

Creating Modern  
Applications with

**Nx, ANGULAR &  
NGRX SIGNAL STORE**

**Fabian Gosebrink**



reviewed and foreword by  
**Marko Stanimirović**

# Contents

|   |           |
|---|-----------|
| <b>Introduction</b>   | <b>6</b>  |
| Motivation . . . . .  | 6         |
| What This Book Covers . . . . .   | 6         |
| About the Author . . . . .  | 8         |
| About the Reviewer . . . . .  | 8         |
| Thanks . . . . .  | 8         |
| GitHub . . . . .  | 8         |
| <br>  |           |
| <b>Building Powerful Angular Applications: Key Mistakes to Avoid</b>                | <b>9</b>  |
| Neglecting the Use of Container and Presentational Components . . . . .             | 9         |
| The Problem with Component Inheritance . . . . .                                    | 14        |
| The Risks of Skipping State Management . . . . .                                    | 15        |
| Is State Management Necessary for My Application? . . . . .                         | 15        |
| Different Types of State in an Angular Application . . . . .                        | 16        |
| Understanding the Role of State, Container, and Presentational Components . . . . . | 18        |
| How State Enhances Your Application’s Logic and Testing . . . . .                   | 21        |
| State Management and Signals in Angular . . . . .                                   | 22        |
| Final Thoughts on Managing State . . . . .  | 25        |
| The Importance of Using Higher-Order Observable Operators . . . . .                 | 27        |
| Refactoring Nested Subscribes for Better Code . . . . .                             | 27        |
| Understanding How Higher-Order Operators Work . . . . .                             | 27        |
| An Overview of the Four Main Operators . . . . .                                    | 28        |
| Final Thoughts on Higher-Order Operators . . . . .                                  | 29        |
| The Dangers of Poor Code Organization . . . . .                                     | 29        |
| Closing Reflections . . . . .   | 31        |
| Summary and Key Takeaways . . . . .   | 31        |
| <br>  |           |
| <b>From Requirements to a first Application Structure with Nx</b>                   | <b>32</b> |
| Introduction to a Nx Workspace . . . . .  | 32        |
| What is Nx? . . . . .   | 32        |
| Comparing to the Angular CLI . . . . .  | 33        |
| Rethinking Libraries . . . . .  | 33        |
| Files and Folders in an Nx workspace . . . . .                                      | 35        |
| Architectural Constraints with scopes and types . . . . .                           | 37        |
| How the Nx affected Command Works at a High Level . . . . .                         | 38        |
| Nx’s visual workspace representation . . . . .                                      | 39        |
| Conclusion . . . . .  | 40        |
| Examine the sample application . . . . .  | 41        |
| Planning the Frontend – From UI Design to a First Architecture . . . . .            | 42        |
| Identifying the top-level features of your application . . . . .                    | 42        |
| Non-Route Based Features . . . . .  | 44        |
| Turning the Output into the First Architectural Sketch . . . . .                    | 45        |
| Reflecting the architecture in state . . . . .                                      | 47        |
| Further thoughts . . . . .  | 48        |
| Conclusion . . . . .  | 49        |

|  |            |
|--|------------|
| <b>Turning Plans Into Reality: Implementing the Sample App</b>                     | <b>50</b>  |
| Setting Up an Nx Workspace   | 51         |
| Working with environments (Eager Loaded)   | 54         |
| Adding the internal library  | 54         |
| Switching environments automatically when building                                 | 55         |
| Including Bootstrap in an application  | 57         |
| Implementing the layout (eagerly loaded)   | 58         |
| Adding the internal layout library   | 59         |
| Connecting the layout with the application   | 60         |
| Implementing the “About” Feature (Lazy Loaded)                                     | 62         |
| Adding the internal about library  | 62         |
| Connecting the lazy loaded feature with the application                            | 63         |
| Adding State Management with the NgRx Signal Store                                 | 65         |
| Implementing the Authentication (Eager Loaded)                                     | 66         |
| Adding the internal and third-party library  | 67         |
| Adding a service for the abstraction   | 68         |
| Creating the state layer   | 69         |
| Using the authentication   | 72         |
| Implementing the General Real Time Functionality (Eager loaded)                    | 75         |
| Adding the internal real time library  | 76         |
| Adding a service for the abstraction   | 76         |
| Creating the state layer   | 76         |
| Using the real time state in the application                                       | 78         |
| Implementing the Notification Feature (Eager Loaded)                               | 81         |
| Adding the internal and third-party library  | 81         |
| Adding a service for the abstraction   | 82         |
| Prepare the notifications to use in the Application                                | 82         |
| Implementing the “Dogs” Feature (Lazy Loaded)                                      | 84         |
| Adding the libraries   | 85         |
| Connecting the lazy loaded feature with the application                            | 87         |
| Planning the container components  | 89         |
| Implementing the domain (services, models and feature state)                       | 89         |
| Implementing the local state for the Main Component                                | 100        |
| Extending the state for loading, selecting and rating a dog                        | 101        |
| Consuming the state in the Main Component  | 106        |
| Creating the presentational component and passing data in to make the data visible | 107        |
| Planning “My Dogs” and “Add Dog” Functionality                                     | 112        |
| Adding the components, states and routes to show my dogs                           | 115        |
| Improving authentication to load secure data                                       | 119        |
| Implementing Adding a New Dog Functionality With State and Components              | 120        |
| Securing the routes using a functional guard                                       | 129        |
| Adding a dog’s detail page   | 131        |
| Improving the My Dogs Overview   | 135        |
| Conclusion   | 137        |
| <b>Ensuring Code Quality in Your Nx Monorepo</b>                                   | <b>138</b> |

|  |     |
|--|-----|
| Maintain Architectural Integrity with Module Boundaries . . . . .          | 138 |
| Using the <code>affected</code> command . . . . .                          | 143 |
| Using Prettier for consistent formatting . . . . .                         | 144 |
| Installing Husky to run pre-commit hooks . . . . .                         | 145 |
| Using Lint-staged to format only changed files . . . . .                   | 146 |
| Checking for blocked words and prevent them from being committed . . . . . | 147 |
| Building the Application Production Ready . . . . .                        | 149 |
| Conclusion . . . . .   | 150 |

|                   |            |
|-------------------|------------|
| <b>Appendices</b> | <b>151</b> |
|-------------------|------------|

*This book brings real-world application design to the forefront, offering a practical approach to building scalable Angular apps with NgRx SignalStore and Nx. It bridges foundational concepts with advanced techniques through clear examples and a project-driven structure, making it an excellent resource for developers working with modern Angular.*

*Marko Stanimirovic - NgRx Core Team*

# Introduction

## Motivation

Over the past ten years, I have consulted on and developed multiple projects of various sizes. These projects have ranged from small personal websites to large portals handling multi-million-dollar software products. During this time, I've encountered many mistakes — some of my own and others that my team and I worked to correct after identifying them. Each project brought new challenges, making the work both enlightening and unique.

This book is intended to serve as both an architectural and practical guide. To be honest, it could have been a series of blog posts, but I thought having everything in one place would be more useful. It will provide you with a guidance on how to start an Angular project the right way, addressing both large-scale considerations and specific details like routing, authentication, Angular signals, state management, and real-time communication updates. The book is designed to be more than just an architectural guide or a manual for particular Angular features. It will offer a solid starting point, explain the reason behind various best practices, and guide you from the beginning, the first thoughts, over sketching the first rough architecture until the (open) end of an Angular project. This approach will help you deliver projects faster, maintain your app at scale, and can help you to make better decision on the journey of your app.

My hope is that this book will prepare you to start your Angular projects with confidence, ready for whatever challenges the future of your project may bring.

The chapters will begin with theoretical concepts, and as we progress, we will first plan and then develop a real-world application that incorporates all the latest features in Angular, including:

- Signals
- Real-time communication with SignalR (WebSockets)
- Nx Workspace with architectural constraints
- State management with NgRx Signal Store
- Functional APIs
- Authentication

Both the application and this book will be continuously updated to reflect the latest standards, incorporating new version syntax and third-party libraries as they become relevant and useful for the application.

## What This Book Covers

The first part of this book is a general chapter and describes how to avoid several common pitfalls I discovered in Angular applications while developing. Best practices are proposed for structuring your application, managing state, and organizing your code to be maintainable and scalable.

You are going to understand why it is so important to separate container and presentational components, what pitfalls lie in component inheritance, and why you should think about state management. It also covers whether your application requires state management, mentions the different types of state, and shows how state management and signals can change your application's logic.

Besides that, the first chapter describes what the higher-order observable operators are, why they are so important, how to use them, and refactoring nested subscribes to keep the code clean and more maintainable.

The second part of this book is turning plans into reality by implementing a real-world Angular application within an Nx workspace using state management with the NgRx signal store. It begins with setting up the workspace and explores environment-specific setups, including how to switch environments automatically and integrate third-party tools like Bootstrap, toasters or authentication libraries.

You'll get to know one way of building an architecture by separating code into libraries, eagerly and lazily loading features like layouts and about pages, and structuring shared logic and UI components.

Authentication and real-time features are implemented using both internal and external libraries, with layered abstractions and strongly typed services. You will learn how to use state management with the NgRx to orchestrate complex interactions like real-time notifications and authenticated data flows.

A feature of showing pictures of dogs and rating them serves as a comprehensive case study. It showcases advanced architectural practices such as planning container components, creating a global and local state, structuring presentational components, handling feature routing, and implementing secure, interactive UI flows—including loading, rating, and managing dog entities. Throughout this part, you'll see how architecture and state go hand in hand to support scalability and maintainability.

The third part of the book begins with “Ensuring Code Quality in Your Nx Monorepo” and shifts focus toward maintaining long-term quality and consistency in your project. After implementing the core application, this chapter guides you through strategies and tooling to keep your codebase clean, enforce architectural boundaries, and enable team collaboration at scale.

It begins by introducing Nx-specific tooling like the affected command to understand how changes on code impact your workspace. You'll see how to enforce clear architectural constraints using Nx's tagging and module boundaries, ensuring that different application layers—such as features, UI, and shared libraries—remain properly isolated and aligned with your intended architecture.

You'll then learn how to integrate automated code formatting using Prettier, how to set up Husky to run pre-commit hooks, and how to use lint-staged to format only changed files. This ensures that code quality is not only enforced but also automated as part of your team's workflow.

Finally, the chapter shows how to build your application production-ready, so that your application is optimized, secure, and reliable before it reaches your users.

By the end of this part, you'll have a solid foundation not just for building Angular applications—but for maintaining a healthy, high-quality, enterprise-ready codebase that can evolve over time without degrading in structure or performance.

## About the Author

Fabian Gosebrink

Fabian is a passionate web developer, Microsoft MVP, Google Developer Expert, trainer, and speaker specializing in Angular, state management, and scalable architecture. With deep experience in the Angular ecosystem, he helps teams build maintainable full-stack applications using Angular, NgRx, Nx, and .NET. Fabian actively bridges communities and shares his expertise across technologies. He supports developers through workshops, courses, and tailored consulting. Fabian regularly speaks at conferences and community events, sharing practical insights on topics like signal-based state management, architectures, and real-world application design. He's also an Nx Champion and contributes to the Angular community through blog posts, open-source projects, and training content—including courses on Pluralsight.

## About the Reviewer

Marko Stanimirovic

Marko is a core member of the NgRx and AnalogJS teams, a Google Developer Expert in Angular, and an organizer of the NG Belgrade conference. He actively contributes to open-source software, shares knowledge through technical articles and talks, and enjoys playing the guitar. Marko holds a Master of Science in Software Engineering from the University of Belgrade.

## Thanks

No book is ever the result of a single person's effort—and this one is no exception.

I would like to express my warmest thanks to everyone who helped bring this project to life.

To my reviewer Marko: thank you for your time, your honesty, and your sharp eye. Your feedback helped shape this book into something better with every iteration. Your attention to both the technical details and the big picture made a real difference and I learned tons!

To the designer of the book's cover Dusko: your visual work brought a unique personality to these pages. Thank you for translating abstract ideas into a clear and beautiful image that gives this book its visual identity.

I'm incredibly grateful for your help.

## GitHub

Source code (which might slightly differ from the samples although I try to keep them up to date are available on) [GitHub Dog Rate App](#)

## Building Powerful Angular Applications: Key Mistakes to Avoid

This might be the most opinionated section, but it's also the one I'm most passionate about. In fact, it's probably what inspired me to write this book. The following principles are some (!) of the principles that *I personally* consider essential for creating maintainable Angular applications — applications that are easy to understand, where developers can quickly get what's happening, and that are simple to test, debug, and extend with new features.

These are not all the recommendations I have for an Angular application. This of course also depends a lot on the application, the team and the product itself. Further, it does *NOT* mean that I'm trying to force you to follow these practices or that your project will *only* be successful by following them.

I'm fully aware that projects can succeed without these guidelines and can differ significantly. However, if I could offer advice based on my experience, these would be my top recommendations out of many. These are my go-to principles or tips that have consistently worked well for me on many projects.

### Neglecting the Use of Container and Presentational Components

In Angular, we can generally work with two types of components: container components and presentational components. These are sometimes referred to as stateful and stateless components, or smart and dumb components. While the names differ, they describe the same concept. Both types are Angular components with metadata, templates, and some logic, but their purposes differ significantly.

Container components are responsible for getting your data and handling data related logic. For example, they can call APIs through services to fetch the data you want to display, such as a user list for example. They are the “workers,” handling main tasks and managing state which influences the application's state, which is why they're often called smart or stateful components.

On the other hand, presentational components don't know where the data comes from. Their role is to take care of how the data is displayed. For instance, if you want to show the user list in a table, the presentational component handles displaying the table and binding the received data to it. If you decide to switch the display from a table to a list for example, you only need to change the presentational component. It receives data through an `@Input()` or a signal `input()` and doesn't concern itself with the data's source — it just displays the data in the correct way. On the other hand, if you decide to replace the data source or state layer, only the container components need to be changed and the presentational components won't be touched at all.

If a presentational component needs to manipulate some data it shouldn't do this directly, instead it informs the container component by emitting an event via an `@Output()` `EventEmitter` or `output()`. The container component being “smart” does the actual manipulation. It has the service injected and knows what methods to call and how to handle the task that the presentational component emitted.

With this separation, it becomes much easier to understand what the application is doing and where specific actions are being performed. Additionally, it improves readability in the HTML, allowing you for example to break down a 100-line markup into multiple, well-structured tags that are both readable and understandable. This makes a significant difference when understanding your code also in the HTML.

Let's consider a to-do application here. The container would fetch and manage the todo items, while the two presentational components would:

- Encapsulate a form and emit an event when the user wants to add a new todo item.
- Display the to-do items in a given format and trigger an event if an item is deleted, as this is a feature of the list in the UI in this case.

This separation of responsibilities makes your application more maintainable, as each component has a clear, focused role.

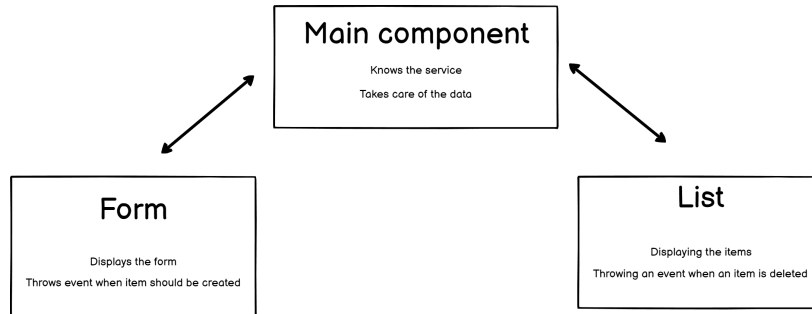


Figure 1: Overview of simple container presentational components

The following examples provide a quick overview of how this concept can be implemented in code.

```

@Component({
  selector: 'app-todo-main',
  imports: [..., TodoFormComponent, TodoListComponent],
  ...
})
export class TodoMainComponent implements OnInit {
  private todoService = inject(TodoService);

  items: ...

  ngOnInit(): void {
    this.todoService.getItems()...;
  }

  addTodo(value: string) {
    this.todoService.addItem(value)...;
  }

  deleteTodo(item: Todo): void {
    this.todoService.removeItem(item.id)...;
  }

  markAsDone(item: Todo): void {
    this.todoService.updateItem(item)...;
  }
}

```

```

}

@Component({
  selector: "app-todo-list",
  imports: [...],
  templateUrl: "./todo-list.component.html",
  styleUrls: ["./todo-list.component.scss"],
})
export class TodoListComponent {
  items = input<Todo[]>([]);
  doneItems = input<Todo[]>([]);

  markAsDone = output<Todo>();
  delete = output<Todo>();

  moveToDone(item: Todo) {
    item.done = !item.done;
    this.markAsDone.emit(item);
  }

  deleteItem(item: Todo) {
    if (confirm("Really delete?")) {
      this.delete.emit(item);
    }
  }
}

```

```

@Component({
  selector: "app-todo-form",
  imports: [ReactiveFormsModule],
  templateUrl: "./todo-form.component.html",
  styleUrls: ["./todo-form.component.scss"],
})
export class TodoFormComponent {
  todoAdded = output<string>();

  private FormBuilder = inject(FormBuilder);

  form = this.FormBuilder.group({
    todoValue: ["", Validators.required],
    done: [false],
  });

  addTodo() {
    if (!this.form.value.todoValue) {
      return;
    }
  }
}

```

```

    }

    this.todoAdded.emit(this.form.value.todoValue);
    this.form.reset();
  }
}

```

And then, putting it all together in the template:

```

<div class="...">
  <app-todo-form (todoAdded)="addTodo($event)"></app-todo-form>

  <app-todo-list
    [items]="items$ | async"
    (delete)="deleteTodo($event)"
    (markAsDone)="markAsDone($event)"
  ></app-todo-list>
</div>

```

As you can see here, we inject the `TodoService` only into the main component. The other two components have no idea where the data is coming from. They are getting passed the data and doing something with it. The presentational components in the template are like methods you can call and pass parameters to it. The form component only focusses on handling the form correctly with a clear defined API (inputs/outputs). The list component does the same for the list.

Of course, you could inject the service into the presentational components as well. They could fiddle with the data directly through service calls themselves. Then they would also have to notify the other components directly to communicate things like adding and deleting a to-do item. That's where things can start to get messy. This is a good example of how we avoid having lots of points in the application that influence the state, the list of to-dos, and instead concentrate this logic to just one place in the container component, which is already a big improvement, even for such a small app as this.

I often compare this to doing groceries with 10 other people helping you, and you have only one grocery list. You could tell each person to get a specific item and have them report back when they've added it to the cart, allowing you to mark it off your list. Or you could give each person their own copy of the grocery list, and everyone runs around collecting items. But then all the people must inform each other that an item has already been picked up, multiplying the communication overhead.

The to-do application is a very simple example and is therefore not a perfect analogy. But it effectively demonstrates the principle. Even at the component level, we can achieve a significant separation of concerns. Angular's built-in input/output features are designed for this purpose, allowing us to create a well-structured, maintainable component-based architecture and let components communicate among each other with a clearly designed API.

Having container and presentational components provides you with an explicit, one-way dataflow: data flows down and events flow up. It makes it easier when something happens to tell where it happened, because it reduces side effects. And makes debugging easier: if the data shown is wrong, you have to know only whether the presentational component is wrong, or the data was wrong when it was passed down. Such separation of concerns can increase the maintainability of your app drastically.

Further, it becomes much easier to test presentational components, as they have fewer dependencies. The developer can just set properties and check what's being rendered, not testing all the other side effects.

This approach also helps when refactoring big components to get an overview of *what* they really do. You can split up injections, maybe you can separate multiple container but even more presentational components even. And you get an idea of what they really do: Over the HTML tags as well as the code of the components typescript file.

The separation between container and presentational components is important to create maintainable Angular applications. I personally would consider this one of the top reasons why some Angular projects are failing or are hardly maintainable. Not the existence of the file separation only, but the mindset and the sense behind it. This also has to do with state and how we manage state - what we are going to discuss later in this book.

*“What if there are deeply nested components with events bubbling up through multiple levels?”*

This can get cumbersome when you have events that need to bubble up through multiple levels, but on the other hand there is a pattern which can keep the consistency in the codebase. If the levels are not that deep (maybe around 3 which is my personal level. Depending on the event itself it can be more or less) bubbling events still work very well. The disadvantage of having components only passing events through is justifiable when you consider the advantage of sticking to an established pattern and stay in reoccurring code which you know and find all over your application.

If you need to go deeper levels, another solution would be to use other communication systems like a service with subjects or signals for communication. This approach, being used with care as you might end up in messy code, can absolutely help.

Also reconsidering the current approach or architecture can be a possibility. Which is led to by the next question:

*“There is not just one(!) container component in my app. There are multiple container components.”*

In apps other than a demo application, you would absolutely have multiple container components. And of course, multiple container components within the same page can be used. This makes the component hosting all the container components a shell component. Inside of the shell component, there will be multiple container components that inject data or stateful services and forward data to the presentational components. In this case, the shell component will be responsible for the coordination between container components within that page.

Obviously, you cannot manage the whole app with one container component, but the same pattern repeats in different parts of the app. It is more about respecting the pattern as having *one* container component. Having to manage 300 stateful components is easier than having 1000 intercommunicating components. This will be even easier by using a state coordination tool, which we will be covering later in the book. Even without that, the idea of having more “dumb” components whose only job is to receive data makes your app more maintainable, predictable and testable, than having every component talk to many others through services, which could lead to messy, hard-to-maintain code, and more importantly code that's hard to test.

## The Problem with Component Inheritance

Components in Angular are Typescript classes with associated metadata and a template. Often, they encapsulate logic, and that logic is sometimes the same across components. Many other programming languages, like Java and C#, share logic between or extend classes through inheritance. In practice, this usually means that logic is extracted into an inherited base component or class shared by several components and is often called something like `BaseComponent` or `ComponentBase`. Deriving classes extend this base class, reaching out to logic which is encapsulated in this base class.

The problem with this approach in Angular applications is that it really complicates the code, and results in tightly coupled components, exactly the opposite of what's intended by a lightweight modular component architecture. If any child component overrides and re-implements some method coming from the base components code, it gets even worse to predict and test. It is also impossible in Angular to inherit a components template.

You also need to handle the lifecycle hooks on both the base and the child. Unit-testing these components is a pain as well, since you may want to mock things that are not even visible or used within the component.

In other words, it is bad practice to inherit components within Angular. How can the logic be shared effectively, then? We can the principle of composition over inheritance. Angular is designed with dependency injection in mind, which enables us to share functionality between components through injectable parts like services rather than inherit from components and avoid all pitfalls with component inheritance.

## The Risks of Skipping State Management

### Is State Management Necessary for My Application?

One ongoing discussion is whether there needs to be a dedicated solution for state on an application. On the one hand it creates much boilerplate and complexity, while on the other hand it is stated that it's necessary in every project. For me personally, this is clear-cut: the only time I wouldn't use a structured state solution is on a proof of concept (PoC) or a demo that needs to show only something small and not directly related to state. In any real application, be it for fun, a side project, a personal project, or a professional one, I would highly recommend it. The one thing I learned is: the state in an Angular app is the most critical aspect to handle and at the same time, it's one of the most complex tasks.

In my experience, the most successful and fastest projects I've worked on have used a tool to handle state. The clear separation of responsibilities in those tools and libraries has a huge impact on productivity. With tools like NgRx<sup>1</sup>, which offers the separation of reducers, actions, effects, and selectors, there's little room for confusion about where to place logic. This is extremely helpful for developers. When implementing a feature, every team member knows exactly what to do and what to expect from a code change in a pull request. Because it is repeating pattern every time. Should a new user be added? Actions, effects, a reducer, and selectors to choose what to display are needed. Should a user be deleted? The same process applies. Need to open a modal? The process remains consistent: actions, an effect, a reducer method, and selectors. It becomes almost routine, and that's a good thing. This reduces technical discussions about how to implement things dramatically and leads to more productivity as well as a clear separation of concerns in code. It provides a specified ruleset for developers which, in my opinion, helps the people a lot.

The technical aspect, which we will also explore, is an important one. However, what stands out the most about NgRx is its clear pattern, well-defined rules, and through that the boost in productivity I've experienced in multiple projects. The projects that used NgRx were the most productive, the fastest and therefore the cheapest.

Speaking about the latest invention, the NgRx Signal Store<sup>2</sup> does not follow the principles of having all those actions, reducer, selectors and effects like the "classic" NgRx had. It is reducing all this boilerplate, but it provides the same: routine. Clear principles to follow and a guide to have a place for logic without the mentioned bits and pieces. We will use the NgRx Signal Store in this book in the implementations so that you can see it in action right away.

This clear separation of concerns makes testing easier, too. Since the implementation follows a consistent pattern, so does the testing. You can test all the effects or reducers etc. all with the same approach.

*Small applications do not need NgRx.*

NgRx often feels heavy and is sometimes skipped because the application seems small. There are two issues with this approach. First: What exactly is "small"? It's difficult to define a threshold for when an application is "big" enough to be "ready" using NgRx. Is it based on lines of code? The number of components? Modules? Features? Folders? There is no definitive answer to this question — if there's an answer at all. It depends on various factors, and it's especially challenging for beginners to know when the app is ready to introduce NgRx. Experienced Angular developers might have a better sense of when the application is big enough, which leads to my second point: Applications can and will likely grow. And

---

<sup>1</sup><https://ngrx.io/>

<sup>2</sup><https://ngrx.io/guide/signals>