Reactive State For Angular with NgRx

Learn to Build Reactive Angular Applications Using NgRx

AMIT GHARAT

bpb

Reactive State for Angular with NgRx

Learn to Build Reactive Angular Applications Using NgRx

Amit Gharat

www.bpbonline.com

FIRST EDITION 2021

Copyright © BPB Publications, India

ISBN: 978-93-89898-248

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

www.bpbonline.com

Dedicated to

My lovely wife, Monika, who has been motivating me throughout the process of writing

About the Author

Amit Gharat is a full-stack engineer and open source contributor. He has built many personal projects in AngularJS/Angular and has also made some of them, like directives, SPAs, and Chrome extensions, open source as well. He has an excessive urge to share his programming experiences in an easy-to-understand language through his blog When not programming, he enjoys reading science-fiction, watching movies, and playing games on his brand new Playstation 4 Pro. This is his third book.

About the Reviewers

Srashti Jain is a senior software developer and is currently working at Hvantage Technologies. She has five years of experience in the IT industry and has worked on .NET, SQL Server, Angular, Node, Vue.

Currently, she's more active towards frontend technology specifically Angular. Srashti is also active in the open-source community. She shares her knowledge via articles, talks, etc. She's also into organising technical meetups/events/workshops, and is the co-organiser at Indore Technical Community and All About Web.

Dan Wellman is an author, web developer, and video instructor based in the South Coast of the United Kingdom. He has worked on the front-end of the web for 15 years and now leads a small team of front-end developers at Unilink Software, providing software solutions for the Ministry of Justice and probationary services in the UK as well as overseas clients in both the public and private sectors. Outside of work, Dan writes books and creates video courses to help share his knowledge with others in the field. He is currently working with PluralSight to deliver quality Angular and TypeScript video courses. He is married and has four children.

With over seven years of experience in web development, **Pankaj Parkar** is a senior software engineer, an enthusiast in full stack web development. He currently works at Deskera as a senior software engineer. His skill in Angular and Web Application Development, and more has seen him become a Microsoft MVP and Google developer expert in Web technology. Pankaj loves to share his knowledge with the community, thus you will find him speaking at technical events and conferences. Pankaj is also the top contributor on stack overflow in India, and he ranks among the top 20 contributors in the world on Angular and AngularJS, etc. He is also a technical reviewer for BPB publications. He now focuses on channeling his knowledge into open source projects and sharing it with the community by mentoring, creating POCS, running workshops, writing blogs to help make the world a better and more developed place.

Acknowledgement

I want to thank the entire BPB team for giving me this opportunity to author this book and guiding me throughout the journey. I want to thank the editors for making the manuscripts look neat and perfect.

Preface

If you have worked in software engineering in recent years, especially in the frontend, you have probably been bombarded with a plethora of buzzwords relating to state management and reactive programming in Javascript. Reactive programming is taking the frontend world by storm. We wrote this book to help you understand the power and significance of reactive application state in Angular and develop the skills to put NgRx as well as other state management libraries out there to work. This book combines the redux philosophy with the Angular reactive nature, and you will learn how to apply the reactive state management to your projects. In a nutshell, we will make Angular application state R.O.C.K., i.e., Reactive, Obvious, Centralize, and Keen using pure functions and immutable data structures.

In the last decade, we have seen many developments in frontend, in the way we design and build Single Page Applications on top of them. In the early days of Angular, it was ideal for storing your application state in services and fetching them over the wire if needed. But as we go along writing more and more complex Angular applications, it becomes the need of the hour to abstract the store functionality out of the project. It consumes it as a standalone library so that duplicity can be avoided. Managing an ever-changing state is hard, and that's why redux was created in 2015 to manage the state of your application data while the React framework took care of the view layer of the application. But there was nothing in Angular yet. Later, what started as a weekend hack in the name of NgRx (Angular + RxJS) by Rob Wormald had become an instant hit and favored by the Angular core team as well. Now NgRx has become the de-facto state management library for Angular.

I had been following the Angular community very closely for many years and was somewhat aware of Redux/Flux/NgRx technologies sprouting within the space but had not tasted the water until I joined a new company, SS&C Technologies. I was fortunate to work with the legacy Angular application early on there. And I realized how easy it would become to learn a new technology when you get hands-on training by working on professionally written software and reading code written by passionate engineers. That's when I found the purpose of writing this book. So, please do not treat this book like any other book; treat it like a colleague you can learn from.

Downloading the code

bundle and coloured images:

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/cc1fe

Errata

Table of Contents

6. NgRx Entity State

CHAPTER 1

R.O.C.K. Solid Application State

You may have kept a phone aside or turned off a Smart TV to solely focus on reading this chapter on your kindle or laptop. That means you have successfully transitioned from one state of being to another. However, we often deal with such different states on a daily basis by performing various actions without paying much attention, via our subconscious mind. Sometimes we multitask as well; that is dealing with multiple states at the same time like singing while driving. And whether you can seamlessly do all the tasks at hand is all controlled by your mind. For that very reason, you end up asking your mind to recall where you had kept the car keys last time even though it's your hand that threw those keys somewhere in the living room. You purposefully do not ask your hand and instead scan your brain to recall the last snapshot of the car keys in the living room under the sofa because the brain is in charge of every action you perform. It tracks the outcomes of every action and uses them as the experience to perform the next action adroitly.

Now imagine, you are not a human being but a **Single Page Application (SPA)** written in JavaScript and running in a browser and a higher-level-energy in terms of a user is guiding you to change the world, probably the VM world. So throwing off the car keys is called the eyes who are letting the brain know about the action being performed is called reconciling the outcomes of the event i.e. where the car keys had landed in the living room is a job of and the brain where we store the memory of the event in order to recall later using **Selector** is called **Application State.**

In this chapter, you will learn to recognize your application state and make it R.O.C.K. solid using Redux principles. Using the simplest form of a shopping cart as an example, we will organize its application state and tame it in such a way that it will become scalable and maintainable as the application grows in the future.

Structure

After a bird's eye view of Application State and Redux jargons above, we will discuss the following topics in great detail in this chapter:

Application State in Angular

The Three Musketeers of Redux

Objectives

In this chapter, you will explore some fundamental ways of thinking about the application state using Redux principles. These principles will guide us through the rest of the book, where we dive into deep technical detail. Along the way, we will create our minimalistic version of NgRx to know the intricacies.

Application State in Angular

You may have already dealt with an application state in your personal or private projects in the form of scattered data lying across the application inconsistently. Instead of treating the application state as a passive variable that is manipulated by the application, it is ideal for an application to respond to state changes in one place by triggering state changes in another place. The interplay and collaboration between state, state changes, and the code that processes them is called **State** The state management libraries have come up with a way to group the distributed states into a centralized store under the umbrella term called Application State. So the application state is nothing but local static data, server responses, cached data, and more in the form of JSON or primitive types that drive the application behavior. The same application could then mutate the associated state when certain changes are propagated by the users of the application such as agreeing to the terms and conditions by clicking on a checkbox, completing registration by hitting the submit button, selecting a tab, pagination controls, and so on. With this notion, it is safe to assume that any kind of data backing the application can be a part of the global Application State.

Figure 1.1 shows the visualization of Redux architecture (Side-effects a.k.a. Redux Effect is excluded to simplify the understanding, which we'll cover in *Chapter 4: NgRx* A component dispatches the action and the reducer acts upon the said action to spit a brand new application state in order to store it in the centralized store. Then, the component subscribes to the state changes using the selector to update the parts of the view:

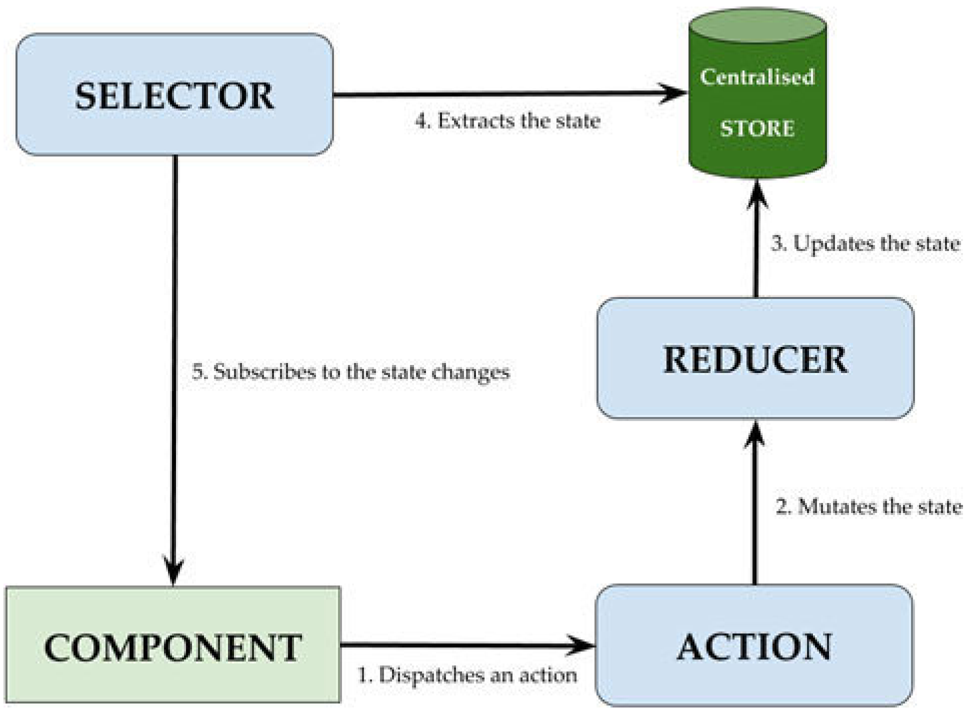


Figure 1.1: *Birds-eye view of Redux*

To understand the big picture and how these pieces come together, we'll create a very simple Angular application next. Let us look at a Dead Simple Shopping Cart Application in Angular in *Figure*

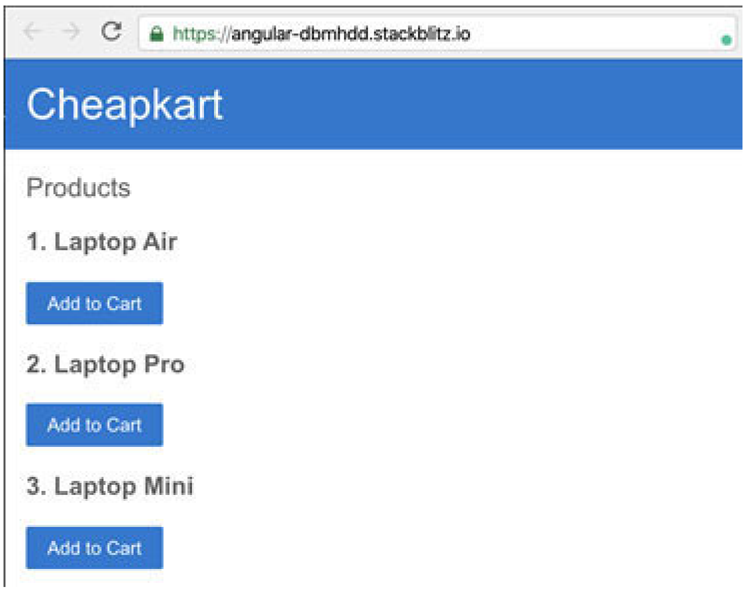


Figure 1.2: *Dead Simple Shopping Cart Application in Angular*

Here is the sample code for the above application. It is recommended to fork it and update the same as we move along each section.

In this application, we have a static list of products stored in a component class, which is then rendered in a component template via the ngFor directive. Additionally, we can add the individual product into a cart by clicking the Action button. If we try to apply the learned Redux terminology to this example then the products property is the application state that is driving our UI. We then mutate the individual product to mark the addedToCart property to true when the product is added to the cart. Below is the app/product-list/product-list.component.ts component for your perusal:

```
import {Component, OnInit} from '@angular/core';
```

```
@Component({
selector: ‘app-product-list’,
template: `
```

Products

*ngFor=”let product of products; let i = index”>

{{i + 1}}. {{product.name}}

(click)=”addToCart(i)”
[textContent]=”product.addedToCart ? ‘Added to Cart’ : ‘Add to Cart’”>
`

```
})
export class ProductListComponent implements OnInit {
public products;
ngOnInit() {
this.products = [
{name: ‘Laptop Air’, price: 799, addedToCart: false},
{name: ‘Laptop Pro’, price: 699, addedToCart: false},
{name: ‘Laptop Mini’, price: 299, addedToCart: false}
];
}

addToCart(i) {
this.products[i].addedToCart = true;
}
}
```

By the sheer look at the code, the Angular component is doing what it is supposed to do and our teeny-tiny Angular application looks just fine, probably the best-written code ever. However, it will blow up when it grows with lots of features and functionalities such as remove-from-cart button, bidding on price, product customizations, payment gateway integration, order tracking, etc. And the reason that the current approach is not scalable is because it has issues (which we will learn more about to fix them going forward), although not so evident at this point which is what brought us to Redux.

The Three Musketeers of Redux

Redux, in its simple language, is a predictable state container a.k.a. centralized store in the form of a JSON object for JavaScript applications that will allow us to manage our application state seamlessly with less boilerplate code. Building a large Angular application has its problems when it comes to managing the application state which is humongously big and complex. It is highly likely in such a complex application that one view updates a model which will affect another view, which, in turn, updates another model to change some other view. And after a while, we'll lose track of which view is updating which model and affecting which other view - it is a downward spiral. Redux makes it easy for applications to read and write states consistently and without any side-effects. It does so consistently because every model change must dispatch an action, that is, notify the centralized store first. Then, Redux takes care of making the necessary mutations to the store object based on what action is implied. With this approach, all the amends to the application state will be tracked by Redux and be done in the centralized store, making the application state consistent and easy to reason about.

Redux is not of the first kind but it imitated Elm's updater—a functional programming language and inspired by Flux—a unidirectional data flow library by Facebook. It also stands on the shoulders of giants, that is, Immutable - a JavaScript library implementing persistent data structures and RxJS - an observable library to deal with asynchronous events. In fact, Redux has three fundamental principles:

The state should be a single source of truth only

The state should be read-only

The state should be mutated using pure functions only

Let us learn through each principle and apply it to ProductListComponent to make it maintainable. Along the way, we'll create our implementation of Redux. Let's call it

Single source of truth

In the real world as well, if you ask different people about a particular incident, you will hear varying plotlines of the same story and none of the plots you heard will be true to the fact. The only way to understand the real story of the incident is to go to the one - a single source of truth. Redux has identified this early on and given us a centralized Redux Store to tackle the inconsistencies or fabrication of the application state. But how?

The state of your whole application should be stored in the form of **POJOs (Plain Old JavaScript Objects)** - a single centralized store. The intuition behind POJOs is that a POJO is an object that only contains data as opposed to JavaScript methods. With a single state tree, you can easily debug or inspect an application. It enables us to persist state between page transitions, avoiding redundant and repetitive XHR calls. It also makes it easy to serialize and hydrate server states, that is, store the data flushed from a web server to web browser into the state tree for faster client-side applications. And hence, the Redux paradigm recommends a single Store per application.

Redux selectors

On a funny note, the browser Window object used to be the centralized store back in the old days of web development. But it was global and the application's performance due to bad **GC (Garbage Collection)** and other major issues. Jokes apart, Redux introduced a robust and extensible API to interact with the store using Redux Selector. Let us understand its pseudo API:

```
type Store {
getState() => State
}
```

This is a store interface that mentions that we can create a JavaScript object of type Store and define the getState() method on the object to get the default/current application state back. So, let’s create an Injectable Angular service that mimics the aforementioned store interface in app/store.ts:

```
import {Injectable} from ‘@angular/core’;

@Injectable({providedIn: ‘root’})
export class MyStore {
private initialState = [
{name: ‘Laptop Air’, price: 799, addedToCart: false},
{name: ‘Laptop Pro’, price: 699, addedToCart: false},
{name: ‘Laptop Mini’, price: 299, addedToCart: false}
];

private currentState = this.initialState;

public getState() {
return this.currentState;
}
}
```

The providedIn: ‘root’ makes sure that service is provided in the root injector turning it into a singleton service i.e. a single instance will be used across the application. Remove it in case you need multiple instances of the same service to be used in different components or services.

All we have done here is moved the existing list of products from ProductListComponent to MyStore as the initial state of the application. Then, we defined the same as the current state of the application since it has not changed yet and finally, allowed to request the same via getState() for whoever needs it. Notice that the MyStore class has been marked as the Angular Injectable service so that we can inject the same into ProductListComponent next using constructor shorthand syntax in Typescript.

Because of this change, the products list is no longer tightly coupled with a single component and can be read/written by other components and services we may create in the future if any. Nonetheless, we have to import MyStore in

```
import {Component, OnInit} from ‘@angular/core’;
import {MyStore} from ‘../store’;

@Component({
selector: ‘app-product-list’,
template: `
```

Products

*ngFor=”let product of products; let i = index”>

{{i + 1}}. {{product.name}}

(click)=”addToCart(i)”
[textContent]=”product.addedToCart ? ‘Added to Cart’ : ‘Add to Cart’”>

`

})
export class ProductListComponent implements OnInit {
public products;

constructor(private readonly store: MyStore) {}

ngOnInit() {
this.products = this.store.getState();
}

addToCart(i) {
this.products[i].addedToCart = true;
}
}

Angular encourages us to use dependency injection techniques to instantiate dependencies in a class. Using constructor shorthand syntax above, we initialized class properties from constructor arguments. This reduces the unnecessary boilerplate code and

makes it easy to mock these dependencies while testing. Without dependency injection, the instantiation of the MyStore class would look like this:

```
export class ProductListComponent implements OnInit {
private readonly store: MyStore;

constructor() {
this.store = new MyStore();
}
}
```

Even though we have improved the existing code a bit by moving the application state into a centralized store, we still mutate the products property of ProductListComponent when the product is added to the cart. However, it also reflects the changes in the store without our notice because the addToCart method directly modifies the existing object reference When the application grows, it will be difficult to reason about such mutations, which urges the need for the second principle of Redux.

Read-only application state

Often, when developers hear that the state should be read-only, they bring up confused faces in complete disguise: wait, if it is read-only then how do I update the state? In contrast, the read-only state means an immutable object whose state cannot be modified after it is created. Instead, a new snapshot of the existing state will be created with all of the changes requested; so the original state has not changed i.e. read-only or immutable but the new state has been created.

Redux actions

Even though Immutability is the key to improve code readability, run-time efficiency, and security compared to mutable objects, Redux's second principle does not recommend Angular views/components/services to directly write to the state (as we were doing before). Instead, they should express intent to transform the state using Redux action. These actions are nothing but plain objects can be logged, serialized, stored, and even replayed for debugging or testing purposes. Let us understand its pseudo API:

```
const ACTION_CREATOR = () => {
type:
payload:
};
```

The aforementioned intent to transform the state can be done by defining an action using action creators. The ACTION_CREATOR is a function to create custom actions such as ADD_TO_CART that accepts the necessary payload such as an index of the product to add to the cart. The ACTION_CREATOR returns an object containing the type of action and the necessary payload altogether for the store to act upon. So, update the addToCart method in ProductListComponent to define the action The ADD_TO_CART action creator takes an index of the product and returns the action. So, update the addToCart method in as follows:

```
addToCart(i) {
const ADD_TO_CART = (index) => {
return {

type: 'ADD_TO_CART',
payload: {index}
}
};
this.store.dispatch(ADD_TO_CART(i));
// original logic commented out
// this.products[i].addedToCart = true;
}
```

Note that the dispatch method has not been defined in MyStore yet so you might get a JavaScript exception that we will fix in the next section.

Do not get confused with payload: {index} since it uses ES6/ES2015 object property value shorthand syntax. This is encouraged to use for readability purposes when the property name and a variable name used for the property value are the same, that is, index. Without the shorthand syntax, we would have written it as follows:

```
payload: {"index": index}
```

Redux dispatcher

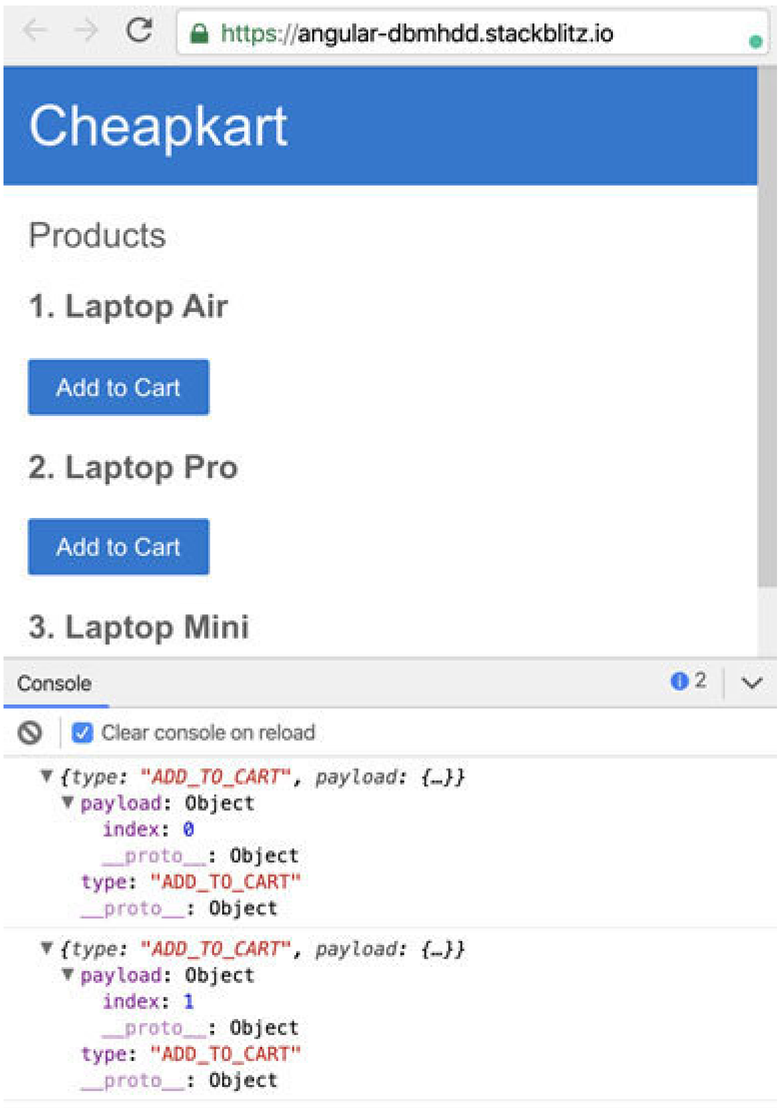
Remember, the only source of information for the store is the action being dispatched. That's why we had sent the ADD_TO_CART action using this.store.dispatch method above is called Redux dispatcher. Here is the pseudo API for it:

```
type Store {
getState() => State,
dispatch(action) => ...
}
```

Let us introduce a dispatch function in the For now, it will simply log the payload to the console for debugging purposes:

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

@Injectable({providedIn: 'root'})
export class MyStore {
private initialState = [
{name: 'Laptop Air', price: 799, addedToCart: false},
{name: 'Laptop Pro', price: 699, addedToCart: false},
{name: 'Laptop Mini', price: 299, addedToCart: false}
];

private currentState = this.initialState;
public getState() {
return this.currentState;
}

public dispatch(action) {
console.log(action);
}
}
```

Now if you click the **Add to Cart** button on the UI for the first two products, you will see the payload logged to the console with the correct index of the product to be added to the cart. *Figure 1.3* shows just that:



```
▼{type: "ADD_TO_CART", payload: {…}}
  ▼payload: Object
     index: 0
     __proto__: Object
   type: "ADD_TO_CART"
   __proto__: Object
▼{type: "ADD_TO_CART", payload: {…}}
  ▼payload: Object
     index: 1
     __proto__: Object
   type: "ADD_TO_CART"
   __proto__: Object
```

Figure 1.3: *Redux Action getting logged to the console*

Although this works, the **Add to Cart** button text does not update to **Added to Cart** as before. And that is because we have not made the amendments to the store yet. This brought us to the third and the last principle of Redux.

Mutate State using Pure functions

In a nutshell, a Pure function is a standard JavaScript function that accepts an input parameter and returns an output i.e. return value without modifying any variable outside its scope. Although the section heading emphasizes mutating the state, Pure functions never mutate the original state; instead, they create a new copy of the original state with all the necessary mutations. And then the original state can be swapped with the newly created state safely. You may think that creating a new object from the old one may not be memory efficient? It is not. But certain optimizations can be done to avoid this fate and one of them is the persistent data structure for immutable values. This data structure always preserves the previous version of the state when it is modified and hence commonly used in functional programming to enforce immutability.

The persistent data structure uses various optimizations in terms of Hash maps tries and Vector tries popularized by Clojure and Scala is a subject of another book. If you are interested to know more about these implementations in JavaScript, it is recommended to read about Immutable.js

Object.assign

On that front, ES6/ES2015 introduced some handy improvements in JavaScript natively, and one of them was the Object.assign method. It copies the values from one or more sources objects to a target object. The target object is the first parameter and is also used as the return value. It has been widely used for merging objects with shallow copying them - meaning updating below ngrx.name later on would not mutate

```
Object.assign(target, ...sources)
// For example,
const redux = {name: “Redux”};
const ngrx = Object.assign({}, redux, {name: “NgRx”});
ngrx.name = “RxJS”;
console.log(redux, ngrx); // {name: “Redux”} {name: “RxJS”}
```

Redux Reducer

The ideal mutation of the state using a pure function is the job of Redux Reducer. The reducer function is again a standard JavaScript function that takes the current state and action as arguments and then returns a new state with necessary amendments. So let us write a reducer in app/store.ts for ADD_TO_CART action:

```
public reducer(state, action) {
switch (action.type) {
case 'ADD_TO_CART':
return state.map((product, index) => {
if (index === action.payload.index) {
return Object.assign({}, product, {addedToCart: true});
}
return product;
});
}
}
```

Ideally, the switch statement was not necessary here to handle the single action but still added to show you that we can handle different actions and their business logic in the same reducer function. Here, we are simply looping over the existing state to check which particular product we want to mutate based on the index provided in the action's payload. When the product matches, we create a new empty object to return. However, before returning we first stuff it with the existing product object and also update addedToCart property from false to true. Otherwise, we return the existing product as is.

Now that we have our reducer function ready, it is time to update the dispatcher function we wrote in the previous section, which was merely logging action to the console. So we must call the aforementioned reducer function from within the dispatch function with the current state of the application and the action being dispatched in

```
public dispatch(action) {
this.currentState = this.reducer(this.currentState, action);
console.log(this.currentState);
}
```

Go ahead and click the **Add to Cart** button for any product. Did you notice anything unusual? You may be wondering why the button text did not change even though the updated current state is logged to the console. The problem is that the consumer, of our application state, is not notified about the change.

Reactive state

That’s when the Reactive State is born. The reactive state uses the observable design pattern in which the object a.k.a. the subject maintains a list of its dependents called observers and automatically notifies them of any changes. Looks like we do not have to reinvent the wheel since RxJS has already taken up the baton to manage reactive state using observables so that all the consumers of the state are notified as soon as the observable state changes.

For that, one important change we have to make is to introduce a new variable called state$ which is an Observable, hence it is suffixed with the dollar ($) sign to denote the same. There are plenty of operators in RxJS to create an observable stream from primitive data types. But the BehaviorSubject operator is specifically useful in our case because it stores the latest value emitted to its consumers and immediately receive the current value whenever a new observer subscribes to it. Also, we can manually emit the new value in the dispatch function whenever any product is being added to the cart.

Let us import BehaviorSubject from RxJS in app/store.ts (some code from the below snippet has been removed for brevity):

```
import {Injectable} from ‘@angular/core’;
import {BehaviorSubject} from ‘rxjs’;
```

```
@Injectable({providedIn: ‘root’})
export class MyStore {
...
...

private state$;

public getState() {

this.state$ = new BehaviorSubject(this.currentState);
return this.state$;
}

public dispatch(action) {
this.currentState = this.reducer(this.currentState, action);
this.state$.next(this.currentState);
}


...
...
}
```

We created an observable state$ from currentState using BehaviorSubject and returned it for ProductListComponent to consume in its template via async pipe. The AsyncPipe is a native angular pipe (apart from etc.) that accepts a Promise or Observable as input and subscribes to the input automatically, eventually returning the emitted value. It asynchronously subscribes to the observable products in the template, which is why we had to use the BehaviorSubject RxJS operator instead of the commonly used etc. operators. We have also renamed store to store$ wherever used for being the observable store now in

```
@Component({
selector: ‘app-product-list’,
template: `
```

Products

***ngFor=”let product of products | async; let i = index”>**

{{i + 1}}. {{product.name}}

(click)=”addToCart(i)”

[textContent]=”product.addedToCart ? ‘Added to Cart’ : ‘Add to Cart’”>

`

})
export class ProductListComponent implements OnInit {
public products;
`

constructor(private readonly store$: MyStore) {}

ngOnInit() {
this.products = this.store$.getState();
}

addToCart(i) {
const ADD_TO_CART = (index) => {
return {
type: ‘ADD_TO_CART’,
payload: {index}
}
};
this.store$.dispatch(ADD_TO_CART(i));
}
}

Look back and compare this version of the application with the one we started with, and you will notice that the application state, no matter how big it grows, will surely ROCK, that is, Reactive using RxJS, Obvious to reason about, Centralized for consistency, and Keen with insights.

Conclusion

We have learned how the application state, if not organized well, can make it unmaintainable and hard to reason about. We then learned how Redux addresses the said issues and proposes a clever way to read/write the application state predictably. We explored every Redux principle in detail and used that learning to create our own Redux implementation - if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is Reducks.

Jokes apart, Redux Selector helps us read the application state in a robust way. Redux Action is the recommended way to mutate the application state, whereas Redux Dispatcher is the harbinger of an intent to mutate the application state. We also learned that Pure functions enforce immutability for consistent and predictable state changes. All these learnings will help you to understand NgRx better. I believe that you must have gained confidence in understanding Redux and its intricacies and are even more excited to learn NgRx.

In the next chapter, we'll build a brand new Angular application using Angular Material and set up NgRx and Redux dev tools that will pave the way for further chapters to come.

Questions

What is Redux and what problem does it solve?

What are the three principles of Redux?

How does Redux maintain a pure application state?

How does Redux manage to mutate the application state predictably?

What's the benefit of immutability?

What is a Reactive state?

CHAPTER 2

Setting up NgRx in Angular

We were taught how the Angular application can quickly become a mess when it grew without a centralized store. For that, we made ourselves accustomed to the Redux principles to manage a single object tree of state per application which will be robust, centralized, and consistent across the application. We narrowed down the Redux architecture for our understanding and wrote our implementation under the name, Reducks. Even though we strictly followed Redux principles to create Reducks in the previous chapter, but, unbeknownst, it was quite inclined towards NgRx for being Reactive. However, it was the simplest Redux-like store we could create ignoring the complexities in building the full-fledged Redux alternative on our own. Instead, it is better not to reinvent the wheel and reuse what is out there which is widely used in the Angular community. Hence, as you have already guessed, we will be learning NgRx going forward.

Angular relies on RxJS to build reactive applications. In spite of that, the Angular application state was not reactive. So, in simple terms, NgRx is a framework for building a reactive state in Angular, making the entire Angular Stack reactive. NgRx not only provides state management but also gives provision to handle side-effects, entity collection, router bindings, code generation via CLI tool which we will learn throughout the book.

Structure

To get the hang of all our learnings from the previous chapter, we will need a sample Angular application that we can use to integrate NgRx within this as well as subsequent chapters. In this chapter, however, we will discuss the following topics in great detail:

Creating Angular Application: Bookmarker

NgRx Debugging: Redux Devtool

Objectives

We will create a sample angular application named Bookmarker - a bookmark manager - in Angular which will pave the way for future chapters. We'll also learn how to set up Redux Devtools to debug the NgRx store and its actions.

Sample Angular Application: Bookmarker

Bookmarker is a bookmark manager. It will allow us to save new bookmarks, edit existing bookmarks, and filter them using fuzzy searching. We'll use Angular CLI to scaffold a brand new angular application with various components, services, utility classes, routes, guards, pipes, and even in-memory data-store so that we can retain saved bookmarks and perform CRUD operation on them. In the end, we'll install NgRx Schematics - useful to create actions/reducers/selectors over command-line and even play with Redux devtools - useful for debugging NgRx data flow. Let us begin then.

Prerequisite

First of all, we need to install NodeJS version 10.13 or later from https://nodejs.org/en/download and post-installation run node --version command in a terminal to verify if it's installed correctly. Along with NodeJS, the NPM, a package manager for NodeJS, must have been installed too whose version can be checked with npm --version command.

Using NPM, we will install Angular CLI globally (with a -g flag) using the following command. You can check the installed version with ng

```
npm install -g @angular/cli@9
```

By the time of writing this book, I was using Node 10.17.0, NPM 6.13.4, and Angular CLI 9.1.11.

Scaffold Angular application using Angular CLI

This part is fairly easy. All we have to do is run a single command and answer a few questions along the way to have a brand new Angular application up and running in no time. So, with the aforementioned ng command, let us create a new Angular application using the following command in a terminal:

```
ng new reactive-state-for-angular-with-ngrx
```

Please answer the asked questions as mentioned below to make sure to create an identical Angular application that we are going to deal with throughout this book.

Q: Would you like to add Angular routing?

A: Press "y".

Q: Which stylesheet format would you like to use?

A: Press Down Arrow key to choose "SCSS" from the given options and then press Enter.

This will take a couple of minutes to create the scaffolding of the application in the reactive-state-for-angular-with-ngrx directory.

One last thing to do before we start the application is to enable non-relative module resolution for ES6 imports. Add the following after lib property in tsconfig.json (or in tsconfig.base.json for Angular 10+):

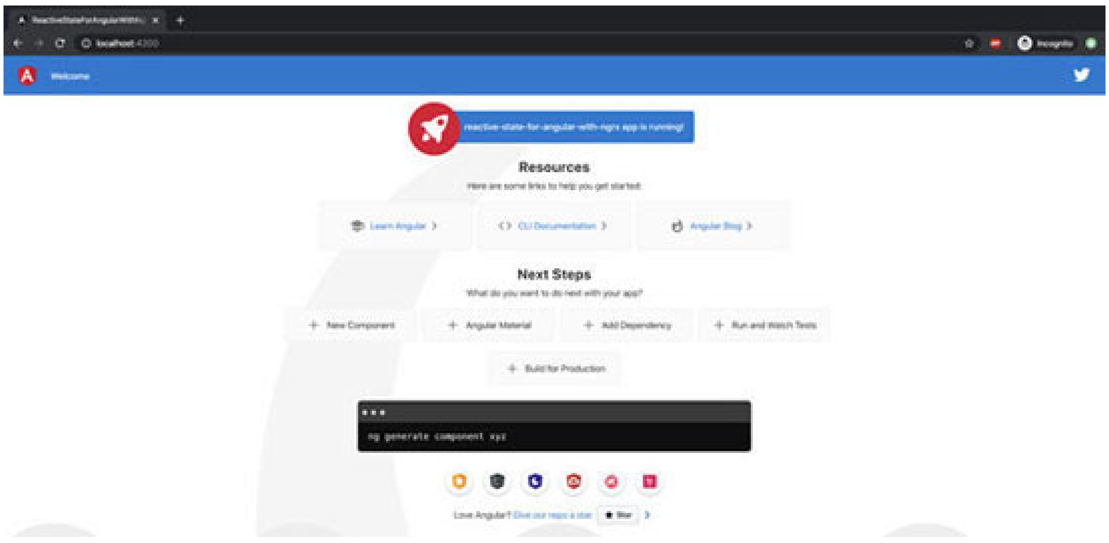
```
"paths": {
"@app/*": ["src/app/*"]
}
```

You will appreciate the benefit of having non-relative module resolution as we progress in the book to deal with various modules' dependencies.

Now go inside the mentioned directory and install Moment.js which is a date library and Angular InMemory Web API which will allow us to simulate CRUD operations over fake RESTful APIs. And then start the Angular application from the terminal:

```
cd reactive-state-for-angular-with-ngrx
npm install --save moment
npm install --save angular-in-memory-web-api
npm start
```

Open a new browser tab and go to **http://localhost:4200** to launch the application which should look something like this for you as per *Figure* Do not worry; we'll remove all these clutter in the next section:


localhost:4200
Welcome
reactive-state-for-angular-with-ngrx app is running!
Resources
Here are some links to help you get started:
Learn Angular
CLI Documentation
Angular Blog
Next Steps
What do you want to do next with your app?
New Component
Angular Material
Add Dependency
Run and Watch Tests
Build for Production
Love Angular? Give our repo a star.
Star

```
ng generate component xyz
```

Figure 2.1: *Sample Angular Application up and running using Angular CLI*

Finally, install the Angular Material library which includes Material Design Components for Angular. Material Design is a visual language based on the science of good design which was developed by Google back in 2014 to make design consistent across platforms. Angular Material is strictly following the guidelines mentioned in the Material Design specs and is actively developed and maintained by Angular Team. For us, it will save our time designing a beautiful responsive application using pre-built and cross-browser compatible UI elements such as Toolbar, Tabs, Grid, Buttons, Icons, and so on.

Let us run the following command in the same terminal as before to install Angular Material components, **CDK (Component Developer Kit),** Themes, and even Angular animations. Note that the CDK is used to customize the existing components or build new ones without worrying much about Material Guidelines but is out of the scope of this book so we will not use it.

ng add @angular/material

You will be bombarded with a similar set of questions like before but do not panic!

Q: Choose a pre-built theme name, or "custom" for a custom theme?

A: Choose "Indigo/Pink" from the given options and then press Enter.

Q: Set up global Angular Material typography styles?

A: Press "y".

Q: Set up browser animations for Angular Material?

A: Press "y".

Voila..! You have successfully scaffolded the application with Angular Material and CLI. Please note that there will not be any difference visually in the browser at the moment since we have not used Angular Material yet which is what we are going to do next.

Sometimes the running Angular application throws wacky errors in the terminal. In that case, just abort the CLI command and restart the application with npm start.

Angular application shell

An application shell is nothing but the toolbar/header and the footer (sometimes even the sidebars) which happens to be the same for all the screens in the application. So when we go from one application route to another such as from /list to only the dynamic page content will be rendered without re-fetching or re-rendering the application shell all over again.

Let us import a few modules for Material Components from Angular Material in the application NgModule in This will allow us to use respective Angular Material components in the application template:

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {BrowserAnimationsModule} from '@angular/platform-
browser/animations';
import {FormsModule} from '@angular/forms';
import {MatDialogModule} from '@angular/material/dialog';
import {MatIconModule} from '@angular/material/icon';
import {MatToolbarModule} from '@angular/material/toolbar';
import {MatFormFieldModule} from '@angular/material/form-field';
import {MatInputModule} from '@angular/material/input';

import {AppRoutingModule} from '@app/app-routing.module';
import {AppComponent} from '@app/app.component';
```

```
@NgModule({
declarations: [
AppComponent
],
imports: [

BrowserModule,
AppRoutingModule,
BrowserAnimationsModule,
```
FormsModule,
MatDialogModule,
MatIconModule,
MatToolbarModule,
MatFormFieldModule,
MatInputModule
],
```
providers: [],
bootstrap: [AppComponent]
})
export class AppModule {}
```

Next, replace the entire CLI generated HTML/CSS in with the following snippet. Note that the /list and /create routes are yet to be created:

```
color="primary">
[routerLink]="['/list']" class="example-header">Bookmarker
class="example-spacer">
search
class="example-full-width" floatLabel="never">
```

```
type="text" matInput placeholder="Filter
bookmarks...">
class="example-spacer">

[routerLink]="['/create']" class="example-icon" matTooltip="Create a
new bookmark">add
```

Finally, add some custom styling for the above markup in src/app/app.component.scss which will help us move certain UI elements to the center/right as per our need:

```
example-header {
cursor: pointer;
}
.example-icon {
padding: 0 14px;
cursor: pointer;
}
example-spacer {
flex: 1 1 auto;
}
```

Additionally, we will also use the change detection strategy to OnPush for all the components in the future in order to make the application performant. Starting with

```
import {Component, ChangeDetectionStrategy} from
'@angular/core';

@Component({
```

```
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.scss'],

changeDetection: ChangeDetectionStrategy.OnPush
})
export class AppComponent {}
```

The default Change Detection strategy assumes nothing about your Angular application and hence whenever anything changes in the component, Angular runs the change detection from the root component down to the last child component that exists in the component tree. This is considered to be the anti-pattern and leads to a poor performing Angular application. On the other hand, the OnPush strategy relies on the immutable data in the component so that the change detection would run along the path from the root component leading to this very component (excluding the rest) and only when

the component depends on changes.

An event triggered by this component or one of its child components.

If all was well at your end too then you should see the following Application Shell rising and shining as per *Figure*

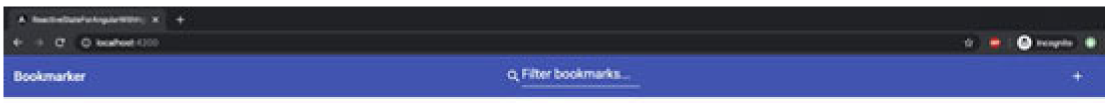


Figure 2.2: *Application Shell in Angular Material*

This application shell will be our gateway to the major functionalities of the sample application. The application logo on the top-left corner would take us to the list of bookmarks; the plus-icon on the top-right corner would allow us to create a new bookmark eventually. Additionally, the filter UI in the middle will be able to search through the list of bookmarks. All these functionalities will come to fruition by the end of this chapter though.

Shared Angular Service: Bookmark

To avoid any backend implementation for our application (which is out of the scope of this book), we are going to rely on a fake RESTful backend using previously installed angular-in-memory-web-api NPM package. Let us create the InMemoryDataService class by running the following command in the terminal:

```
ng g class shared/services/in-memory-data/InMemoryDataService --
skipTests=true
```

The g is an alias for generating in Angular CLI. This will generate an ES6 class without any unit tests in src/app/shared/services/in-memory-data directory. Using the same command, we can even generate Angular components, services, pipes, guards, directives, etc. later in the chapter.

Overwrite the generated src/app/shared/services/in-memory-data/in-memory-data-service.ts with the following snippet:

```
import {InMemoryDbService} from 'angular-in-memory-web-api';
import * as moment from 'moment';

export class InMemoryDataService implements InMemoryDbService
{
createDb() {
return {
```

```
bookmarks: [
{
id: 1,
name: 'Angular',
url: 'https://angular.io/',
created: moment().toDate(),


},
{
id: 2,
name: 'NgRx',
url: 'https://ngrx.io/',
created: moment().subtract(1, 'days').toDate()
},
{
id: 3,
name: 'Typescript - Javascript that scales',
url: 'https://www.typescriptlang.org/',
created: moment().subtract(2, 'days').toDate()
},
{
id: 4,
name: 'RxJS - A reactive programming library for JavaScript',
url: 'https://rxjs.dev/',
created: moment().subtract(3, 'days').toDate()
}
]
};
}
}
```

Notice how easy it is to manipulate the current date using Moment.js to have a few today's, yesterday's, and older sample bookmarks to begin with. Let us hook the service up in the application NgModule as well in

```
import {BrowserModule} from '@angular/platform-browser';

import {NgModule} from '@angular/core';
import {BrowserAnimationsModule} from '@angular/platform-
browser/animations';
import {MatIconModule} from ‘@angular/material/icon’;
import {MatToolbarModule} from ‘@angular/material/toolbar’;
import {MatFormFieldModule} from ‘@angular/material/form-field’;
import {MatInputModule} from ‘@angular/material/input’;
import {MatDialogModule} from ‘@angular/material/dialog’;
import {FormsModule} from ‘@angular/forms’;

import {HttpClientModule} from '@angular/common/http';
import {HttpClientInMemoryWebApiModule} from 'angular-in-
memory-web-api';
import {InMemoryDataService} from '@app/shared/services/in-
memory-data/in-memory-data-service';
import {AppRoutingModule} from '@app/app-routing.module';
import {AppComponent} from '@app/app.component';

@NgModule({
declarations: [
AppComponent
],
imports: [
BrowserModule,
AppRoutingModule,
BrowserAnimationsModule,
FormsModule,
MatIconModule,
MatToolbarModule,
MatFormFieldModule,

MatInputModule,
HttpClientModule,
HttpClientInMemoryWebApiModule.forRoot(
InMemoryDataService, {
dataEncapsulation: false,
passThruUnknownUrl: true,
put204: false // return entity after PUT/update
}
),
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule {}
```

The intended usage of the Angular in-memory-web-api package is to intercept Angular HTTP or HTTPClient requests for demos and testing purposes only. Without it, the requests would go to a remote server.

Time has come to create a bookmark service that will make fake HTTP calls to the in-memory data service in order to fetch/save/update bookmarks. This will be an Angular service so let us use Angular CLI to generate it in the Terminal:

```
ng g service shared/services/bookmark/bookmark --skipTests=true
```

Next, update the generated src/app/shared/services/bookmark/bookmark.service.ts with:

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

import {HttpClient} from '@angular/common/http';

import {Observable} from 'rxjs';

export interface Bookmark {
id: number;
name: string;
url: string;
created: Date;
}

@Injectable({
providedIn: 'root'
})
export class BookmarkService {
allBookmarks: Bookmark[]; // this will be filled up by a guard of
/list
editBookmark: Bookmark; // this will be filled up by a guard of
/edit/:bookmarkId
filterText: string; // this will be filled up by a filter textbox from
toolbar
constructor(
private http: HttpClient
) {}
public getAll(): Observable {
return this.http.get('api/bookmarks');
}

public getById(bookmarkId): Observable {
return this.http.get(`api/bookmarks/${bookmarkId}`);
}

public save(bookmark: Bookmark) {

return this.http.post(`api/bookmarks`, bookmark);
}

public update(bookmark): Observable {
return this.http.put(`api/bookmarks`, bookmark);
}
}
```

Few things to note here:

We have defined the interface of a bookmark including its and date of creation. This kind of typing makes refactoring large applications a breeze.

We have also injected HttpClient making the real HTTP requests to fetch/save/update bookmarks. However, those requests will be intercepted by in-memory-web-api that we hooked up earlier and will not throw any HTTP errors in the browser. In fact, the getAll function would return an array of 4 dummy bookmarks that we had added before whereas the getById function would return a single bookmark matching bookmarkId provided in the parameter. While save and update functions will add a new bookmark and update the existing one respectively.

Unfortunately, we cannot test the bookmark service in the browser yet. But before we do that, let us create a shared error dialog component next.

Shared Angular Component: Error Dialog

The Error Dialog is going to be a custom component built on top of the Angular Material Dialog component that will be shared across components and guards in the future. If any route is failed to load or any HTTP request refused to complete, we will use this error dialog to show an error message and expect to take further action.

Since this error dialog component will not depend on any route, we have to make it an entry component. Run the following command in the terminal:

```
ng g component shared/components/error-dialog --
entryComponent=true --module=app --skipTests=true
```

Let's see the command in detail:

The --module=app flag will automatically import the component and declare it in the application Just a time-saving trick!

The --entryComponent=true flag will load this component imperatively i.e. without a route or a custom HTML tag. In fact, it will be bootstrapped just like AppComponent in the application Take a look at src/app/app.module.ts for the curious minds.

Then, update src/app/shared/components/error-dialog/error-dialog.component.html with:

mat-dialog-title>Error occurred!

mat-dialog-content>

{{error.errorMessage}}

mat-dialog-actions>

mat-button (click)="okClick()" cdkFocusInitial>Ok

Also, update src/app/shared/components/error-dialog/error-dialog.component.ts with:

```
import {Component, Inject, ChangeDetectionStrategy} from
'@angular/core';
import {MatDialogRef, MAT_DIALOG_DATA} from
'@angular/material/dialog';
import {Router} from '@angular/router';

export interface Error {
errorMessage: string;
redirectTo: string;
}

@Component({
selector: 'app-error-dialog',
templateUrl: './error-dialog.component.html',
styleUrls: ['./error-dialog.component.scss'],
changeDetection: ChangeDetectionStrategy.OnPush
})
export class ErrorDialogComponent {
constructor(
public dialogRef: MatDialogRef,
@Inject(MAT_DIALOG_DATA) public error: Error,
private router: Router
) {}

okClick(): void {
if (this.error.redirectTo) {
this.router.navigate([this.error.redirectTo]);


}
this.dialogRef.close();
}
}
```

Few things to note here:

We can pass the data from outside to this Error component via error property.

The **Ok** button will simply close the dialog or redirect to the provided URL.

Shared Angular Class: Util

The Util class will have any generic logic that needs to be shared across various components and services. Let us generate the class using Angular CLI first in the Terminal:

```
ng g class shared/util --skipTests=true
```

Then, update src/app/shared/util.ts with two utility methods that will be used later on:

```
import * as moment from 'moment';

export function isToday(created) {
return moment(created).isSame(moment().startOf('day'), 'd');
}

export function isYesterday(created) {
return moment(created).isSame(moment().subtract(1,
'days').startOf('day'), 'd');
}
```

They are simply checking if a date provided in the parameter is today or yesterday. These helper methods will be useful to filter a list of bookmarks by today's or yesterday's date later on.

Shared Angular Pipe: Fuzzy

The Fuzzy pipe will allow us to filter the list of bookmarks. Let us generate the class using Angular CLI in the Terminal:

ng g pipe shared/pipes/fuzzy --skipImport=true --skipTests=true

Then, update src/app/shared/pipes/fuzzy.pipe.ts with:

```
import {Pipe, PipeTransform} from '@angular/core';

import {Bookmark} from
'@app/shared/services/bookmark/bookmark.service';

@Pipe({
name: 'fuzzy'
})
export class FuzzyPipe implements PipeTransform {
transform(bookmarks: Bookmark[], ...args: any[]): any {
const filterTerm = args[0];
return filterTerm ? bookmarks.filter((b: Bookmark) =>
!!b.name.toLowerCase().includes(filterTerm.toLowerCase())) :
bookmarks;
}
}
```

We are now ready with all the necessary things needed to roll out the application's true functionality that is to create a bookmark. Let us do just that next.

Create bookmark

In a nutshell, we will create a new route, that is, to load the create-bookmark component. This component is supposed to ask a bookmark name and its url from a user and then save the bookmark (along with a date of creation) into the in-memory data store.

The following command will generate the /create route along with a lazy-loaded module,

```
ng g module create --route=create --module=app
```

Let's see the command in detail:

The --route=create flag will generate a new route its component and its module CreateModule comprising the same component.

The --module=app flag will declare the new route /create with a lazy-loaded module CreateModule in The lazy-loaded module is good for performance reasons since the module will be lazily loaded only when a user navigates to the /create route.

The above command must have created a few component-specific files under src/app/create directory. Let us quickly update them one-by-one.

First, in add a bunch of Angular Material modules:

```
import {NgModule} from '@angular/core';
import {CommonModule} from '@angular/common';
import {ReactiveFormsModule} from '@angular/forms';

import {MatFormFieldModule} from '@angular/material/form-field';
import {MatExpansionModule} from '@angular/material/expansion';
import {MatIconModule} from '@angular/material/icon';
import {MatInputModule} from '@angular/material/input';
import {MatButtonModule} from '@angular/material/button';
import {MatDialogModule} from ‘@angular/material/dialog’;

import {CreateRoutingModule} from '@app/create/create-
routing.module';
import {CreateComponent} from '@app/create/create.component';

@NgModule({
declarations: [CreateComponent],
imports: [
CommonModule,
CreateRoutingModule,
ReactiveFormsModule,
MatFormFieldModule,
MatExpansionModule,
MatIconModule,
MatInputModule,
MatButtonModule,
MatDialogModule
]
})
export class CreateModule {}
```

Then, update the component template in If you see certain errors in the template (in the form of red wiggly underlines), ignore them for now since they will be gone when we update the component class later:

```
[formGroup]="bookmarkForm" (ngSubmit)="onSubmit()">
class="create-bookmark-container">
[expanded]="true" hideToggle>
Create New Bookmark
Type bookmark name and url
bookmark_border

matInput placeholder="Name"
formControlName="name">

matInput placeholder="URL"
formControlName="url">

type="submit" mat-raised-button color="primary" [disabled]="!bookmarkForm.valid">Create Bookmark
```

Next, update the component class in Check if the errors in the template have gone, they should!

```
import {Component, OnDestroy, ChangeDetectionStrategy} from
'@angular/core';
import {FormBuilder, Validators, FormGroup} from
'@angular/forms';
import {Router} from '@angular/router';
import {Subscription} from 'rxjs';
import {MatDialog} from '@angular/material/dialog';
import * as moment from 'moment';

import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';

@Component({
selector: 'app-create',
templateUrl: './create.component.html',
styleUrls: ['./create.component.scss'],
changeDetection: ChangeDetectionStrategy.OnPush
})
export class CreateComponent implements OnDestroy {
bookmarkForm: FormGroup;
bookmarkCreate$: Subscription;

constructor(

private fb: FormBuilder, private bookmarkService: BookmarkService,
private router: Router,
private readonly dialog: MatDialog
) {
this.bookmarkForm = this.fb.group({
```

```
name: ['', Validators.required],
url: ['', Validators.required]
});
}

onSubmit() {
this.bookmarkCreate$ = this.bookmarkService.save({...
this.bookmarkForm.value, created: moment().toDate()}).subscribe(
() => {this.router.navigate(['/list']);},
() => this.dialog.open(ErrorDialogComponent, {
width: '400px',
data: {errorMessage: 'Sorry, unable to create bookmark.'}
})
);
}

ngOnDestroy() {
if (this.bookmarkCreate$) {
this.bookmarkCreate$.unsubscribe();
}
}
}
```

Few things to note here:

We have instantiated the reactive form using FormBuilder since they are more powerful compared to the template-driven forms when it comes to complex validation.

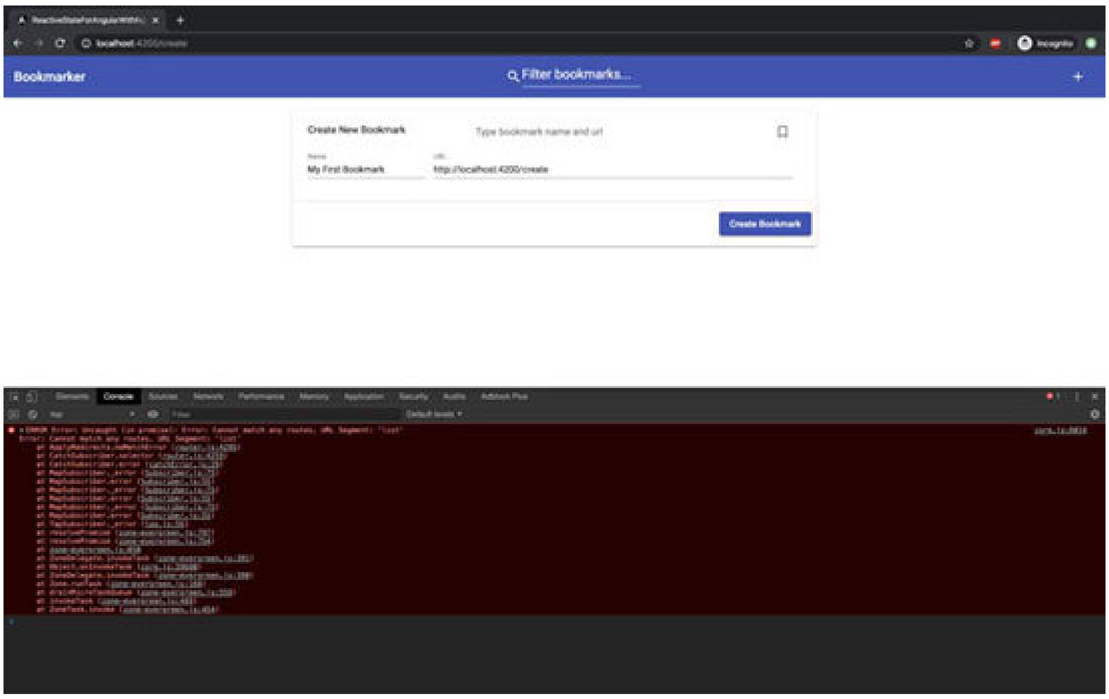
On we call the bookmark service to save the new bookmark and redirect it to /list route. Otherwise, show an error message using the shared error dialog created earlier.

At the end, customize the used Angular Material components to suit our needs in Note that applying the ::ng-deep pseudo-class to any CSS rule completely disables view-encapsulation for that rule. That means any style with ::ng-deep applied becomes a global style:

```
.create-bookmark-container {
mat-expansion-panel {
max-width: 800px;
margin: 20px auto 0 auto;
}

.mat-expansion-panel-header-title {
flex-basis: 0;
}

.mat-expansion-panel-header-description {
justify-content: space-between;
align-items: center;
flex-basis: 0;
}

::ng-deep .mat-expansion-panel-body {
display: flex;

::ng-deep mat-form-field:nth-child(2) {
```

```
flex-grow: 1;
}

}

mat-form-field {
margin-right: 12px;
}
}
```

Now, go to the browser and hit **http://localhost:4200/create** to see the create-bookmark screen as shown below. If you fill-up the form and hit the **Create Bookmark** button then it should add this new bookmark into the list of existing bookmarks as per *Figure*



```
ERROR Error: Uncaught (in promise): Error: Cannot match any routes. URL Segment: 'list'
Error: Cannot match any routes. URL Segment: 'list'
    at [illegible]
```

Figure 2.3: *Create Bookmark screen*

Oh, wait!? Did you get an error? No worries, that's because we have not created the /list route yet to which we redirect to after creating the new bookmark. Let us fix that next.

List bookmarks

In a nutshell, we will create a new route i.e. /list to load the list bookmark component. This component is supposed to show a list of bookmarks and order them by their date of creation.

The following command will generate the /list route along with a lazy-loaded module,

ng g module list --route=list --module=app

Let's see the command in detail:

The --route=list flag will generate a new route its component and its module ListModule comprising the same component.

The --module=app flag will declare the new route /list with a lazy-loaded module ListModule in

The above command must have created a few component-specific files under the src/app/list directory. Let us quickly update them one-by-one.

First, in add a bunch of Angular Material modules:

import {NgModule} from '@angular/core';

```
import {CommonModule} from '@angular/common';
import {MatListModule} from '@angular/material/list';
import {MatIconModule} from '@angular/material/icon';
import {FormsModule} from '@angular/forms';

import {ListRoutingModule} from '@app/list/list-routing.module';
import {ListComponent} from '@app/list/list.component';
import {FuzzyPipe} from '@app/shared/pipes/fuzzy.pipe';



@NgModule({
declarations: [ListComponent, FuzzyPipe],
imports: [
CommonModule,
ListRoutingModule,
MatListModule,
MatIconModule,
FormsModule


]
})
export class ListModule {}
```

Then, update the component template in If you see certain errors in the template (in the form of red wiggly underlines), ignore them for now since they will be gone when we update the component class later:

```
*ngIf="bookmarkService.filterText; else groupedBookmarks">
```

mat-subheader>Results for "*{{bookmarkService.filterText}}*"

*ngFor="let allBookmark of allBookmarks | fuzzy: bookmarkService.filterText">
mat-list-icon>bookmark

mat-line>{{allBookmark.name}}

[href]="allBookmark.url" mat-line> {{allBookmark.url}}
[routerLink]="['/edit', allBookmark.id]">mat-list-icon>edit

#groupedBookmarks>

mat-subheader>Today

*ngFor="let todaysBookmark of todaysBookmarks">
mat-list-icon>bookmark

mat-line>{{todaysBookmark.name}}

[href]="todaysBookmark.url" mat-line> {{todaysBookmark.url}}
[routerLink]="['/edit', todaysBookmark.id]">mat-list-icon>edit

mat-subheader>Yesterday

*ngFor="let yesterdaysBookmark of yesterdaysBookmarks">
mat-list-icon>bookmark

mat-line>{{yesterdaysBookmark.name}}

[href]="yesterdaysBookmark.url" mat-line>
{{yesterdaysBookmark.url}}

[routerLink]="['/edit', yesterdaysBookmark.id]">mat-list-icon>edit

mat-subheader>Older

*ngFor="let olderBookmark of olderBookmarks">
mat-list-icon>bookmark

mat-line>{{olderBookmark.name}}

[href]="olderBookmark.url" mat-line> {{olderBookmark.url}}
[routerLink]="['/edit', olderBookmark.id]">mat-list-icon>edit

Next, update the component class in src/app/list/list.component.ts. Check if the errors in the template have gone, they should! Here, we fetch all the bookmarks first and then filter them out by today's and yesterday's date. And the rest fell into the older category:

```
import {Component, OnInit} from '@angular/core';

import {BookmarkService, Bookmark} from
'@app/shared/services/bookmark/bookmark.service';
import {isToday, isYesterday} from '@app/shared/util';

@Component({
selector: 'app-list',
templateUrl: './list.component.html',
styleUrls: ['./list.component.scss']
})
export class ListComponent implements OnInit {
```

```
allBookmarks: Bookmark[];
todaysBookmarks: Bookmark[];
yesterdaysBookmarks: Bookmark[];
olderBookmarks: Bookmark[];

constructor(public readonly bookmarkService: BookmarkService) {}

ngOnInit() {
this.allBookmarks = this.bookmarkService.allBookmarks;

this.todaysBookmarks = this.allBookmarks.filter((bookmark) =>
isToday(bookmark.created));
this.yesterdaysBookmarks = this.allBookmarks.filter((bookmark) =>
isYesterday(bookmark.created));
this.olderBookmarks = this.allBookmarks.filter((bookmark) => {
return !this.todaysBookmarks.find((b) => b.id === bookmark.id) &&
!this.yesterdaysBookmarks.find((b) => b.id === bookmark.id);
});
}
}
```

At the end, customize the used Angular Material components to suit our needs in

```
mat-list {
max-width: 800px;
margin: 20px auto 0 auto;
}

.mat-list-icon {
```

```
color: rgba(0, 0, 0, 0.54);
}
```

Unlike the create bookmark screen, we need to fetch the list of bookmarks for this component to work properly. For that, let us generate an Angular Route Guard:

ng g guard list/list --skipTests=true

Q: Which interfaces would you like to implement?

A: Press to choose "CanActivate" from the given options (if not pre-selected) and then press .

This command will generate a new guard file for the /list route to use. Update src/app/list/list.guard.ts as follows. Here, we fetch a list of all the bookmarks over the wire and then store them on the allBookmarks property from the bookmark service for the ListComponent component to use. Otherwise, we show an error message and prevent the route from activating:

```
import {Injectable} from '@angular/core';
import {CanActivate} from '@angular/router';
import {Observable, of} from 'rxjs';
import {first, switchMap, map, catchError} from 'rxjs/operators';

import {MatDialog} from '@angular/material/dialog';
import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
```

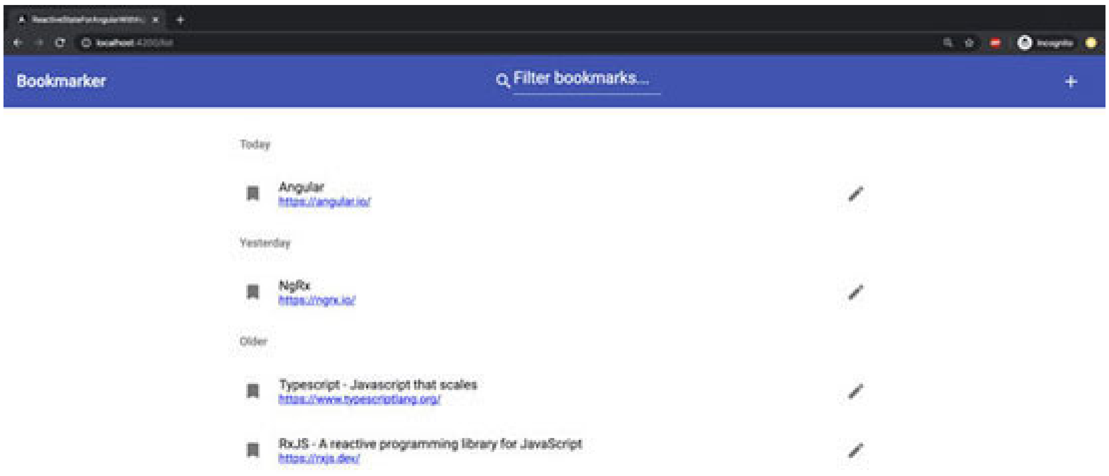
```
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';

@Injectable({
providedIn: 'root'
})
export class ListGuard implements CanActivate {
constructor(
private readonly bookService: BookmarkService,
private readonly dialog: MatDialog
) {}

canActivate(): Observable {
return this.bookService.getAll().pipe(
first(),

map((bookmarks) => this.bookService.allBookmarks = bookmarks),
switchMap(() => this.activateRoute()),
catchError(() => {
this.dialog.open(ErrorDialogComponent, {
width: '400px',
data: {errorMessage: 'Sorry, unable to fetch bookmarks.'}
});
return of(false);
})
);
}

activateRoute() {
return of(true);
}
```

}

And also update routes in src/app/list/list-routing.module.ts to inject the Guard:

```
import {ListGuard} from '@app/list/list.guard';
const routes: Routes = [{path: '', canActivate: [ListGuard],
component: ListComponent}];
```

That's it. Go to the browser and hit **http://localhost:4200/list** to see a list of bookmarks as per *Figure*


Bookmarker

Filter bookmarks...

Today

Angular
https://angular.io/

Yesterday

NgRx
https://ngrx.io/

Older

Typescript - Javascript that scales
https://www.typescriptlang.org/

RxJS - A reactive programming library for JavaScript
https://rxjs.dev/

Figure 2.4: *List Bookmark screen*

Edit bookmark

In a nutshell, we will create a new route i.e. /edit to load the edit bookmark component. The following command will generate the /edit route along with a lazy-loaded module,

```
ng g module edit --route=edit/:bookmarkId --module=app
```

Let's see the command in detail:

The --route=edit flag will generate a new route its component and its module EditModule comprising the same component.

The --module=app flag will declare the new route /edit with a lazy-loaded module EditModule in

The above command must have created a few component-specific files under src/app/edit directory. Let us quickly update them one-by-one.

First, in add a bunch of Angular Material modules:

```
import {NgModule} from '@angular/core';
import {CommonModule} from '@angular/common';
import {ReactiveFormsModule} from '@angular/forms';
import {MatFormFieldModule} from '@angular/material/form-field';
import {MatExpansionModule} from '@angular/material/expansion';
import {MatIconModule} from '@angular/material/icon';
import {MatInputModule} from '@angular/material/input';
import {MatButtonModule} from '@angular/material/button';



import {EditRoutingModule} from '@app/edit/edit-routing.module';
import {EditComponent} from '@app/edit/edit.component';

@NgModule({
declarations: [EditComponent],
imports: [
CommonModule,
EditRoutingModule,
ReactiveFormsModule,
MatFormFieldModule,
MatExpansionModule,
MatIconModule,
MatInputModule,
MatButtonModule
]
})
export class EditModule {}
```

Then, update the component template in If you see certain errors in the template (in the form of red wiggly underlines), ignore them for now since they will be gone when we update the component class later:

```
[formGroup]="bookmarkForm" (ngSubmit)="onSubmit()">
class="edit-bookmark-container">
```

[expanded]="true" hideToggle>
Edit Existing Bookmark
Type bookmark name and url

bookmark_border
matInput placeholder="Name"
formControlName="name">
matInput placeholder="URL"
formControlName="url">

type="submit" mat-raised-button color="primary" [disabled]="!bookmarkForm.valid">Update Bookmark

Next, update the component class in Check if the errors in the template have gone, they should!

```
import {Component, OnDestroy, ChangeDetectionStrategy} from
'@angular/core';
import {FormBuilder, Validators, FormGroup} from
'@angular/forms';
import {Router} from '@angular/router';
import {Subscription} from 'rxjs';
import {MatDialog} from '@angular/material/dialog';

import * as moment from 'moment';

import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';

@Component({
```

```
selector: 'app-edit',
templateUrl: './edit.component.html',
styleUrls: ['./edit.component.scss'],
changeDetection: ChangeDetectionStrategy.OnPush
})
export class EditComponent implements OnDestroy {
bookmarkForm: FormGroup;
bookmarkUpdate$: Subscription;

constructor(
private fb: FormBuilder,
private bookmarkService: BookmarkService,
private router: Router,
private readonly dialog: MatDialog
) {
this.bookmarkForm = this.fb.group({
name: [this.bookmarkService.editBookmark.name, Validators.required],
url: [this.bookmarkService.editBookmark.url, Validators.required]
});
}

onSubmit() {
this.bookmarkUpdate$ = this.bookmarkService.update({

id: this.bookmarkService.editBookmark.id,
...this.bookmarkForm.value,
created: moment().toDate()
}).subscribe(
() => {this.router.navigate(['/list']);},
() => this.dialog.open(ErrorDialogComponent, {
width: '400px',
data: {errorMessage: 'Sorry, unable to update bookmark.'}
})
);
}

ngOnDestroy() {
if (this.bookmarkUpdate$) {
this.bookmarkUpdate$.unsubscribe();
}
}
}
```

Similar to we instantiate the reactive form along with validators here as well. However, the input fields will be auto-filled with the values being edited. On we update the bookmark and redirect it back to the /list route. Otherwise, show an error message if the update API fails for some reason.

At the end, customize the used Angular Material components to suit our needs in

```
.edit-bookmark-container {
mat-expansion-panel {
max-width: 800px;
margin: 20px auto 0 auto;

}

.mat-expansion-panel-header-title {
flex-basis: 0;
}

.mat-expansion-panel-header-description {
justify-content: space-between;
align-items: center;
flex-basis: 0;
}

::ng-deep .mat-expansion-panel-body {
display: flex;

::ng-deep mat-form-field:nth-child(2) {
flex-grow: 1;
}
}

mat-form-field {
margin-right: 12px;
}
}
```

Just like the list bookmark screen, we need to fetch the bookmark details for this component to edit properly. For that, let us generate an Angular Route Guard:

ng g guard edit/edit --skipTests=true

Q: Which interfaces would you like to implement?

A: Press to choose "CanActivate" from the given options (if not pre-selected) and then press .

This command will generate a new guard file for the /edit/:bookmarkId route to use. Update as follows:

```
import {Injectable} from '@angular/core';
import {CanActivate, ActivatedRouteSnapshot} from
'@angular/router';
import {Observable, of} from 'rxjs';
import {MatDialog} from '@angular/material/dialog';
import {first, map, switchMap, catchError} from 'rxjs/operators';

import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';

@Injectable({
providedIn: 'root'
})
export class EditGuard implements CanActivate {
constructor(
private readonly bookService: BookmarkService,
private readonly dialog: MatDialog
) {}

canActivate(route: ActivatedRouteSnapshot): Observable {
return this.bookService.getById(route.params.bookmarkId).pipe(
first(),
map((bookmark) => this.bookService.editBookmark = bookmark),
switchMap(() => this.activateRoute()),
catchError(() => {
this.dialog.open(ErrorDialogComponent, {

width: '400px',
data: {errorMessage: 'Sorry, unable to edit bookmark.', redirectTo:
'/list'}
});
return of(false);
})
);
}

activateRoute() {
return of(true);
}

}
```

Few things to note here:

First of all, we fetch the bookmark by its id and store it in the editBookmark property from the bookmark service and activate the /edit/:bookmarkId route.

Otherwise, show an error message using the shared error dialog and prevent the route from activating.

Finally, update routes in src/app/edit/edit-routing.module.ts to inject the same guard:

```
import {EditGuard} from '@app/edit/edit.guard';
const routes: Routes = [{path: '', canActivate: [EditGuard],
component: EditComponent}];
```

That's it. Go to the browser and hit **http://localhost:4200/list** to see a list of bookmarks and then try to edit one of the bookmarks.

Filter bookmarks

Upon typing into the filter textbox up on the header, you will notice that the bookmarks list is not filtered. The problem is that we have not updated filterText property in So update

```
color="primary">
[routerLink]="['/list']" class="example-header">Bookmarker
class="example-spacer">
search
class="example-full-width" floatLabel="never">
type="text" matInput placeholder="Filter
bookmarks..." [(ngModel)]="bookmarkService.filterText">
class="example-spacer">
[routerLink]="['/create']" class="example-icon" matTooltip="Create a
new bookmark">add
```

Then, update src/app/app.component.ts:

```
import {Component, ChangeDetectionStrategy} from
'@angular/core';

import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';

@Component({
selector: 'app-root',

templateUrl: './app.component.html',
styleUrls: ['./app.component.scss'],
changeDetection: ChangeDetectionStrategy.OnPush
})
export class AppComponent {
constructor(public readonly bookmarkService: BookmarkService) {}
}
```

Go to the browser and hit **http://localhost:4200/list** and type ngrx into the filter textbox to see a list of bookmarks filtered before your eyes.

Default Route

We are almost done with the application. There is one minor thing pending though. I would recommend simply launching **http://localhost:4200/** in a browser. Did you notice a blank page? I reckon, yes. And that's because the empty path matches none of the routing configurations defined in

Let us add the empty path and redirect to the /list route if none matches:

```
import {NgModule} from '@angular/core';
import {Routes, RouterModule} from '@angular/router';

const routes: Routes = [
{path: 'create', loadChildren: () =>
import('@app/create/create.module').then(m => m.CreateModule)},
{path: 'list', loadChildren: () =>
import('@app/list/list.module').then(m => m.ListModule)},
{path: 'edit/:bookmarkId', loadChildren: () =>
import('@app/edit/edit.module').then(m => m.EditModule)},
{path: '', redirectTo: '/list', pathMatch: 'full'}
];

@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule]
})
export class AppRoutingModule {}
```

We are done with the Angular application that we needed to experiment NgRx on. I would suggest you play with the application a bit; try adding a new bookmark, editing an existing one, or searching through the list before moving to the next section. The next section could easily be a part of the next chapter, however, I really do not want this to come in your way while learning NgRx Store in the next chapter since we are simply installing a couple of packages here without going into the underpinnings of NgRx. So it is safe to go through the next section now before moving on but feel free to skip it if you do not wish to debug NgRx in future chapters.

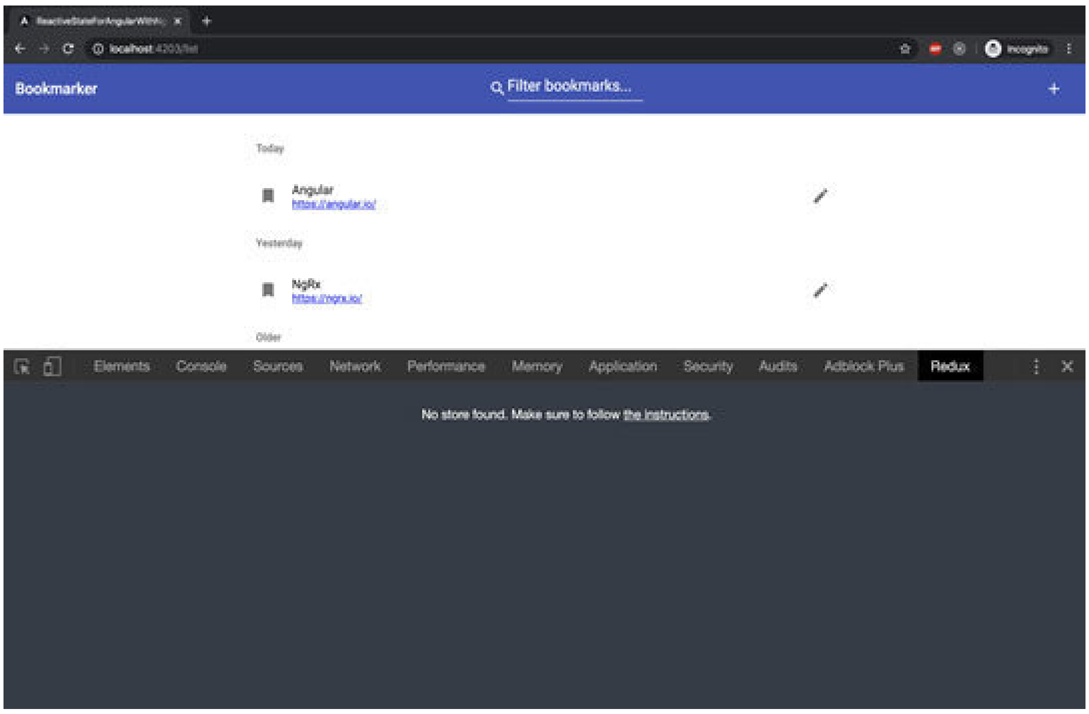
NgRx Debugging: Redux Devtool

When we will make state changes using NgRx action/dispatcher/reducer in the next chapter, we will need some sort of development workflow. This development workflow will showcase us with various NgRx actions along with the data we dispatch which will help us debug the state changes. Fortunately, Redux has its browser extension to power-up Redux development workflow, and luckily, it is compatible with NgRx as well. The Redux browser extension can be used on Chrome and Firefox respectively:

https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd (or Google devtools in

https://addons.mozilla.org/en-US/firefox/addon/reduxdevtools/ (or Google devtools in

Upon installing, launch **http://localhost:4200/** in Chrome/Firefox and open the browser devtool to select the Redux tab as shown in the following screenshot:


Bookmarker
Filter bookmarks...
Today
Angular
https://angular.io/
Yesterday
NgRx
https://ngrx.io/
Older
Elements Console Sources Network Performance Memory Application Security Audits Adblock Plus Redux

No store found. Make sure to follow the instructions.

__Figure 2.5:__ Redux Devtools without NgRx Devtool installed

You will see the above note about the missing Store which is obvious because we have not installed the NgRx Store yet. Let us just install NgRx Store and StoreDevTools now in the terminal using one of the following commands:

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

Then, import both the packages in src/app/app.module.ts as follows:

```
import {StoreModule} from '@ngrx/store';
import {StoreDevtoolsModule} from '@ngrx/store-devtools';
```

And setup NgRx Store and Redux Devtools (do not forget to turn it off for the Production environment) under NgModule imports property:

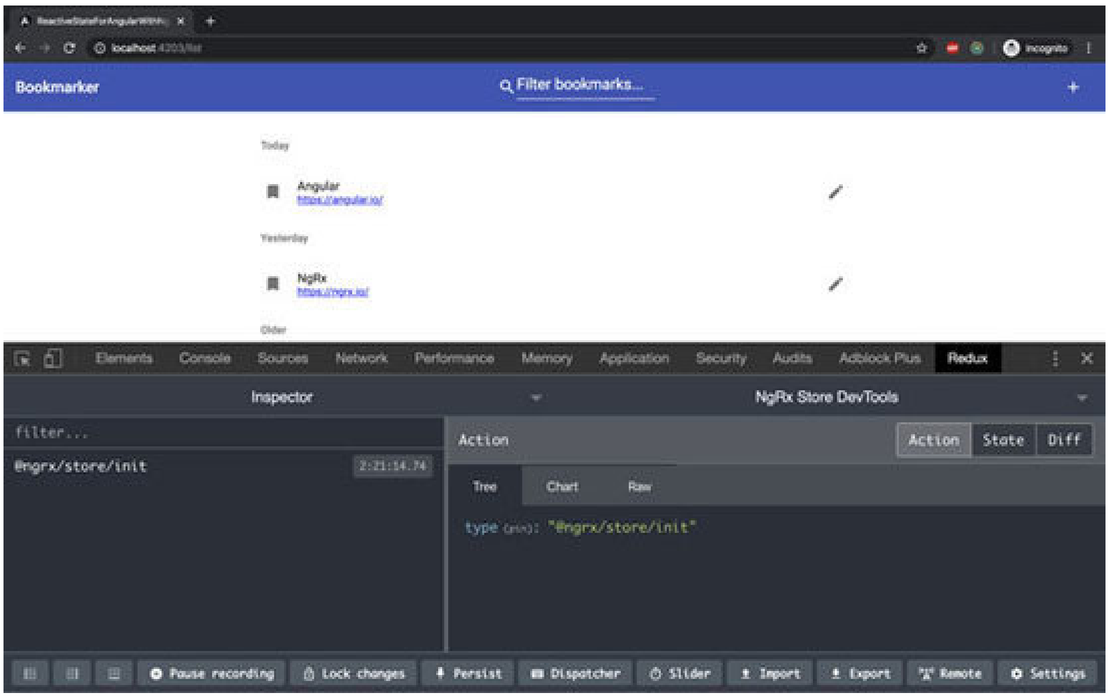
```
StoreModule.forRoot([], {
runtimeChecks: {

strictStateImmutability: true,
strictActionImmutability: true,
}
}),
StoreDevtoolsModule.instrument()
```

Let's see the above code in detail:

strictStateImmutability prevents from modifying the state directly which helps the state predictably updated.

strictActionImmutability is the same as strictState Immutability but for NgRx Actions.

Now if you reload **http://localhost:4200/** in Chrome/Firefox, you will see the Redux devtool kicked in. *Figure 2.6* shows the first action being dispatched by NgRx itself:



```
@ngrx/store/init                2:21:14.74
type (pin): "@ngrx/store/init"
```

Figure 2.6: *Redux Devtools with NgRx Devtool installed*

This brought us to the end of the second chapter but we will not stop here. Throughout this book, we will keep our eyes open for the code-level improvements that allow us to extract rigid and scattered application states used here into a well-defined and reusable

Conclusion

In this chapter, we have created a sample Angular application from scratch using the Angular CLI. We have learned to create a shared Angular Service, a shared Angular Error Dialog, and a shared Angular Pipe in the beginning. We then used those shared services/components/pipes across the application. Using Angular CLI, we easily created various lazy routes to list/create/edit bookmarks. In short, we managed to build the end-to-end single page application in Angular which is kind of an added bonus in this book.

This chapter in fact taught you to build production-ready applications in no time. Near the end of the chapter, we got ourselves acquainted with NgRx Debugging tool which we will continue to use to debug the application state in future chapters.

In the next chapter, we will begin with the NgRx Store and improve the existing application for the better.

Questions

What is Angular CLI and how to scaffold an application using it?

What is an Angular application shell?

What is a shared component/service in Angular?

How to create a lazy route in Angular?

What is an Angular Guard and why to use it?

What is an Angular pipe and how to create one?

What is Redux Devtool?

CHAPTER 3

NgRx Store

Managing Application State is tricky. We have analyzed the consequences of having the application state scattered across the application. We then made our way up to get acquainted with Redux Principles in the first chapter. We also wrote our version of NgRx, Reducks, to learn more about Redux terminologies in terms of Selector, Action, Dispatcher, and Reducer and how they are wired up in a sequence. We learned that Reducks is made to work with a single component only (which is far from the reality) but never addressed it head-on. Even though we have not touched NgRx until this chapter, the understanding from the previous chapters has not gone in vain but in fact, it will help us understand NgRx better.

This chapter is structured slightly differently from the previous ones because we expect you to know most of the Redux terminologies from the first chapter by now. Given that you have a good understanding of Redux or Reducks, we take the opportunity to introduce you to another similar technology called NgRx. You can use this library in conjunction with an Angular application, and we think they are well worth your time learning about it, especially if you are looking to scale your Angular applications. Fortunately, the most difficult hurdle is the mental leap of changing paradigms, and you have successfully done that because you are here now. At a high level, the goal of this chapter is to teach you the components of NgRx architecture.

Understanding how NgRx works can be overwhelming at first and a basic understanding of Redux would help you go through this chapter, but it's certainly not necessary.

Structure

In this chapter, we will leverage @ngrx/schematics to scaffold or auto-generate NgRx related code. We will write our first Application State in NgRx and migrate the existing components (from the previous chapter) to make use of it.

What is NgRx?

@ngrx/schematics

What is the root store?

What is feature store?

Objectives

We will understand the concept behind the root store and create our very first root store in NgRx to integrate it into the sample application that we built in the previous chapter. We will also learn about the feature store and how/why it is different from the root store and when to prefer one or the other.

What is NgRx?

NgRx Store, as you already know, is RxJS powered state management for Angular applications, inspired by Redux. Hence, NgRx is a framework for building reactive applications in Angular. With NgRx, the entire tech stack in Angular will become reactive. NgRx provides consistent and easy to reason about state management for Angular applications. It is good at isolating the side-effects produced by the asynchronous API calls using NgRx Effects which will be explored in the next chapter. It also provides a CLI tool, NgRx Schematics, for code generation which instantly saves developers time needed for writing boilerplate code to integrate NgRx, enhancing the experience, and boosting the productivity of developers. Although, there are four pillars on which NgRx is resting peacefully:

Serializability: NgRx as a whole is a package that normalizes state changes and passes them through observables making the state serializable and predictable. The serializability enables us to save the entire state into external storage as well such as IndexDB. In addition, we can even inspect the state, download it, upload it back, and dispatch actions using the Redux Devtools making it playful to tinker with.

Type Safety: NgRx is backed by the solid type system from Typescript. It expects every state, action, and even reducer to have imperative typing. This helps big time while building and refactoring large applications or dealing with humongous codebases.

Isolation: The side effects are nothing but interaction with external resources such as network requests, web socket, etc, and any business logic that can be isolated from the UI. Even though NgRx reducers are pure functions as per the 3rd rule of Redux (we learned in the first chapter), it does not neglect side effects. NgRx Effects is there to manage such side-effects with a breeze without breaking the aforementioned rule.

Testability: Testing the NgRx code is pretty easy and straightforward because of the use of pure functions for reading or mutating the state and the ability to isolate side-effects from the UI.

Imagine NgRx like a sentient bot who knows way better than anybody else about managing the application state well enough to drive performance and consistency in your application. Being the bot-like but not self-aware (like *us* conscious human beings), it provides ways using which we are allowed to save or fetch the application state (or a slice of the application state) is what shown in *Figure 3.1* below:

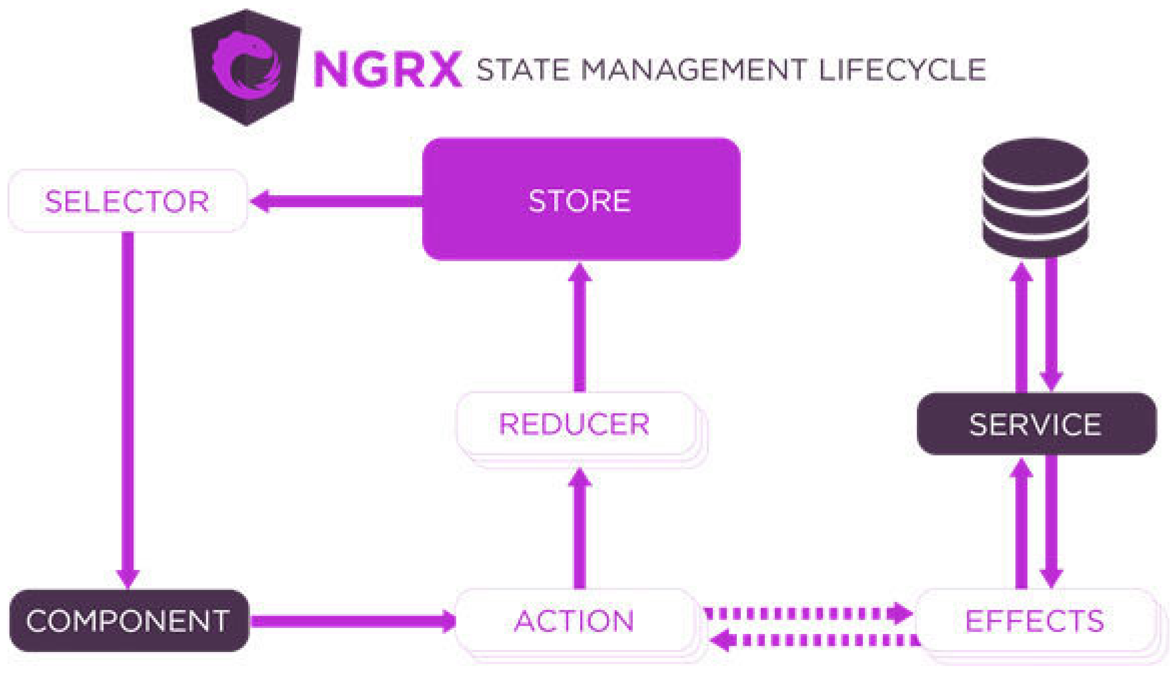


Figure 3.1: *NgRx Store data flow*

By now we know it all but once again, let us look at those ways quickly:

NgRx Action: An angular component/service/directive/class dispatches an action with an optional payload towards NgRx Reducer.

NgRx Reducer: A pure function that receives the payload and combines it with the existing state in order to generate a new state and hurls it towards the store.

NgRx Selector: A highly optimized helper function that obtains the slice of the application state stored by the reducer that can be used by the Angular Component/Template.

NgRx Effect: Handles asynchronous calls a.k.a. side-effects that the NgRx reducer should not handle itself for being a pure function. We will cover this in great detail in the next chapter.

@ngrx/schematics

It is a common pain-point for Angular developers using NgRx that it requires a lot of boilerplate code to wire things up. And you must have run into a similar situation while using Reducks in the first chapter. Hence, the NgRx team has created this scaffolding library based on Angular Schematics which will provide Angular CLI commands to auto-generate store/action/reducer/effects files. Angular Schematics is nothing but a code generator based on templates that support complex logic. They are a set of instructions for transforming an Angular project by generating a new or modifying the existing code. The NgRx Schematics commands would automatically hook reducers to respective Angular modules without any manual effort which is a big time saver, especially in large Angular applications. But before we can use any of those commands, let us first install the package:

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

Please answer the asked questions as mentioned below:

Q: Do you want to use @ngrx/schematics as the default collection?

A: Press "y".

This must-have updated angular.json for Angular CLI to take effect. We are now off to create our NgRx Store in the next section.

What is the root store?

The Root Store comprises the entire application state. As we learned in the first chapter, the Root Store is nothing but a JSON object. For the application we built in *Chapter 2: Setting up NgRx in Angular* (take a look at *Figure* in NgRx, the application state can be represented as separate slices as per *Figure* each slice holds the state of a different component:

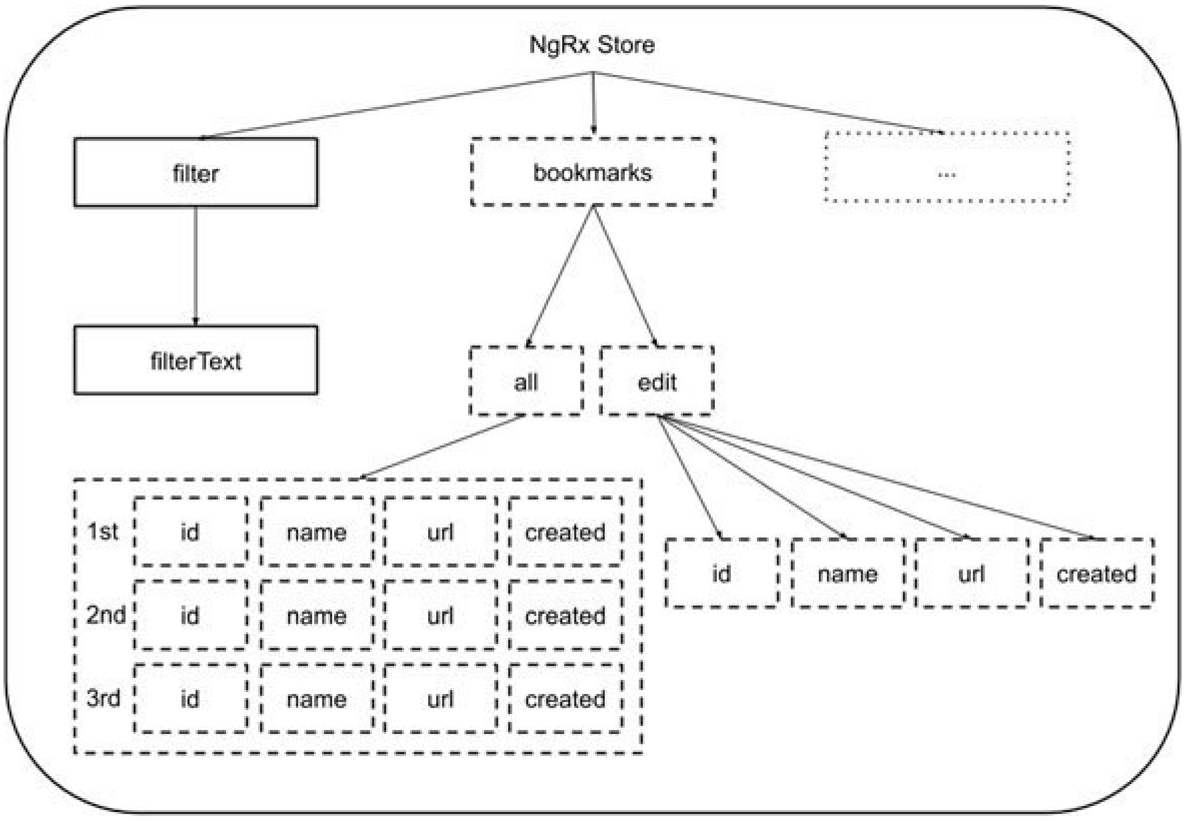


Figure 3.2: *Application State in NgRx Store is JSON object*

Each slice of the application state should have a unique key. The first slice is a filter property which would hold filterText that we type into the filter input box. Since it belongs to the application shell, we will make it a part of the root store. This means that the root store will be created immediately with the initial states defined after the application is loaded in the browser. On the other hand, the second slice is a bookmarks property that will hold either all the bookmarks or the one you want to edit. The edit slice will further have the bookmark's metadata in terms of its **id, name, url,** and **created** Whereas the all slice will have a similar set of metadata but grouped. Since we lazy load /list and /edit routes with feature modules using Angular Router, we will make bookmarks as a feature store (which is why it is highlighted with the dashed boxes). This means that the feature store will be created as and when the lazy-loaded route is resolved and activated.

Creating a root store

Since we installed the NgRx Schematics in the previous section, we can leverage Angular CLI to create our NgRx store using the following command. Few things to note:

The --stateInterface flag expects the type of the state, just like we had MyStore type in the first chapter.

The --root flag makes it a root store. If not provided, it will make it a feature store.

The --statePath flag creates an src/app/store directory to save the generated files.

The --module flag automatically plugs the root store into

ng g store --stateInterface AppState --root --statePath store --module app.module.ts

Please answer the asked questions as mentioned below:

Q: What should be the name of the state?

A: Type "AppState" and press Enter.

You should see the following snippet in the generated src/app/store/index.ts file. The important bits in the below code are ActionReducerMap and

```
import {
ActionReducer,
ActionReducerMap,
createFeatureSelector,
createSelector,
MetaReducer
} from '@ngrx/store';

import {environment} from '../../environments/environment';
export interface AppState {
}
export const reducers: ActionReducerMap = {
};
export const metaReducers: MetaReducer[] =
!environment.production ? [] : [];
```

We will get rid of unnecessary imports as we do not need them and will also rename reducers to ROOT_REDUCERS for brevity. So, final src/app/store/index.ts should look like:

```
import {ActionReducerMap, MetaReducer} from '@ngrx/store';
import {environment} from '../../environments/environment';

export interface AppState {
}
```

```
export const ROOT_REDUCERS: ActionReducerMap = {
};
export const metaReducers: MetaReducer[] =
!environment.production ? [] : [];
```

This is perfect for the example at hand. Adding ROOT_REDUCERS to your application module, later on, instructs NgRx to update the root store by executing the defined reducers. But before we do that, let us briefly understand this reducer jargon next.

ActionReducerMap

Consider ActionReducerMap as a JavaScript Map object which lets us register multiple reducers. Just like the Map object, it takes a unique name of the reducer as a key, and a reducer function itself as a value. Then, we configure all the reducers defined in ROOT_REDUCERS in the application module using Using the .forRoot method, we are ensuring that all the reducers that we may define under ROOT_REDUCERS will make up the root store when the application loads in the browser.

metaReducers

Each NgRx action is tapped by the NgRx reducer to alter the state one way or the other. In case we want to pre-process each of such actions before its reducer is invoked, we can leverage For example, if you want to track every action along with its payload on Google Analytics or Mixpanel then you do not have to copy-paste the tracking code all over the application right before dispatching the action. Instead of that, you can use the power of the meta reducers since they intercept each action right before dispatching the associated reducer. That means you can stuff your tracking logic inside the meta reducer. Additionally, you can add multiple meta reducers. Plus, it is easy to learn about the meta reducers once you understand the entire **action -> dispatcher -> reducer** pipeline in NgRx. Hence, we are not going to cover it in this book but you can explore more on the NgRx Website if needed.

Moving on, as we have renamed few exports above, we must have to fix the related imports in src/app/app.module.ts for the application to work properly (just replace reducers with

```
import metaReducers} from '@app/store';
{....}
```

Now that we have created the container, it's time to create our very first reducer next.

Root NgRx Reducer

Modifying the state using a pure function is a job of NgRx Reducer. The reducer function takes the initial state and action as arguments and then returns a new state with necessary amendments. As per the *Figure 3.2* that we saw earlier, we first have to create a reducer for one of the root store properties, filter, and then we will hook it up in ROOT_REDUCERS at the end. As of now, our application is quite minimalistic and only has a provision for searching the bookmarks but you can extend the toolbar to have more functionality such as filtering bookmarks by domain or arrange them in cards format instead of list format, etc. Hence, we will call it, toolbar.

Creating your first reducer

Let us create our first Reducer using the following command. Here, the reducer command tells the NgRx Schematics to generate a reducer file. The store/toolbar/toolbar is the path where we want to store the generated reducer file. It assumes based on the Angular CLI configurations that the store directory should be created under src/app directory so the absolute path is not necessary. In that, the toolbar directory and then ultimately naming the reducer file as

```
ng g reducer store/toolbar/toolbar --skipTests=true
```

The --skipTests flag prevents us from generating the spec files since we do not want to write unit tests. However, in real-world projects, you may have to. In that case, just do not pass the flag.

Please answer the asked questions as mentioned below:

Q: Should we add success and failure actions to the reducer?

A: Type "No" and press Enter. These will be used in NgRx Effects in the next chapter though.

Q: Do you want to use the create function?

A: Type "Yes" and press Enter. Starting from NgRx version 7.4.0, the new API called Action Creator was introduced over a class-based approach for being light-weight and concise.

The aforementioned command must have generated the following code in

```
import {Action, createReducer, on} from '@ngrx/store';

export const toolbarFeatureKey = 'toolbar';

export interface State {
}

export const initialState: State = {

};

const toolbarReducer = createReducer(
initialState,
);

export function reducer(state: State | undefined, action: Action) {
return toolbarReducer(state, action);
}
```

We will amend the above code to suit our needs as follows. We will:

Get rid of toolbarFeatureKey that will be needed only by the feature store.

Rename State to FilterState to have filterText property in it.

Update initialState to have an empty filterText property for now. We will update it when you type something into the filter input box from the application shell.

Add a comment in toolbarReducer to tap into NgRx actions to mutate filterText in the next section.

```
import {Action, createReducer, on} from '@ngrx/store';

export interface FilterState {
filterText: string;
}

export const initialState: FilterState = {
filterText: ''
};

const toolbarReducer = createReducer(

initialState,
// tap into NgRx actions to mutate the state
);
```

```
export function reducer(state: FilterState | undefined, action:
Action) {
return toolbarReducer(state, action);
}
```

When we will run this reducer in the browser, the root store filter will have an empty filterText value correctly as per the initial state we had defined above. But for it to happen, we must hook this reducer in the aforementioned ROOT_REDUCERS next.

Connecting your first reducer

Even though we have created our first reducer, the Redux Devtools would not be showing you anything yet. And that is because we have not connected the reducer toolbarReducer with ROOT_REDUCERS yet. This part is easy though! All we have to do is import the FilterState and reducer function from src/app/store/toolbar/toolbar.reducer.ts in src/app/store/index.ts as follows:

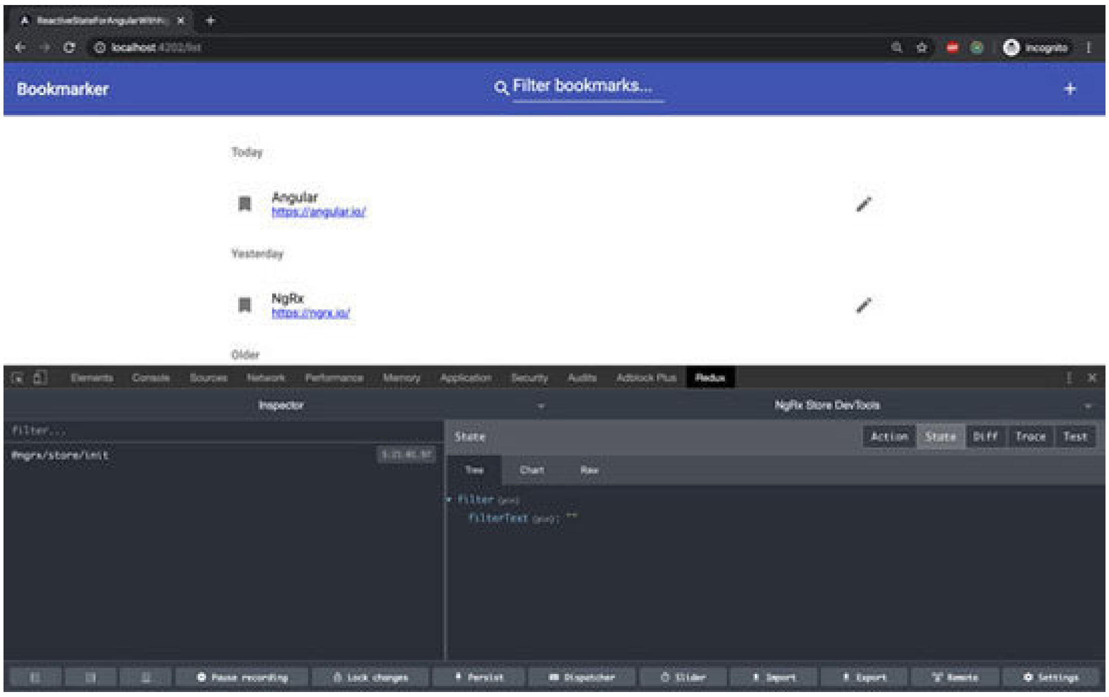
```
import {ActionReducerMap, MetaReducer} from '@ngrx/store';

import {environment} from '../../environments/environment';
import * as fromToolbar from '@app/store/toolbar/toolbar.reducer';

export interface AppState {
filter: fromToolbar.FilterState;
}

export const ROOT_REDUCERS: ActionReducerMap = {
filter: fromToolbar.reducer
};

export const metaReducers: MetaReducer[] =
!environment.production ? [] : [];
```

For readability purposes, it is recommended to import all the exports from src/app/store/toolbar/toolbar.reducer.ts as fromToolbar. So we hooked,

FilterState into AppState

Toolbar reducer into ROOT_REDUCERS

This will automatically inject an empty filterText in the filter property of the application state on page load based on the initialState that we defined earlier in the toolbar reducer. Just reload **http://localhost:4200** to see the core action @ngrx/store/init triggered by NgRx as per *Figure*



```
@ngrx/store/init
```

```
▾ filter (pin)
    filterText (pin): ""
```

Figure 3.3: *Root Store with initialState*

However, NgRx Reducer is of no use unless there is NgRx Action. Hence, let us create our very first action and hook it up in the abovementioned toolbarReducer in the next section.

Root NgRx Action

NgRx Action expresses intent towards NgRx Reducer to transform the state. The action creator is a function to create a custom action such as FILTER_BOOKMARKS that accepts the necessary payload such as filterText to filter the list of bookmarks. The action creator returns an object containing the type of action and the necessary payload altogether for the reducer to act upon when dispatched. NgRx action uses the creator syntax given below:

createActionextends string, P extends object>(
type: T,
config: Props

)
props

extends object>(): Props

The createAction method is just a pure function that takes type i.e. a unique action name as its first parameter and config i.e. payload as its second parameter. With the first syntax in mind, we can write the NgRx action as follows. This expects the filterText property as part of the payload and then destructures it returning the key/value pair. However, this approach is a bit lengthy, especially if you have plenty of properties in the payload:

```
createAction(
'[Toolbar] Filter Bookmarks',
(payload: {filterText: string}) => ({filterText})
)
```

The alternate way uses the aforementioned props function which will automatically destructure the payload properties for you. This is the recommended way and we are going to use it throughout the book. For example, the definition of the filter bookmark action with a single property in the payload:

```
createAction(
'[Toolbar] Filter Bookmarks',
props<{filterText: string}>()
)
```

NgRx action creator methods are really handy in a true sense. They are introduced recently in the newer version of NgRx to create NgRx actions easily and effortlessly. The NgRx action creator, takes a name of the action you want to create and a props function which makes it a breeze to add properties without actual payload, sort of creating a harness for the payload with type-casting. Please note that the name of the action should be unique for debugging purposes hence it is ideal to prefix the name [Toolbar] Filter Bookmarks (it also indicates which store the action belongs to).

Creating your first action

Let us quickly generate our first action. Here, The action command tells the NgRx Schematics to generate an action file. The store/toolbar/toolbar is the path where we want to store the generated action file,

```
ng g action store/toolbar/toolbar --skipTests=true
```

Please answer the asked questions as mentioned below:

Q: Should we generate success and failure actions?

A: Type "No" and press Enter. These will be used in NgRx Effects in the next chapter though.

Q: Do you want to use the create function?

A: Type "Yes" and press Enter.

This command must have generated src/app/store/toolbar/toolbar.actions.ts which can be updated with:

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

```
export const FILTER_BOOKMARKS = createAction(
'[Toolbar] Filter Bookmarks',
props<{filterText: string}>()
);
```

When we will connect this action to the aforementioned toolbarReducer next, the root store filter will be updated with an actual search term that we may type in. But for it to happen, we must hook this action in.

Connecting your first action

We now have to import the same action and hook it up with toolbarReducer in src/app/store/toolbar.reducer.ts as follows:

```
import {FILTER_BOOKMARKS} from './toolbar.actions';

const toolbarReducer = createReducer(
initialState,
on(FILTER_BOOKMARKS, (_, {filterText}) => ({filterText}))
);
```

Few things to note here,

The on function associates actions with given state change functions. Here we are tapping into the type of action, and getting the provided payload, and returning the same as a new state which will be put into the store eventually. We have used ES6/ES2015 object property value shorthand syntax that we learned in the first chapter.

The _ refers to the existing state of the filter but we do not need it to formulate the new state.

Now that we have connected the action to the correct reducer in the previous section, the root store filter will not be updated immediately unless we dispatch the FILTER_BOOKMARKS action.

Dispatching your first action

Unlike the previous bit, dispatching the NgRx Action is a piece of cake. All we need to do is inject the Store object in the constructor of a component and dispatch FILTER_BOOKMARKS action with filterText as its parameter. So ideally, when we type into the filter input box, it should dispatch the same action with an updated filterText value as payload. But before that, we have to first change the way we filter the list of bookmarks. If you remember, we had data-bound Angular ngModel on the filter input box to update bookmarkService.filterText property in Now, we do not have to rely on the bookmark service to store the filter text since we have set up NgRx Store for the same.

Let us bind an input event on the filter textbox to trigger a handler callback, The input event will trigger the callback as you add or remove characters from the input box:

```
type="text"
(input)="onFilterChange($event.target.value)" matInput
placeholder="Filter bookmarks...">
```

Then, define the said method in the component class to dispatch the action by replacing bookmarkService with Store in

```
import {Component, ChangeDetectionStrategy} from
'@angular/core';
```

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

import {AppState} from '@app/store';
import {FILTER_BOOKMARKS} from
'@app/store/toolbar/toolbar.actions';

@Component({
selector: 'app-root',

templateUrl: './app.component.html',
styleUrls: ['./app.component.scss'],
changeDetection: ChangeDetectionStrategy.OnPush
})
export class AppComponent {
constructor(public readonly store$: Store) {}

onFilterChange(filterText) {
this.store$.dispatch(FILTER_BOOKMARKS({filterText}));
}
}
```

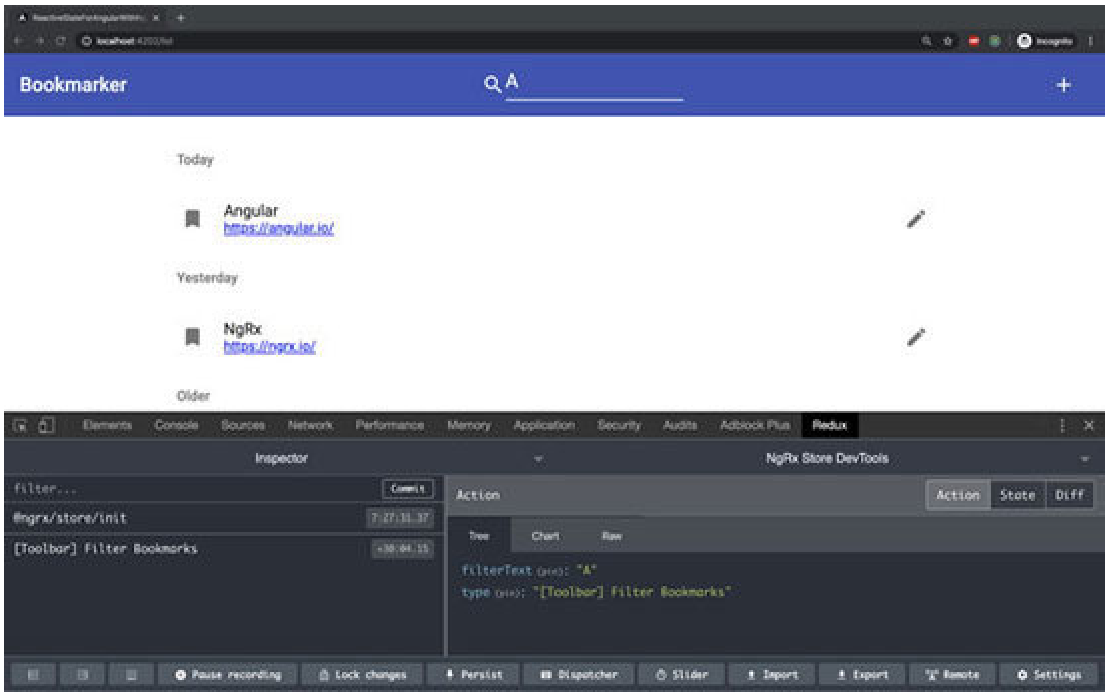
Few things to note here:

First, we import the Store from the NgRx NPM package.

Second, we instantiate store$ for being observable. The Store type is a typescript generic so we can pass any type of the state which gets type information when using selectors. Here, we have used AppState to be safe but using FilterState would have been fine too.

The store$ object has plenty of methods at its disposal and one of them is to dispatch the NgRx action, And obviously, the NgRx action is supposed to take params like filterText in this case.

Now go back to **http://localhost:4200** and type something in the filter textbox while paying attention to the Redux Devtools. As you see in *Figure* as I typed into the filter textbox, it had triggered [Toolbar] Filter Bookmarks action with filterText equals

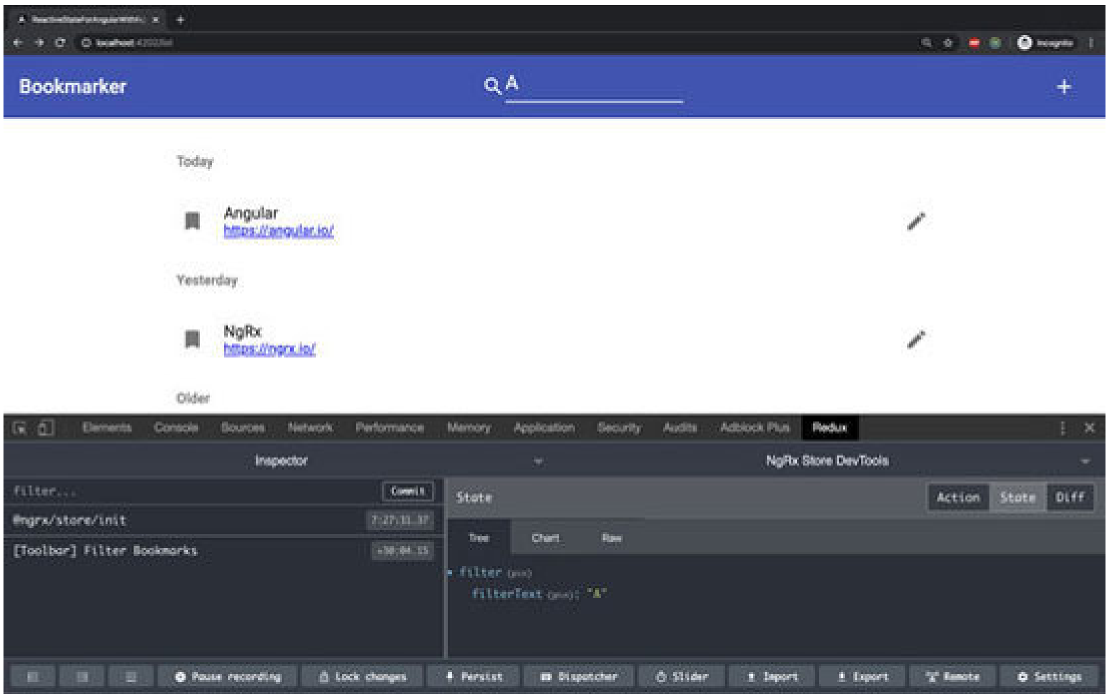


```
@ngrx/store/init                 7:27:31.37
[Toolbar] Filter Bookmarks       +30:04.15
```

```
filterText (pin): "A"
type (pin): "[Toolbar] Filter Bookmarks"
```

Figure 3.4: *NgRx Action dispatched*

Further, if you click on the State button on the right-hand side in the Redux Devtools, you will see that the application state has been updated as well - compare the *Figure 3.5* with the previous *Figure*



```
@ngrx/store/init                 7:27:31.37
[Toolbar] Filter Bookmarks       +30:04.15
```

```
filter (pin)
  filterText (pin): "A"
```

Figure 3.5: *Updated Application State after dispatching the NgRx action.*

You must have noticed already that the list of bookmarks did not filter out even though the application state had been updated. And that is because the template of ListComponent is still relying on the filterText property from the bookmark service in order to filter the list of bookmarks that we no longer modify. Instead, it should read filterText from the store, and this is what we will do next.

Root NgRx Selector

You must have been familiar with the approach of extracting value from an object, mainly via object property accessors. For your reference, the name property of const obj = {name: "NgRx"} can be read as obj.name. But it does not scale well if the object properties are too deeply nested. For example, consider the following response from some random API:

```
const state = {
data: {
AccountHolding: [{
accountCode: "3019205"
}]
}
}
```

If you are asked to read the accountCode property, you might read it this way:

```
state.data.AccountHolding[0].accountCode
```

To further complicate the matter, when you are asked to handle the null checks for all you might do something like this:

```
state &&
state.data &&
state.data.AccountHolding &&
state.data.AccountHolding[0].accountCode
```

This works but may go out of hand and quickly become error prone and hard to read. How about we create a selector function that will return the slice of the state for the next selector function to consume and return the slice of that for the next selector function to consume and so on. By the same token, we may create three selector functions in this case to get to the accountCode property as follows:

```
function getData(state) {return state && state.data;}
function getAccountHolding(data) {return data &&
data.AccountHolding;}
function getAccountCode(accountHolding) {
return accountHolding && accountHolding[0].accountCode;
}

console.log(getAccountCode(getAccountHolding(getData(state)))); //
returns 3019205
```

First, the getData method takes the original state variable that we had defined above and returns the value of the data property from it. Second, the getAccountHolding holds on to the data property we got back from the previous method to extract the AccountHolding property. Third, we pass the value of AccountHolding to pluck accountCode off of it. And finally, we run these functions right to left in a sequence to get what we need.

Creating your first selector

Isn't the above code looking scary to you!? That's where NgRx selector creators help. Similar to NgRx action creators, NgRx has selector creators that obviate the need for such custom functions. The NgRx selectors are nothing but pure functions used for obtaining slices of the application state. It takes n number of functions i.e other NgRx selectors as parameters. Each of those parameters or selector functions will be evaluated during run time and their values will be available in the last function in the form of its parameters in order to calculate the final value that it must return. This approach provides performance benefits using memoization, particularly with selectors that perform expensive computation.

Consider the following example where we have three selectors that simply return static values. The exported finalSelector selector composes the aforementioned selectors in a sequence and returns their evaluated values in the same sequence at the end. Then, we just concat the values and return a single comma separated string as a result:

```
const firstSelector = () => 'one'
const secondSelector = () => 'two'
const thirdSelector = () => 'three'
export const finalSelector = createSelector(
firstSelector,
secondSelector,
thirdSelector,
(one: string, two: string, third: string) => [one, two, third].join(',')
);
```

The NgRx createSelector function only allows 8 or fewer selectors to compose which is sufficient in most cases. However, you can create your selector to increase the maximum limit using

Now that we understand how to create a custom selector or compose multiple selectors to form a new one, it is time to write our first selector to get filterText off of the store. Unfortunately, Angular CLI does not have a provision to create the selector file unlike Action/Reducer so we have to manually create one for us. Let us create an empty toolbar.selectors.ts file in src/app/store/toolbar/ directory and update it as given below:

```
import {createSelector} from '@ngrx/store';

import {AppState} from '@app/store';
import {FilterState} from '@app/store/toolbar/toolbar.reducer';

export const selectFilter = (state: AppState) => state.filter;

export const getFilterText = createSelector(
selectFilter,
(filterState: FilterState) => filterState.filterText
);
```

Few things to note here:

We first read the root store property filter off of the state. The AppState interface was used just to avoid the type error while doing so.

The NgRx createSelector function takes the slice of the application state, that is, whatever is returned by the selectFilter function and then further extracts filterText from it. The FilterState interface was used for the same reason to avoid the type error.

Although, during run time in a browser, the actual application state, that is, the one we saw in *Figure 3.3* above will be passed down in the parameter of selectFilter and will return the filter object for getFilterText to consume in order to finally get

Although it looks like the return type will be a primitive type, that is, String in this case but it will be an Observable of Remember, NgRx is a reactive framework.

Using your first selector

The NgRx selector is a JavaScript function, callable with parameters. By the same token, we can directly call getFilterText selector by passing the selectFilter selector in the parameter which in turn further takes the AppState in its parameter. This approach, however, expects us to resolve the dependencies, which is neither optimal nor correct. We will run into the same problem that we faced while extracting accountCode in the previous section. Hence, NgRx uses the RxJS select method which lets us extract the slice of the state without much fuss. It takes either the name of the slice of the state or a selector function.

If we rely on the former approach then we have to run nested select methods to get filterText off of filter as follows:

```
import {select} from '@ngrx/store';
this.filterText$ = this.store$.select('filter').pipe(select('filterText'));
```

If we rely on the latter approach then we just have to pass the selector function we wrote previously as follows. This is the recommended way in NgRx though. We can make use of the selector in Here we have:

Injected store$ of type AppState into the constructor of the component.

Renamed filterText to filterText$ and changed the type to be an observable of string.

Run the selector getFilterText using the NgRx select method on the store. This will extract the state filterText from the root filter store.

```
import {Component, OnInit, ChangeDetectionStrategy} from
'@angular/core';

import {Store} from '@ngrx/store';
import {Observable} from 'rxjs';

import {BookmarkService, Bookmark} from
'@app/shared/services/bookmark/bookmark.service';
import {isToday, isYesterday} from '@app/shared/util';
import {AppState} from '@app/store';
import {getFilterText} from '@app/store/toolbar/toolbar.selectors';

@Component({
selector: 'app-list',
templateUrl: './list.component.html',
styleUrls: ['./list.component.scss'],
changeDetection: ChangeDetectionStrategy.OnPush
})
export class ListComponent implements OnInit {
allBookmarks: Bookmark[];
todaysBookmarks: Bookmark[];
yesterdaysBookmarks: Bookmark[];
olderBookmarks: Bookmark[];
filterText$: Observable;
constructor(
public readonly bookmarkService: BookmarkService,
public readonly store$: Store
) {}

ngOnInit() {
this.allBookmarks = this.bookmarkService.allBookmarks;


this.todaysBookmarks = this.allBookmarks.filter((bookmark) =>
isToday(bookmark.created));
this.yesterdaysBookmarks = this.allBookmarks.filter((bookmark) =>
isYesterday(bookmark.created));
this.olderBookmarks = this.allBookmarks.filter((bookmark) => {
return !this.todaysBookmarks.find((b) => b.id === bookmark.id) &&
!this.yesterdaysBookmarks.find((b) => b.id === bookmark.id);
});
this.filterText$ = this.store$.select(getFilterText);
}
}
```

Since the filterText$ class member is an observable now, we have to subscribe to it directly in the Angular template using an async pipe and then pass the resolved value filterText into the fuzzy pipe. So, update the first mat-list block in src/app/list/list.component.html with the following as follows:

```
*ngIf="filterText$ | async as filterText; else groupedBookmarks">
```

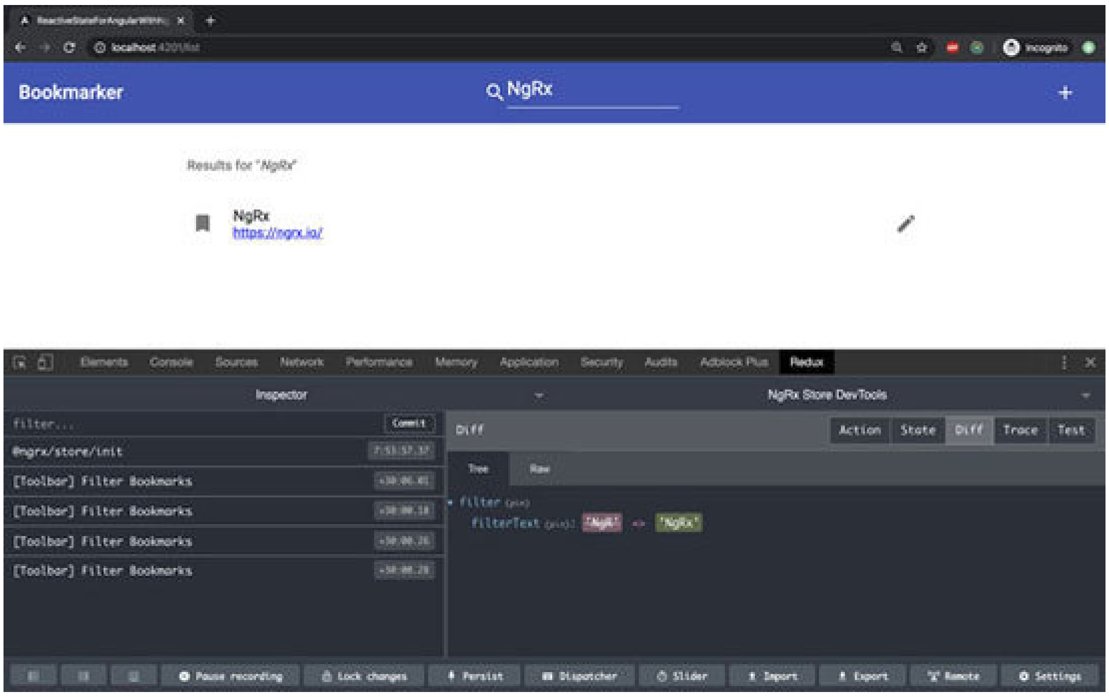
mat-subheader>Results for "*{{filterText}}*"

*ngFor="let allBookmark of allBookmarks | fuzzy: filterText">
mat-list-icon>bookmark

mat-line>{{allBookmark.name}}

[href]="allBookmark.url" mat-line> {{allBookmark.url}}
[routerLink]="['/edit', allBookmark.id]">mat-list-icon>edit

Now if you go to **http://localhost:4200** and type NgRx into the filter input box, you will see the list of bookmarks being filtered correctly as per [Figure](#)



```
@ngrx/store/init                 7:51:57.37
[Toolbar] Filter Bookmarks       +30:06.01
[Toolbar] Filter Bookmarks       +30:00.18
[Toolbar] Filter Bookmarks       +30:00.26
[Toolbar] Filter Bookmarks       +30:00.28
```

```
▾ filter (pin)
  filterText (pin): 'NgR' => 'NgRx'
```

Figure 3.6: *Filtering a list of bookmarks*

There are four filter actions triggered for 4 characters NgRx we typed in which may not be efficient if we were to make a remote API call each time. In that case, you could use debounce or delay or distinctUntilChanged RxJS operators to reduce the number of API calls to make the search functionality more robust.

What is the feature store?

In the previous section, you may have noticed that the root store, filter, was created with the empty filterText value as soon as the application booted up in the browser. Then we could search for the bookmarks and update the value of filterText (by typing into the filter input box) to filter the list of bookmarks. In this case, the filter input box as well as the list of bookmarks were hydrated/available on page load itself. However, in certain situations, you might lazy load components using Angular Router and the data on which the state depends will not be available immediately. So in this case, you have to first lazy load the route, hydrate associated components, instantiate related services/pipes and then create the store for it. However, the root store does not cater to this requirement which is why the NgRx team has come up with the **Feature Store.**

The feature store is no different than the root store in anyway except that it is created by the lazy loaded module and not the root module i.e. AppModule in our case. When the feature store is instantiated then its slice of the store is combined with the root store making up the entire application state. Note that the feature store in *Figure 3.7* below is highlighted with dashed boxes:

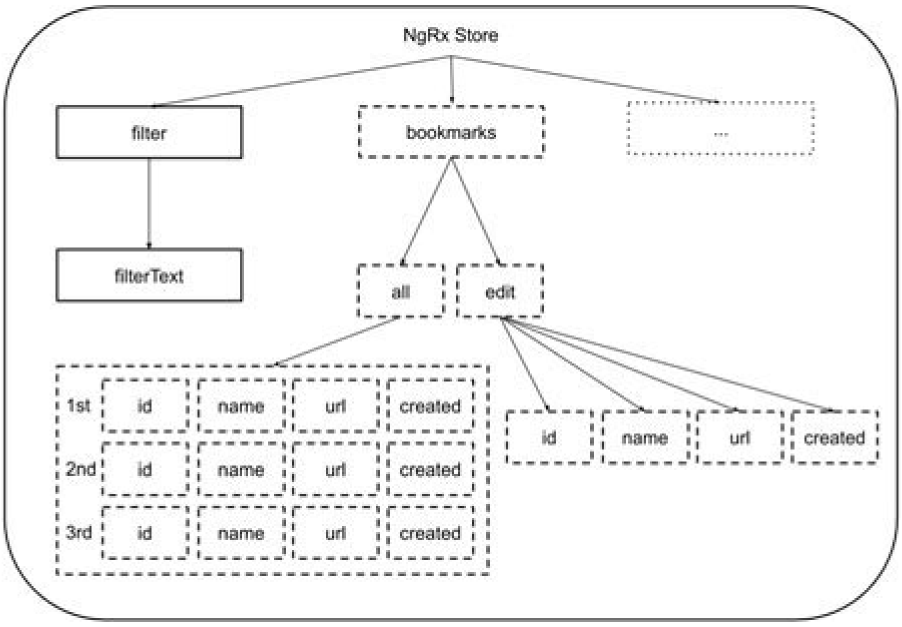


Figure 3.7: *Feature Store "bookmarks" in NgRx*

Time has come to go hands-on and create our first feature store using the following command. Few things to note:

The --stateInterface flag expects the type of the state,

The --root flag is not needed since we want it to be a feature store.

The --statePath flag creates src/app/store/bookmarks directory to save the generated files.

The --module flag automatically plugs the feature store in ListModule.

```
ng g store --stateInterface BookmarksState --statePath
store/bookmark --module list/list.module.ts
```

Please answer the asked questions as mentioned below:

Q: What should be the name of the state?

A: Type "bookmarks" and press Enter. This will become the key for the feature store as per Figure 3.7 above.

You should see the following snippet in the generated src/app/store/bookmark/index.ts file:

```
import {
ActionReducer,
ActionReducerMap,
createFeatureSelector,
createSelector,
MetaReducer
} from '@ngrx/store';
import {environment} from '../../../environments/environment';

export const bookmarksFeatureKey = 'bookmarks';

export interface BookmarksState {
}

export const reducers: ActionReducerMap = {
};
```

```
export const metaReducers: MetaReducer[] =
!environment.production ? [] : [];
```

Above file is quite similar to the one we created for the root store earlier. Except that it has exported the feature key named This will help us hook up the feature store to the previously created root store on the fly:

```
import {NgModule} from '@angular/core';
import {CommonModule} from '@angular/common';
import {MatListModule} from '@angular/material/list';
import {MatIconModule} from '@angular/material/icon';
import {ListRoutingModule} from '@app/list/list-routing.module';
import {ListComponent} from '@app/list/list.component';

import {FuzzyPipe} from '@app/shared/pipes/fuzzy.pipe';
import {StoreModule} from '@ngrx/store';
import * as fromBookmarks from '../store/bookmark';

@NgModule({
declarations: [ListComponent, FuzzyPipe],
imports: [
CommonModule,
ListRoutingModule,
MatListModule,
MatIconModule,
StoreModule.forFeature(fromBookmarks.bookmarksFeatureKey,
fromBookmarks.reducers, {metaReducers:
fromBookmarks.metaReducers})
]
})
```

```
export class ListModule {}
```

With a proper distinction, Angular CLI has injected the feature store using forFeature (instead of method into ListModule in

Feature NgRx Reducer

Unlike the root NgRx reducer, the feature NgRx reducer should be hooked into the feature store. In this section, we will create the feature reducer using the same command we had used earlier to create the root reducer. Let us create our first feature reducer using the following command:

```
ng g reducer store/bookmark/bookmark --skipTests=true
```

Please answer the asked questions as mentioned below:

Q: Should we add success and failure actions to the reducer?

A: Type "No" and press Enter. These will be used in NgRx Effects in the next chapter though.

Q: Do you want to use the create function?

A: Type "Yes" and press Enter.

The aforementioned command must have generated the following code in src/app/store/bookmark/bookmark.reducer.ts. If the generated code does not match line-by-line for you, it is fine since we will modify it anyway:

```
import {Action, createReducer, on} from '@ngrx/store';

export const bookmarkFeatureKey = 'bookmark';

export interface State {}

export const initialState: State = {};

const bookmarkReducer = createReducer(
initialState,
);

export function reducer(state: State | undefined, action: Action) {
return bookmarkReducer(state, action);
}
```

You may have already noticed that this code is quite similar to the feature store we created in And that's because both files are reducers files, however, we will use this file to keep the reducer functions only for brevity. Alternatively, you can merge both the files if needed. Let us update src/app/store/bookmark/bookmark.reducer.ts as follows:

```
import {createReducer, on} from '@ngrx/store';

import {Bookmark} from
'@app/shared/services/bookmark/bookmark.service';
```

```
export const initialAllBookmarksState: Bookmark[] = [];

export const initialEditBookmarkState: Bookmark | null = null;

export const allBookmarksReducer = createReducer(
initialAllBookmarksState,
// tap into NgRx actions to mutate the state
);

export const editBookmarkReducer = createReducer(
initialEditBookmarkState,
// tap into NgRx actions to mutate the state
);
```

Few things to note here:

First of all, we have defined the initial states. For /list route, it's an empty array and for /edit route, it is null.

Then we passed the initial states to the reducer functions.

Later on, we will plug in the necessary actions (LOAD_ALL_BOOKMARKS and EDIT_BOOKMARK) to update the default states respectively once we create them in the next section. For now, we have just added two comments there.

Now connect these two reducer functions to the feature store as follows in src/app/store/bookmark/index.ts.

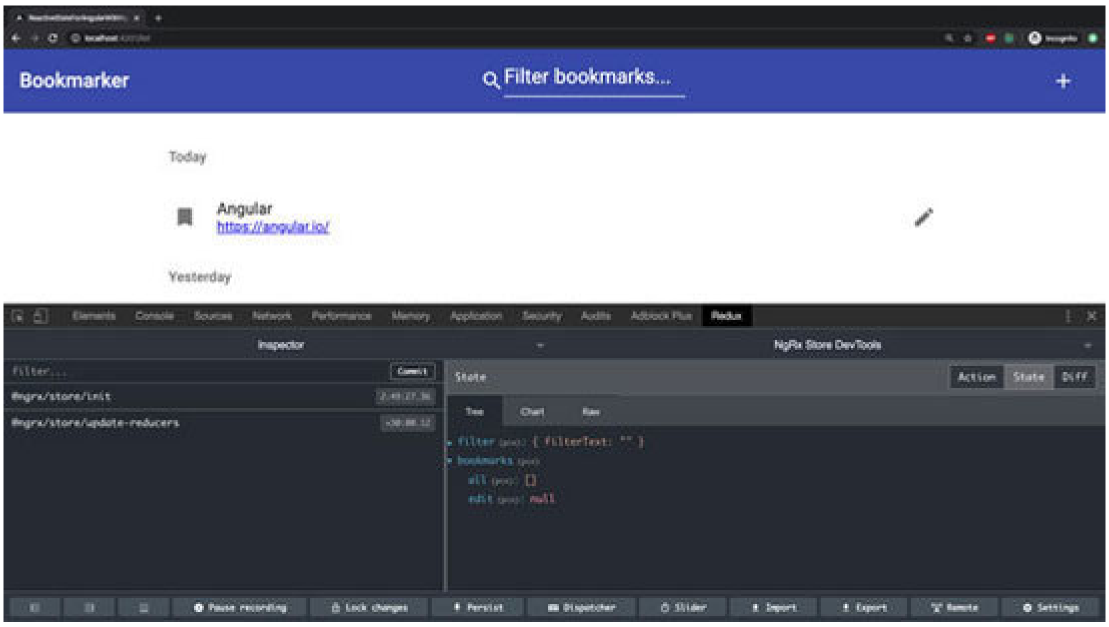
```
import {
Action,
ActionReducerMap,
MetaReducer
} from '@ngrx/store';
import {environment} from '../../../environments/environment';
import {Bookmark} from
'@app/shared/services/bookmark/bookmark.service';
import * as fromBookmark from
'@app/store/bookmark/bookmark.reducer';

export const bookmarksFeatureKey = 'bookmarks';

export interface BookmarksState {
all: Bookmark[];
edit: Bookmark;
}
export const reducers: ActionReducerMap = {
all: (state: Bookmark[], action: Action) =>
fromBookmark.allBookmarksReducer(state, action),
edit: (state: Bookmark, action: Action) =>
fromBookmark.editBookmarkReducer(state, action),
};

export const metaReducers: MetaReducer[] =
!environment.production ? [] : [];
```

For readability purposes, it is recommended to import all the exports from src/app/store/bookmark/bookmark.reducer.ts as So we hooked:

The states for all and edit keys into

Bookmark reducer functions into the action reducer map, reducers.

This will automatically inject the initial state (that we defined earlier) in the bookmarks feature store on page load, that is, an empty array of bookmarks in, all, property, and the null value in edit property. Just reload **http://localhost:4200** to see it in action as per [Figure]()



```
@ngrx/store/init
@ngrx/store/update-reducers
```

```
▸ filter (pin): { filterText: "" }
▾ bookmarks (pin)
    all (pin): []
    edit (pin): null
```

Figure 3.8: *Feature store "bookmarks" with the initial state*

The state for all the bookmarks is empty but we still see a list of bookmarks rendered on top. That means our state is not in sync with the UI. Let us fix it with the feature NgRx action next.

Feature NgRx Action

Ideally, there is no difference between the root NgRx action and the feature NgRx action. It is just that they will be used by the root store and the feature store respectively. Let us create a feature NgRx action using the following command:

ng g action store/bookmark/bookmark --skipTests=true

Please answer the asked questions as mentioned below:

Q: Should we generate success and failure actions?

A: Type "No" and press Enter.

Q: Do you want to use the create function?

A: Type "Yes" and press Enter.

This command must have generated src/app/store/bookmark/bookmark.actions.ts which can be updated with the following snippet. Here, we have defined two NgRx actions namely, LOAD_ALL_BOOKMARKS and Both the actions take a single parameter when invoked mainly bookmarks and bookmark respectively. This simply means that when we invoke, say, LOAD_ALL_BOOKMARKS action from a component, we'll pass an array of bookmarks in its parameter:

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

import {Bookmark} from
'@app/shared/services/bookmark/bookmark.service';

export const LOAD_ALL_BOOKMARKS = createAction(
'[Bookmark] Load All Bookmarks',
props<{bookmarks: Bookmark[]}>()
);

export const EDIT_BOOKMARK = createAction(
'[Bookmark] Edit Bookmark',
props<{bookmark: Bookmark}>()

);
```

Please note that the name of the action should be unique for debugging purposes hence it is ideal to prefix the name [Bookmark] Load All Then, we have to put some intelligence into the feature store to save the incoming data in parameters. For that, we will replace the previously added comments in src/app/store/bookmark/bookmark.reducer.ts as follows:

```
import {createReducer, on} from '@ngrx/store';

import {Bookmark} from
'@app/shared/services/bookmark/bookmark.service';
```

```
import {LOAD_ALL_BOOKMARKS, EDIT_BOOKMARK} from
'@app/store/bookmark/bookmark.actions';

export const initialAllBookmarksState: Bookmark[] = [];
export const initialEditBookmarkState: Bookmark | null = null;

export const allBookmarksReducer = createReducer(
initialAllBookmarksState,
on(LOAD_ALL_BOOKMARKS, (_, {bookmarks}) => bookmarks)
);

export const editBookmarkReducer = createReducer(
initialEditBookmarkState,
on(EDIT_BOOKMARK, (_, {bookmark}) => bookmark)
);
```

Few things to note here:

The on function associates actions with given state change functions. Here we are tapping into the type of action, and getting the provided payload, bookmarks, and returning the same as a new state which will then be saved into the feature store.

The _ refers to the existing state of all/edit but we do not need it to formulate the new state.

We have decided to use the same feature store for both /list and /edit routes but you can make a separate feature store for each lazy-loaded module if needed.

Feature NgRx Selector

Now that we have connected the feature actions with the feature reducers, it is time to read the data off of the feature store and use it in the component. Just like we could not use forRoot method to hook up the feature store to the application state (instead used forFeature method if you remember), we cannot use createSelector method to query the feature store. Instead of that, we should use the createFeatureSelector method which allows us to get a top-level feature store property, bookmarks. However, to further extract the slices of the feature store we can continue to use the createSelector method as before.

Let us create an empty bookmark.selectors.ts file in the src/app/store/bookmark/ directory and update it as given below.

```
import {createFeatureSelector, createSelector} from '@ngrx/store';

import {BookmarksState} from '@app/store/bookmark';

const getBookmarksState = createFeatureSelector('bookmarks');

export const getAllBookmarks = createSelector(
getBookmarksState,
state => state.all
);
```

```
export const getEditBookmark = createSelector(
getBookmarksState,
state => state.edit
);
```

Few things to note here:

We first queried the feature store bookmarks off of the application state using

We then used the NgRx createSelector function to extract the values of all and edit properties off of the bookmarks state.

Let us make use of these selectors in src/app/list/list.component.ts as follows:

```
import {Component, OnInit, ChangeDetectionStrategy} from
'@angular/core';
import {Store} from '@ngrx/store';
import {Observable} from 'rxjs';
import {map} from 'rxjs/operators';

import {Bookmark} from
'@app/shared/services/bookmark/bookmark.service';
import {isToday, isYesterday} from '@app/shared/util';
import {AppState} from '@app/store';
import {getFilterText} from '@app/store/toolbar/toolbar.selectors';
import {getAllBookmarks} from
'@app/store/bookmark/bookmark.selectors';

@Component({
selector: 'app-list',
templateUrl: './list.component.html',
styleUrls: ['./list.component.scss'],
changeDetection: ChangeDetectionStrategy.OnPush
})
export class ListComponent implements OnInit {
allBookmarks$: Observable;
todaysBookmarks$: Observable;
yesterdaysBookmarks$: Observable;
olderBookmarks$: Observable;

filterText$: Observable;

constructor(public readonly store$: Store) {}

ngOnInit() {
this.allBookmarks$ = this.store$.select(getAllBookmarks);
this.todaysBookmarks$ = this.allBookmarks$.pipe(
map(allBookmarks => allBookmarks.filter((bookmark) =>
isToday(bookmark.created)))
);
this.yesterdaysBookmarks$ = this.allBookmarks$.pipe(
map(allBookmarks => allBookmarks.filter((bookmark) =>
isYesterday(bookmark.created)))
);
this.olderBookmarks$ = this.allBookmarks$.pipe(
map(allBookmarks => allBookmarks.filter((bookmark) =>
!isToday(bookmark.created) && !isYesterday(bookmark.created)))
);
this.filterText$ = this.store$.select(getFilterText);
}
}
```

Here we have:

Injected store$ of type AppState into the constructor of the component.

Changed the type for all the variables up top to be observables since the NgRx selector returns the observable value.

Run the getAllBookmarks selector using the RxJS select method on the store. This will extract the state all from the bookmarks feature store.

Mapped all the bookmarks using the RxJS map method to filter todays, yesterdays, and older bookmarks.

With the above changes in mind, the template of the component must be yelling at you! And it is obvious because the template did not expect the observable values before. To resolve the observables in the template itself, we can leverage the async pipe. Let us update the entire src/app/list/list.component.html file with:

```
*ngIf="filterText$ | async as filterText; else groupedBookmarks">
```

mat-subheader>Results for "*{{filterText}}*"

***ngFor="let allBookmark of allBookmarks$ | async | fuzzy: filterText">**
mat-list-icon>bookmark

mat-line>{{allBookmark.name}}

[href]="allBookmark.url" mat-line> {{allBookmark.url}}
[routerLink]="['/edit', allBookmark.id]">mat-list-icon>edit

#groupedBookmarks>

mat-subheader>Today

***ngFor="let todaysBookmark of todaysBookmarks$ | async">**
mat-list-icon>bookmark

mat-line>{{todaysBookmark.name}}

[href]="todaysBookmark.url" mat-line> {{todaysBookmark.url}}
[routerLink]="['/edit', todaysBookmark.id]">mat-list-icon>edit

mat-subheader>Yesterday

***ngFor="let yesterdaysBookmark of yesterdaysBookmarks$ | async">**
mat-list-icon>bookmark

mat-line>{{yesterdaysBookmark.name}}

[href]="yesterdaysBookmark.url" mat-line>
{{yesterdaysBookmark.url}}

[routerLink]="['/edit', yesterdaysBookmark.id]">mat-list-icon>edit

mat-subheader>Older

*ngFor="let olderBookmark of olderBookmarks$ | async">
mat-list-icon>bookmark

mat-line>{{olderBookmark.name}}

[href]="olderBookmark.url" mat-line> {{olderBookmark.url}}
[routerLink]="['/edit', olderBookmark.id]">mat-list-icon>edit

Now if you reload you will notice that the list of bookmarks is gone and not even the JS exception in the console is visible. What went wrong? Nothing. Notice that the feature store is empty and so does the UI - both the store and UI are in sync. However, the only problem is that the feature store is empty because of the initial state we filled it with. Let us update it with the actual data next.

Updating Feature Store

If you remember, we fetched a list of bookmarks over the wire in list.guard.ts and stored it against the allBookmarks property from the bookmark service. We do not need that property anymore since we can save the same in the feature store directly. All we need is to dispatch the LOAD_ALL_BOOKMARKS action with the list of bookmarks in the payload. For that, let us update

```
import {Injectable} from '@angular/core';
import {CanActivate} from '@angular/router';
import {Observable, of} from 'rxjs';
import {first, switchMap, map, catchError} from 'rxjs/operators';
import {MatDialog} from '@angular/material/dialog';
import {Store} from '@ngrx/store';

import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';
import {AppState} from '@app/store';
import {LOAD_ALL_BOOKMARKS} from
'@app/store/bookmark/bookmark.actions';

@Injectable({
providedIn: 'root'
})
```

```
export class ListGuard implements CanActivate {
constructor(
private readonly bookService: BookmarkService,
private readonly dialog: MatDialog,

public readonly store$: Store
) {}

canActivate(): Observable {
return this.bookService.getAll().pipe(
first(),
map((bookmarks) =>
this.store$.dispatch(LOAD_ALL_BOOKMARKS({bookmarks}))),
switchMap(() => this.activateRoute()),
catchError(() => {
this.dialog.open(ErrorDialogComponent, {
width: '400px',
data: {errorMessage: 'Sorry, unable to fetch bookmarks.'}
});
return of(false);
})
);
}

activateRoute() {
return of(true);
}
}
```

Again we have:

Injected store$ of type AppState into the constructor of the guard.

Dispatched the LOAD_ALL_BOOKMARKS action with the payload we received from the bookmark service.

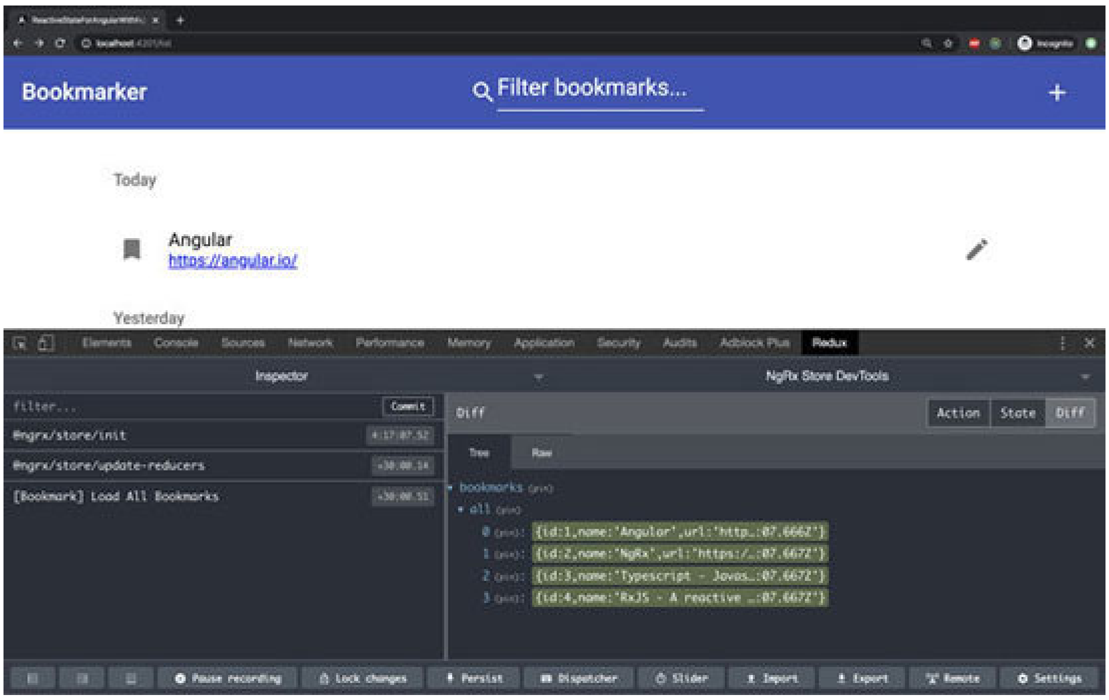
That's it. Now the NgRx flow for the /list route would be like,

Fetch the list of bookmarks over the wire in

Save the list of bookmarks into the feature store by dispatching the LOAD_ALL_BOOKMARKS action.

Extract the list of bookmarks off of the feature store using the getAllBookmarks selector in src/app/list/list.component.ts to render the list of bookmarks in the UI.

Reload **http://localhost:4200** to see the list of bookmarks as before. Also, pay attention to the NgRx store which is now in sync with the UI as per *Figure*



```
@ngrx/store/init
@ngrx/store/update-reducers
[Bookmark] Load All Bookmarks
```

```
bookmarks
  all
    0: {id:1,name:'Angular',url:'http…:07.666Z'}
    1: {id:2,name:'NgRx',url:'https:/…:07.667Z'}
    2: {id:3,name:'Typescript - Javas…:07.667Z'}
    3: {id:4,name:'RxJS - A reactive …:07.667Z'}
```

Figure 3.9: *Feature store "bookmarks" with the actual state*

Similarly, you can dispatch the EDIT_BOOKMARK action from src/app/edit/edit.guard.ts to update the edit slice when any bookmark is being edited. But I will not cover it in the chapter since it's quite easy to do. You can consider it as a task for yourself to learn by doing. However, the working solution will be included in the companion codebase provided with this book, in case you are stuck.

Conclusion

In this chapter, we have gained an understanding of root as well as feature store in NgRx. We learned how lazy-loaded Angular modules can leverage the feature store to sanely scale the NgRx store. We improved the existing application that we built in the previous chapter by making use of the NgRx store without relying on the singleton Angular bookmark service. We purposefully used the root store for the filter bookmarks functionality and then created the feature store for the list (and edit) bookmark routes so that we can learn both the approaches in detail. Along the way, NgRx schematics helped us a long way for scaffolding the boilerplate code needed to create and manage the store.

In a nutshell, we learned about NgRx architecture as a whole, NgRx Schematics to scaffold NgRx files over the command line, when to use root store versus feature store, and how to leverage action -> dispatcher -> reducer pipeline in your projects to deal with synchronous data flows.

In the next chapter, we will explore the possibility of using a similar action -> dispatcher -> reducer pipeline while creating/updating bookmarks. But there we call the asynchronous APIs which we could not move into the reducer, otherwise, we will break the third rule of Redux. That's when NgRx Effects come into the picture.

Questions

What is the NgRx root store?

What is the NgRx feature store?

When to prefer a root or feature store?

How to install and use NgRx Schematics?

How to create NgRx Reducer?

How to create NgRx Action?

How to create NgRx Selector?

CHAPTER 4

NgRx Effects

In the previous chapter, we firmly rooted the notion that NgRx Store is merely a JSON object split into two halves: *Root Store* and *Feature Store* . You can think about it as the orchestrator through which new states are pushed and combined via lazy-loaded Angular routes. The reason for these discussions is that other than asynchronous events, most of your applications will involve the use of the same NgRx patterns we learned so that Angular components can be clear of the business logic and just dispatch necessary actions to morph the data to save it in the store eventually.

Now you have entered into new, more sophisticated territory. This chapter is our point of inflection where we will look at handling asynchronous events using the action-dispatcher-reducer pipeline and will see what problem we face before going towards the remedy. Brave developers like you would at least try to implement their learnings from the previous chapter to the untouched parts of the application such as create bookmarks and update bookmark journeys. For instance, a user may submit the create bookmark button on the /create route, and instead of calling the API directly to save a new bookmark, (s)he may dispatch an action and let an associated reducer call the aforementioned API to save the bookmark. But there is a catch! If you remember the third rule of Redux (i.e. Mutate State using Pure functions only) from the 1st chapter then you will notice that you have flouted the rule by calling the API (an asynchronous event) from within the reducer (a pure function). By the same token, the API became the side-effect in this scenario. This happens quite often with developers, especially when they are new to NgRx. Thus, in this chapter, we will focus on how to handle side-effects in NgRx using an RxJS powered side-effect model for NgRx Store called **NgRx**

Structure

In this chapter, we will leverage @ngrx/effects package to scaffold or auto-generate NgRx Effects files. We will use the side-effect model in the existing create/update bookmark journeys:

What are the side effects?

Installing @ngrx/effects package

What is feature NgRx effects?

What is root NgRx effects?

Objectives

We will understand the concept behind the side-effect model and create our very first Effect in NgRx to integrate it in the sample application.

What are the side effects?

A function is said to have a side-effect if it modifies some state variable value outside its local environment. That is to say that it has an observable effect besides returning a value. Example side effects include modifying a non-local variable, modifying a static local variable, modifying a mutable argument passed by reference, performing I/O, or calling other side-effect functions. Even Functional Programming is based on the very premise that your functions should not have side effects since they are considered evil in the paradigm of pure functions. In the NgRx world, asynchronous events are considered to be the side effects and hence cannot be tackled with a traditional action-dispatcher-reducer pipeline that we learned about in the previous chapter.

Being evil does not mean that side-effects are not needed. In fact, without them, our functions will do only calculations. However, at some point in time, we must do something with the calculated value such as write them to Databases, write them to Files, send them to external systems over the wire, etc. Thus, NgRx provides built-in support for an RxJS powered side-effect model to tackle such side-effects effectively in NgRx with the help of With the NgRx Effects in place, as shown in *Figure 4.1* below, a component dispatches an action to update the store but before a reducer would take charge of the dispatched action, an effect intercepts the action and executes the side-effect and then dispatches a new action (notice double arrows between action and effects below) which the reducer this time would tap into to finally update the store:

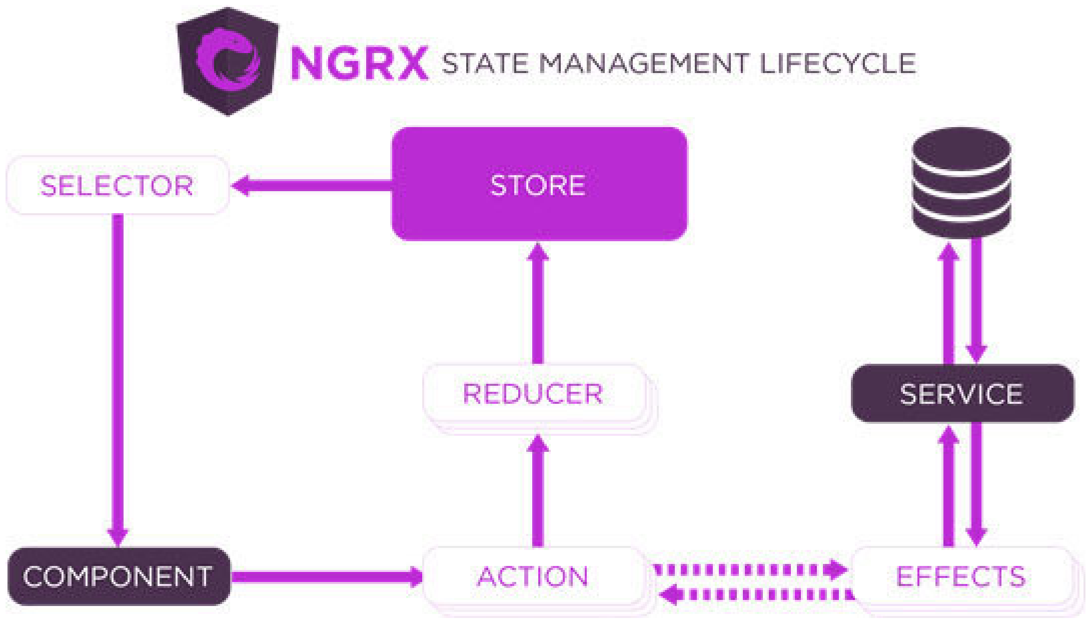


Figure 4.1: *NgRx Effects*

If you pay more attention then you will notice that we have successfully side-lined the side-effects without compromising the action-dispatcher-reducer pipeline. In fact, we have used the same pipeline twice. First, when the component dispatches an action. Second, when the Effect (after executing the side-effect) dispatches another action. On another note, keep in mind that with NgRx Effects in place you try to avoid a reducer control whenever possible. Like when the component dispatched the action in our example here, we deliberately did not have a reducer for that action because we did not want to update the store before executing the side-effect. However, when the effect dispatched the new action, we did have a reducer for it to update the store with whatever value we may get back from the side-effect run. Now, why you should avoid unnecessary reducer controls as much as possible has more to do with the notion of avoiding abnormal behavior with store updates as well as to continue the fluent flow in NgRx pipeline. In cases where you absolutely need to update the store even before the side-effect runs, you can do so but with caution.

Installing @ngrx/effects package

In the sample application that we built in the second chapter, components are responsible for interacting with an external resource, angular-in-memory-web-api directly through a service called However, NgRx effects provide a way to interact with those services and isolate them from the components, enabling the usage of more pure components that just select states and dispatch actions. NgRx effects are like long-running services that listen to every action dispatched in order to intercept them and run necessary side-effects. For that to happen, NgRx effects intercept actions based on their types using an operator called And at the end, NgRx effects perform side-effects, which may be synchronous or asynchronous events and return a new action as stated before they complete the loop by updating the store.

Prior to starting the show, let us first install the NgRx effects package in a terminal.

```
ng add @ngrx/effects
```

This must-have updated angular.json for Angular CLI to scaffold NgRx effects files later on. This must-have added EffectsModule in src/app/app.module.ts as follows:

```
EffectsModule.forRoot([])
```

From what you had learned about NgRx Store in the previous chapter, NgRx Effects too work in the same way. That is to say that there can be Root as well as Feature Effects in NgRx. Just like the feature store, the feature effects would not work without having the root effects in place which is why the above command had auto-magically hooked up the root effects in the application module by default. Since we have not scaffolded the root effects yet, it has been using an empty array there.

The EffectsModule.forRoot([]) method must be added to your application module even though you do not register any root-level effects.

Although we are going to touch the same files while creating root and feature effects in this chapter, I want to restrict the redundant movement of the code to not confuse the readers. Hence, we will first start with feature effects and then end the chapter with root effects in place. We are now off to create the feature NgRx Effects in the next section.

What is feature NgRx effects?

The feature NgRx Effects creates a provision to handle side-effects when the feature store is being updated via NgRx actions/reducers. Having said that, it has to be added to the feature module using EffectsModule.forFeature([]) so that it will be ready to tap feature-module-level NgRx actions that our application may dispatch. If you skim through the code that we have written so far, you will notice that in a couple of components, mainly create and edit components, we are saving the bookmark details using the BookmarkService service over the wire. Since the used APIs are asynchronous, we cannot use the traditional action-dispatcher-reducer pipeline there and that is why it is the perfect problem to resolve with the NgRx Effect model. Thus, let us create our very first NgRx Effects next.

Creating Feature NgRx Effects

Since we had installed the NgRx Schematics in the previous chapter, we can leverage Angular CLI to create our first NgRx Effects using the following command. Few things to note:

The effect command argument generates the effect file.

The path defines the name of the effect file and where it's going to be created. We want to create bookmark.effects.ts in the src/app/store/bookmark/ directory.

The --module flag automatically plugs the feature effects into the CreateModule feature module.

The --skipTests flag prevents generating the spec files since we are not covering any tests in this book.

```
ng g effect store/bookmark/bookmark --
module=create/create.module.ts --skipTests=true
```

Please answer the asked questions as mentioned below:

Q: Should we wire up success and failure actions?

A: Type "Yes" and press Enter.

Q: Do you want to use the create function?

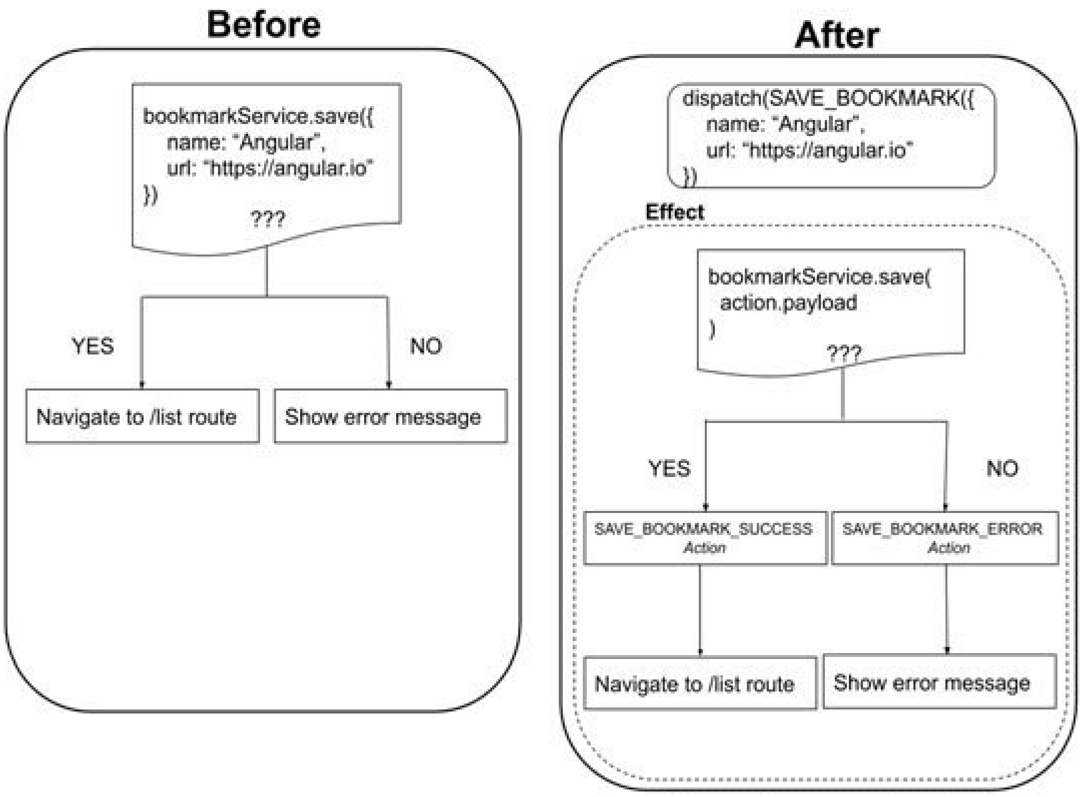
A: Type "Yes" and press Enter. This will import a createEffect function for us to set up our first effect.

You should see the following snippet in the src/app/store/bookmark/bookmark.effects.ts file:

```
import {Injectable} from '@angular/core';
import {Actions, createEffect} from '@ngrx/effects';

@Injectable()
export class BookmarkEffects {
constructor(private actions$: Actions) {}
}
```

Before we modify the aforementioned auto-generated effects file to suit our needs, we must understand how the data flow would change with the NgRx effect in place.

In *Figure 4.2* given below, the before-section depicts the existing implementation of our business logic and data flow in the component while creating a new bookmark. In which, we simply call the bookmark service to save a new bookmark, and depending on the response of the API call, we perform certain actions i.e. navigating to the /list route or showing the error message. But this setup is less ideal since the business logic sits inside the component. However, there is a possibility of moving the business logic out of the component which is what we intend to do:



Before
bookmarkService.save({ name: "Angular", url: "https://angular.io" })
???
YES
NO
Navigate to /list route
Show error message
After
dispatch(SAVE_BOOKMARK({ name: "Angular", url: "https://angular.io" }))
Effect
bookmarkService.save(action.payload)
???
YES
NO
SAVE_BOOKMARK_SUCCESS Action
SAVE_BOOKMARK_ERROR Action
Navigate to /list route
Show error message

Figure 4.2: *NgRx Effects in action to create bookmark journey*

There are many ways to isolate the business logic out of the component to make it a pure/dump component i.e. devoid of business logic. But since we are exploring NgRx Effect in this chapter, we will take the effects route to conquer the said problem. We will look at the following strategies:

Expressing the intent with dispatching NgRx Effect: This strategy is useful for the component to decide what data to save since it has hold of it via Angular FormBuilder and dispatch the same using SAVE_BOOKMARK action for an effect to tap into.

Manage positive/negative outcome with non-dispatching NgRx Effect: This one is used when the bookmark is successfully created or not saved for some reason. And a desperate action needs to be taken such as navigating to the /list route or an error message that needs to be shown respectively.

Let us make these changes in the application and see how it goes.

Expressing the intent with dispatching NgRx Effect

If you take a look at the src/app/create/create.component.ts file right now, you will notice that it has a lot of clutter lying just there such as BookmarkService to save the bookmark, Router service to manage the aforementioned positive outcome, and ErrorDialogComponent to handle the error while doing so. All of this just bloats the component for no apparent reason and can be moved out elsewhere so that the sole purpose of the component will become just to express the intent for saving a new bookmark. The simplest solution for that is to use NgRx action to express the intent.

First of all, create a new action named SAVE_BOOKMARK which can take a payload of type Bookmark as its parameter in src/app/store/bookmark/bookmark.actions.ts as follows:

```
export const SAVE_BOOKMARK = createAction(
'[Bookmark] Save Bookmark',
props<{bookmark: Bookmark}>()
);
```

Then use the defined action to express the intent in src/app/create/create.component.ts by removing the above mentioned clutter from the component. Now CreateComponent should look slim and trim as follows:

```
import {Component, ChangeDetectionStrategy} from
'@angular/core';
import {FormBuilder, Validators, FormGroup} from
'@angular/forms';
import * as moment from 'moment';
import {Store} from '@ngrx/store';

import {AppState} from '@app/store';

import {SAVE_BOOKMARK} from
'@app/store/bookmark/bookmark.actions';

@Component({
selector: 'app-create',
templateUrl: './create.component.html',
styleUrls: ['./create.component.scss'],
changeDetection: ChangeDetectionStrategy.OnPush
})
export class CreateComponent {
bookmarkForm: FormGroup;

constructor(
private fb: FormBuilder,
public readonly store$: Store
) {
this.bookmarkForm = this.fb.group({
name: ['', Validators.required],
url: ['', Validators.required]
});
}
onSubmit() {
this.store$.dispatch(SAVE_BOOKMARK({bookmark: {...
this.bookmarkForm.value, created: moment().toDate()}}));
}
}
```

All we have done here is injected the Store object to dispatch the SAVE_BOOKMARK action with the same old payload. However, the bookmark will not be saved yet since we have removed the business logic. Again, you cannot move the earlier business logic in the reducer because it was an asynchronous API call. Thus, we will move it in src/app/store/bookmark/bookmark.effects.ts as follows:

```
import {Injectable} from '@angular/core';
import {Actions, createEffect, ofType} from '@ngrx/effects';
import {exhaustMap, map, catchError} from 'rxjs/operators';
import {of} from 'rxjs';

import {SAVE_BOOKMARK, SAVE_BOOKMARK_SUCCESS,
SAVE_BOOKMARK_ERROR} from
'@app/store/bookmark/bookmark.actions';
import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';

@Injectable()
export class BookmarkEffects {
constructor(
private actions$: Actions,
private bookmarkService: BookmarkService
) {}

public saveBookmark$ = createEffect(() => this.actions$.pipe(
ofType(SAVE_BOOKMARK),
exhaustMap((action) => this.bookmarkService.save({...
action.bookmark}).pipe(
map(() => SAVE_BOOKMARK_SUCCESS()),
catchError(() => of(SAVE_BOOKMARK_ERROR()))
))
), {useEffectsErrorHandler: false});
}
```

You will notice some TSLint errors in your IDE for not having the SAVE_BOOKMARK_SUCCESS and SAVE_BOOKMARK_ERROR actions defined yet, but do not worry, we will fix those errors in some time. For now, let us understand the sorcery behind this effect.

CREATEEFFECT

Now, here is something to think about when creating an effect is The createEffect function takes two parameters i.e. your business logic wrapped in a function and a bunch of configurations for the effect to consider while running. One of such configurations is useEffectsErrorHandler used above to disable the default error handling in NgRx Effect, especially when we have our error handling mechanism in place via the catchError operator. If not provided, NgRx Effect would automatically resubscribe to an error (up to 10 times but can be customized). Since we do not want the default behavior and immediately want to dispatch the

SAVE_BOOKMARK_ERROR action in the first attempt itself, we had turned useEffectsErrorHandler off. By default, it is on though. In the end, the response of the createEffect method must be stored on a public/private class member i.e. saveBookmark$ in our case.

OFTYPE

This is extremely important because as soon as the EffectsModule.forFeature is registered, every action is scrutinized by the saveBookmark$ effect. Just like the feature store once initiated is conjoined with the root store becoming the part of the application state altogether, both root as well as feature effects will be commingled in a way that every NgRx action dispatched anywhere in the application would go through all the activated Effects one by one. Hence, the ofType operator makes sense to use more and more to avoid unintended consequences that may arise without it. That means if you forgot to filter an incoming action, the effect's business logic i.e. the body of exhaustMap in our case will run for any type of action which might lead to confusion within the application. Hence, it is ideal to specify what type of action is your effect catering to. That's exactly what we have done in the above code - we want to tap onto the SAVE_BOOKMARK action. When the said action is dispatched by our effect would kick in and execute the business logic immediately.

Always use the ofType operator to target a specific action in your effect. Additionally, you can use comma-separated actions to target multiple actions in the same effect.

EXHAUSTMAP

This you will laugh as well as be amazed at. Let me ask you one thing: have you ever disabled a button on the first click in order not to submit the form or trigger any function more than once? You cannot see but I have raised my hand already! The exhaustMap operator in RxJS saves us from this kind of mishap without extraneous efforts by not projecting the source value if the previous one has not emitted output value yet. Meaning, the new bookmark will not be saved twice (introducing duplicates) if you happen to double click the **Create Bookmark** button.

On another note, we are projecting the bookmarkService.save method as a source value which upon completing may trigger either the SAVE_BOOKMARK_SUCCESS or SAVE_BOOKMARK_ERROR action.

Managing outcome with non-dispatching NgRx Effect

Every NgRx effect expects an NgRx action as a return value which will be dispatched automatically by the effect but sometimes we do not want to dispatch anything once the effect has run – this is called a non-dispatching NgRx Effect.

In cases where you need to isolate the specific logic when the previous effect runs successfully, you can rely on a new NgRx action to trigger another effect in return. It may seem more verbose to create a separate effect for such a small update, but as the complexity of the application grows, handling such outcomes will become a tedious process. By requiring the separate effect to handle the positive outcome of the main effect, you can more easily expand this and continue adding more logic as needed. Having said that, it is recommended to trim down the effect as much as possible to make it less confusing to understand as well as easy to test. Likewise, we have moved the existing success/positive response of the bookmarkService.save API into a separate effect that we do next.

HANDLING THE POSITIVE OUTCOME

For that, in we have to first define the NgRx action which we have already dispatched when the new bookmark is saved successfully:

```
export const SAVE_BOOKMARK_SUCCESS = createAction(
'[Bookmark] Save Bookmark Success'
);
```

Then tap onto the said action in src/app/store/bookmark/bookmark.effects.ts to handle the /list route navigation as follows:

```
import {Injectable} from '@angular/core';
import {Actions, createEffect, ofType} from '@ngrx/effects';
import {exhaustMap, map, catchError, tap} from 'rxjs/operators';
import {of} from 'rxjs';
import {Router} from '@angular/router';
import {SAVE_BOOKMARK, SAVE_BOOKMARK_SUCCESS,
SAVE_BOOKMARK_ERROR} from
'@app/store/bookmark/bookmark.actions';
import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';

@Injectable()
export class BookmarkEffects {
constructor(
private actions$: Actions,
private router: Router,
private bookmarkService: BookmarkService
) {}
public saveBookmark$ = createEffect(() => this.actions$.pipe(
ofType(SAVE_BOOKMARK),
exhaustMap((action) => this.bookmarkService.save({...
action.bookmark}).pipe(
map(() => SAVE_BOOKMARK_SUCCESS()),
catchError(() => of(SAVE_BOOKMARK_ERROR()))
))
), {useEffectsErrorHandler: false});
public saveBookmarkSuccess$ = createEffect(() => this.actions$.pipe(

ofType(SAVE_BOOKMARK_SUCCESS),
tap(() => this.router.navigate(['/list']))
), {dispatch: false});
}
```

You will still notice one TSLint error in your IDE for not having the SAVE_BOOKMARK_ERROR action defined yet which we will fix soon.

Here, the RxJS tap operator would help us perform the side-effect (page navigation) of the observable stream but return an observable that is identical to the source for later use. Also, notice the configuration regarding the dispatch behavior passed. We had disabled the default behavior in this effect since we do not want to dispatch any NgRx action once the effect has run.

HANDLING NEGATIVE OUTCOME

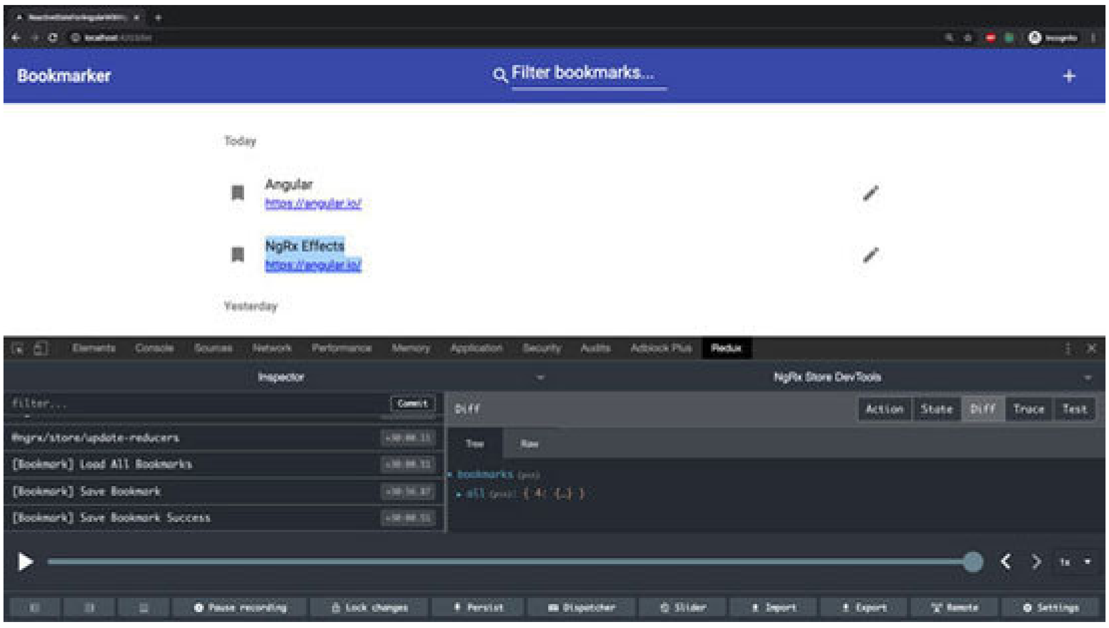
This is quite similar to the previous section and we already have a fair understanding of having multiple such effects to deal with the complexity that growing applications may bring. For that, in we will define the below NgRx action:

```
export const SAVE_BOOKMARK_ERROR = createAction(
'[Bookmark] Save Bookmark Error'
);
```

Then tap onto the said action in src/app/store/bookmark/bookmark.effects.ts to handle the error that we may run into while creating the new bookmark as follows. Note that the TSLint errors should have vanished:

```
import {Injectable} from '@angular/core';
import {Actions, createEffect, ofType} from '@ngrx/effects';

import {exhaustMap, map, catchError, tap} from 'rxjs/operators';
import {of} from 'rxjs';
import {Router} from '@angular/router';
import {MatDialog} from '@angular/material/dialog';

import {SAVE_BOOKMARK, SAVE_BOOKMARK_SUCCESS,
SAVE_BOOKMARK_ERROR} from
'@app/store/bookmark/bookmark.actions';
import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';

@Injectable()
export class BookmarkEffects {
constructor(
private actions$: Actions,
private router: Router,
private bookmarkService: BookmarkService,
private readonly dialog: MatDialog
) {}

public saveBookmark$ = createEffect(() => this.actions$.pipe(
ofType(SAVE_BOOKMARK),
exhaustMap((action) => this.bookmarkService.save({...
action.bookmark}).pipe(
map(() => SAVE_BOOKMARK_SUCCESS()),
catchError(() => of(SAVE_BOOKMARK_ERROR()))
))
), {useEffectsErrorHandler: false});



public saveBookmarkSuccess$ = createEffect(() =>
this.actions$.pipe(
ofType(SAVE_BOOKMARK_SUCCESS),
tap(() => this.router.navigate(['/list']))
), {dispatch: false});

public saveBookmarkError$ = createEffect(() => this.actions$.pipe(
ofType(SAVE_BOOKMARK_ERROR),
tap(() => this.dialog.open(ErrorDialogComponent, {
width: '400px',
data: {errorMessage: 'Sorry, unable to create bookmark.'}
}))
), {dispatch: false});
}
```

Figure 4.3 shows that the new bookmark has been successfully created on the list bookmarks page. Notice the **[Bookmark] Save Bookmark** and **[Bookmark] Save Bookmark Success** actions dispatched in Redux Devtool and the newly added bookmark, NgRx Effects, highlighted:



```
@ngrx/store/update-reducers
[Bookmark] Load All Bookmarks
[Bookmark] Save Bookmark
[Bookmark] Save Bookmark Success
```

```
▸ bookmarks (pin)
  ▸ all (pin): { 4: {…} }
```

Figure 4.3: *Created a new bookmark using NgRx Bookmark Effects flow*

Similarly, you can create another set of NgRx effects when any bookmark is being edited. But I will not cover that in this chapter since it's quite easy to do. You can consider it as a task for yourself to learn by doing. However, the working solution will be included in the companion codebase provided with this book, in case you are stuck.

What is root NgRx Effect

The root NgRx Effect creates a provision to handle side-effects when the root store is being updated via NgRx actions/reducers. Having said that, it has to be added to the root module using EffectsModule.forRoot([]) so that it will be ready to tap root-level NgRx actions that our application may dispatch on page load. If you skim through the code that we have written so far, you will notice that in a couple of components/guards/effects we are navigating users to the /list route using a Router service. Although, there is nothing wrong with the implementation until we decide to rename the route name to something else. We will have to find every occurrence of the same and replace it correctly to prevent bugs. Surefire, we can do this but would not it be great if we could avoid this effort of renaming altogether? And since the router.navigate method used is a promise i.e. an asynchronous event it is the perfect problem to resolve with the Root NgRx Effect.

Since we had installed the NgRx Schematics in the previous chapter, we can leverage Angular CLI to create our first NgRx Effect using the following command. Few things to note:

The effect flag generates the effect file.

The path defines the name of the effect file and where it's going to be created. We want to create router.effects.ts in the src/app/store/router/ directory to manage all sorts of router navigation in one place.

The --module flag automatically plugs the feature effect into

The --skipTests flag prevents generating the spec files since we are not covering any tests in this book.

The --root flag is extremely important to hook up the generated NgRx Effect using If not provided, EffectsModule.forFeature will be used.

ng g effect store/router/router --module=app.module.ts --skipTests=true --root

Please answer the asked questions as mentioned below:

Q: Should we wire up success and failure actions?

A: Type "Yes" and press Enter.

Q: Do you want to use the create function?

A: Type "Yes" and press Enter. This will import a createEffect function for us to set up our first effect.

You should see the following snippet in the src/app/store/router/router.effects.ts file:

```
import {Injectable} from '@angular/core';
import {Actions, createEffect} from '@ngrx/effects';

@Injectable()
export class RouterEffects {
constructor(private actions$: Actions) {}
}
```

Before we modify the aforementioned auto-generated effects file to suit our needs, let us create an action file for the router effects to consume. If you take a look at the src/app/bookmark/bookmark.effects.ts file right now, you will notice that it has some clutter lying around in terms of an Angular Router service. All of this just bloats BookmarkEffects for no apparent reason and can be moved out elsewhere so that the sole purpose of the effect will become just to express the intent for page navigation.

So, first of all, let us create a new action file for all sorts of router navigation in the application:

```
ng g action store/router/router --skipTests=true
```

Please answer the asked questions as mentioned below:

Q: Should we generate success and failure actions?

A: Type "No" and press Enter.

Q: Do you want to use the create function?

A: Type "Yes" and press Enter.

This command must have generated **src/app/store/router/router.actions.ts** which can be updated with:

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

export const REDIRECT_TO_LIST_ROUTE = createAction(
'[Router] Redirect to /list route'
);
```

Then, we'll make use of the above action in src/app/store/bookmark/bookmark.effects.ts and get rid of the saveBookmarkSuccess$ effect and the SAVE_BOOKMARK_SUCCESS action previously used:

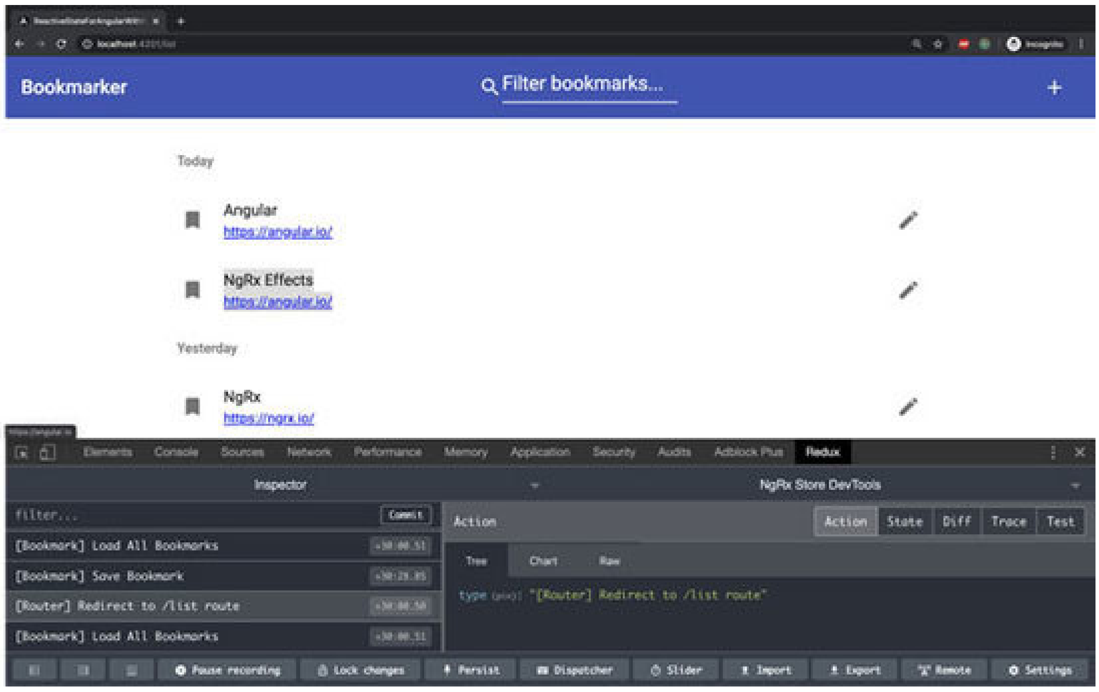
```
import {Injectable} from '@angular/core';
import {Actions, createEffect, ofType} from '@ngrx/effects';
import {exhaustMap, map, catchError, tap} from 'rxjs/operators';
import {of} from 'rxjs';
import {MatDialog} from '@angular/material/dialog';
import {SAVE_BOOKMARK, SAVE_BOOKMARK_ERROR} from
'@app/store/bookmark/bookmark.actions';
import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
```

```
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';
import {REDIRECT_TO_LIST_ROUTE} from
'@app/store/router/router.actions';

@Injectable()
export class BookmarkEffects {
constructor(
private actions$: Actions,
private bookmarkService: BookmarkService,
private readonly dialog: MatDialog
) {}
public saveBookmark$ = createEffect(() => this.actions$.pipe(
ofType(SAVE_BOOKMARK),
exhaustMap((action) => this.bookmarkService.save({...
action.bookmark}).pipe(
map(() => REDIRECT_TO_LIST_ROUTE()),
catchError(() => of(SAVE_BOOKMARK_ERROR()))
))
), {useEffectsErrorHandler: false});
public saveBookmarkError$ = createEffect(() => this.actions$.pipe(
ofType(SAVE_BOOKMARK_ERROR),
tap(() => this.dialog.open(ErrorDialogComponent, {
width: '400px',
data: {errorMessage: 'Sorry, unable to create bookmark.'}
}))
), {dispatch: false});
}
```

Next, we'll tap onto the REDIRECT_TO_LIST_ROUTE action in This is the same as what we were doing with the saveBookmarkSuccess$ effect earlier, just moved it in the generic router effects with a different name,

```
import {Injectable} from '@angular/core';
import {Actions, createEffect, ofType} from '@ngrx/effects';
import {Router} from '@angular/router';
import {tap} from 'rxjs/operators';

import {REDIRECT_TO_LIST_ROUTE} from
'@app/store/router/router.actions';

@Injectable()
export class RouterEffects {
constructor(
private actions$: Actions,
private router: Router
) {}

public navigateToListRoute$ = createEffect(() => this.actions$.pipe(
ofType(REDIRECT_TO_LIST_ROUTE),
tap(() => this.router.navigate(['/list']))
), {dispatch: false});
}
```

With the above changes in place, *Figure 4.4* shows that the page navigation happens successfully after creating a new bookmark via the router effects in Redux Devtool with the **[Router] Redirect to /list route** action dispatched:



Today

Angular
https://angular.io/

NgRx Effects
https://angular.io/

Yesterday

NgRx
https://ngrx.io/

```
filter...
[Bookmark] Load All Bookmarks
[Bookmark] Save Bookmark
[Router] Redirect to /list route
[Bookmark] Load All Bookmarks
```

```
type (pin): "[Router] Redirect to /list route"
```

Figure 4.4: *Created a new bookmark using NgRx Router Effects flow*

Similarly, you can create another NgRx router effect to handle any page navigation (not just the /list route) that happens when the error dialog is shown by But we will not cover that in this chapter since it's quite easy to do. You can consider it as a task for yourself to learn by doing. However, the working solution will be included in the companion codebase provided with this book, in case you are stuck.

Conclusion

In this chapter, we have gained an understanding of root as well as feature effects in NgRx. We learned how to create NgRx effects in general and spy on incoming actions to run asynchronous code without breaking the third rule of Redux (i.e. Mutate State using Pure functions only). We improved the existing components by isolating the business logic out of them. This brought two things into our notice: First, we learned to create pure a.k.a. dump component without any business logic in them and Second, we managed to execute side-effects without affecting a traditional action-dispatcher-reducer pipeline. We purposefully used the feature effect for the create bookmark (as well as edit bookmark) journeys since they are invoked lazily using Angular Router. Along the way, NgRx schematics helped us generate scaffolding boilerplate code needed to create and manage the NgRx effect files.

Our application needs lots of improvements such as a loader while navigating between different routes, pagination while listing bookmarks, and many more such things. Thus, in the next chapter, we will explore the NgRx Router Store which connects the Angular Router to the NgRx store.

Questions

What is the NgRx root effect?

What is the NgRx feature effect?

When to prefer the root or feature effect?

How to create the NgRx effect?

When the exhaust map operator is useful?

CHAPTER 5

NgRx Router Store

The previous chapter examined how isolating business logic into an RxJS powered effect model can simplify their consumption and reduce the management overhead. This mechanism is important because it allows you to reuse the effect to handle it by one or more components/services without sacrificing any of the Redux laws. The various strategies for how these different types of side-effects (dispatching NgRx Effects, non-dispatching NgRx Effects, or handling positive/negative outcomes of the base Effects) occurred, was determined based on the configurations defined in the effects. We also showed examples like route navigation and error handling that uses the feedback from one effect to signal the start or completion of another. The representational power of a single store per application is limitless.

Remember that in the previous chapters you were able to store every minute data or events in the store so that it can be fetched as and when needed in a robust way. It's important to realize that, by design, NgRx Store is our source of truth for any type of information that the application may need; therefore, relying on any other source, inadvertently, is an anti-pattern that should always be avoided at any cost. By now, you've managed to update the store with all the bookmarks and filter values irrespective of where they come from i.e. over the wire or user-inputted but still, you are looking at a different source unknowingly. And that is the state of the router. We still read a bookmark ID from the URL using a few helper methods provided by Angular Router while editing bookmarks. But there are times when such manual parsing of the URL to extract URL fragments, query params, or path params are insufficient and redundant because you may need external libraries or custom code to interact and manage all this. You could, alternatively, take help from NgRx. And the latter over the former is what is highly recommended.

In this chapter, we'll continue with this theme and expand what we left off in *Chapter 4: NgRx* You will learn that you do not always have to care about the router state if you simply want to offload the work to parse and extract necessary router states and instead focus on your application. Furthermore, we'll explore scenarios to have a custom serializer to take out the parts we need and dump the rest from the vast router information provided by NgRx as well as to have an indicator in terms of a loader/spinner during route navigation. The interplay of having a router state and using it to show/hide the loader/spinner forms the foundation of the more complex logic to be handled gracefully by connecting the Angular Router to the Store. To illustrate this, we'll leverage NgRx Router Store throughout the chapter.

Structure

In this chapter, we will leverage the @ngrx/router-store package which provides bindings to connect the Angular Router to the Store. During each router navigation cycle, multiple actions are dispatched for changes in the router's state that we will listen to.

The following topics will be covered in this chapter:

Installing @ngrx/router-store package

Overriding Default Router State Serializer

Tapping native actions of Router Store

Objectives

We will understand the concept behind the router store and integrate it into the sample application to end up having a single source of truth for the application which is nothing but an NgRx Store.

Installing @ngrx/router-store package

In our sample application that we built in the second chapter; guards are responsible for interacting with URLs using the Angular Router service. However, NgRx Router Store as well provides a way to interact with those URLs and connect them to the store. In fact, NgRx Router Store parses every Angular route to extract various information in terms of a URL, fragments, path params, query params, etc and makes it possible for us to query it using NgRx selectors.

Prior to starting the show, let us first install the NgRx Router Store package in a terminal:

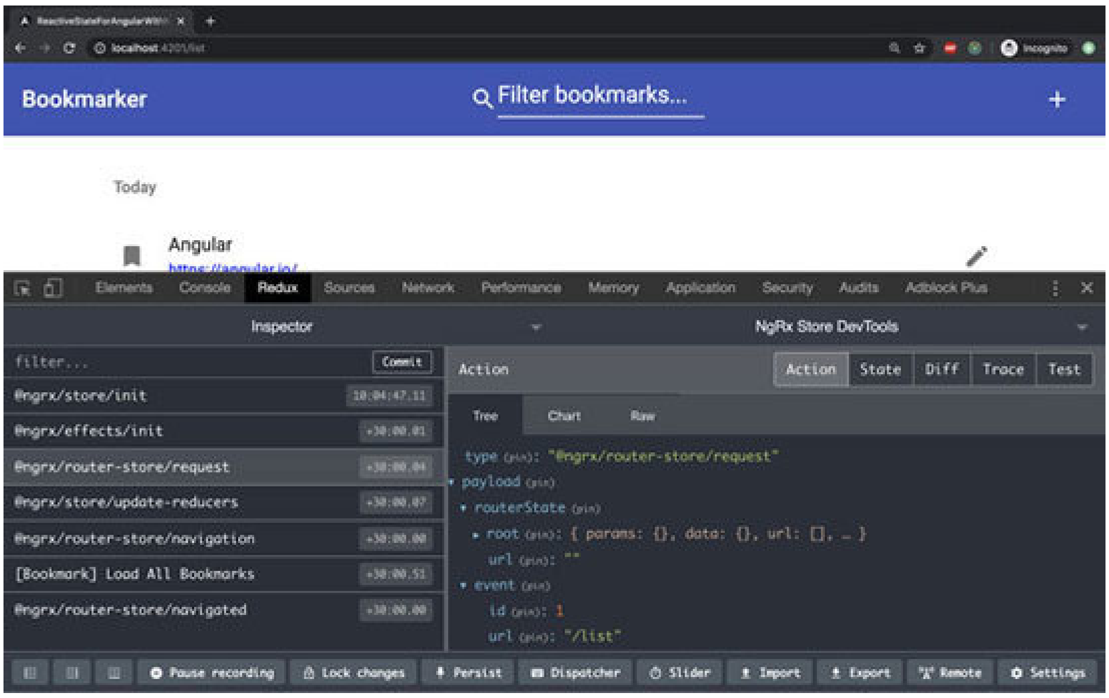
```
ng add @ngrx/router-store
```

This must-have added following in src/app/app.module.ts as follows:

```
StoreRouterConnectingModule.forRoot()
```

All StoreRouterConnectingModule.forRoot() takes is a configuration (have not specified above yet but will soon) to customize the stateKey and the serializer. Also, there is no StoreRouterConnectingModule.forFeature() method unlike Store/Effects.

Figure 5.1 depicts the connection of the Angular Router to the store wherein the default NgRx action had been dispatched with the payload. The payload is nothing but a parsed output of the current application route i.e. /list in the form of a serialized JavaScript object. That means the NgRx Router Store has successfully connected the Angular Router to the NgRx Store as shown below:



```
@ngrx/store/init                      10:04:47.11
@ngrx/effects/init                    +30:00.01
@ngrx/router-store/request            +30:00.04
@ngrx/store/update-reducers           +30:00.07
@ngrx/router-store/navigation         +30:00.00
[Bookmark] Load All Bookmarks         +30:00.51
@ngrx/router-store/navigated          +30:00.00
```

```
type (pin): "@ngrx/router-store/request"
▾ payload (pin)
  ▾ routerState (pin)
    ▸ root (pin): { params: {}, data: {}, url: [], … }
      url (pin): ""
  ▾ event (pin)
      id (pin): 1
      url (pin): "/list"
```

Figure 5.1: *NgRx Router Store in action*

There are several native NgRx actions dispatched by the NgRx Router Store depending upon what we are doing in the application. When any router navigation is about to start in the application, the @ngrx/router-store/request action is dispatched with the necessary payload showcasing the current state of the route and where we intend to go. For example, Figure 5.1 above clearly shows that the intended route we want to go to is the /list route which is mentioned under the event property in the payload above.

During navigation, before any Angular Guards or Resolvers run, the @ngrx/router-store/navigation action is dispatched. After successful navigation, the @ngrx/router-store/navigated action is dispatched (look at the last action dispatched above). For some reason, if the navigation is canceled or errored then the @ngrx/router-store/cancel and @ngrx/router-store/error actions will be dispatched respectively.

At a first glance, it is obvious that the NgRx Router Store has properly extracted params, data, URL, and queryParams from the route; however, there is a lot of junk in there if you expand the root node under routerState property in the payload. Furthermore, the junk will be prominent enough under properties and their sub-properties when you edit any bookmark and select the newly fired @ngrx/router-store/request or @ngrx/router-store/navigated action in the Redux Devtools. So, the question is can we do any better? And the answer to that is obviously yes. But please note that the junk as we referred to it is not junk; it is just extra information extracted by DefaultRouterStateSerializer used by the NgRx Router Store which may be needed by other applications but we do not need. Hence, we are allowed to write a custom serializer to get just what we want from the route.

Overriding Default Router State Serializer

The idea behind it is that you often have information that is completely subject to the requirement of the application. If you want to keep it as it is then there is no need for a custom serializer. But if you need a clutter-free router state and that too without much code/effort then it is recommended to serialize it in your way. When the route navigation happens in the application, the aforementioned native actions are dispatched with a snapshot of the router state (of type in its payload, however, it's a complex structure, containing many pieces of information about the current router state and what's rendered by the router. In most cases, you may only need a subset of information from For that, you end up writing a custom serializer.

In let us write a custom serializer as given below. Make sure that it implements the abstract RouterStateSerializer class from the NgRx Router Store and must return a new altered snapshot of the router state:

```
import {RouterStateSnapshot} from '@angular/router';
import {RouterStateSerializer} from '@ngrx/router-store';
import {Params} from '@angular/router';

export interface RouterState {
url: string;
params: Params;
queryParams: Params;
}

export class CustomRouterStateSerializer implements
RouterStateSerializer {

serialize(routerState: RouterStateSnapshot): RouterState {
let route = routerState.root;
while (route.firstChild) {
route = route.firstChild;
}
const {url, root: {queryParams}} = routerState;
const {params} = route;
return {url, params, queryParams};
}
}
```

All we have done here is,

Defined the interface, of the new snapshot of the router state. Notice how it's a trimmed-down version of the much complex router state that we saw earlier.

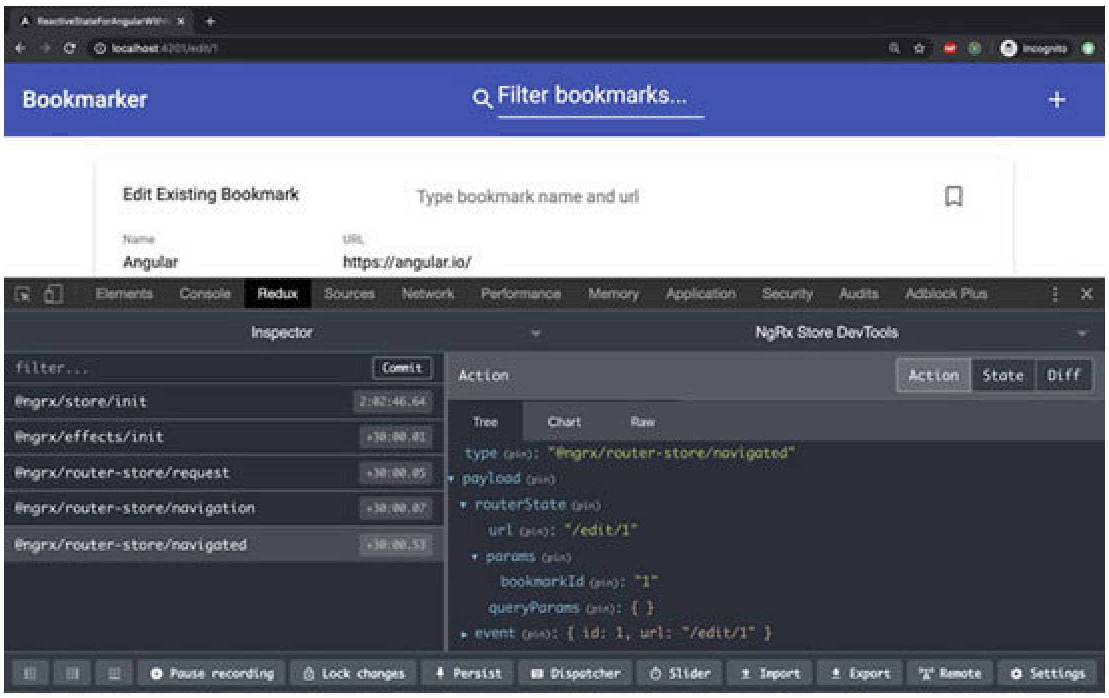
Implemented RouterStateSerializer which allowed us to override the native serialize method with our logic.

Extracted the necessary pieces from the default snapshot of the router state such as and

Now, we have our custom serializer ready to plug into the application module. Here is how you can hook it up under imports in src/app/app.module.ts replacing the existing method StoreRouterConnectingModule.forRoot() with a custom serializer in place:

```
import {
CustomRouterStateSerializer

} from '@app/store/router/custom-router-state.serializer';

StoreRouterConnectingModule.forRoot({serializer:
CustomRouterStateSerializer})
```

Then, go to the /list route and edit any bookmark as shown in *Figure 5.2* below. In the Redux Devtools, look for the @ngrx/router-store/navigated action to find the customized payload containing the new router state. There are so many things you can do, and it all depends on your needs. In this case, in the kind of application we are building throughout this book, we needed only path params and query params but you can extract the scheme from the url, url fragments, the host, the domain, and what not and all the complex logic can reside inside the custom serializer we wrote above. This is a clean and maintainable router state right there at your disposal and *Figure 5.2* proves that the custom serializer worked as intended. Notice the payload highlighted below contains only what the custom serializer extracted for us and nothing else:



```
@ngrx/store/init                    2:02:46.64
@ngrx/effects/init                  +30:00.01
@ngrx/router-store/request          +30:00.05
@ngrx/router-store/navigation       +30:00.07
@ngrx/router-store/navigated        +30:00.53
```

```
type (pin): "@ngrx/router-store/navigated"
▾ payload (pin)
  ▾ routerState (pin)
      url (pin): "/edit/1"
    ▾ params (pin)
        bookmarkId (pin): "1"
      queryParams (pin): { }
  ▸ event (pin): { id: 1, url: "/edit/1" }
```

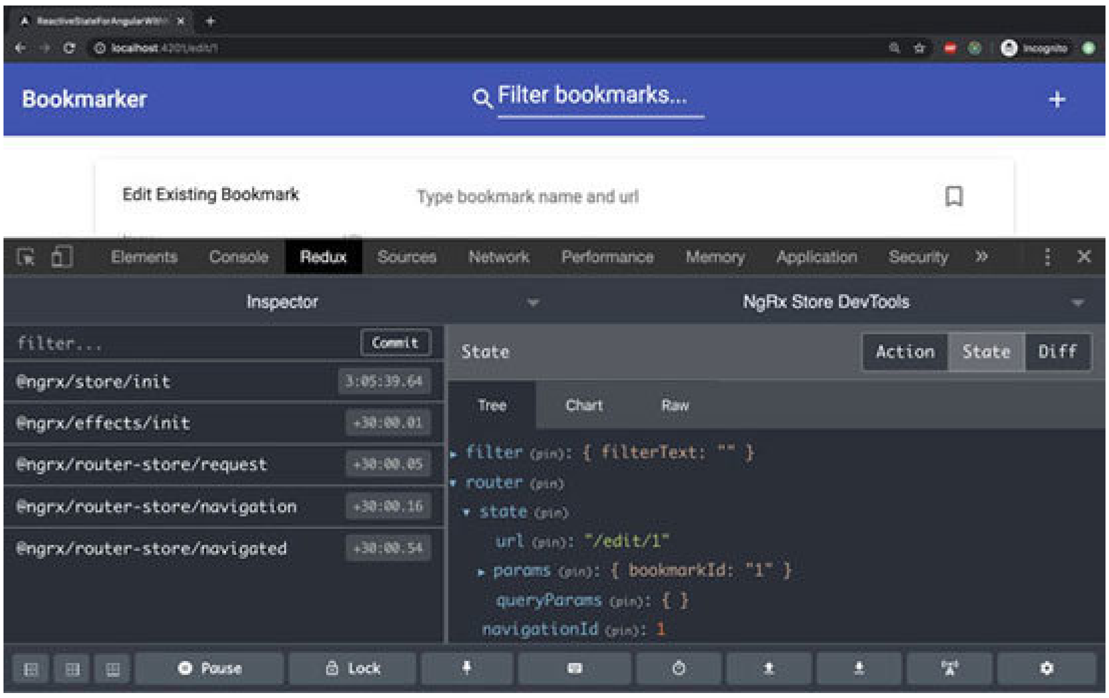
Figure 5.2: *Lightweight Router State using the custom serializer*

Since we are on the /edit route, the params property is not empty as before and, in fact, it correctly shows the id of the bookmark being edited. Does that mean that this new router state is part of the store as well? Hmm, not yet! And that's because, up until, we have not set up the reducer for it which is a piece of cake. By the same token, we do not have to create a reducer ourselves as we can reuse the existing routerReducer provided by the NgRx Router Store under ROOT_REDUCERS that we created in *Chapter 3: NgRx Store* in

import {ActionReducerMap, MetaReducer} from '@ngrx/store';
import * as fromRouterStore from '@ngrx/router-store';
import {environment} from '../../environments/environment';
import * as fromToolbar from '@app/store/toolbar/toolbar.reducer';

```
export interface AppState {
filter: fromToolbar.FilterState;
router: fromRouterStore.RouterReducerState;
}

export const ROOT_REDUCERS: ActionReducerMap = {
filter: fromToolbar.reducer,
router: fromRouterStore.routerReducer
};
export const metaReducers: MetaReducer[] =
!environment.production ? [] : [];
```

We have followed the same pattern that we decided to use early on which is to import everything from the NgRx Router Store under the fromRouterStore variable and pluck the necessary router reducer state and the reducer itself respectively. *Figure 5.3* shows that the store has been updated with the router state correctly below:



```
@ngrx/store/init                 3:05:39.64
@ngrx/effects/init               +30:00.01
@ngrx/router-store/request       +30:00.05
@ngrx/router-store/navigation    +30:00.16
@ngrx/router-store/navigated     +30:00.54
```

```
▸ filter (pin): { filterText: "" }
▾ router (pin)
  ▾ state (pin)
      url (pin): "/edit/1"
    ▸ params (pin): { bookmarkId: "1" }
      queryParams (pin): { }
    navigationId (pin): 1
```

Figure 5.3: *Custom Router State is now part of the NgRx Store*

The StoreRouterConnectingModule.forRoot() method above takes one more optional parameter apart from serializer and that is The default configuration sets stateKey to the router automatically, however, say you choose a different key name, foobar in ROOT_REDUCERS then it is necessary to define stateKey as foobar there as well. The benefit of that is that you will be opted-in for time-travel debugging.

The Time Travel Debugging (TTD) is a tool that allows you to record the execution of NgRx actions dispatched and the state updates by the associated reducers which you can then replay later both forwards and backward. TTD can help you debug issues easier by letting you "rewind" your debugger session, instead of having to reproduce the issue until you find the bug. Even Redux Devtools allows TTD - notice the button strips at the bottom in

Figure 5.2 and click any one of the 3 buttons with icons from the left to start the TTD session. There are plenty of articles on the Internet/Miasma to learn Redux Devtools for the better.

Next, we'll read that router state from the store instead of extracting it off of the route all over again in the edit bookmark journey. By now, you must be proficient in writing an NgRx selector which will read the bookmarkId property from the router state mentioned above. Let us quickly create the src/app/store/router/router.selectors.ts file as follows:

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import * as fromRouterStore from '@ngrx/router-store';

import {RouterState} from '@app/store/router/custom-router-
state.serializer';

const getRouterState = createFeatureSelector>('router');

export const getBookmarkId = createSelector(
getRouterState,
routerState => routerState.state.params.bookmarkId
);
```

As you may have already guessed, we have taken out the piece i.e. router from the root store as Notice the router state as per the Figure 5.3 has two properties in it mainly state and navigationId (both are part of the RouterReducerState interface), and the state object further has and queryParams (all are part of the RouterState interface we had defined in the custom serializer earlier), hence, the type of the getRouterState selector is RouterReducerState. Later, we extracted the bookmarkId param off of getRouterState creating a new selector getBookmarkId in return which we can use later on.

Remember that, unlike the create bookmark journey, the edit bookmark journey (including the edit component and guard) was not updated in *Chapter 3: NgRx Store* and it was given as a task for readers to learn by doing. Unfortunately, if you could not crack it then you should use the aforementioned selector in src/app/edit/edit.guard.ts as follows. Here, we read bookmarkId from the router store directly and pass it along to bookmarkService in order to fetch the edit bookmark details:

```
import {Injectable} from '@angular/core';
import {CanActivate} from '@angular/router';
import {Observable, of} from 'rxjs';
import {MatDialog} from '@angular/material/dialog';
import {first, map, switchMap, catchError} from 'rxjs/operators';
import {Store} from '@ngrx/store';

import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';
import {AppState} from '@app/store';
import {getBookmarkId} from '@app/store/router/router.selectors';

@Injectable({
```

```
providedIn: 'root'
})

export class EditGuard implements CanActivate {
constructor(
private readonly bookService: BookmarkService,
private readonly dialog: MatDialog,
public readonly store$: Store
) {}

canActivate(): Observable {
return this.store$.select(getBookmarkId).pipe(
first(),
switchMap((bookmarkId) =>
this.bookService.getById(bookmarkId).pipe(
first(),
map((bookmark) => this.bookService.editBookmark = bookmark),
switchMap(() => this.activateRoute()),
catchError(() => {
this.dialog.open(ErrorDialogComponent, {
width: '400px',
data: {errorMessage: 'Sorry, unable to edit bookmark.', redirectTo:
'/list'}
});
return of(false);
})
))
);
}

activateRoute() {
```

```
return of(true);
}
}
```

Fortunately, if you could manage to update the edit bookmark journey as requested in *Chapter 3: NgRx Store* (kudos to you) then you should ignore the non-highlighted parts from the canActivate method given in the above snippet and just use the highlighted parts to read bookmarkId using the same selector.

Similarly, you can use the same selector in src/app/edit/edit.component.ts as well but I will not cover it in this chapter since it's quite easy to do. You can consider it as a task for yourself to learn by doing. However, the working solution will be included in the companion codebase provided with this book, in case you are stuck.

Now go and check the edit bookmark functionality works as intended with the above changes in mind. I am optimistic that it will for you.

Tapping native actions of Router Store

Imagine someone making a page transition to see a list of bookmarks requesting considerable data from the server. But shortly after spawning this call, the user navigates away from the page by clicking elsewhere. What happens to the original request? Should we allow the user to navigate away so hastily? We're guessing no. Page navigation, as it exists in Angular Router, has a deterministic lifespan defined almost entirely by you. Additionally, there is little support within JavaScript for route management because this has historically been left to browser manufacturers to worry about. Although, this feature of tapping the lifecycle of page navigation makes Angular a marvelously useful framework. Angular Router as well emits such events etc.) which we can leverage on to deduce whether the page navigation is in process or not. However, this becomes way easier with NgRx native actions. An important point to remember when dealing with NgRx native actions is that the life-time of action begins when it is dispatched internally by NgRx which we could track in the Redux Devtools as seen earlier. Hence, there is little overhead in tapping into these native actions. The other naive option to achieve the same result is to dispatch your NgRx actions or rely on Angular Service to track the status of when the navigation starts or ends, however, this approach needs duplicate code/efforts in every page/component whenever there is ongoing page navigation about to happen. All this is not to sound alarmist. We expect that the best solution supersedes a better solution which indeed supersedes a good solution. That is why we need sophisticated libraries like NgRx. With NgRx, our strategy would be:

Tap NgRx Router Store actions in a reducer to toggle loading root-level state.

Control progress bar's visibility via the loading state.

NgRx Router Store provides a straight forward mechanism for tapping the native NgRx Router Store actions so let us begin.

Managing the loading state with native actions

It is important to understand that when you tap a native NgRx Router action, you are creating a provision to listen to it no matter where the navigation happens in the application. And hence, the scope of the callbacks that are passed into the action should remain small and light-weight. Since NgRx Store uses the small and pure functions at its helm, using reducers to tap into such native actions makes it a perfect fit.

Let us create the router reducer using the following command, ultimately naming the reducer file as

```
ng g reducer store/router/router --skipTests=true
```

Please answer the asked questions as mentioned below:

Q: Should we add success and failure actions to the reducer?

A: Type "No" and press Enter.

Q: Do you want to use the create function?

A: Type "Yes" and press Enter.

The aforementioned command must have created Then, override it with a following:

```
import {Action, createReducer, on} from '@ngrx/store';
import {routerNavigationAction, routerNavigatedAction,
routerCancelAction, routerErrorAction} from '@ngrx/router-store';

export interface LoaderState {
loading: boolean;
}
export const loaderState: LoaderState = {
loading: false
};

export const pageNavigationStartReducer = createReducer(

loaderState,
on(routerNavigationAction, (_) => ({loading: true})),
on(routerNavigatedAction, (_) => ({loading: false})),
on(routerCancelAction, (_) => ({loading: false})),
on(routerErrorAction, (_) => ({loading: false}))
);

export function reducer(state: LoaderState | undefined, action:
Action) {
return pageNavigationStartReducer(state, action);
}
```

We are doing a few important things here:

As we agreed above that we'll rely on the loader state to show/hide a progress bar. So first of all, we had defined the interface, and then the actual state, with a default falsy value.

NgRx Router Store dispatches various native actions during the page navigation process and we tapped into a few of them to toggle the loader state. So we turned the loader state ON when the page navigation begins and turned it OFF when it is either completed successfully or canceled/errored abruptly.

The next ideal thing for us to do is to hook up the aforementioned reducer into ROOT_REDUCERS in So update it as follows:

```
import {ActionReducerMap, MetaReducer} from '@ngrx/store';
import * as fromRouterStore from '@ngrx/router-store';

import {environment} from '../../environments/environment';

import * as fromToolbar from '@app/store/toolbar/toolbar.reducer';
import * as fromRouter from '@app/store/router/router.reducer';

export interface AppState {
filter: fromToolbar.FilterState;
router: fromRouterStore.RouterReducerState;
loader: fromRouter.LoaderState;
}

export const ROOT_REDUCERS: ActionReducerMap = {
filter: fromToolbar.reducer,
router: fromRouterStore.routerReducer,
loader: fromRouter.reducer
};

export const metaReducers: MetaReducer[] =
!environment.production ? [] : [];
```

Now if you check the state in the Redux Devtools then you should see the loader state in place alongside other root-level states. Also, pay attention to the value of the loader.loading property which quickly jumps from falsy to truthy and falsy again as you navigate to any page.

Show/Hide progress bar based on the loading state

Being able to use this loader state using an NgRx selector is certainly beneficial because it frees you from having to write them yourself (with respect to the naive approach we thought about earlier), reducing the probability for bugs to occur. Now, all we have to do is fetch the loader state and use its computed value (for being an observable) to toggle the DOM, that is, show/hide a progress bar. On that note, let us first write a selector to read the above-mentioned loader state in src/app/store/router/router.selectors.ts as follows:

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import * as fromRouterStore from '@ngrx/router-store';

import {RouterState} from '@app/store/router/custom-router-state.serializer';
import {LoaderState} from '@app/store/router/router.reducer';

const getRouterState = createFeatureSelector>('router');
const getLoaderState = createFeatureSelector('loader');

export const getBookmarkId = createSelector(
getRouterState,
routerState => routerState.state.params.bookmarkId
);
```

```
export const isPageLoading = createSelector(
getLoaderState,
loaderState => !!loaderState.loading
);
```

I'm pretty sure that by now you are well acquainted with the above code based on your learnings from the previous chapters and I do not have to explain at all. Moving on, we are going to use the Material Progress Spinner component for the loader in the DOM. For that, we'll import the associated module from @angular/material/progress-spinner as follows in src/app/app.module.ts (the entire snippet is not shown for brevity):

```
import {MatProgressSpinnerModule} from
'@angular/material/progress-spinner';

@NgModule({
.
.
.
imports: [
.
.
.
MatProgressSpinnerModule
.
.
.
]
.
.
.
})
export class AppModule {}
```

Then, add the progress spinner at the top of the file in

***ngIf="isPageLoading$ | async"** class="example-loader">

   mode="indeterminate" value="50">

Furthermore, position it in the center of the page afloat over existing DOM in

```
.example-loader {
position: absolute;
left: 0;
top: 0;
bottom: 0;
right: 0;
background: rgba(0, 0, 0, 0.5);
z-index: 9999;

> mat-progress-spinner {
position: relative;
top: calc(50% - 70px);
left: calc(50% - 70px);
}
}
```

Notice that we had used the class property isPageLoading$ via async pipe above in the template but have not imported it in the application component yet. Let us fix that in

```
import {Component, ChangeDetectionStrategy, OnInit} from
'@angular/core';
import {Store} from '@ngrx/store';
import {Observable} from 'rxjs';

import {AppState} from '@app/store';

import {FILTER_BOOKMARKS} from
'@app/store/toolbar/toolbar.actions';
import {isPageLoading} from '@app/store/router/router.selectors';

@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.scss'],
changeDetection: ChangeDetectionStrategy.OnPush
})
export class AppComponent implements OnInit {
isPageLoading$: Observable;

constructor(public readonly store$: Store) {}

onFilterChange(filterText) {
this.store$.dispatch(FILTER_BOOKMARKS({filterText}));
}
ngOnInit() {
this.isPageLoading$ = this.store$.select(isPageLoading);
}
}
```

Here, we've simply imported the isPageLoading selector and consumed it in the component/template using NgRx select method. With everything sorted now, if you go to the application in a browser and start navigating to various routes in the application, you should see an animated loader in the center of the screen for a time-being until the page, expected to visit, loads. It is most prominently visible on the /list bookmark route since it takes a bit longer to load compared to /edit or /create routes.

Conclusion

In this chapter, we have managed to make the NgRx Store, one and only source of truth for the application data. We have purposely learned to override the default router store serializer with our own in order to truncate the mammoth of the default router state. Additionally, we chose to tap the native actions triggered by the NgRx Router Store in order to reason about when the page navigation starts and ends. With that knowledge of the page navigation events, we could improve the experience of lazy-loaded routes by showing an animated loader by the time the route is fully loaded.

The end of this chapter brought us to the last chapter of this book on the NgRx Entity adapter for managing record collections. In the final chapter, we'll leverage NgRx Entity in the list bookmark journey and improve the existing listing for the better. It also provides APIs for manipulating and querying entity collections.

Questions

What is NgRx Router Store?

What is the Default Router Store Serializer?

Why and How to override the Default Router Store Serializer?

How to tap native Router Store actions and what's the benefit?

CHAPTER 6

NgRx Entity State

This is probably one of the most complex topics in the NgRx world which is why I intentionally kept it at the end. Up until now, we have had streams of data whether it comes from creating a new bookmark or editing an existing one. But all in all, we have only dealt with a singular state, or rather just looked at it that way. Now, we take the opposite approach as we look at an individual bookmark as a separate entity. What I mean by that is that for now there was no easy way to add/update the bookmark without manipulating the whole list of bookmarks. In fact, we had not tried to do so since all we did was fetch/save/update bookmarks using a remote API and never modified the individual bookmark manually at a local level and then sync the store with the remote data. That is because we fetch a fresh new list of bookmarks every time a user navigates to the /list route which off-course has a performance overhead. What if we not refetch the same list of bookmarks over and over again and instead just updates the local instance of the bookmarks whenever a new or any existing bookmark is added or updated respectively. Well, if we do so, the local copy of the list would not show the updates until we manually update the store after the act. However, the edit act is not that easy compared to the create act because to update the existing bookmark in the local copy needs some heavy lifting to loop over the entire list of the bookmarks in order to find it. What if we give its own space for every bookmark and editing or removing it would be performant as well? That's where **NgRx Entity State Adapter** really shines and helps us manage a collection of records, that is, bookmarks in our case, way more easily than anything else.

In this chapter, we will, for the last time, make further improvements to the sample application. You will learn that the key to elegant collection management is done in part by the effective use of Entity and Entity State. Given the idea in place, the next step is to learn about the various ways of manipulating collections using the Entity Adapter Collection methods. Before we get started, it is important for you to understand that you need to put aside the imperative collection handling techniques you are accustomed to, like in favor of Entity Adapter operations against an entity as implemented by NgRx Entity.

Structure

In this chapter, we will leverage the @ngrx/entity package which provides Entity State Adapter for managing record collections. The following topics will be covered:

What is Entity State?

Installing @ngrx/entity package

Replacing empty Entity Collection using .setAll

Reading Entity State using Entity Selectors

Using cached Entity State Collection

Adding/Updating Entities

Sorting Entity State Collection

Objectives

We will understand the concept behind the Entity State and integrate it into the sample application to end up having a collection of records for the list of bookmarks to easily manipulate them with performance in mind.

What is Entity State?

An Entity can be any singular, identifiable, and separate object. In our case, every bookmark can be a separate entity if we want it to be so. Each entity is supposed to have certain attributes; the same way each bookmark has its attributes in terms of and the date of its creation. With the concept of entity, each bookmark can have its own space in the store to make it easy to touch upon without thinking about the whole collection of bookmarks all the time. Many such entities come together to form a collection called **Entity State** Let us reinvent the NgRx Entity (just like we did in the first chapter with Reducks) for the sake of learning. We will call it Reducks Entity.

Let us take a look at the existing state of the list of bookmarks. As you know, they are represented by an array of data structure at the moment as given below. Each bookmark has its own unique identification number in terms of an Getting a hold of a particular bookmark is cumbersome here and not performance sensitive. Why so?

```
[
{
id: 1,
name: 'Angular',
url: 'https://angular.io/',
created: '2020-05-17T15:37:44.047Z'
},
{
id: 2,
name: 'NgRx',
url: 'https://ngrx.io/',

created: '2020-05-16T15:37:44.048Z'
}
]
```

Imagine we want to get a hold of the item from the above list. With the existing data structure in place, the only fine way to do that is to loop over the entire list (regardless of its size/length) and keep matching the item's ID with the one we want to find. This has to be repeated for every item in a sequence until we find a match. That means the longer the list, the longer will it take to find a matching item, especially for those items that are lying near the end of the list. In that case, we have to keep searching from the start of the list and then all the way to the end to find a match. After some thoughts, you will come to know that the problem lies not in the search algorithm but in the data itself. That the data structure we have used to represent the above list of bookmarks is not an optimal one and hence, we need some sort of mechanism to normalize the data.

Normalisation of State

Normalization is the process of taking data from a problem and reducing it to a set of relations while ensuring data integrity and eliminating data redundancy. This really means that there should be exactly one place that contains the true value of something. For example, the bookmark details with a unique 1, can be used elsewhere by not copying over the entire details but merely referencing its This way the true value of this particular bookmark lies in just one place, sort of, eliminating the data redundancy and also ensuring its integrity since we know the bookmark details that exist in just one place are real and not tampered or stale or whatever. So, the basic ideas behind the normalization of state (with respect to the list of bookmarks mentioned above) are:

Each bookmark must have its place in the state.

Each bookmark must reference itself by its **Id** only.

Each bookmark must have a unique identification to identify itself.

Each bookmark must represent its properties in the form of an object such as name, url, and created properties in our case.

Ordering of the bookmarks must be depicted in the form of an array of Ids without defying the idea number 4 above.

Now let's apply these ideas to the aforementioned list of bookmarks in chronological order for the betterment of our learning and understanding of the normalization concept. As per rule 1 and 2, let us first define the shape of the state, in the form of an empty object. And inside of it, create a space for an individual bookmark as follows. And reference them by their Ids. Note that those numbers are not incrementing numbers but the unique identification number associated with the respective bookmark. You can use strings too if needed:

```
{
bookmarkEntities: {
entities: {
"1": {},
"2": {}
}
}
}
```

When we do this, we must adhere to the rule 3 which says that the individual bookmark should be identified using a unique identification number (or string). So, let us agree on the unique identification for each bookmark to be an Id property here:

```
{
bookmarkEntities: {
entities: {
"1": {
id: 1
},
"2": {
id: 2
}
}
}
}
```

Now as per rule 4, decorate the bookmark object with few more properties to have some meaning to it since it is quite dull at the moment:

```
{
bookmarkEntities: {
entities: {
"1": {
id: 1
name: 'Angular',
url: 'https://angular.io/',
created: '2020-05-17T15:37:44.047Z'

},
"2": {
id: 2,
name: 'NgRx',
url: 'https://ngrx.io/',
created: '2020-05-16T15:37:44.048Z'
}
}
}
}
```

And finally, as per rule 5, we can control the sorting order of the list which can be represented by the ids array as follows. This simply means that when we fetch the list of bookmarks later on, they will be in descending order as planned, that is, the 2nd bookmark with the Id equals 2 is at the top followed by the 1st bookmark with the Id equals 1:

```
const state = {
bookmarkEntities: {


ids: [2, 1],
entities: {
"1": {
id: 1,
name: 'Angular',
url: 'https://angular.io/',
created: '2020-05-17T15:37:44.047Z'


},
"2": {
id: 2,
name: 'NgRx',
url: 'https://ngrx.io/',
created: '2020-05-16T15:37:44.048Z'
}
}
}
}
```

Now that we have normalized the list of bookmarks, you must be wondering how does it help? Well, find me a bookmark whose Id is 2. You can do it very easily and that too in a single line of code with state.bookmarkEntities.entities["2"] (without sluggish loops and all).

Entity State Interface

Certainly, we need an Interface to represent the aforementioned state.bookmarkEntities object. Typescript turns it into an easy task though. You already know the desired traits of the state you write an interface for. Let us take a look:

```
export interface EntityState {
ids: string[] | number[];
entities: Dictionary;
}
```

We can also extend the above interface to add up more properties if needed. For example, what if we want to know whether the entities are loaded or not, just to prevent redundant remote HTTP calls:

```
export interface State extends EntityState {
loaded: boolean;
}
```

Now that we have the list of bookmarks in the form of entities and also the state interface to represent the same with three properties, ids, entities, and loaded. All that is remaining is a bridge to connect the consumer and the producer. In this case, the producer is our very entity and the consumer could be a component or a service that wants to read or write this state. Let us call that bridge, *Entity*

Entity Adapter

The Entity Adapter is nothing but a class providing various methods for performing operations against the collection such as adding new bookmarks, editing/deleting existing bookmarks, etc. Here we have written the simplest implementation of the much complex Entity Adapter (that we will later learn about in NgRx Entity) named The getInitialState method forms the initial state by spitting out the empty state which we can pass along to other methods such as The updateOne method takes the existing state and the new entity data we want to replace it with. It takes the id of the bookmark and changes to make (Notice that code snippet given below is abbreviated for brevity):

```
class createEntityAdapter{
getInitialState(props): State {
return {...state.bookmarkEntities, ...props}
}

updateOne(
update: {id: string | number; changes: object},
existingState: State
): State {
existingState.entities[update.id] = {
...existingState.entities[update.id],
...update.changes
};
return existingState;
}
.
.


.
}
```

Next, we must instantiate the adapter to call the aforementioned methods to manipulate the store as follows:

```
const adapter = new createEntityAdapter();

const initialState: State = adapter.getInitialState({
loaded: false
});
adapter.updateOne({id: 2, changes: {name: 'NgRx2'}}, initialState);
```

That's our Reducks Entity in action. Here we have updated the second bookmark's name by its Id, hence the updated state would look like this:

```
{
bookmarkEntities: {
ids: [2, 1],
entities: {
"1": {
id: 1
name: 'Angular',
url: 'https://angular.io/',
created: '2020-05-17T15:37:44.047Z'
},
"2": {
id: 2,
name: 'NgRx2',
url: 'https://ngrx.io/',
created: '2020-05-16T15:37:44.048Z'

}
}
}
}
```

This may be too much grunt work for you and you are right. However, this paves the way for us to understand and use NgRx Entity more sanely than ever, starting with installing the NgRx Entity package in the next section.

Installing @ngrx/entity package

In the sample application that we built in the second chapter, guards are responsible for fetching data from remote APIs and storing the same in the store afterward. However, the existing list of bookmarks is not in the form of entities. So, let's install NgRx Entity to override the existing behavior and store each and every bookmark as a separate entity:

```
ng add @ngrx/entity
```

This must-have added a package in package.json. Next, we will auto-generate bookmark-entity as follows:

```
ng g entity store/bookmark-entity/bookmark-entity --skipTests=true
```

Please answer the asked questions as mentioned below:

Q: Do you want to use the create function?

A: Type "Yes" and press Enter.

This must-have created 3 files mainly action, reducer, and model ones in src/app/store/bookmark-entity/ directory. This is where the NgRx Entity shines where all the necessary actions/reducers to manipulate the entities are auto-generated for you. But we do not need the bookmark-entity.model.ts file since we have the model defined already in app/shared/services/bookmark/bookmark.service.ts which we can reuse. So delete the bookmark-entity.model.ts file and reference the Bookmark model as follows in

```
import {createAction, props} from '@ngrx/store';
import {Update} from '@ngrx/entity';



import {Bookmark} from
'@app/shared/services/bookmark/bookmark.service';

export const loadBookmarkEntitys = createAction(
'[BookmarkEntity/API] Load BookmarkEntitys',
props<{bookmarkEntitys: Bookmark[]}>()
);

export const addBookmarkEntity = createAction(
'[BookmarkEntity/API] Add BookmarkEntity',
props<{bookmarkEntity: Bookmark}>()
);

export const upsertBookmarkEntity = createAction(
'[BookmarkEntity/API] Upsert BookmarkEntity',
props<{bookmarkEntity: Bookmark}>()
);

export const addBookmarkEntitys = createAction(
'[BookmarkEntity/API] Add BookmarkEntitys',
props<{bookmarkEntitys: Bookmark[]}>()
);

export const upsertBookmarkEntitys = createAction(
'[BookmarkEntity/API] Upsert BookmarkEntitys',
props<{bookmarkEntitys: Bookmark[]}>()
);

export const updateBookmarkEntity = createAction(
'[BookmarkEntity/API] Update BookmarkEntity',
props<{bookmarkEntity: Update}>()
);

export const updateBookmarkEntitys = createAction(
'[BookmarkEntity/API] Update BookmarkEntitys',
props<{bookmarkEntitys: Update[]}>()
);



export const deleteBookmarkEntity = createAction(
'[BookmarkEntity/API] Delete BookmarkEntity',
props<{id: string}>()
);

export const deleteBookmarkEntitys = createAction(
'[BookmarkEntity/API] Delete BookmarkEntitys',
props<{ids: string[]}>()
);

export const clearBookmarkEntitys = createAction(
'[BookmarkEntity/API] Clear BookmarkEntitys'
);
```

Entity State Interface

Remember that the interface and the adapter we wrote at the beginning of the chapter for Reducks Entity are no longer needed here since the NgRx Entity provides us with the generic interface by default. Quite similarly what we did above, we will remove the unwanted model and replace it in src/app/store/bookmark-entity/bookmark-entity.reducer.ts with the Bookmark model as follows (this is not the full source code of the file since untouched code is not added for brevity):

```
import {Action, createReducer, on} from '@ngrx/store';
import {EntityState, EntityAdapter, createEntityAdapter} from
'@ngrx/entity';

import {Bookmark} from
'@app/shared/services/bookmark/bookmark.service';
import * as BookmarkEntityActions from './bookmark-entity.actions';

export const bookmarkEntitiesFeatureKey = 'bookmarkEntities';

export interface State extends EntityState {
loaded: boolean;
}
export const adapter: EntityAdapter = createEntityAdapter();
export const initialState: State = adapter.getInitialState({
loaded: false
});
```

Here, we have extended the default generic EntityState interface to add new properties alongside Ids and entities such as loaded. The loaded property will be false by default as set in initialState and will be set to true once we load the list of bookmarks into this entity state later on.

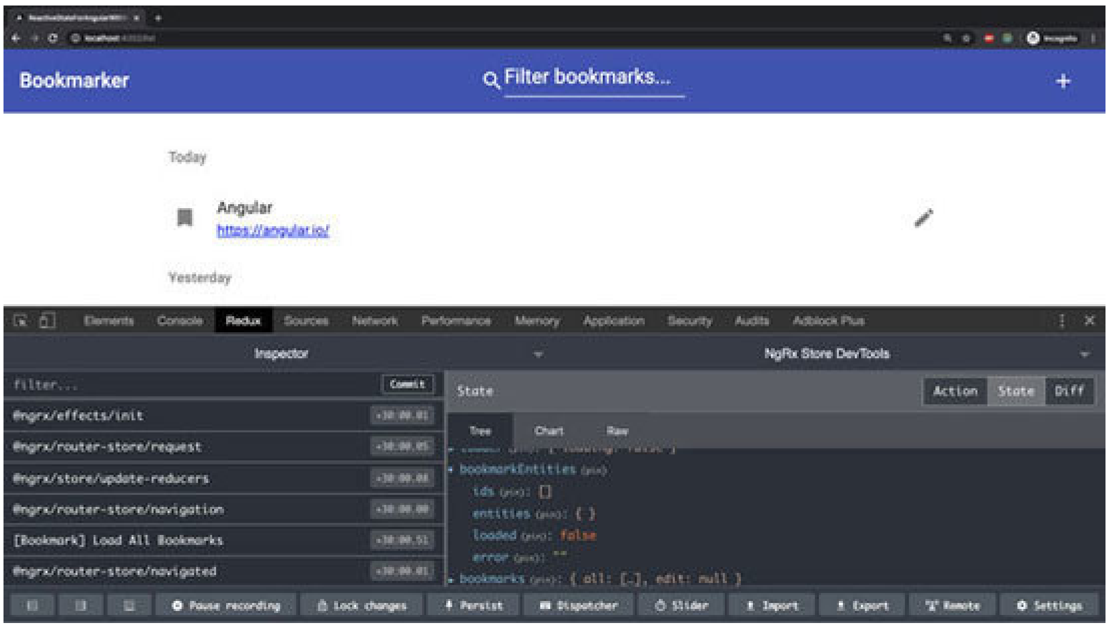
Entity Adapter

Using the createEntityAdapter method, we have instantiated our adapter for the Bookmark entity. This process exposes a bunch of pre-defined operation methods to manipulate the store one way or the other such as etc. Some of them will be used in the sample application going forward. However, we should not be invoking these adapter methods manually but again via pre-defined NgRx actions such as etc, respectively.

In order to understand this, we must first hook up the BookmarkEntity reducer to the store in

```
import {ActionReducerMap, MetaReducer} from '@ngrx/store';
import * as fromRouterStore from '@ngrx/router-store';

import {environment} from '../../environments/environment';
import * as fromToolbar from '@app/store/toolbar/toolbar.reducer';
import * as fromRouter from '@app/store/router/router.reducer';
import * as fromBookmarkEntity from '@app/store/bookmark-
entity/bookmark-entity.reducer';

export interface AppState {
filter: fromToolbar.FilterState;
router: fromRouterStore.RouterReducerState;
loader: fromRouter.LoaderState;
[fromBookmarkEntity.bookmarkEntitiesFeatureKey]:
fromBookmarkEntity.State
}

export const ROOT_REDUCERS: ActionReducerMap = {

filter: fromToolbar.reducer,
router: fromRouterStore.routerReducer,
loader: fromRouter.reducer,
[fromBookmarkEntity.bookmarkEntitiesFeatureKey]:
fromBookmarkEntity.reducer
};

export const metaReducers: MetaReducer[] =
!environment.production ? [] : [];
```

Now if you visit the http://localhost:4200/list page in the browser then the Redux Devtools must show you the feature key bookmarkEntities with initial states which is empty and not loaded:



```
@ngrx/effects/init
@ngrx/router-store/request
@ngrx/store/update-reducers
@ngrx/router-store/navigation
[Bookmark] Load All Bookmarks
@ngrx/router-store/navigated
```

```
▾ bookmarkEntities (pin)
    ids (pin): []
    entities (pin): { }
    loaded (pin): false
    error (pin): ""
▸ bookmarks (pin): { all: [...], edit: null }
```

Figure 6.1: *Bookmark Entities with initial states*

So the obvious next thing for us to do is to load the list of bookmarks into the bookmarkEntities store.

Replacing empty Entity Collection using .setAll

Earlier we used to load the list of bookmarks (after fetching from a remote API) in the bookmarks store using a custom written LOAD_ALL_BOOKMARKS action. However, we longer need it because we can use the auto-generated .loadBookmarkEntitys action to load the same list of bookmarks into the bookmarkEntities store in src/app/list/list.guard.ts as follows. Internally, the said action would stuff the bookmarkEntities store using the entity adapter .setAll method:

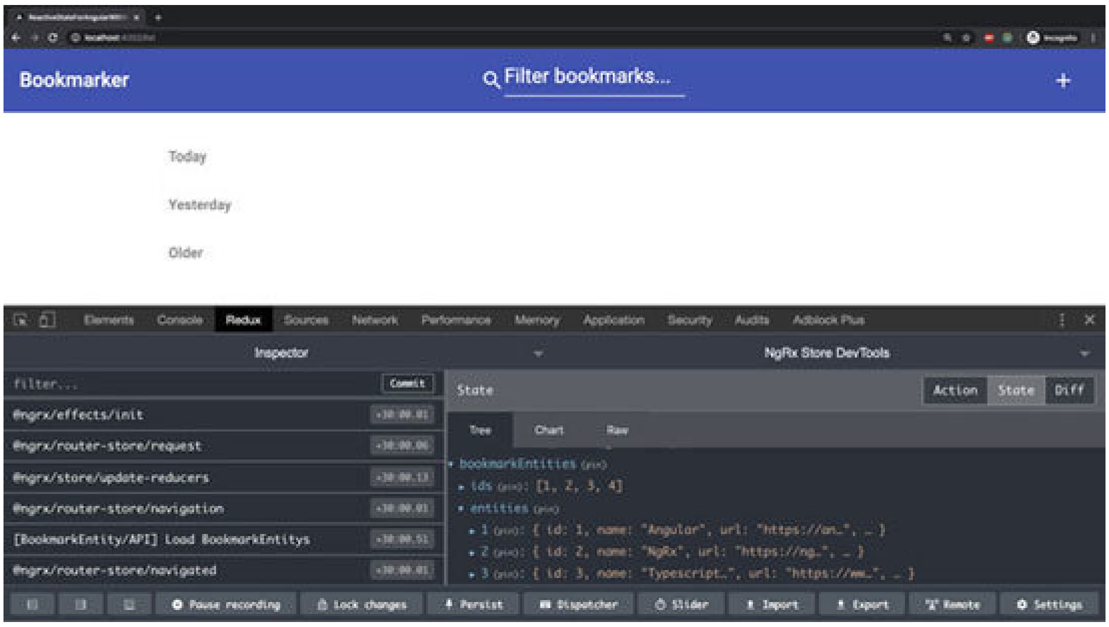
```
import {Injectable} from '@angular/core';
import {CanActivate} from '@angular/router';
import {Observable, of} from 'rxjs';
import {first, switchMap, tap, catchError} from 'rxjs/operators';
import {MatDialog} from '@angular/material/dialog';
import {Store} from '@ngrx/store';

import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';
import {AppState} from '@app/store';
import * as BookmarkEntityActions from '@app/store/bookmark-
entity/bookmark-entity.actions';

@Injectable({
```

```
providedIn: 'root'
})
export class ListGuard implements CanActivate {
constructor(
private readonly bookService: BookmarkService,
private readonly dialog: MatDialog,

public readonly store$: Store
) {}

canActivate(): Observable {
return this.bookService.getAll().pipe(
first(),
tap((bookmarkEntitys) => this.store$.dispatch(
BookmarkEntityActions.loadBookmarkEntitys({bookmarkEntitys})
)),
switchMap(() => this.activateRoute()),
catchError(() => {
this.dialog.open(ErrorDialogComponent, {
width: '400px',
data: {errorMessage: 'Sorry, unable to fetch bookmarks.'}
});
return of(false);
})
);
}

activateRoute() {
return of(true);
}
}
```

In addition to this, we also have to set the loaded property to true in the guise of having the entity state stuffed with the necessary data. So, update the loadBookmarkEntitys action in src/app/store/bookmark-entity/bookmark-entity.reducer.ts as follows. Here, using the ES6 spread operator, we first spread the existing properties in the state object and then override one of its properties. So that the loaded property will be truthy in the store as soon as the entity store is updated:

```
on(BookmarkEntityActions.loadBookmarkEntitys, (state, action) =>
adapter.setAll(action.bookmarkEntitys, {...state, loaded: true})),
```

Meanwhile, you can go to the list bookmarks route and take a look at the bookmarkEntities store for more detail as per [Figure 6.2]()below:



```
@ngrx/effects/init
@ngrx/router-store/request
@ngrx/store/update-reducers
@ngrx/router-store/navigation
[BookmarkEntity/API] Load BookmarkEntitys
@ngrx/router-store/navigated
```

```
▾ bookmarkEntities (pin)
  ▸ ids (pin): [1, 2, 3, 4]
  ▾ entities (pin)
    ▸ 1 (pin): { id: 1, name: "Angular", url: "https://an…", … }
    ▸ 2 (pin): { id: 2, name: "NgRx", url: "https://ng…", … }
    ▸ 3 (pin): { id: 3, name: "Typescript…", url: "https://ww…", … }
```

Figure 6.2: *Bookmark Entities with a list of bookmarks*

With these changes in mind, you will notice that the list component seemed broken but it is not actually. It however still tries to read the bookmarks from the old bookmarks store which we need to fix next.

Reading Entity State using Entity Selectors

The Entity Adapter, apart from the operation methods that we have seen before, also exposes a few selectors to read the whole or subset of the entity state. These selectors are lying at the bottom in They are,

selectIds: returns the value of the ids property as per *Figure*

selectEntities: returns the value of the entities property as per *Figure*

selectTotal: returns a total number of bookmarks that exist in the store.

selectAll: converts the value of the entities property into an array of objects, just like how it used to be in the bookmarks store earlier. Hence, this selector is ideal to use in the list component to fix the broken list of bookmarks in the UI.

For the sake of consistency, we must move these selectors into its selector file. For that, create the src/app/store/bookmark-entity/bookmark-entity.selectors.ts file and update it with:

```
import {createFeatureSelector} from '@ngrx/store';
import {State, bookmarkEntitiesFeatureKey, adapter} from
'@app/store/bookmark-entity/bookmark-entity.reducer';

const getBookmarkEntityState =
createFeatureSelector(bookmarkEntitiesFeatureKey);

export const {
selectIds,


selectEntities,
selectAll,
selectTotal,
} = adapter.getSelectors(getBookmarkEntityState);
```

So now we will use the selectAll selector in src/app/list/list.component.ts replacing the getAllBookmarks selector that we were using earlier in order to get the list of bookmarks:

```
import {Component, OnInit, ChangeDetectionStrategy} from
'@angular/core';
import {Store} from '@ngrx/store';
import {Observable} from 'rxjs';
import {map} from 'rxjs/operators';

import {Bookmark} from
'@app/shared/services/bookmark/bookmark.service';
import {isToday, isYesterday} from '@app/shared/util';
import {AppState} from '@app/store';
import {getFilterText} from '@app/store/toolbar/toolbar.selectors';
import * as bookmarkEntitySelectors from '@app/store/bookmark-
entity/bookmark-entity.selectors';

@Component({
selector: 'app-list',
templateUrl: './list.component.html',
styleUrls: ['./list.component.scss'],
changeDetection: ChangeDetectionStrategy.OnPush
})
export class ListComponent implements OnInit {
allBookmarks$: Observable;
todaysBookmarks$: Observable;

yesterdaysBookmarks$: Observable;
olderBookmarks$: Observable;
filterText$: Observable;

constructor(public readonly store$: Store) {}

ngOnInit() {
this.allBookmarks$ =
this.store$.select(bookmarkEntitySelectors.selectAll);
this.todaysBookmarks$ = this.allBookmarks$.pipe(
map(allBookmarks => allBookmarks.filter((bookmark) =>
isToday(bookmark.created)))
);
this.yesterdaysBookmarks$ = this.allBookmarks$.pipe(
map(allBookmarks => allBookmarks.filter((bookmark) =>
isYesterday(bookmark.created)))
);
this.olderBookmarks$ = this.allBookmarks$.pipe(
map(allBookmarks => allBookmarks.filter((bookmark) =>
!isToday(bookmark.created) && !isYesterday(bookmark.created)))
);
this.filterText$ = this.store$.select(getFilterText);
}
}
```

Go to the list bookmarks route to confirm if the list of bookmarks in the UI appears normal as before. It should.

Using cached Entity State Collection

Since it is way easier to add/update an individual bookmark (which we will look at soon) with NgRx Entity State, we can leverage the cached entities to prevent redundant remote API calls to fetch the same set of bookmarks over and over again every time we go to the list route. For that, we can rely on the new entity property, loaded, we added last time to check if the entity state is empty or not. If it is not loaded then we must call the remote API to fetch a fresh list of bookmarks and store them into the entity store. Otherwise, just read the existing data off of the Store to pass on for the list component to render.

First of all, in let us write a custom selector:

```
import {createFeatureSelector, createSelector} from '@ngrx/store';

import {State, bookmarkEntitiesFeatureKey, adapter} from
'@app/store/bookmark-entity/bookmark-entity.reducer';
import {getBookmarkId} from '@app/store/router/router.selectors';

const getBookmarkEntityState =
createFeatureSelector(bookmarkEntitiesFeatureKey);

export const {
selectIds,
selectEntities,
selectAll,
selectTotal,
} = adapter.getSelectors(getBookmarkEntityState);

export const haveBookmarkEntitiesLoaded = createSelector(

getBookmarkEntityState,
(state) => !!state.loaded
);
```

Here, the haveBookmarkEntitiesLoaded selector gets a hold of the entire bookmark entity state and confirms if the loaded property is truthy/falsy. Next, we will use the said selector in src/app/list/list.guard.ts as follows:

```
import {Injectable} from '@angular/core';
import {CanActivate} from '@angular/router';
import {Observable, of} from 'rxjs';
import {first, switchMap, tap, catchError} from 'rxjs/operators';
import {MatDialog} from '@angular/material/dialog';
import {Store} from '@ngrx/store';

import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';
import {AppState} from '@app/store';
import * as BookmarkEntityActions from '@app/store/bookmark-
entity/bookmark-entity.actions';
import {haveBookmarkEntitiesLoaded} from '@app/store/bookmark-
entity/bookmark-entity.selectors';
```

```
@Injectable({
providedIn: 'root'
})
export class ListGuard implements CanActivate {
constructor(
private readonly bookService: BookmarkService,
private readonly dialog: MatDialog,

public readonly store$: Store
) {}

canActivate(): Observable {
return this.store$.select(haveBookmarkEntitiesLoaded).pipe(
first(),
switchMap((haveBookmarkEntitiesLoaded) =>
haveBookmarkEntitiesLoaded
? this.activateRoute()
: this.bookService.getAll().pipe(
tap((bookmarkEntitys) => this.store$.dispatch(
BookmarkEntityActions.loadBookmarkEntitys({bookmarkEntitys})
)),
switchMap(() => this.activateRoute()),
catchError(() => {
this.dialog.open(ErrorDialogComponent, {
width: '400px',
data: {errorMessage: 'Sorry, unable to fetch bookmarks.'}
});
return of(false);
})
)
)
)
}

activateRoute() {
return of(true);
}

}
```

Here we simply check whether the entity state is already loaded or not. If loaded then just activate the list route. Otherwise, go and fetch the fresh list of bookmarks over the wire using a remote API and update the entity store before activating the list route. Although, this implementation would introduce one major bug in the overall flow, especially while creating or updating a bookmark that the list is not showing the updates (simply because we prevented refetching to avoid redundant remote API calls, right?). That means we also have to update the local entity store whenever a new bookmark is created or any existing bookmark is updated.

Adding/Updating Entities

We had started using the NgRx Effects to call remote APIs in *Chapter 4: NgRx Effects* whenever we create/update bookmarks. That means we have to put the logic to update the entity store there. But before that, we should be aware of NgRx Entity actions provided by the Entity Adapter reducer which will be suitable for these acts. There are in fact two operation methods we can leverage:

addBookmarkEntity: adds one entity into the existing entity collection. This internally uses an adapter.addOne operation method.

updateBookmarkEntity: updates one entity in the existing entity collection. This also supports partial updates; meaning we can control one or more properties to update in the entity. This internally uses an adapter.updateOne operation method.

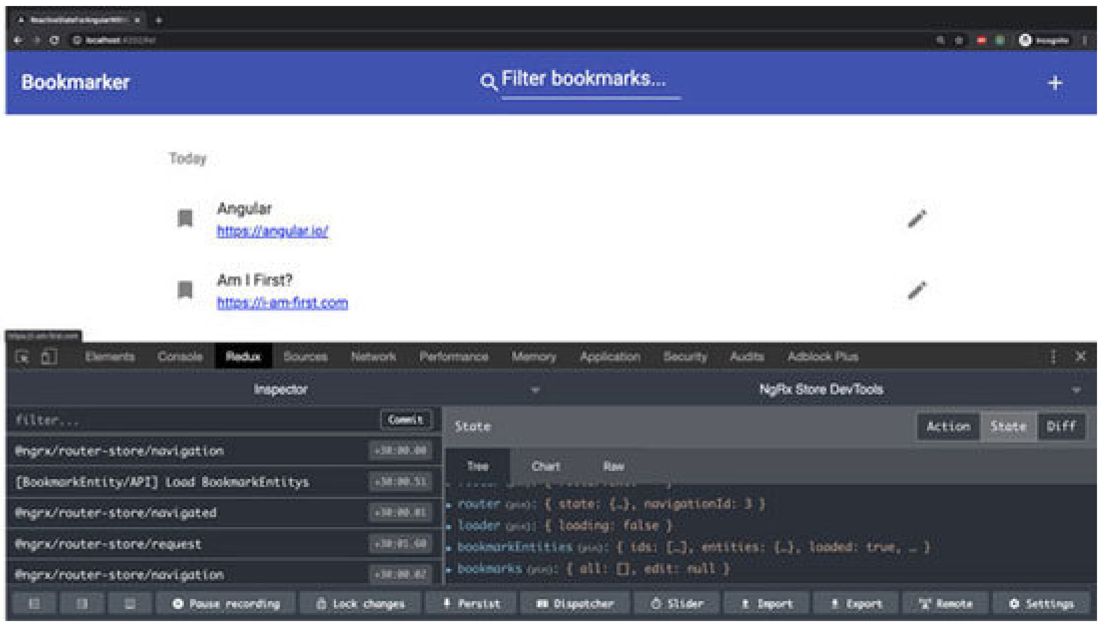
So update the saveBookmark$ effect in src/app/store/bookmark/bookmark.effects.ts as follows. Note that we have to pass the unique entity Id before storing the entity. For now, I have used the timestamp as the Id which may not be bulletproof in the production application but okay to use in the sample application here:

```
import {Injectable} from '@angular/core';
import {Actions, createEffect, ofType} from '@ngrx/effects';
import {exhaustMap, map, catchError, tap} from 'rxjs/operators';
import {of} from 'rxjs';
import {MatDialog} from '@angular/material/dialog';
import {Store} from '@ngrx/store';



import {AppState} from '@app/store';
import {SAVE_BOOKMARK, SAVE_BOOKMARK_ERROR} from
'@app/store/bookmark/bookmark.actions';
import {BookmarkService} from
'@app/shared/services/bookmark/bookmark.service';
import {ErrorDialogComponent} from
'@app/shared/components/error-dialog/error-dialog.component';
import {REDIRECT_TO_LIST_ROUTE} from
'@app/store/router/router.actions';
import * as BookmarkEntityActions from '@app/store/bookmark-
entity/bookmark-entity.actions';

@Injectable()
export class BookmarkEffects {
constructor(
private actions$: Actions,
private bookmarkService: BookmarkService,
private readonly dialog: MatDialog,
private readonly store$: Store
) {}

public saveBookmark$ = createEffect(() => this.actions$.pipe(
ofType(SAVE_BOOKMARK),
exhaustMap((action) => this.bookmarkService.save({...
action.bookmark}).pipe(
tap(() => this.store$.dispatch(
BookmarkEntityActions.addBookmarkEntity(
{bookmarkEntity: {...action.bookmark, id: new Date().getTime()}}
)
)),

map(() => REDIRECT_TO_LIST_ROUTE()),
catchError(() => of(SAVE_BOOKMARK_ERROR()))
))
), {useEffectsErrorHandler: false});

public saveBookmarkError$ = createEffect(() => this.actions$.pipe(
ofType(SAVE_BOOKMARK_ERROR),
tap(() => this.dialog.open(ErrorDialogComponent, {
width: '400px',
data: {errorMessage: 'Sorry, unable to create bookmark.'}
}))
), {dispatch: false});
}
```

If you had used the effect model in the edit route as suggested in *Chapter 4: NgRx Effects* (since we had not covered it there) then update the updateBookmark$ effect in the above file as well; ignore otherwise. Furthermore, the updateBookmarkEntity action takes the Id of the entity you want to update and the partial changes to make which is of type

```
public updateBookmark$ = createEffect(() => this.actions$.pipe(
ofType(UPDATE_BOOKMARK),
exhaustMap((action) => this.bookmarkService.update({...
action.bookmark}).pipe(
tap(() => this.store$.dispatch(
BookmarkEntityActions.updateBookmarkEntity(
{bookmarkEntity: {id: action.bookmark.id, changes:
action.bookmark}}
)
)),

map(() => REDIRECT_TO_LIST_ROUTE()),
catchError(() => of(UPDATE_BOOKMARK_ERROR()))
))
), {useEffectsErrorHandler: false});
```

Go ahead and add a new bookmark to make sure that the above changes work as intended:



Today

Angular
https://angular.io/

Am I First?
https://i-am-first.com

```
@ngrx/router-store/navigation
[BookmarkEntity/API] Load BookmarkEntitys
@ngrx/router-store/navigated
@ngrx/router-store/request
@ngrx/router-store/navigation
```

```
router (pin): { state: {…}, navigationId: 3 }
loader (pin): { loading: false }
bookmarkEntities (pin): { ids: […], entities: {…}, loaded: true, … }
bookmarks (pin): { all: [], edit: null }
```

Figure 6.3: *Adding a new bookmark entity*

Voila! It worked. But if you had paid attention from the first chapter itself, you may have noticed that the sorting of the bookmarks is a bit screwed. That the latest bookmark does not show up at the top of the list. We could easily fix the sorting in the list component after reading the entity collection. However, the NgRx Entity makes it even easier than that in the next section.

Sorting Entity State Collection

While creating the Entity Adapter, we are provided with two configurations that we had not used yet. As learned before that each entity must have a unique property and we decided to use the Id property from the entity to uniquely identify itself. However, in real-world applications, there might be a combination of one or more such properties or a completely different property altogether. In that case, we are allowed to override the default Id property using selectId configuration.

In update the existing adapter as:

```
export const adapter: EntityAdapter = createEntityAdapter({
selectId: entity => entity.id, //optional when there is an id property
in the entity
});
```

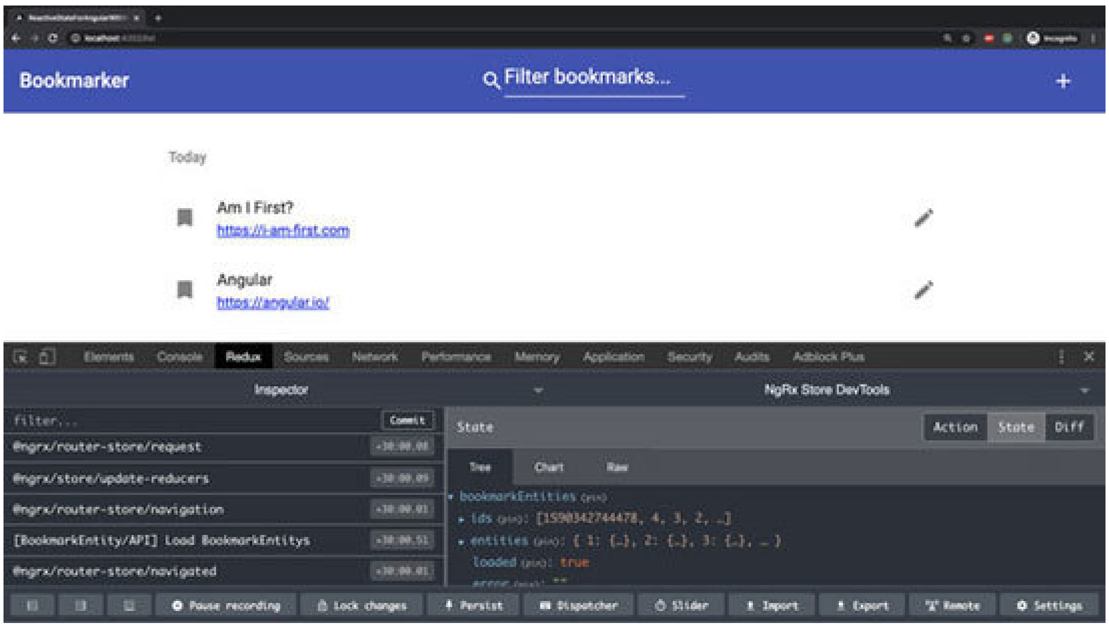
You can use any other property or even a combination of two or more properties from the entity. Try changing it to entity.name and check the Redux Devtools for fun!

Additionally, you can pre-sort the entity state collection while fetching, using the sortComparer configuration as follows. You can use any numeric property from the entity to sort the entity collection in ascending or descending order. Here, we specifically want to sort it in descending order by Id so that the newly created or edited bookmark would be listed on top:

export const adapter: EntityAdapter = createEntityAdapter({
selectId: entity => entity.id, //optional when there is an id property in the entity

sortComparer: (a: Bookmark, b: Bookmark) => b.id - a.id
});

Go ahead and add a new bookmark to find the sorting has been fixed as per *Figure*



```
@ngrx/router-store/request
@ngrx/store/update-reducers
@ngrx/router-store/navigation
[BookmarkEntity/API] Load BookmarkEntitys
@ngrx/router-store/navigated
```

```
▾ bookmarkEntities (pin)
  ▸ ids (pin): [1590342744478, 4, 3, 2, …]
  ▸ entities (pin): { 1: {…}, 2: {…}, 3: {…}, … }
    loaded (pin): true
```

Figure 6.4: *Entity Collection sorted in descending order*

Alright, the time has come to part ways. This journey through NgRx has taken you to some new places: from the fundamentals of reactive state to all the way to a fully-fledged web application, completely reactive, in Angular. Along the way, you have explored how Angular applications are built from scratch using Angular CLI and Angular Material in such a short time. You looked at how the imminent problem of state management in large applications could be tamed with Redux Paradigms by using a pure functional action-dispatcher-reducer pipeline. You experimented with your version of NgRx as well as NgRx Entity (in this chapter) to use that understanding to learn NgRx better. You also learned to isolate asynchronous events via NgRx Effects to keep the action-dispatcher-reducer pipeline intact. We eventually moved the router state into the store in order to ultimately make the NgRx Store a single source of truth. As I said at the outset, this book was not meant as a definitive guide to all things NgRx (since many new things in NgRx are not covered yet such as NgRx Data, NgRx Components, etc.), instead, by focusing on a variety of topics and how they pertain to your real applications here are merely the beginning of what you can do with NgRx. Also, remember that the Redux principles we covered in the first chapter are not confined to mere NgRx because, with the same learning, you can explore other state management libraries such as NGXS, XState, Akita, etc. Finally, I hope that this book has encouraged you not just to use NgRx in your projects but also to examine your code with a more critical eye toward many of the concepts covered in the book. I cannot wait to see what you build next.

Conclusion

In this chapter, we have improved the loading of the list of bookmarks using the NgRx Entity state. With NgRx Entity, we have treated each bookmark as a separate entity, standing tall without the herd. This allowed us to interact with the individual bookmark without thinking about other bookmarks while creating/editing with obvious performance gain. At the beginning of the chapter, we repeated ourselves (just like in *Chapter 1: R.O.C.K. solid Application* to make our own Reducks Entity in order to learn NgRx Entity better. Once we got the hang of the components of NgRx Entity in terms of Entity State Interface, Entity Adapter, Entity Adapter operation methods, and Entity Adapter configurations, we made some useful improvements in the application as well. We also prevented redundant remote API calls in the list component by using the cached entities.

The end of this chapter brought us to the end of this book. However, this is not the end of your life-long learning. I hope that this book has given you a necessary nudge to explore more such technologies or frameworks or libraries. And I wish that you have enjoyed and learned a lot by reading this book, just like me by writing it.

Questions

What is NgRx Entity State?

What is the benefit of NgRx Entity State?

What is the NgRx Entity Interface and what's the use of it?

What is NgRx Entity Adapter and what it does?