every place where we need them. Let's say you're adding a detail route: a specific issue might then be loaded when navigating from the list, but could be missing upon page reload.

Another option for fetching initial data is presented by **effect lifecycle** hooks. Similar to component hooks, you're able to implement certain methods for tapping into the lifecycle of your effect classes. The `OnInitEffects` hook is to effects what `OnInit` is to components. It allows us to implement the method `ngrxOnInitEffects()` from which we can return an action to be dispatched once the effects class is initialized:

```
// issue.effects.ts
import { OnInitEffects } from "@ngrx/effects";

@Injectable()
export class IssueEffects implements OnInitEffects {
  load$ = createEffect(() =>
    this.action$.pipe(
      ofType(IssueActions.load),
      switchMap(() => this.issues.getAll()),
      map((issues) => IssueActions.loadSuccess({ issues })))
  );

  ngrxOnInitEffects(): Action {
    return IssueActions.load();
  }
}
```

Returning our loading action will fetch the issues once `IssueEffects` is registered with a root or feature module. However, this also means that the underlying HTTP request will be kicked-off upon app start or at the latest when the corresponding feature module is lazy-loaded. Consequently, you will have less availability problems then before, but in turn you might be running a lot of work upfront.



You might be tempted to base effects for initial data upon NgRx lifecycle actions like `INIT` or `ROOT_EFFECTS_INIT`. Yet, the former only denotes the initialization of the store and will be dispatched too late as to be picked up by any effect. This is not the case for the latter, however, as its name would suggest, the action of type `ROOT_EFFECTS_INIT` will only be dispatched once after all root effects are registered. So, you won't be able to leverage it inside feature effects. Therefore, you're mostly better off with the other options presented here.



Initial Data and Effects

[Changes](#) | [Source Code](#) | [Live Demo](#)

13.7 Non-Dispatching Effects

Normally, effects are required to be observables emitting actions which can be dispatched. However, there are cases where you want to run some logic based on a dispatched action without the need to notify

the store after it's done. What you want is an effect that doesn't result in an action. For example, we might want to display a notification once an issue is successfully saved. After filtering for the corresponding action type, we'd invoke a `NotificationService` via the `tap()` operator. It allows us to run some code every time a value passes through without altering the stream itself. Without anything else, the `notification$` observable would now emit any already dispatched `submitSuccess` action again, effectively creating an endless loop. In order to prevent this, we can pass a second configuration parameter to `createEffect()` with a property `dispatch` set to `false`. NgRx will then ignore any actions coming from this effect:

```
// issue.effects.ts
@Injectable()
export class IssueEffects {
  notification$ = createEffect(
    () =>
      this.action$.pipe(
        ofType(IssueActions.submitSuccess),
        tap(({ issue }) => {
          this.notifications.info(`Issue submitted: ${issue.title}`);
        })
      ),
    { dispatch: false }
  );

  constructor(
    private action$: Actions,
    private issues: IssueService,
    private notifications: NotificationService
  ) {}
}
```

Such non-dispatching effects also allow you to return observables from the effect creator that aren't of type `Observable<Action>`. So, you're free to transform the effect stream to your needs without running into type conflicts.

The `NotificationService` used in this example could be a regular Angular service creating a [Material Snackbar](#), [Bootstrap Toast](#) or some other tasty pop-up. For this example it just opens a browser alert.



Non-Dispatching Effects

[Changes](#) | [Source Code](#) | [Live Demo](#)

13.8 Other Effect Sources

Now we know that you don't need to let your effects lead to actions, but there's more: you also don't need to base them on actions. So far all our effects have been observables piping off of the `Actions` observable, however, you can use whatever you want as an effect source. Having said this, I'll have to add that it makes sense to connect at least one end of any effect to the store. Therefore an effect should either produce actions or incorporate the `action$` stream at some point - otherwise the resulting observable is probably not store-related and should rather be placed in another service.

Here's an incomplete example that listens to a `media query` toggling `dark mode` based on browser preferences:

```
@Injectable()
export class ThemeEffects {
  darkMode$ = createEffect(() => {
    const darkModeMatcher = this.document.defaultView.matchMedia(
      "(prefers-color-scheme: dark)"
    );
    return fromEvent(darkModeMatcher, "change").pipe(
      map((event: MediaQueryListEvent) => event.matches),
      startWith(darkModeMatcher.matches),
      distinctUntilChanged(),
      map((darkMode) => ThemeActions.toggleDarkMode({ darkMode }))
    );
  });

  constructor(@Inject(DOCUMENT) private document: Document) {}
}
```



The `rxjs-web` library created by Jan-Niklas Wortmann provides reactive wrappers around native Web APIs.

`WebSockets` could pose another effect source. Imagine we'd be implementing a real-time chat application where new messages are pushed to clients. A corresponding effect could be based on an `RxJS WebSocketSubject` created with the `websocket()` factory function:

```
import { websocket } from "rxjs/webSocket";
import { Message } from "../message.state.ts";
import * as MessageActions from "../message.actions.ts";

@Injectable()
```



```

export class MessageEffects {
  private socket = WebSocket<Message>("ws://localhost:8081");

  receivedMessage$ = createEffect(() =>
    this.socket.pipe(map((message) => MessageActions.receive({ message })))
  );

  sendMessage$ = createEffect(
    () =>
      this.action$.pipe(
        ofType(MessageActions.send),
        tap((action) => this.socket.next(action.message))
      ),
    { dispatch: false }
  );

  constructor(private action$: Actions) {}
}

```

The `receivedMessage$` effect maps messages coming in from the socket to actions that are dispatched to the store. Meanwhile, the non-dispatching `sendMessage$` effect sends messages to the socket based on actions of type `MessageActions.send`.

Note that you're always free to start such effects off with actions and switch to other streams afterwards (e.g. with operators like `switchMap()` or `exhaustMap()`). Both examples could also benefit from extracting details into separate services for media matching and websocket management. Also, you'll want to think about error handling when implementing this in a real application.

As you can see, effects are a great way of connecting various external interactions to the store without making it considerably harder to reason about state transitions. By mapping all state-relevant events to actions, nothing changes for our state management. After all, we still get a new state for each dispatched action - we just have new sources where these actions can come from.

Other effects may incorporate - but are not limited to - the following sources:

- timers and intervals
- browser events
 - key events
 - mouse events
 - ... many more Web APIs
- router events
- web worker messages

- observables from services or libraries (e.g. [Angular Firebase SDK](#))



Don't use store selections as effect sources unless you really know what you're doing - even then you should probably only use those for non-dispatching effects. Otherwise your state transitions might cause effect loops that are hard to debug.



Recommended Reads

Deciding whether an effect is the right choice for your situation can be difficult. Here are two great reads for getting a better feel for this topic:

[Stop Using NgRx Effects for That](#) by Michael Pearson

[Start Using NgRx Effects for This](#) by Tim Deschryver

Chapter 14

Testing

Although functional programming concepts might seem tedious at some points, it's precisely because of them that testing of the resulting code is incredibly straight-forward. NgRx embraces functional programming while providing us with powerful testing utilities. So, we're in a good position to ensure that our issue tracker works as expected.

In this chapter I'll show you how to test all parts of your store as well as any code that interacts with it. We'll only be looking at unit / integration tests since end-to-end testing shouldn't care for the state management library you're using under the hood. I'll also say some words on which parts should receive the most testing attention and where you might let things slide a bit - just promise that the latter stays between you and me.

I'll be writing the tests with the [Jasmine](#) testing framework as it's the default for applications created through the Angular CLI - you're free to choose an alternative like [Jest](#) though. Overall, there's a bunch of things you could be doing different in this chapter, but for the sake of keeping this book actionable I'll be showing you some techniques that work well for NgRx.



Getting comfortable with Angular testing takes some time. If anything here looks confusing, circle back to the [official testing docs](#). They're a good place to start and nothing's ever wrong with freshening up the basics!

14.1 Testing Reducers

We've learned that reducers are pure functions. A pure function computes a return value only by reading from input values - no mutations, no side-effects. It only expresses itself through its output which will always be the same for the same inputs no matter how often we call it. Therefore we can validate a pure function by just validating its return value once for a certain [category of inputs](#). We don't need to mock dependencies or set up any kind of state around it. We just need to call the function and check that the return value looks like we expect it to.

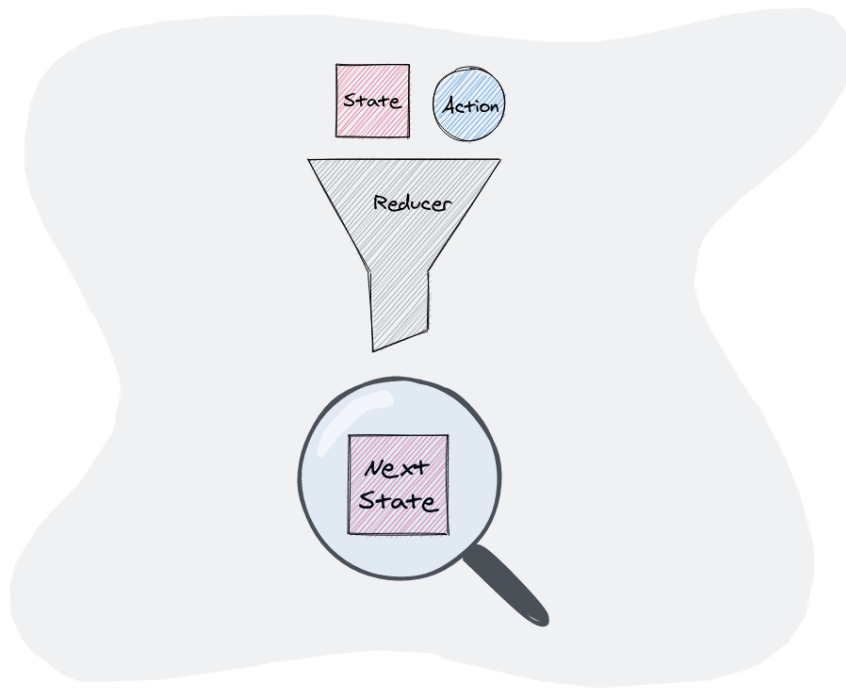


Figure 14.1: Verifying that a reducer works correctly can be done simply by asserting the next state it produces

There are two input parameters for a reducer function: the current state and an action. Both of these are plain objects that we can easily construct for our tests. Passing them to a reducer will give us the next state which we can then assert for correctness. That's it. Repeat this pattern for each combination of action and action metadata and you're done. You can follow a structure where you wrap all tests in an outer `describe()` group named after the reducer and subsequently group by type of action. Inside each `describe()` for an action type you can place multiple tests with `it()` for different action payloads and states. Feel free to introduce additional groups where necessary.

There are only two edge cases:

- initially NgRx will call the reducer with an undefined state to get the initial state
- all reducers receive every action, a reducer can therefore receive an unknown action

You'll want to start your reducer test suite with these edge cases:

```
// issue.reducer.spec.ts
import { issuesReducer, initialState } from "../issues.reducer";
import { INIT } from "@ngrx/store";

describe("Issue Reducer", () => {
  describe("init action", () => {
    it("should return the initial state", () => {
      const nextState = issuesReducer(undefined, { type: INIT });
      expect(nextState).toBe(initialState);
    });
  });
});
```

```

    });
  });

  describe("unknown action", () => {
    it("should return the previous state", () => {
      const nextState = issuesReducer(initialState, {} as any);
      expect(nextState).toBe(initialState);
    });
  });
});

```

For the initial action we just use the one that the framework will actually dispatch while expecting our initial state to be returned. For the second edge case we want the state to stay the same, so we'll just re-use the initial state and match it against the result. To ensure that the unknown action stays unknown, we provide an empty object without a type. This requires a type assertion to `any` because an empty object isn't really an action. Instead you might also choose a type called `"UNKNOWN"`, but watch out not to use it for a real action in the future.

These tests will be the same for any reducer, however, because the edge cases are not actually handled by us but rather by the `createReducer()` function you might decide to skip them - after all, they're already somewhat [tested in the NgRx suite](#).

Either way, all other tests can now be dedicated to our custom actions. Here's a test for resolving an issue:

```

// issue.reducer.spec.ts
import { IssueState } from "../issue.state";
import * as IssueActions from "../issue.actions";

describe("resolve", () => {
  it("should resolve issue", () => {
    const issueId = "issue-1";
    const state: IssueState = {
      ...initialState,
      loaded: true,
      entities: {
        [issueId]: {
          id: issueId,
          title: "Test Issue",
          description: "This is a test description",
          priority: "low",
          resolved: false,

```

```

    },
  },
};
const nextState = issueReducer(state, IssueActions.resolve({ issueId }));
expect(nextState.entities[issueId].resolved).toBeTruthy();
});
});

```



Try not to mimic the reducer functionality by deriving the target state from your inputs and checking that it equals the reducer's return value. Either perform specific assertions or manually craft the target state inside of an `toEqual()` assertion.

Note that I'm incorporating the initial state while constructing the input state. This way I don't have to come up with valid test doubles for parts of the state that aren't relevant to the current action. For more complex state shapes you can leverage test object factories.

Reducers should be rigorously tested because they contain a lot of logic. At the same time they're probably the parts that are the easiest to test - so, you can't make excuses, but also don't need to!



Testing Reducers

[Changes](#) | [Source Code](#) | [Live Demo](#)

14.2 Test Object Factories

Manually defining states over and over again per test case will bloat your test suites. Sharing test data between different suites could be one solution, but this also couples individual tests to each other. Changing the data for a single case means you might break another one.

A better solution involves test object factories. These utility classes can provide different instances of dummy data to each test case. Here's a test object factory for the issue state:

```

// issue.factory.spec.ts
import { Issue } from "../../models/issue";
import { initialState, Issues, IssueState } from "../issue.state";

export class IssueFactory {
  private lastId = 0;

  entity(issue?: Partial<Issue>): Issue {
    const id = this.lastId++;
    return {

```

```

    id: `issue-${id}`,
    title: `Title ${id}`,
    description: `Description ${id}`,
    priority: `medium`,
    resolved: false,
    ...issue,
  };
}

entities(...issues: Issue[]): Issues {
  const entities: Issues = {};
  issues.forEach((issue) => (entities[issue.id] = issue));
  return entities;
}

state(state: Partial<IssueState>): IssueState {
  return {
    ...initialState,
    ...state,
  };
}
}

```

Now, our test cases can be more concise while we can still override the test data, e.g. by passing a [partial](#) issue to `entity()`. It's advisable not to rely too much on pre-defined values from the factory - otherwise you're again coupled to shared data. Instead, you're better off asserting by reference or explicitly overriding values that are essential to a specific test case.

Let's refactor our reducer test for resolving an issue to use the factory:

```

// issue.reducer.spec.ts
import { IssueFactory } from "../issue.factory.spec";

describe("Issue Reducer", () => {
  let factory: IssueFactory;

  beforeEach(() => {
    factory = new IssueFactory();
  });
}

```

```
describe("resolve", () => {
  it("should resolve issue", () => {
    const issue = factory.entity();
    const state = factory.state({
      loaded: true,
      entities: factory.entities(issue),
    });
    const nextState = issueReducer(
      state,
      IssueActions.resolve({ issueId: issue.id })
    );
    expect(nextState.entities[issue.id].resolved).toBeTruthy();
  });
});
```

Here's another handy factory function that allows you to mock the root state based on the initial state and some overrides:

```
// index.spec.ts
import { INIT } from "@ngrx/store";
import { reducers, RootState } from ".";

export const mockState = (override: Partial<RootState> = {}): RootState => {
  const initialState = {};
  Object.entries(reducers).forEach(([key, reducer]) => {
    initialState[key] = reducer(undefined, { type: INIT });
  });
  return {
    ...initialState,
    ...override,
  } as RootState;
};
```

Defining this in a central place allows your individual test suites to be more decoupled from the root state. Now, when you add another reducer, you won't have to update several test suites when they relied on mocking the root state.



Test Object Factories

[Changes](#) | [Source Code](#) | [Live Demo](#)

14.3 Testing Action Creators

Technically, the tests from the previous chapter did not only test a reducer but also the corresponding action creators. We could separate these units by defining the resulting actions inline when invoking the reducer from a test. Then the action creators could receive their own test suite:

```
// issue.actions.spec.ts
import * as IssueActions from "../issue.actions";

describe("Issue Actions", () => {
  describe("resolve", () => {
    it("should return resolve action", () => {
      const issueId = "issue-1";
      const action = IssueActions.resolve({ issueId });
      expect(action.issueId).toBe(issueId);
    });
  });
});
```

However, since actions are merely data structures and action creators utility functions for creating them, there's not much to verify - the interesting aspects are already covered by the type system. Consequently, the resulting tests are incredibly trivial and don't really serve a purpose. Maintaining them will probably cost you more than they're worth. Therefore you may omit individual action creator tests and rather verify them in combination with reducers as we've done before.



Testing Action Creators

[Changes](#) | [Source Code](#) | [Live Demo](#)

14.4 Testing Selectors

Selectors are also pretty decoupled from the actual store. They're basically pure functions operating on one parameter, the root state. This makes them almost as smooth to test as reducers. We drop a pre-defined state in and assert the output. Here's how we could test the feature selector for the issue state:

```
// issue.selectors.spec.ts
import { RootState } from "../..";
import { selectFeature } from "../issue.selectors";
import { initialState } from "../issue.state";

describe("selectFeature", () => {
```

```

it("should select feature state", () => {
  const issueState = factory.state({
    entities: factory.entities(factory.entity(), factory.entity()),
  });
  const rootState = mockState({
    issue: issueState,
  });
  expect(selectFeature(rootState)).toEqual(issueState);
});
});

```

Though, I think it's obvious from this example that testing trivial selectors by themselves might not be worth your time. Therefore you probably just want to test more complex selectors in a corresponding test suite.

Usually, such selectors are created with `createSelector()` and although you can test them in the same way, you might get away with testing only a specific part: the projector. The third argument passed to `createSelector()` is actually the only one that really can (or should) contain logic. The selectors you pass before are either trivial or also defined with their own projector while everything in-between is again taken care of by the type system. That's why a `MemoizedSelector` exposes the projector as a property we can leverage for our test:

```

// issue.selectors.spec.ts
import { initialState, IssueState } from "../issue.state";
import { selectFiltered } from "../issue.selectors";

describe("selectFiltered", () => {
  it("should select all for empty filter", () => {
    const issues = [factory.entity(), factory.entity()];
    const filtered = selectFiltered.projector(issues, { text: "" });
    expect(filtered).toEqual(issues);
  });

  it("should filter issues for non-empty filter", () => {
    const first = factory.entity({
      title: "First",
      description: "This is a Test",
    });
    const second = factory.entity({
      title: "Second",

```

```

        description: "This is a Test",
    });
    let filtered = selectFiltered.projector([first, second], {
        text: "First",
    });
    expect(filtered).toEqual([first]);
    filtered = selectFiltered.projector([first, second], {
        text: "test",
    });
    expect(filtered).toEqual([first, second]);
    });
});

```

As you might've noticed, testing only the projector also spares us from mocking the whole root state. Instead, we can just provide mocks for the result of upstream selectors.

Parameterized selectors are no different. Their projector only has one additional parameter you need to supply:

```

// issue.selectors.spec.ts
import { selectOne } from "../issue.selectors";

describe("selectOne", () => {
    it("should select issue by id", () => {
        const first = factory.entity();
        const second = factory.entity();
        const entities = factory.entities(first, second);
        const selected = selectOne.projector(entities, first.id);
        expect(selected).toEqual(first);
    });
});

```

Testing pipeable selectors is different as we're basically testing RxJS operators or rather observables that incorporate these operators. Let's explore observable testing in a separate section - we'll be needing this for testing the remaining parts of our app and a pipeable selector is a perfect example for getting started.

14.5 Testing Observables

From my point of view there are two main approaches for testing observables:

1. [asynchronous testing](#) with `toArray()` operator

2. marble testing

Either one is fine and you can even mix them throughout your codebase depending on what seems easier for a specific test case. Claims that one or the other is better are purely based on opinion and definitely not [scientifically proven](#).

Async Testing

A Jasmine testing function (the one you pass to `it()`) can be made asynchronous by declaring a parameter, usually called `done`. This parameter will then contain a callback function which we can invoke to inform the framework that our test has completed.



Jasmine also offers the option to pass an [async function](#) or return a promise. However, observables cannot be [awaited](#) directly. You'd have to turn them into promises which might be a [bit tricky right now](#). Angular also provides a bunch of [async helpers](#), but let's keep it simple and solid.

When testing observables we can then call `done()` from inside of observer callbacks (`next`, `error`, `complete`). Many times though, we have observables that will invoke the `next` callback with multiple emissions. In those cases we don't want to complete the test case upon the first emission. Instead we'd like to assert all emissions that occur until the observable completes. That's where the `toArray()` operator comes into play. It'll collect all emissions and emit them as an array once the underlying observable completes.

Equipped with this knowledge we're able to define a test for our pipeable `selectAllLoaded()` selector:

```
// issue.selectors.spec.ts
import { of } from "rxjs";
import { toArray } from "rxjs/operators";
import { IssuesState } from "../issues.state";
import { selectLoadedIssues } from "../issues.selectors";

describe("Issue Selectors", () => {
  describe("selectAllLoaded", () => {
    it("should emit issues once loaded", (done) => {
      const unloadedState = mockState({
        issue: factory.state({ loaded: false }),
      });
      const issue = factory.entity();
      const loadedState = mockState({
```

```

    issue: factory.state({
      loaded: true,
      entities: factory.entities(issue),
    }),
  });
});
of(unloadedState, loadedState)
  .pipe(selectAllLoaded(), toArray())
  .subscribe((states) => {
    expect(states.length).toBe(1);
    expect(states[0]).toEqual([issue]);
    done();
  });
});
});
});

```

After defining two subsequent states and passing them to `of()` we get an observable that will mimic the state transition from unloaded to loaded before completing. Piping it through our `selectAllLoaded()` operator should skip the first state emission. We can verify this inside the `next` callback passed to `subscribe()` where we have access to all emissions thanks to `toArray()`. There we can assert our expectations in order to then finish the test by calling `done()`.

Note that you'll have to mock the root state (here represented by `unloadedState` and `loadedState`) for pipeable selectors that are directly applied to the store.



You don't need to unsubscribe in these kinds of tests. If the observable doesn't complete, `toArray()` won't emit and the test will timeout.



Technically, `of()` is a bit different from the actual store as it emits synchronously upon subscription. Therefore you might want to instead create the observable with `scheduled()` while passing `asyncScheduler` when you need to run code after the subscription but before the observer is invoked.

Marble Testing

I've mentioned [marble diagrams](#) before as a graphical representation for observables. Now, instead of having images with these graphical representations we could also *draw* them with plain ASCII characters. Here's an observable that emits two values `a` and `b` before completing as denoted by the pipe symbol at the end:

```
--a--b--|
```

Time elapses with each character when moving from left to right while a dash represents one *frame* of time where nothing happens. What we end up with is a domain-specific language (DSL) that can be used to precisely describe any observable you could imagine. This DSL is called marble syntax and RxJS provides [testing utilities](#) for matching expressions of this syntax against actual observables.

Transferring this concept to our test case we'd say that `a` and `b` in the previous marble diagram are two consecutive states - `a` could contain the `loaded` flag set to `false` whereas `b` would have it set to `true`. When we now apply the `selectAllLoaded()` operator, we'd expect to have a new observable that only emits the second state. In marble syntax this observable would look like this:

```
-----b--|
```

The emission of `a` is gone and has been replaced with a dash - so time passes while nothing emits past the operator at that point.

In order to progress observables correctly along a diagram, RxJS uses a virtual clock. This way our tests can remain synchronous and deterministic. It also allows us to test code in a fraction of the time that it would usually take for execution (e.g. when using the `delay()` operator). This virtual timezone is initialized through a test scheduler which accepts a function for matching two values with the testing framework of our choice - here it is for Jasmine:

```
// issue.selector.spec.ts
import { TestScheduler } from "rxjs/testing";

const scheduler = new TestScheduler((actual, expected) => {
  // Jasmine-specific equality check
  expect(actual).toEqual(expected);
});
```

When passing code to the scheduler's `run()` function, all observables will be executed synchronously in virtual time. We also get a set of helper functions for parsing [cold and hot observables](#) from marble strings as well as matching an existing observable against them. Now that we know how our observable should look before and after the `selectAllLoaded()` operator is applied, the only thing left to do is write that into a test case:

```
// issue.selector.spec.ts
it("should emit issues once loaded", () => {
  scheduler.run(({ hot, expectObservable }) => {
    const unloadedState = {
      issue: factory.state({ loaded: false }),
    };
    const issue = factory.entity();
```

```

const loadedState = {
  issue: factory.state({
    loaded: true,
    entities: factory.entities(issue),
  }),
};

const selection = hot("--a--b--|", {
  a: unloadedState,
  b: loadedState,
}).pipe(selectAllLoaded());
expectObservable(selection).toBe("-----b--|", { b: [issue] });
});
});

```

Note that the run helpers `hot()` and `cold()` accept representatives for each emitted marble (here `a` and `b`) in form of a dictionary. The same applies for the `toBe()` matcher on `expectObservable()`. Also, I'm using `hot()` for creating the underlying observable, because the `NgRx` store is using a `BehaviorSubject` - therefore a hot observable - under the hood.

Personally, I'd say it's easier to get started with async observable testing. Using an additional operator like `toArray()` might incorporate logic that isn't essential to the code we're testing, but so does any testing utility. It only gets tricky when you want to test time-related observable logic - in those cases you're probably better off with marble testing where time is virtualized. In the following sections I'll write async tests so that you're not forced to learn the marble DSL right now.



Recommended Video

[Unit Testing NgRx RxJS with Marbles](#) by Sam Brennan & Keith Stewart



Testing Selectors and Observables

[Changes](#) | [Source Code](#) | [Live Demo](#)

14.6 Testing Effects

Testing effects means testing observables. So, we can directly apply what we've learned. However, since effects are defined as injectable classes we might want to spin up an Angular `TestBed` first:

```

import { TestBed } from "@angular/core/testing";
import { provideMockActions } from "@ngrx/effects/testing";

describe("IssueEffects", () => {

```

```

let action$: Observable<Action>;
let effects: IssueEffects;
let serviceSpy: jasmine.SpyObj<IssueService>;
let factory: IssueFactory;

beforeEach(() => {
  factory = new IssueFactory();
  serviceSpy = jasmine.createSpyObj("IssueService", ["save"]);
  TestBed.configureTestingModule({
    providers: [
      IssueEffects,
      { provide: IssueService, useValue: serviceSpy },
      provideMockActions(() => actions$),
    ],
  });
  effects = TestBed.inject(IssueEffects);
});
});

```

Here I'm performing a setup step before each test where all dependencies of the effect class are mocked in a testing module. The `IssueService` is mocked with a `Jasmine spy` in order to exclude the HTTP layer from this test suite. Additionally, NgRx provides a testing utility function `provideMockActions()` which allows us to replace the action observable with an arbitrary observable. Under the hood, it's using the RxJS observable creator `defer()`. Therefore, the callback returning our `action$` replacement will only be called upon subscription to an individual effect inside a single test. This way each test can provide an observable mock that fits best.

Eventually, the actual instance of `IssueEffects` is retrieved from the testbed. Frankly though, it's not totally necessary to use a testbed for unit-testing services. You could also pass the mocks directly into the class constructor - the subsequent test cases will look exactly the same.

Now we can describe each effect observable through test cases for all possible scenarios. Let's start with the happy path for the issue submission:

```

describe("submit$", () => {
  it("should save issue and dispatch success", (done) => {
    const first = factory.entity();
    const second = factory.entity();
    action$ = of(
      IssueActions.submit({ issue: first }),
      IssueActions.submit({ issue: second })
    );
    effects.submit().subscribe({
      next: () => {
        expect(action$.value).toEqual(IssueActions.submit({ issue: first }));
        expect(action$.value).toEqual(IssueActions.submit({ issue: second }));
        done();
      },
    });
  });
});

```



```

    );
    serviceSpy.save.and.returnValue(of(first), of(second));
    effects.submit$.pipe(toArray()).subscribe((actions) => {
      expect(actions).toEqual([
        IssueActions.submitSuccess({ issue: first }),
        IssueActions.submitSuccess({ issue: second }),
      ]);
      done();
    });
  });
});

```

I'm faking two `submit` actions through an observable created with the RxJS `of()` function. Meanwhile, the issue service spy is prepared for two subsequent invocations where it'll respond with observables containing complete issues (normally, these would be coming from a server). Testing the effect then only requires us to assert that NgRx would receive the proper success actions. Note that I'm not checking for any invocations on the spy since that's rather an implementation detail. You might deviate from this approach in some situations - especially for non-dispatching effects.

Here's another test verifying that the effect dispatches a failure action when the service call fails while recovering afterwards:

```

it("should dispatch failure and recover on error", (done) => {
  const first = factory.entity();
  const second = factory.entity();
  action$ = of(
    IssueActions.submit({ issue: first }),
    IssueActions.submit({ issue: second })
  );
  serviceSpy.save.and.returnValue(
    throwError(new Error("Validation Error")),
    of(second)
  );
  effects.submit$.pipe(toArray()).subscribe((actions) => {
    expect(actions).toEqual([
      IssueActions.submitFailure(),
      IssueActions.submitSuccess({ issue: second }),
    ]);
    done();
  });
});

```

```
});
```

The failing HTTP request is mocked with the RxJS `throwError()` function and a plain error. In reality that'd probably be an `HttpErrorResponse`, so feel free to expand this setup if you're performing more specific error handling.

Limiting effects to orchestration especially pays off for testing where we're then able to mock implementation details like HTTP requests. Effects that rely on **other sources** can be tested in the same way. However, our implementations for toggling dark mode and communicating in real-time are using APIs like `fromEvent()` and `websocket()` that are difficult to mock. Therefore it would make sense to introduce designated services for media matching and web socket creation - the former is also available [from the Angular CDK](#). While you'll be having the same problems when testing these new services, you'd end up with an improved separation of concern and you'd only have a hard time mocking the mentioned APIs once.

If your effect is accessing the state, you also need to mock the NgRx store. We'll be looking into that in a separate section as it's also required for testing components and services which depend on the store.



Testing Effects

[Changes](#) | [Source Code](#) | [Live Demo](#)

14.7 Testing Components and Services

We've discussed testing each part of an NgRx store on its own. Now we'll take a look at units that depend on the store such as components, services or even directives. While nothing in general changes to the way you'd test these units, we've got two different options for dealing with the state management. We can either leave it aside by mocking the store or include it into the testing scope with integration tests.

Mocking the Store

Mocking the store in many different test suites is cumbersome. That's why NgRx provides another testing utility function `provideMockStore()` that can do the work for us. Among other things, it creates a `MockStore` which you'll import into your providers when setting up the testbed for a component test:

```
// issue-list.component.spec.ts
describe('IssueListComponent', () => {
  let component: IssueListComponent;
  let fixture: ComponentFixture<IssueListComponent>;
  let store: MockStore<RootState>;
  let factory: IssueFactory;
  let dispatchSpy: jasmine.Spy;
```

```

beforeEach(async () => {
  factory = new IssueFactory();
  await TestBed.configureTestingModule({
    declarations: [IssueListComponent],
    providers: [provideMockStore<RootState>({ initialState: mockState() })],
    imports: [RouterTestingModule],
  }).compileComponents();
  store = TestBed.inject(MockStore);
  dispatchSpy = spyOn(store, "dispatch");
  fixture = TestBed.createComponent(IssueListComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});
});

```

Optionally, you can pass a `MockStoreConfig` for providing an initial state. After retrieving the mocked store instance from the testbed, you may also set the state explicitly in each test suite in order to assert correct component rendering:

```

// issue-list.component.spec.ts
it("should display issues", () => {
  let elements = fixture.debugElement.queryAll(By.css("li"));
  expect(elements.length).toBe(0);
  const issues = [factory.entity(), factory.entity()];
  store.setState(
    mockState({
      issue: factory.state({
        loaded: true,
        entities: factory.entities(...issues),
      }),
    })
  );
  fixture.detectChanges();
  elements = fixture.debugElement.queryAll(By.css("li"));
  expect(elements.length).toBe(issues.length);
});

```

Additionally, you can override the values returned from selectors through the `overrideSelector()` method. This will give you a regular `MemoizedSelector` whose result you can also update later on via `setResult()` and a subsequent call to the store's `refreshState()` method:

```
// issue-list.component.spec.ts
it("should display issues (selector override)", () => {
  let elements = fixture.debugElement.queryAll(By.css("li"));
  const selector = store.overrideSelector(fromIssue.selectFiltered, []);
  fixture.detectChanges();
  expect(elements.length).toBe(0);
  const issues = [factory.entity(), factory.entity()];
  selector.setResult(issues);
  store.refreshState();
  fixture.detectChanges();
  elements = fixture.debugElement.queryAll(By.css("li"));
  expect(elements.length).toBe(issues.length);
});
```

The difference is that you don't have to elaborately setup the state beforehand while you isolate the selectors from the test. Otherwise you'd also be implicitly testing the selectors. On the other hand, you now have to know which selector is actually used by the component internally. Consequently, you're somewhat violating encapsulation principles.

Currently, there's no designated utility for verifying action dispatches with the mock store, however, you can still get that done yourself. A straight-forward approach may involve spying on the store's `dispatch()` method. Here's how this could look when mimicking a search input:

```
// issue-list.component.spec.ts
it("should dispatch search", () => {
  const dispatchSpy = spyOn(store, "dispatch");
  const text = "abc";
  const input = fixture.debugElement.query(By.css("input"));
  input.nativeElement.value = text;
  input.nativeElement.dispatchEvent(new Event("input"));
  fixture.detectChanges();
  expect(dispatchSpy).toHaveBeenCalledWith(IssueActions.search({ text }));
});
```

Alternatively, you could setup an async test based on the `scannedActions$` property of the mock store.

Integration Testing

Instead of mocking the store, we could also integrate the real thing into the testbed. This way we would assure that both units work well together (as it's the case in your actual application). For this purpose,

you'd import the regular `StoreModule` into your setup:

```
// issue-list.component.spec.ts
let component: IssueListComponent;
let fixture: ComponentFixture<IssueListComponent>;
let store: Store<RootState>;
let factory: IssueFactory;

beforeEach(async () => {
  factory = new IssueFactory();
  await TestBed.configureTestingModule({
    declarations: [IssueListComponent],
    imports: [RouterTestingModule, StoreModule.forRoot(reducers)],
  }).compileComponents();
  store = TestBed.inject(Store);
  fixture = TestBed.createComponent(IssueListComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});

it("should display issues", () => {
  let elements = fixture.debugElement.queryAll(By.css("li"));
  fixture.detectChanges();
  expect(elements.length).toBe(0);
  store.dispatch(IssueActions.submitSuccess({ issue: factory.entity() }));
  fixture.detectChanges();
  elements = fixture.debugElement.queryAll(By.css("li"));
  expect(elements.length).toBe(1);
});
```

I wouldn't advise you to write both unit and integration tests for the same component. You'll have twice the amount of code to maintain, while your confidence in the code probably won't increase that much.



Testing Components and Services

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 15

Performance

I've been touching on performance here and there already, but actually there aren't too many NgRx-specific points to consider. The concepts embedded into the framework will already provide you with a solid foundation for lightning fast applications - anything else mostly relates to Angular in general.

While I'm not a fan of dismissing performance optimizations as the root of all evil, make sure to invest your time where it matters. Grab low-hanging fruits and compare your solutions to others when something feels off, but don't engage in extensive optimization of non-critical parts. Instead, measure application performance, narrow problems down and weigh costs versus benefits. Let's recap some internal considerations before extending our view to how NgRx relates to Angular performance.

It might seem counter-intuitive to some, but functional programming is what makes NgRx fast. Pure functions and immutability are the foundation for memoization and simple equality checks. This way we can defer expensive computations and view updates to a minimum.

When memoizing selectors you're trading memory for computation resources. Caching the result of a selector function means we don't have to re-calculate it all the time. That's a powerful technique, especially for a single-threaded language like JavaScript. Therefore you should leverage selector memoization when you need to transform the state before making it available to consumers. Remember to reset selectors when their results are no longer required though.

Selection observables will only emit distinct states, therefore the view can only update when it really needs to. Underneath we're creating shallow state copies and that's also incredibly fast since we're reusing most of the state. Anything that drops out of the state will just be picked up by the garbage collection. At the same time, state normalization ensures that we're not keeping any redundancies in memory.

One more interesting detail: Angular event bindings (i.e. calling a class method from the template) will block the change detection until the handler code completes. With NgRx we're mostly just dispatching an action in order to then directly return the control flow to Angular. This way the view stays responsive while state transitions take place "in the background".

15.1 OnPush Change Detection

Angular has two [strategies for detecting changes](#) to the component state that need to be rendered into the view. There's a default one which triggers in a multitude of cases; in very simple terms, it basically runs all the time. That's not inherently bad and Angular, by default, delivers decent performance. Running the change detection less often still provides potential for improvement and that's why the OnPush strategy also exists. It'll trigger only in the following three cases:

1. when a component's inputs are reassigned
2. when events occur on a component or one of its children
3. when a component is *dirty*, meaning it's explicitly marked for change detection through a call to `markForCheck()` on a `ChangeDetectorRef` (like it's done inside of the `AsyncPipe`)

You can configure a component to use OnPush change detection by specifying the `changeDetection` metadata property in its decorator:

```
// issue-list.component.ts
import { Component, OnInit, ChangeDetectionStrategy } from "@angular/core";

@Component({
  selector: "app-issues",
  templateUrl: "./issues.component.html",
  styleUrls: ["./issues.component.scss"],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class IssueListComponent {}
```

Since all selections from the store come in the form of an observable, applying this performance optimization is pretty straightforward. Note, however, that you'll have to invoke the change detection manually when involving component state that isn't bound through the AsyncPipe.



Libraries like [@ngrx/component](#) and [@rx-angular/template](#) expand on this topic in order to optimize rendering even further.

15.2 Tracking List Elements

When rendering lists with the `NgFor` directive, Angular replaces DOM elements once the underlying object reference changes. In order to help the framework to optimize this process, you can provide a function for re-identifying elements with `trackBy` . While shallow copying won't change too many object references at the same time, you can still benefit from applying this optimization, especially for transient elements produced from selector projectors.

```
// issue-list.component.ts
trackByIssues(index: number, issue: Issue): string {
  return issue.id;
}

<!-- issue-list.component.ts -->
<li *ngFor="let issue of issues$ | async; trackBy: trackByIssues">...</li>
```

15.3 Efficient Handling of Remote Data

Try not to replicate your whole database on the client-side by fetching everything into the store. Keep the amount of cached data reasonable and try to clear parts that aren't used anymore.

You also don't have to store the complete resource returned from a server response. It's totally fine to omit properties that aren't required for your application. Destructuring and the object property shorthand syntax are your friends here.

Additionally, you can fill up your store incrementally with approaches like pagination. Unfortunately, NgRx doesn't provide pagination support out-of-the-box. However, maybe you're also better off keeping paginated collections out of the store depending on your use-case. This way you can easily clean up obsolete resources when a component is destroyed.

Lastly, remember to normalize entities by replacing nesting with key references before putting them into the state. Otherwise you're storing redundant information that you'll also have to update multiple times.



Recommended Read

Here's an article I wrote on how to manage pagination in a reusable way with the data source abstraction from Angular Material: [Angular Material Pagination Data Source](#)



Performance

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 16

Patterns

In this chapter we'll explore some patterns that relate to the use of NgRx in varying extends. I'm intentionally refraining from calling them "best practices" because I know that those two words are synonymous with "the only right way of doing things" to some people - of course that's not true for you! You're a well-reflected developer who weighs their options before choosing the right thing for themselves. I know that since you obviously made the incredibly wise decision to read this book.

16.1 Container and Presentational Components

There's this architectural concept in UI development where you separate components into the following two categories:

Presentational Components (aka dumb components) are purely concerned with looks. They don't maintain state, depend on services or perform computations. They just render their template inputs into the view and emit events based on user interaction. These constraints allow them to be reusable and easily tested. Consequently, you wouldn't inject the store into a presentational component. This way, it's loosely coupled to the application logic and you can easily change where the underlying state is managed. Usually, presentational components also only contain other presentational components.

Container Components (aka smart components) are organizational wrappers around presentational or other container components. They deal with services and business logic, may contain state, but don't really care that much for how things look. While dependency injection still allows for some loose coupling, container components are generally less-likely to be reused. They select state from the store in order to forward the data to presentational components via input bindings. Upon user interaction, the presentational components notify the containers via event emission which in turn will dispatch actions to the store.

This pattern doesn't specifically relate to NgRx or even Angular, but rather to component-based UI development in general. However, you're likely to encounter it in combination with state management

solutions. There's even an NgRx [schematic for generating a container component](#) that depends on the store:

```
ng generate @ngrx/schematics:container issues
```

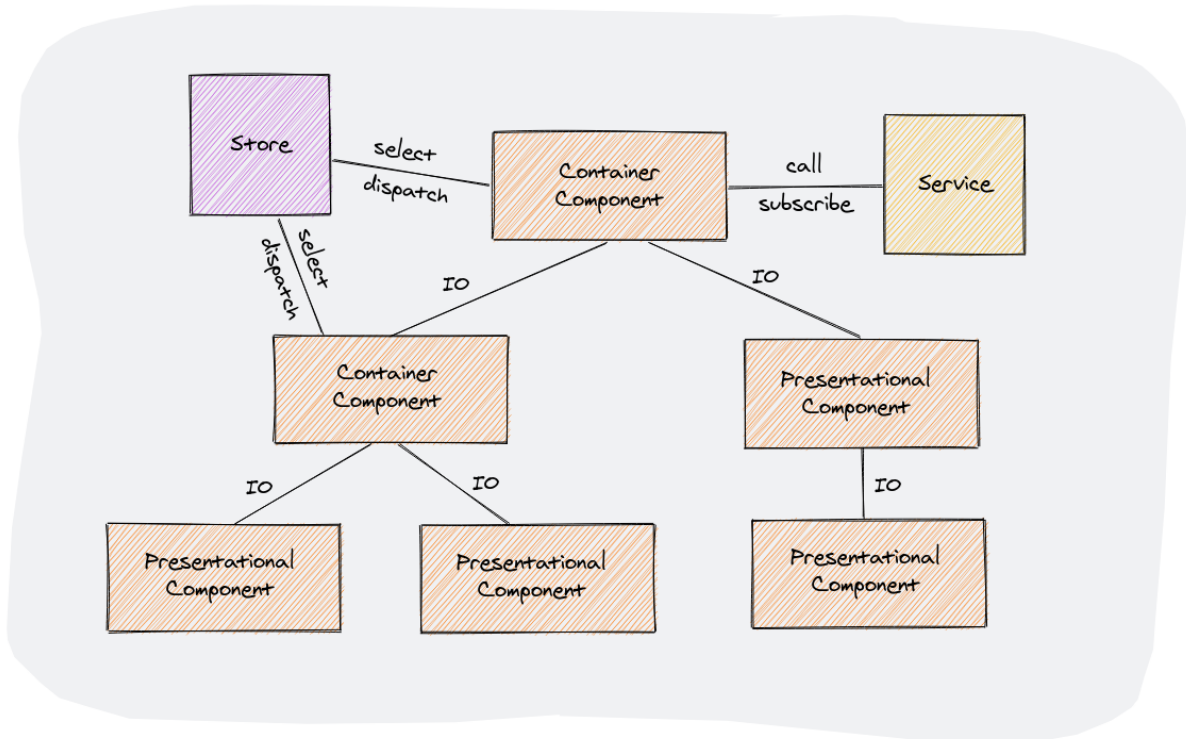


Figure 16.1: The separation between container and presentational components prevents logic and state management details from leaking across the view hierarchy.

In our case, the `IssuesComponent` could serve as a container component while `NewIssueComponent` and the `IssueListComponent` would be presentational ones:

```
// issues.component.ts
@Component({ ... })
export class IssuesComponent implements OnInit {
  issues$: Observable<Issue[]>;

  constructor(private store: Store) {}

  ngOnInit(): void {
    this.issues$ = this.store.pipe(fromIssue.selectAllLoaded());
  }

  onSearch(text: string): void {
    this.store.dispatch(IssueActions.search({ text }));
  }
}
```

```

onResolve(issue: Issue): void {
    this.store.dispatch(IssueActions.resolve({ issueId: issue.id }));
}

onSubmit(issue: Issue): void {
    this.store.dispatch(IssueActions.submit({ issue }));
}
}

```

```

<!-- issues.component.html -->
<app-new-issue (submitNew)="onSubmit($event)"></app-new-issue>
<app-issue-list
    *ngIf="issues$ | async as issues; else loading"
    [issues]="issues"
    (resolve)="onResolve($event)"
    (search)="onSearch($event)"
></app-issue-list>
<ng-template #loading>Loading...</ng-template>

```

```

// issue-list.component.ts
@Component({ ... })
export class IssueListComponent {
    @Input()
    issues: Issue[];

    @Output()
    search = new EventEmitter<string>();

    @Output()
    resolve = new EventEmitter<Issue>();

    constructor() {}

    onSearch(text: string): void {
        this.search.emit(text);
    }

    onResolve(issue: Issue): void {
        this.resolve.emit(issue);
    }
}

```

```

    }
}

@Component({ ... })
export class NewIssueComponent {
    form: FormGroup;

    @Output()
    submitNew = new EventEmitter<Issue>();

    constructor(private fb: FormBuilder) {
        this.form = this.fb.group({
            title: ['', Validators.required],
            description: ['', Validators.required],
            priority: ['low', Validators.required],
        });
    }

    onSubmit(): void {
        const issue = this.form.value;
        this.submitNew.emit(issue);
    }
}

```

I'm not a fan of overly strict boundaries although I like the notion of separating concerns. Introducing lots of components just because it's the cool thing to do often results in solid chaos. I've also found that prematurely optimizing components for reuse can make development unnecessarily complicated - many times you don't really know how components are going to be reused beforehand. There's no one-size-fits-all solution and you should evaluate your options per instance.

After all, we're basically in the same territory that **motivated** us to integrate NgRx in the first place. NgRx already provides various tools for separating concerns and moving them out of components. Dispatching actions to the store is comparable to the event binding of presentational components, providing you with the sought-after indirection. Moreover, state transitions are handled in reducers, selectors map the state to models for the view while effects are isolated to dedicated services.

Nevertheless, this pattern is something to keep in mind for when your components are getting too smart, e.g. because they're juggling numerous dependencies or managing huge chunks of state. Yet, breaking components apart is often only half of the solution. The other half might be rethinking your approach and possibly introducing new services, directives or pipes.



16.2 Facades

The [facade pattern](#) is not specific to NgRx, Angular or even web development. It's a general software-design pattern originating from object-oriented programming. The idea is this: when you're connecting a unit of code to one or more distinct units, instead of letting those interact directly you're defining a clear boundary: a facade. By masking how things work on the other side, you prevent different concerns from leaking into each other. Eventually, a facade is an abstraction that's meant to hide complexity.

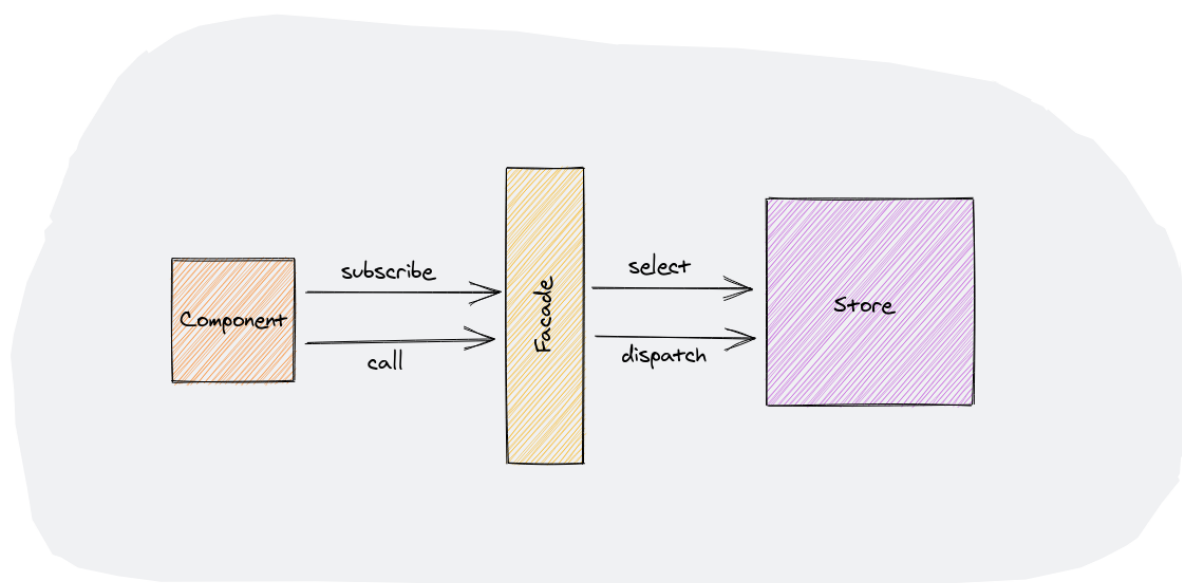


Figure 16.2: Facades are services sitting between components and the store. They expose selections as observables and map method calls to dispatched actions in order to abstract the state management

When applying the pattern to NgRx, components are interacting with the store only through a facade. The components don't directly depend on the store anymore. Meanwhile, the facade exposes observables based on selections from store and provides methods that dispatch actions. Here's what a facade for our issues could look like:

```
// issue.facade.ts
import { Injectable } from "@angular/core";
import { Store } from "@ngrx/store";
import { Issue } from "../../models/issue";
import * as IssueActions from "../issue.actions";
import * as fromIssue from "../issue.selectors";

@Injectable({ providedIn: "root" })
export class IssueFacade {
```

```

issues$ = this.store.pipe(fromIssue.selectAllLoaded());

constructor(private store: Store) {}

submit(issue: Issue): void {
  this.store.dispatch(IssueActions.submit({ issue }));
}

resolve(issueId: string): void {
  this.store.dispatch(IssueActions.resolve({ issueId }));
}

search(text: string): void {
  this.store.dispatch(IssueActions.search({ text }));
}
}

```

```

// issues.component.ts
@Component({ ... })
export class IssuesComponent implements OnInit {
  issues$: Observable<Issue[]>;

  constructor(private facade: IssueFacade) {}

  ngOnInit(): void {
    this.issues$ = this.facade.issues$;
  }

  onSearch(text: string): void {
    this.facade.search(text);
  }

  onResolve(issue: Issue): void {
    this.facade.resolve(issue.id);
  }

  onSubmit(issue: Issue): void {
    this.facade.submit(issue);
  }
}

```

```
}
```

You end up with a single service having a concise interface that abstracts the state management from the component's perspective. The notion is comparable to the separation of container and presentational components, but instead of outsourcing implementation details to a parent, you put them in a service. Note that you may place the facade into a feature sub-directory inside `store/`.

While this pattern sounds intriguing on paper, you might notice that we didn't win that much in terms of complexity. The relation between properties and selections as well as methods and actions is still 1-to-1, yet we almost doubled the amount of code. Some might say that the facade abstraction only lives up to its potential when selection logic gets more complicated. Most of the time though, you'll probably be fine with a tool that's already in your NgRx belt: selectors. That's where you can abstract from the shape of the actual state and produce additional presentational types if necessary. They're also fast, easy to test and composable beyond reducer boundaries.

There's another problem with facades: you're breaking the indirection between cause and effect by grouping actions and selections into the same service. You've put in all the effort to adopt event-sourcing in your application in order to go back to a service-oriented architecture at the last moment.

Dispatching actions from a service also promotes reusing the same action type for different underlying events. This makes it harder to trace back what's happening in your app. The proposed solution is adding a generic `dispatch()` method to the facade. A component would then still dispatch actions by itself while reading state from the facade. At that point you really have to ask yourself what you're trying to achieve with a leaky facade whose benefits are already questionable. Take a look at alternative state management options and see if something fits better in your case.

When masking NgRx with facades, you need to consider: what's the complexity you're trying to hide? The store has essentially two methods: one for reading data and another one for communicating events. Those two methods represent the basic idea of NgRx, Redux as well as CQRS and event-based programming in general. Meanwhile, a component isn't exposed to the store internals like reducers or effects. The store itself is already a facade and the public API are selectors and actions.

The facade pattern is definitely useful in software-design, Angular (e.g. the `HttpClient` is a facade) and maybe even in combination with NgRx. However, I doubt that it should be the default approach for dealing with the state management solution. A good use-case might be interop with other stateful services. In general though you should [avoid hasty abstractions](#).



Recommended Read

Here's the article that introduced facades to the NgRx world and a discussion of their usefulness:

[NgRx + Facades: Better State Management](#) by Thomas Burleson

[NgRx Facades: Pros and Cons](#) by Sam Julien



16.3 Re-Hydration

It's a common requirement: persisting NgRx state in order to load it back up when the application is restarted. This process of populating an empty object with domain data is called re-hydration. While it's common to persist the store data to the browser storage (mostly `localStorage`), you might also re-hydrate from a server-side cache.

There are some pitfalls to watch out for when applying this pattern. For one thing, you should take care not to store sensitive data in potentially insecure storages. Consider factors such as multiple users working on the same machine. Additionally, the state you're storing can become outdated. Consequently, you might incorporate techniques like validation and partial re-hydration. For this example we'll develop a simplified solution that saves the whole root state to the `localStorage`.

The popular approach for implementing re-hydration is based on meta-reducers. Such a re-hydration meta-reducer would have to do two things:

1. Persist the resulting state after each action has been processed by the actual reducer(s)
2. Provide persisted state upon initialization

Recalling our logging meta-reducer, persisting the result state is pretty straight-forward: we simply replace the log statements with browser API calls for serializing an object to JSON and putting it into the `localStorage`. Since we've taken care to keep our state serializable, this should work right-away. Additionally, we've already learned that NgRx calls reducers once with an undefined state and an `INIT` action to retrieve the initial state. This would be the place for parsing a potentially existing stored state and returning it instead of the underlying reducer's initial state. Here's how a corresponding meta-reducer might look:

```
// hydration.reducer.ts
import { ActionReducer, INIT } from "@ngrx/store";
import { RootState } from "..";

export const hydrationMetaReducer = (
  reducer: ActionReducer<RootState>
): ActionReducer<RootState> => {
  return (state, action) => {
    if (action.type === INIT) {
      const storageValue = localStorage.getItem("state");
      if (storageValue) {
        try {
```



```

        return JSON.parse(storageValue);
    } catch {
        localStorage.removeItem("state");
    }
}
}
const nextState = reducer(state, action);
localStorage.setItem("state", JSON.stringify(nextState));
return nextState;
};
};

```

Note that I'm wrapping the parsing into a try-catch block in order to recover when there's invalid data in the storage.

Since we're trying to re-hydrate the whole store, we'll have to register the meta-reducer at the root:

```

// index.ts
import { hydrationMetaReducer } from "../hydration.reducer";

export const metaReducers: MetaReducer[] = [hydrationMetaReducer];

// app.module.ts
@NgModule({
  imports: [
    StoreModule.forRoot(reducers, { metaReducers })
  ]
})

```



There's a well-known library called [ngrx-store-localstorage](#) you might utilize to sync your store to the [localStorage](#). It's leveraging the plain meta-reducer approach and offers some advantages over a custom implementation.



Re-Hydration: Meta-Reducer

Note that the re-hydrated state will be overridden by the data loaded from the in-memory database. Check the devtools to trace the re-hydration.

[Changes](#) | [Source Code](#) | [Live Demo](#)

Serialization, parsing and persistence are processes that clearly sound like side-effects to me. Just because [JSON.stringify\(\)](#), [JSON.parse\(\)](#) and the [localStorage](#) are synchronous APIs, doesn't mean

they're pure. Placing them into a reducer (or meta-reducer) is in itself a violation of NgRx principles. That doesn't mean it's not allowed to implement re-hydration this way, but there might be value in a different approach

Let's rethink re-hydration based on the NgRx building blocks. Interactions with browser APIs should go into effects. However, setting the state is not possible from an effect, so we'll still need a reducer, or rather a meta-reducer. It would only hydrate the state based on an action dispatched by an effect.

We'll start by defining an action that kicks-off the hydration as well as two additional actions that indicate whether a stored state could be retrieved:

```
// hydration.actions.ts
export const hydrate = createAction("[Hydration] Hydrate");

export const hydrateSuccess = createAction(
  "[Hydration] Hydrate Success",
  props<{ state: RootState }>()
);

export const hydrateFailure = createAction("[Hydration] Hydrate Failure");
```

Our meta-reducer can be incredibly simple and thus remain pure: it just has to replace the state based on `hydrateSuccess` actions. In any other case it'll execute the underlying reducer.

```
// hydration.reducer.ts
import { Action, ActionReducer } from "@ngrx/store";
import * as HydrationActions from "../hydration.actions";

function isHydrateSuccess(
  action: Action
): action is ReturnType<typeof HydrationActions.hydrateSuccess> {
  return action.type === HydrationActions.hydrateSuccess.type;
}

export const hydrationMetaReducer = (
  reducer: ActionReducer<unknown>
): ActionReducer<unknown> => {
  return (state, action) => {
    if (isHydrateSuccess(action)) {
      return action.state;
    } else {
      return reducer(state, action);
    }
  };
}
```

```

    }
  };
};

```

The `isHydrateSuccess()` helper function implements a [user-defined type guard](#). This way we can safely access the `state` payload property based on the action type of `hydrateSuccess`.

Now we can write the effect that dispatches `hydrateSuccess` and `hydrateFailure` actions based on whether there's a serialized state available from the `localStorage`. It'll be started by a `hydrate` action that we return through the `OnInitEffects` lifecycle. We'll then try to retrieve a value from the storage using the constant key `"state"` in order to parse it and return the corresponding hydration actions. If we're successful in parsing the state, it'll end up at our meta-reducer which puts it into the NgRx store.

```

// hydration.effects.ts
@Injectable()
export class HydrationEffects implements OnInitEffects {
  hydrate$ = createEffect(() =>
    this.action$.pipe(
      ofType(HydrationActions.hydrate),
      map(() => {
        const storageValue = localStorage.getItem("state");
        if (storageValue) {
          try {
            const state = JSON.parse(storageValue);
            return HydrationActions.hydrateSuccess({ state });
          } catch {
            localStorage.removeItem("state");
          }
        }
        return HydrationActions.hydrateFailure();
      })
    )
  );

  constructor(private action$: Actions, private store: Store<RootState>) {}

  ngrxOnInitEffects(): Action {
    return HydrationActions.hydrate();
  }
}

```

```
}
```

What's still missing is an effect that persists the current state to the `localStorage` in the first place. We'll base it off of the actions stream in order to wait for either an `hydrateSuccess` or `hydrateFailure`. This way we won't overwrite an existing state before the re-hydration is done. Then we stop looking at actions and instead subscribe to the store with the `switchMap()` operator. Slap a `distinctUntilChanged()` on top and you'll have a stream that emits the state any time it changes. Lastly, we'll mark the effect as non-dispatching and serialize the state to the `localStorage` inside of a `tap()` operator.

```
// hydration.effects.ts
serialize$ = createEffect(
  () =>
    this.action$.pipe(
      ofType(HydrationActions.hydrateSuccess, HydrationActions.hydrateFailure),
      switchMap(() => this.store),
      distinctUntilChanged(),
      tap((state) => localStorage.setItem("state", JSON.stringify(state)))
    ),
  { dispatch: false }
);
```

Don't forget to register the new effect class in your module declaration. Additionally, you'd be better off [injecting the `localStorage`](#) and/or outsourcing the whole parsing and persistence process into another service.

Apart from complying with the NgRx principles, this effect-based re-hydration implementation additionally allows us to

- leverage dependency injection and thus ease testing
- incorporate time-based filtering (e.g. RxJS operators like `auditTime()`)
- perform advanced error handling
- re-hydrate from asynchronous sources

The only disadvantage would be that we can't provide a stored state as a direct replacement for the initial state. If that's a requirement, you might try to [register reducers via dependency injection](#) in order to still get around an impure implementation.



Keep in mind that the shape of your application state can change between different releases. Meanwhile, your clients will have old versions in their storage - carelessly re-hydrating those will probably break your app. Possible solutions might involve tracking some kind of version or deep-checking state keys. Depending on the outcome you could discard or migrate serialized states.



Recommended Read

I've written up a similar approach for autosaving the state to a server:

[Angular Autosave for Forms, Services and NgRx](#)



Re-Hydration: Meta-Reducer + Effects

Note that the re-hydrated state will be overridden by the data loaded from the in-memory database. Check the devtools to trace the re-hydration.

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 17

Router Store

The router store connects the Angular router to NgRx. It's not a replacement for the Angular router but rather a bridge for working with the router state (path, query params, data) using the tools we know (reducers, selectors, effects). There are two things the router store does:

1. serialize the router state into the store
2. emit actions based on navigation to update this state

While it's not strictly necessary to leverage this module for interacting with the router, it has certain advantages.

Firstly, the store can remain the single source of truth. Take a look at the issue detail component that we implemented. It doesn't just depend on the store but also requires the activated route to retrieve the correct issue. With the router store the ID route param would be available from the state. This way we could write a selector that abstracts the connection between route and issue for the component.

Secondly, writing effects based on navigation gets easier. While it's perfectly fine to use [events emitted by the router](#) as an effect source, it might be more convenient to have a set of well-defined actions instead.

Additionally, the router store will propagate state changes back to the router and navigate accordingly. Time-travelling through application states with the dev tools will therefore work across routes. Moreover, you can influence the route serialization and thus easily store additional information that might seem a bit lost elsewhere.

17.1 Installation

Run one of the following commands to install the Node module either with npm or yarn:

```
npm install @ngrx/router-store
```

```
yarn add @ngrx/router-store
```

There's also a schematic which will directly register the library within your app module:

```
ng add @ngrx/router-store@latest
```

Otherwise you need to import the `StoreRouterConnectingModule` yourself using its `forRoot()` method. Make sure to import the module after the store and effects as well as the router itself.

```
// app.module.ts
...
import { StoreRouterConnectingModule, RouterState } from "@ngrx/router-store";

@NgModule({
  imports: [
    ...
    StoreRouterConnectingModule.forRoot({
      routerState: RouterState.Minimal
    }),
  ],
  ...
})
export class AppModule {}
```

You can pass a `StoreRouterConfig` while initializing the module allowing you to configure the following:

- `stateKey` : The property under which the router state is serialized into the store. Defaults to `router`
- `serializer` : A custom implementation of `RouterStateSerializer`
- `navigationTiming` : Whether navigation actions should be dispatched before or after guards and resolvers have run. By default, it's before.
- `routerState` : Member of `RouterState` enum for using either the `default` or a `minimal` serialization approach. Doesn't take effect when you're providing a custom serializer.



Only the minimal serialization is actually serializable to JSON. You should prefer it to conform with NgRx and Angular Ivy runtime checks. To the same end, you should take care to keep any route data (provided in the route config or by resolvers) serializable.

After that the module will happily dispatch corresponding actions during routing like `routerNavigatedAction` to indicate a successful navigation. Take a look at the docs for [all available actions and their possible orders](#).

However, actions that aren't processed by a reducer won't result in state transitions. So, in order to serialize the route information into the state we still need to register a reducer with the store. Luckily, we don't need to write one ourselves this time. NgRx already provides the `routerReducer` which we can add to our reducer mapping. At the same time we also need to extend the state typing with the `RouterReducerState` type. Note that the property key we're using corresponds to the `stateKey` from the router store configuration.

```
// index.ts
import { routerReducer, RouterReducerState } from "@ngrx/router-store";

export interface RootState {
  issue: IssueState;
  router: RouterReducerState;
}

export const reducers: ActionReducerMap<RootState> = {
  issue: issueReducer,
  router: routerReducer,
};
```

Open up the dev tools and you will see how router state transitions in sync with the routing. On top of that, travelling back and fourth through states will also navigate your application accordingly. This way you get time-travelling debugging across different routes.

17.2 Selecting Router State

Given you're using one of the existing route serializations, NgRx provides a bunch of selectors out of the box. You can get a grip on them by calling `getSelectors()` imported from `@ngrx/router-store` with a selector for the router state. In order to benefit from memoization you should only retrieve the selectors once though. You might want to introduce a file like `router.selectors.ts` in the store directory for this purpose. There you can re-export the selectors again with a destructuring assignment:

```
// router.selectors.ts
import { getSelectors } from "@ngrx/router-store";
import { RootState } from "..";

export const selectFeature = (state: RootState) => state.router;

export const {
  selectCurrentRoute,
  selectFragment,
  selectQueryParams,
```



```

    selectQueryParam,
    selectRouteParams,
    selectRouteParam,
    selectRouteData,
    selectUrl,
  } = getSelectors(selectFeature);

```

Now we can leverage these selectors to build another selector that retrieves a specific issue for our detail view. While `selectRouteParams` would definitely be an option, `selectRouteParam` might be even more convenient. That's actually not a selector by itself but a rather a factory function for one. When passing a route parameter key we'll receive a selector for this specific parameter only. Then we combine that with our existing `selectEntities` selector:

```

// issue.selectors.ts
import * as fromRouter from "../router/router.selectors";

export const selectActiveId = fromRouter.selectRouteParam("id");

export const selectActive = createSelector(
  selectEntities,
  selectActiveId,
  (entities, id) => entities[id]
);

```

The resulting selector can then be used in our detail component while allowing us to drop the dependency on `ActivatedRoute` :

```

// issue-detail.component.ts
import * as fromIssue from '../../store/issue/issue.selectors';

@Component({ ... })
export class IssueDetailComponent {
  issue$: Observable<Issue>;

  constructor(private store: Store) {
    this.issue$ = this.store.select(fromIssue.selectActive);
  }
}

```

The component now contains even less logic and thus becomes easier to test.

17.3 Reacting to Router Actions

The router store actions aren't strictly reserved for the router reducer. We can also listen to them in our reducers or effects like any other action.

Maybe we would like to display a loading indicator at the top of the page during navigation. For this we could flip a loading flag to `true` upon `routerRequestAction` and back to `false` once a navigation succeeds, fails or gets canceled as indicated by `routerNavigatedAction`, `routerErrorAction` and `routerCancelAction` respectively. Here's how that could look in an additional reducer:

```
// navigation.reducer.ts
import {
  routerCancelAction,
  routerErrorAction,
  routerNavigatedAction,
  routerRequestAction,
} from "@ngrx/router-store";
import { createReducer, on } from "@ngrx/store";
import { initialState } from "../navigation.state";

export const navigationReducer = createReducer(
  initialState,
  on(routerRequestAction, (state) => ({
    ...state,
    loading: true,
  })),
  on(routerNavigatedAction, routerErrorAction, routerCancelAction, (state) => ({
    ...state,
    loading: false,
  })),
);
```

As an example for a router-based effect one could write a non-dispatching effect that notifies some analytics tool about page views:

```
// router.effects.ts
import { routerNavigatedAction } from "@ngrx/router-store";

@Injectable()
export class RouterEffects {
  pageView$ = createEffect(
    () =>
```

```

    this.action$.pipe(
      ofType(routerNavigatedAction),
      tap((action) =>
        this.analytics.trackPageView(action.payload.routerState.url)
      )
    ),
    { dispatch: false }
  );

  constructor(private action$: Actions, private analytics: AnalyticService) {}
}

```

Some people also use router-based effects for pre-loading data similar to a [route data resolve](#). Personally, I'm not a big fan of that. Depending on the use-case, I'd rather chose an actual resolve or dispatch the loading action from the routed component.



Router Store

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 18

Entity Abstraction

Managing entities like our issues can get repetitive. You'll mostly be performing the same actions for any entity collection: creating, reading, updating, deleting. At the same time the state will probably look really similar between different entities. That seems like we'd be creating a good chunk of very similar code aka boilerplate - yuck!

The NgRx entity adapter provides a generic approach for minimizing boilerplate when working with entity collections. On a conceptional level it's basically the same approach that's used for plain collections aka arrays. Instead of creating a new array implementation for any element type, we're reusing the same implementation in a generic fashion, only specifying the element type per instance. Just like an array has methods for operating on the underlying elements, the entity adapter provides functionality for updating entity collections in the state. Meanwhile, the NgRx concepts remain untouched. The NgRx entity abstraction is rather elegant, but there's no magic - we'll just defer some operations to the adapter instead of implementing them ourselves.

18.1 Installation

Here are the commands to install the NgRx entity package:

```
npm install @ngrx/entity
```

```
yarn add @ngrx/entity
```

There's also a schematic, however, at this point it doesn't do anything other than installing the dependency:

```
ng add @ngrx/entity@latest
```

18.2 Entity State and Adapter

In order to manage a state slice in a generic fashion it needs to conform to the `EntityState` interface. It already includes the properties for handling an entity collection:

```
interface EntityState<T> {  
  ids: string[] | number[];  
  entities: Dictionary<T>;  
}
```

Therefore, we'll refactor the issue state to extend this base type:

```
// issue.state.ts  
import { EntityState } from "@ngrx/entity";  
  
export interface IssueState extends EntityState<Issue> {  
  filter: Filter;  
  loaded: boolean;  
  loading: boolean;  
}
```

At the same time, we'll ditch our custom `entities` property as it's already defined in the parent. The dictionary type will be inferred from the generic parameter on `EntityState<Issue>`. NgRx has a custom `Dictionary` type for this purpose. That's basically the same as our index type `Issues` but reusable for different types of values.

The `ids` property specified by `EntityState` is supposed to be an array containing the IDs of all entities, therefore the keys for the `entities` dictionary. While this might seem redundant at first, since those are already accessible via `Object.keys(entities)`, storing them also in an array serves a purpose that a dictionary can not: retaining order. This way, looking up a specific entity can remain fast, but we can also have our entities sorted.

Afterwards we can create the centerpiece of the entity abstraction: the `EntityAdapter`. There's a factory function `createEntityAdapter()` which accepts the entity type as a generic parameter:

```
// issue.state.ts  
import { EntityState, createEntityAdapter } from "@ngrx/entity";  
import { Issue } from "../../models/issue";  
  
export const adapter = createEntityAdapter<Issue>();
```

During creation you can also pass a function for selecting IDs that are available under a key other than `id` (e.g. `uuid`, `guid` or `ID`). You can also define a sort comparer which defaults to a sorting based on the ID. Passing `false` instead of a function for the `sortComparer` option will disable the sorting.

The adapter has a function for initializing the generic entity state where we can pass additional custom properties as a parameter:

```
// issue.state.ts
export const initialState: IssueState = adapter.getInitialState({
  filter: {
    text: "",
  },
  loaded: false,
  loading: false,
});
```

Now that we're using the adapter, we can replace any collection related selectors with generic once. Create and re-export them within a destructuring assignment of the adapter's `getSelectors()` method. You'll have to pass the feature selector in order to let the adapter know where the entity state is registered in the root state.:

```
// issue.selectors.ts
import { adapter } from "../issue.state.ts";

export const selectFeature = (state: RootState) => state.issues;

export const {
  selectIds,
  selectEntities,
  selectAll,
  selectTotal,
} = adapter.getSelectors(selectFeature);
```

The adapter doesn't provide us with actions. You'll still have to define those yourself. However, we also don't need any specific actions, instead we can keep using our existing ones. You might want to take a look at the available [adapter collection methods](#) to get some inspiration though. We'll apply those inside the reducer - that's where we reap the benefits of the adapter abstraction:

```
// issue.reducer.ts
import { createReducer, on } from "@ngrx/store";
import { resolve, submitSuccess } from "../issues.actions";
import { adapter, initialState } from "../issues.state";

export const issueReducer = createReducer(
  initialState,
  on(submitSuccess, (state, { issue }) => adapter.addOne(issue, state)),
```

```

on(resolve, (state, { id }) =>
  adapter.mapOne(
    {
      id,
      map: (issue) => ({
        ...issue,
        resolved: true,
      }),
    },
    state
  )
),
on(IssueActions.resolveFailure, (state, { issueId }) =>
  adapter.mapOne(
    {
      id: issueId,
      map: (issue) => ({
        ...issue,
        resolved: false,
      }),
    },
    state
  )
),
on(IssueActions.loadSuccess, (state, { issues }) => ({
  ...adapter.setAll(issues, state),
  loaded: true,
})))
);

```

Every adapter collection method allows us to perform a generic operation on our entities while returning a new state. Take a look at how we're now handling `submitSuccess` actions. Passing the incoming issue to `adapter.addOne()` will create a copy of the existing state where the issue is then placed into the `entities` dictionary - same functionality as before, but less code.

When transformations get more specific, we can utilize advanced adapter methods like `updateOne()` which expects an `Update<T>` next to the current state. The [corresponding type definitions](#) look like this:

```
export interface UpdateStr<T> {
  id: string;
  changes: Partial<T>;
}
export interface UpdateNum<T> {
  id: number;
  changes: Partial<T>;
}
export declare type Update<T> = UpdateStr<T> | UpdateNum<T>;
```

There are two definitions based on whether your ID is a number or strings. The adapter will try to find an entity with the passed ID and potentially update the corresponding entity with the partial representation specified under the `changes` property.

Note that you can still write state change functions that don't use the adapter. Moreover, you might also combine adapter operations with manual logic by either transforming the state before passing it to an adapter method or expanding on the result (see the handling of `loadSuccess` above). Keep in mind though that you still must not directly mutate the state at any point.

18.3 When to Use

The entity abstraction is definitely well thought out. While it allows us to handle entity collections in a generic way, it also doesn't restrict us. We can still add custom properties, write additional reducer logic or define actions the way we want. Yet, with solutions like `immer.js` in the picture, it doesn't drastically reduce the code we're writing. At the same time, even if it doesn't dictate fat actions, it somewhat promotes their use. You'll be tempted to define a single action creator for updating an entity which might then be used all over your application. This contradicts our rules for writing actions, specifically regarding categorization based on event source. However, the charm of NgRx entity is that it's non-obstructive - you can easily introduce it at a later point of time. Therefore you might start out managing entities manually and add the abstraction when it fits.



Entity Abstraction

[Changes](#) | [Source Code](#) | [Live Demo](#)

Chapter 19

Data Abstraction

[NgRx data](#) is a conceptual extension of the entity abstraction that takes care of even more than implementing state transitions and selectors for collections. Per entity collection it'll also give you a service for persisting to a remote server via HTTP, corresponding actions and effects as well as a facade for interacting with entities in an abstract manner. Additionally, it supports intricacies like transaction support or entity change tracking. Exploring all available extension points is beyond the scope of this book, however, I'd like to show you how we can re-create our existing functionality.

You can install the `@ngrx/data` library with one of the following commands:

```
npm install @ngrx/data
```

```
yarn add @ngrx/data
```

You might also get installation support with the `ng add` schematic:

```
ng add @ngrx/data@latest
```

This will already create a file `entity-metadata.ts` for you and register its `EntityDataModuleConfig` while importing the `EntityDataModule` into your app module - otherwise you'll have to do this yourself.

With NgRx Data, Instead of implementing each code unit for managing an entity collection ourselves, we'll provide the abstraction with metadata which it then uses to derive the necessary functionality under the hood. We don't need to work with an entity adapter anymore, actually, we can ditch most of our custom code for the part of the store that is concerned with issues like reducer, actions, selectors, effects - even the facade. As long as we're okay with how NgRx handles our entities, we just need to provide metadata for the issue entity. That's done by defining an `EntityMetadataMap` where the entity name in its singular form is used as the key. The corresponding value is a partial representation of `EntityMetadata` which offers certain options for configuring how the entity is handled - in our case we'll pass our custom filter function here:

```
// entity-metadata.ts
import { EntityMetadataMap, EntityDataModuleConfig } from "@ngrx/data";

const entityMetadata: EntityMetadataMap = {
  Issue: {
    filterFn: (issues, text) => {
      if (text) {
        const lowercased = text.toLowerCase();
        return issues.filter(
          ({ title, description }) =>
            title.toLowerCase().includes(lowercased) ||
            description.toLowerCase().includes(lowercased)
        );
      } else {
        return issues;
      }
    },
  },
};

export const entityConfig: EntityDataModuleConfig = {
  entityMetadata,
};

// app.module.ts
@NgModule({
  imports: [EntityDataModule.forRoot(entityConfig)],
})
export class AppModule {}
```

These few lines of code are all that's required for generating the following units for the issue entity collection at runtime:

- various actions for all entity operations (i.a. create, update, delete) based on the container type `EntityAction`
- an `EntityCollectionReducer` that manages an `EntityCollection` based on entity actions; the reducer and state type are respectively comparable to the collection methods and entity state type from the plain entity abstraction
- an `EntityCollectionDataService` for performing HTTP requests for persisting entity state to the server based on the entity name

- `EntityEffects` that connect entity actions to the `EntityCollectionDataService`
- an `EntityCollectionService`, the facade for interacting with the entity collection from components and services, plus corresponding `EntitySelectors`

We can adapt the default facade by extending `EntityCollectionServiceBase`. This way we can expose additional selections and methods:

```
// issue-collection.service.ts
import { Injectable } from "@angular/core";
import {
  EntityCollectionServiceBase,
  EntityCollectionServiceElementsFactory,
} from "@ngrx/data";
import { Observable } from "rxjs";
import { Issue } from "../models/issue";
import * as fromIssue from "../store/issue/issue.selectors";

@Injectable({ providedIn: "root" })
export class IssueCollectionService extends EntityCollectionServiceBase<Issue> {
  active$: Observable<Issue>;

  constructor(elementsFactory: EntityCollectionServiceElementsFactory) {
    super("Issue", elementsFactory);
    this.active$ = this.store.select(fromIssue.selectActive);
  }

  resolve(issue: Issue): Observable<Issue> {
    return this.update({ ...issue, resolved: true });
  }
}
```

Our container component can then use the collection service to interact with the entity state:

```
// issues.component.ts
import { IssueCollectionService } from '../services/issue-collection.service';

@Component({ ... })
export class IssuesComponent implements OnInit {
  issues$: Observable<Issue[]>;

  constructor(private issues: IssueCollectionService) {
```

```

    this.issues$ = this.issues.filteredEntities$;
  }

  ngOnInit(): void {
    this.issues.load();
  }

  onSearch(text: string): void {
    this.issues.setFilter(text);
  }

  onResolve(issue: Issue): void {
    this.issues.resolve(issue);
  }

  onSubmit(issue: Issue): void {
    this.issues.add(issue);
  }
}

```

Note that I cleared out all files in `store/issue/` except for `issue.selectors.ts`. Here we still need to define the `selectStats` and `selectActive` selectors. We can do so by retrieving the existing entity selectors through a `EntitySelectorsFactory` (similar to `getSelectors()` from the plain entity abstraction). This allows us to re-create our custom selectors:

```

// issue.selectors.ts
import { EntitySelectorsFactory } from "@ngrx/data";
import { createSelector } from "@ngrx/store";
import { Issue } from "../../models/issue";
import * as fromRouter from "../../router/router.selectors";

export const {
  selectEntities,
  selectEntityMap,
} = new EntitySelectorsFactory().create<Issue>("Issue");

export const selectStats = createSelector(
  selectEntities,
  (issues): IssueStats => {
    const resolved = issues.filter((issue) => issue.resolved);

```

```

    return {
      total: issues.length,
      resolved: resolved.length,
    };
  }
);

export const selectActiveId = fromRouter.selectRouteParam("id");

export const selectActive = createSelector(
  selectEntityMap,
  selectActiveId,
  (entities, id) => entities[id]
);

```

Now we can also update the detail view to leverage the collection service:

```

// issue-detail.component.ts
import { IssueCollectionService } from '../services/issue-collection.service';

@Component({ ... })
export class IssueDetailComponent {
  issue$: Observable<Issue>;

  constructor(private issues: IssueCollectionService) {
    this.issue$ = this.issues.active$;
  }
}

```

Meanwhile, you can still directly select the entity state from the store as we might do for keeping our stats in the application header.

As you can see, almost all functionality we need is already provided by NgRx Data and what's missing can be implemented through extension points (at least most of the time). This makes the abstraction extremely powerful since we get the same result with a fraction of the code that we had prior.

When you're already using NgRx this abstraction is a seamless addition for managing data. However, it also promotes action re-use, thus diminishes the value of a Redux-like architecture to some extent. Moreover, there is currently no support for things like pagination and the documentation is a bit sparse. Therefore, evaluate carefully whether you need these big guns for your project.



Data Abstraction

[Changes](#) | [Source Code](#) | [Live Demo](#)

Resources

Documentation

- [NgRx Documentation](#)
- [Angular Documentation](#)
- [TypeScript Documentation](#)
- [RxJS Documentation](#)
- [Redux Style Guide](#)

Community

- [NgRx Discord Server](#)
- [Angular Discord Server](#)
- [Angular Checklist](#)

Alternatives

- [NGXS](#)
- [Akita](#)
- [RxState](#)
- [NgRx ComponentStore](#)
- [XState](#)

Tools used for this Book

- [Pandoc](#)
- [Excalidraw](#)