NGRX

Reactive State Management for Angular

Nils Mehlhorn

NgRx - Reactive State Management for Angular

Nils Mehlhorn

This version was published on 2020-11-24.

Contents

Preface

Share This Book

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

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

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

Feedback

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

Acknowledgements

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

About the Author

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

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

Chapter 1

Introduction

1.1 Motivation

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

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

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

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

This gets tricky when some part of state has to be shared between components that are fairly distant in terms of the view hierarchy. You'd pull at lot of state into higher up components, if not into the root component itself while other components might forward data that they're not really concerned with. We end up with these messy hybrids of state containers and view-state bridges only to serve the way in which the latter are organized.

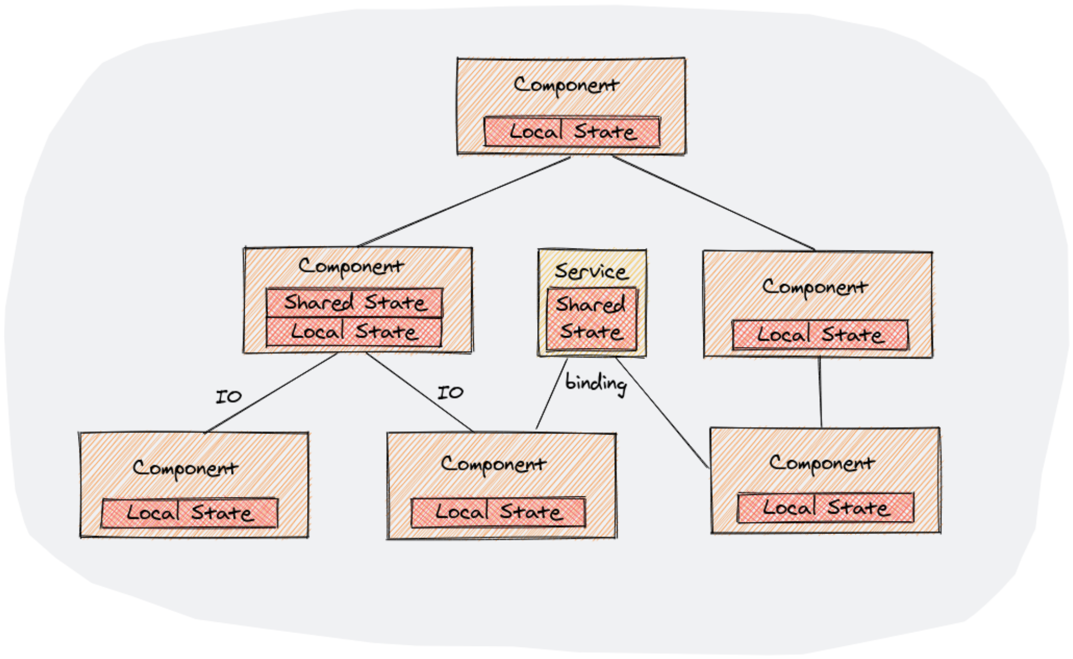

Figure 1.1: In Angular, state can be shared via inputs and outputs but also by binding to services outside of the view hierarchy. This way you don't need to funnel state across various parents when connecting otherwise distant components.

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

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

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

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

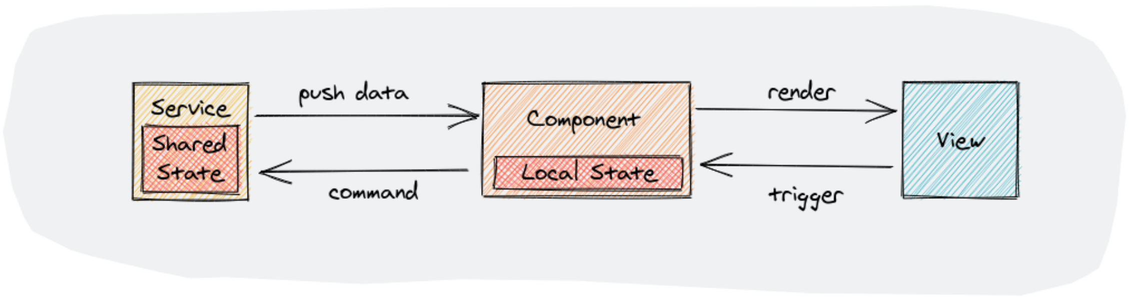

Figure 1.2: Separating commands and queries allows for unidirectional data flow and therefore optimized change detection and interception

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

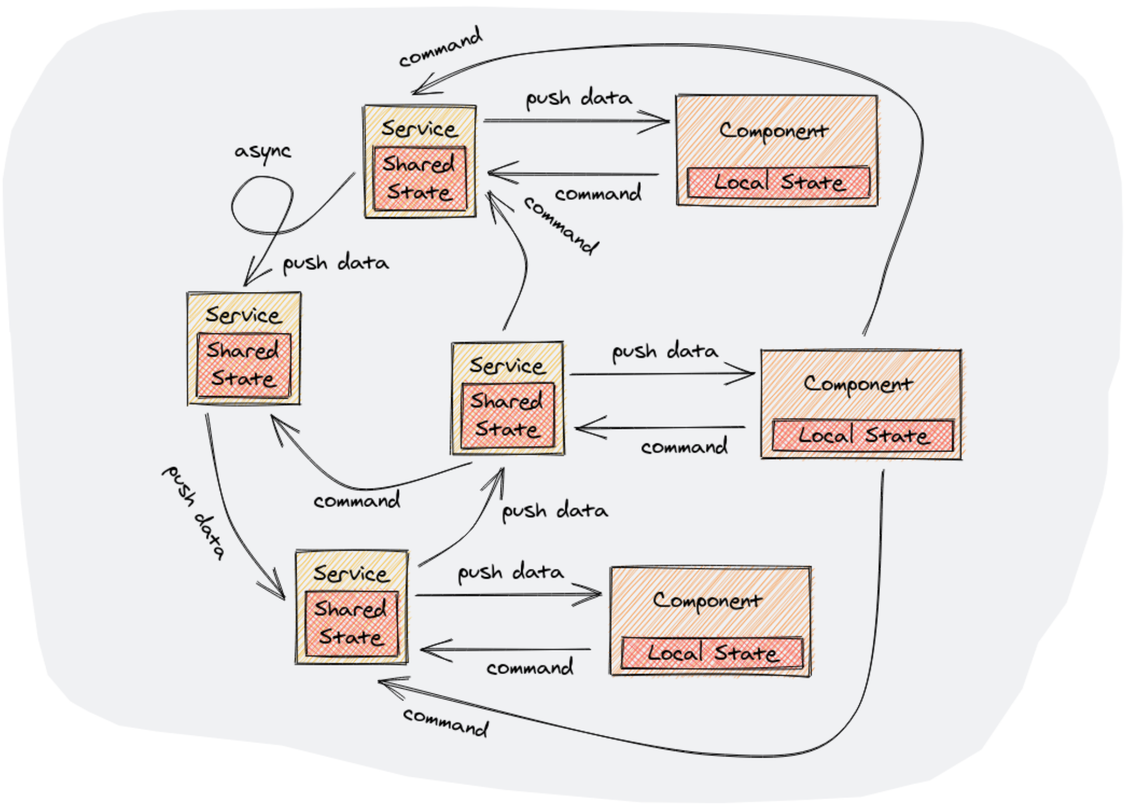

Figure 1.3: Synchronizing shared states can get messy, especially with a command-oriented architecture and when async tasks are involved

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

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

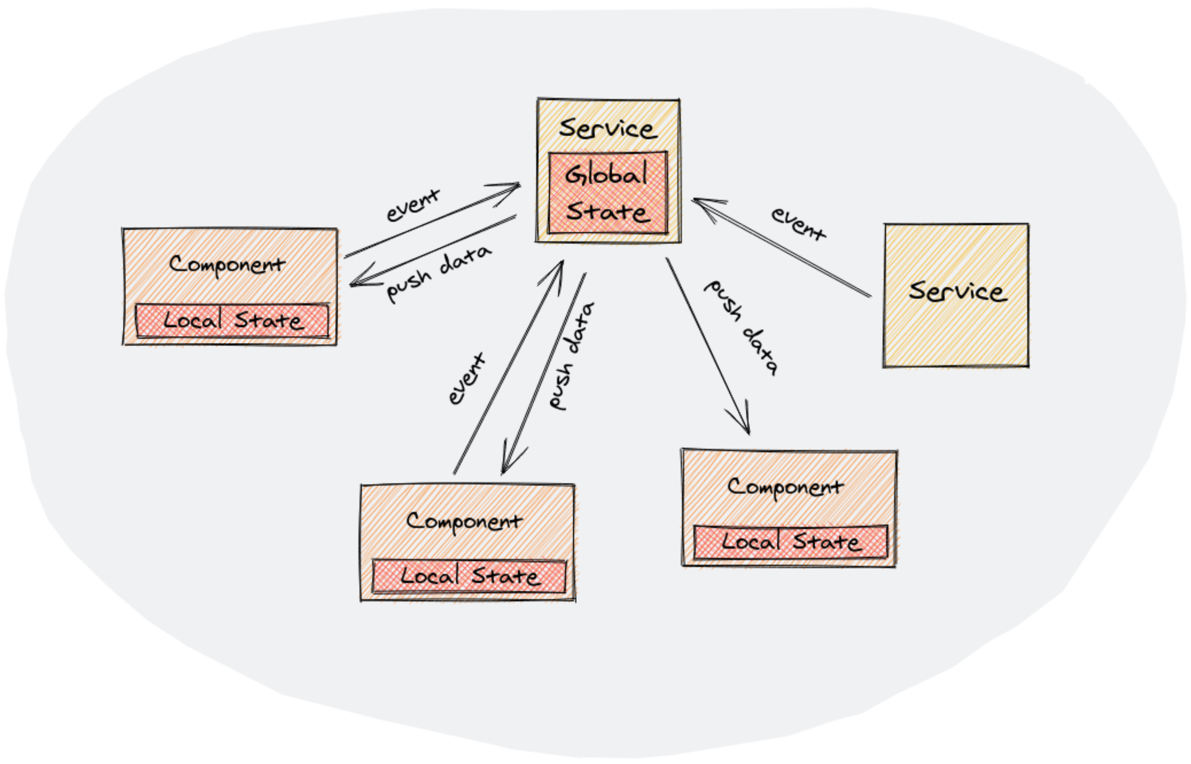

Figure 1.4: With NgRx, shared state is elevated to global state that updates based on events

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

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