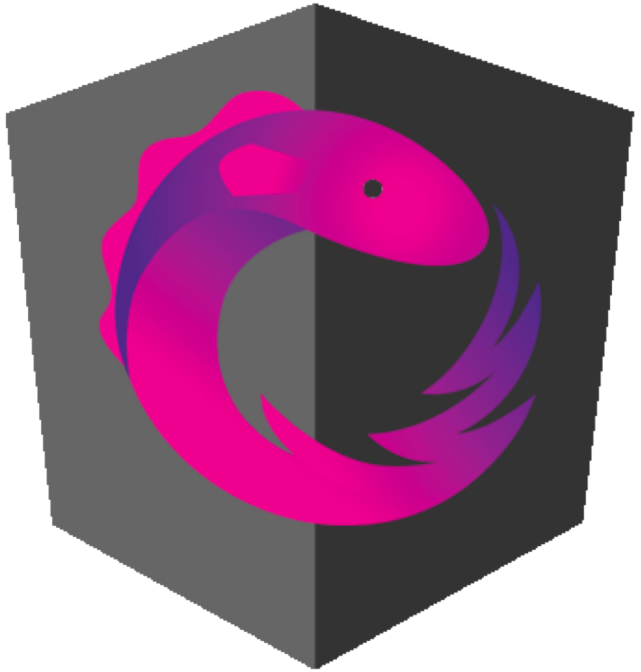
Angular Advanced
State management with @ngrx/store

Peter Kassenaar –
info@kassenaar.com

What is State Management?

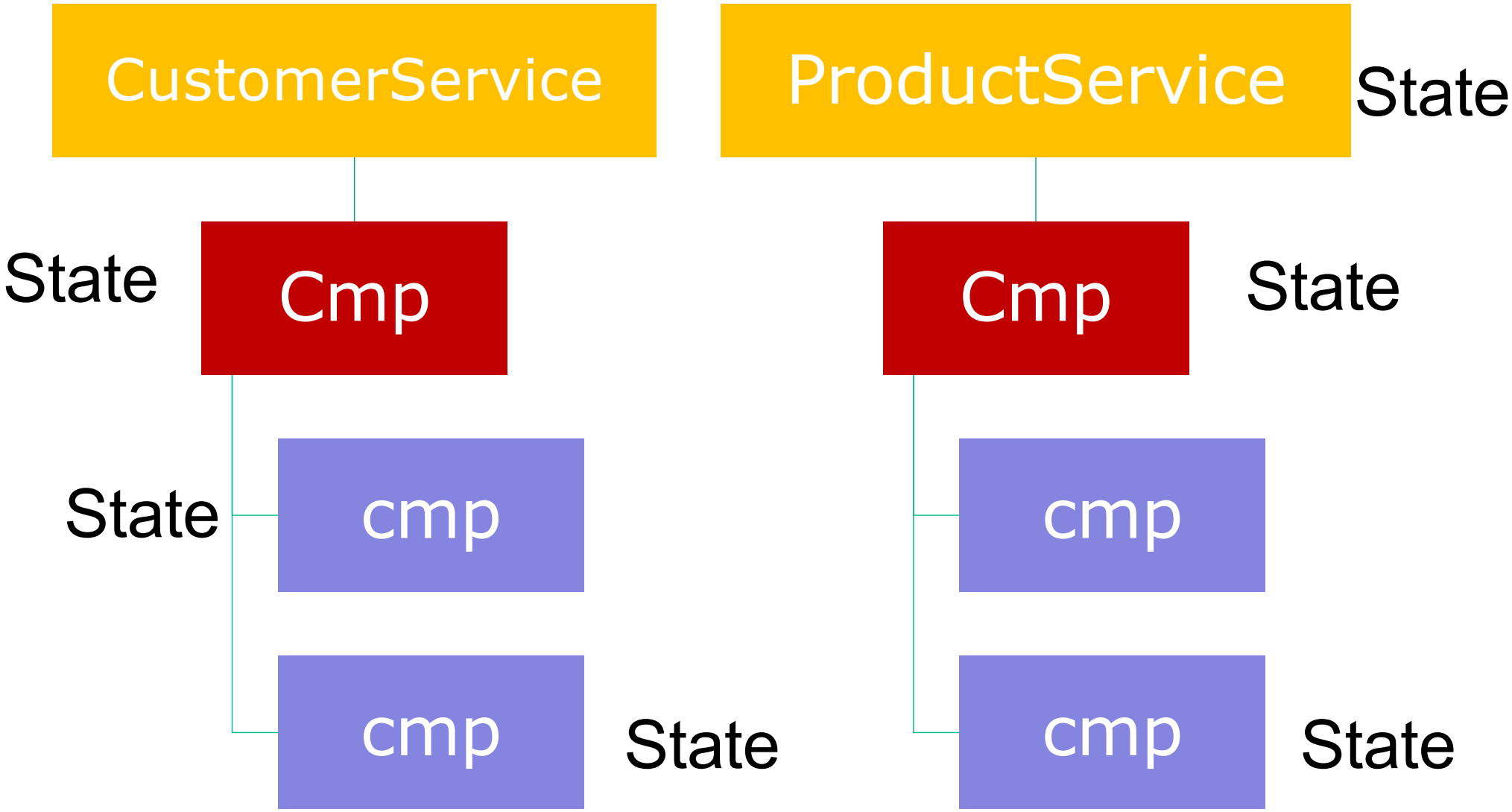
- Various design patterns, used for managing *state* (data in its broadest sense!) in your application.
- Multiple solutions possible – depends on application & framework



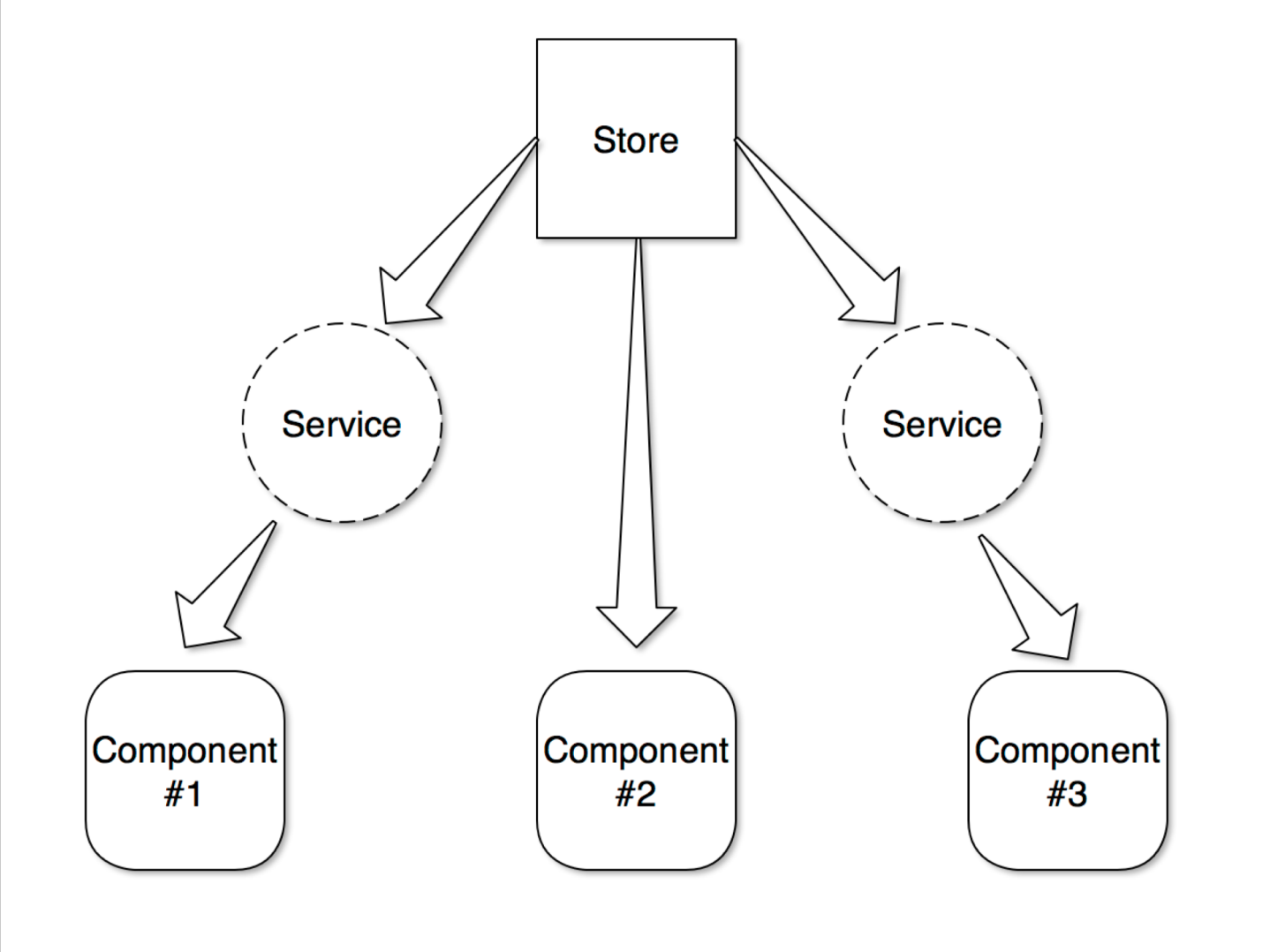
Benefits of using a store

- State is only changed in a controlled way
- Component state is also driven from the store
- Based on immutable objects – b/c they are predictable
- In Angular – immutability is fast
  - Because no changes can appear, no change detection is needed!
- Developer tools available to debug and see how the store changes over time
  - “Time travelling Developer tools”

State management without a store

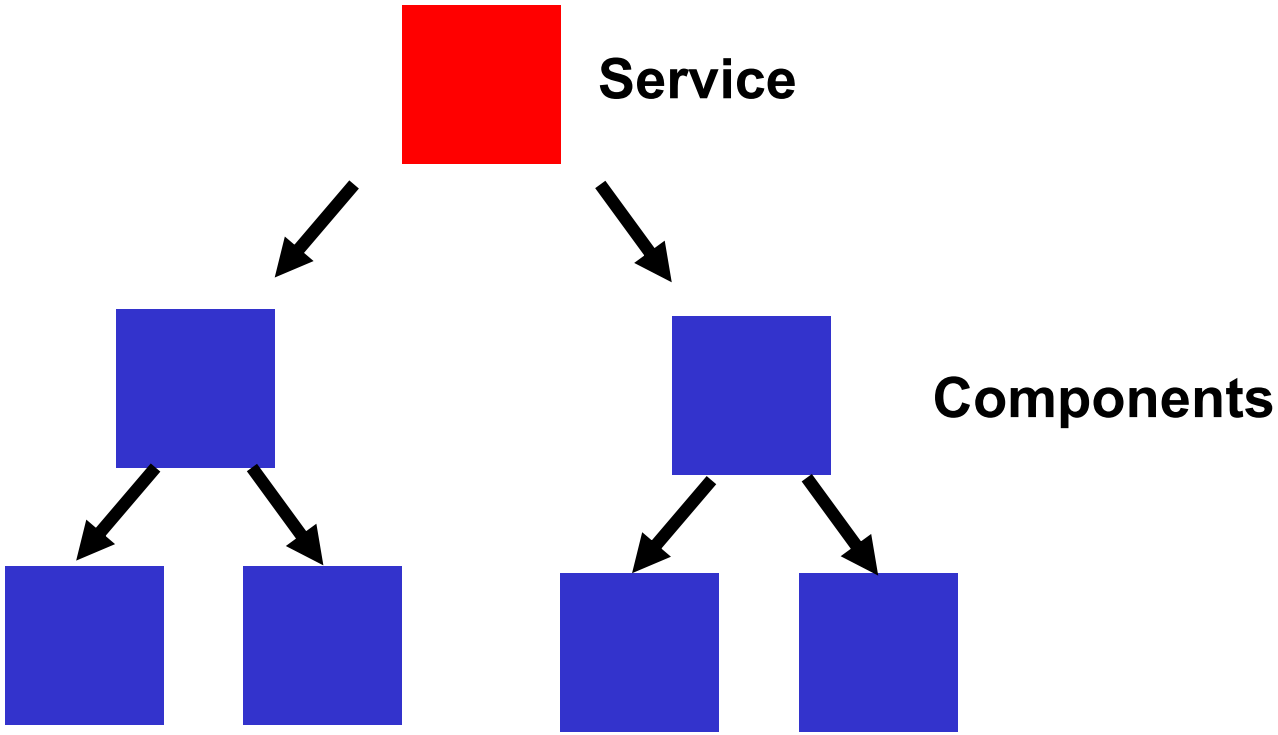


Store architecture - #1



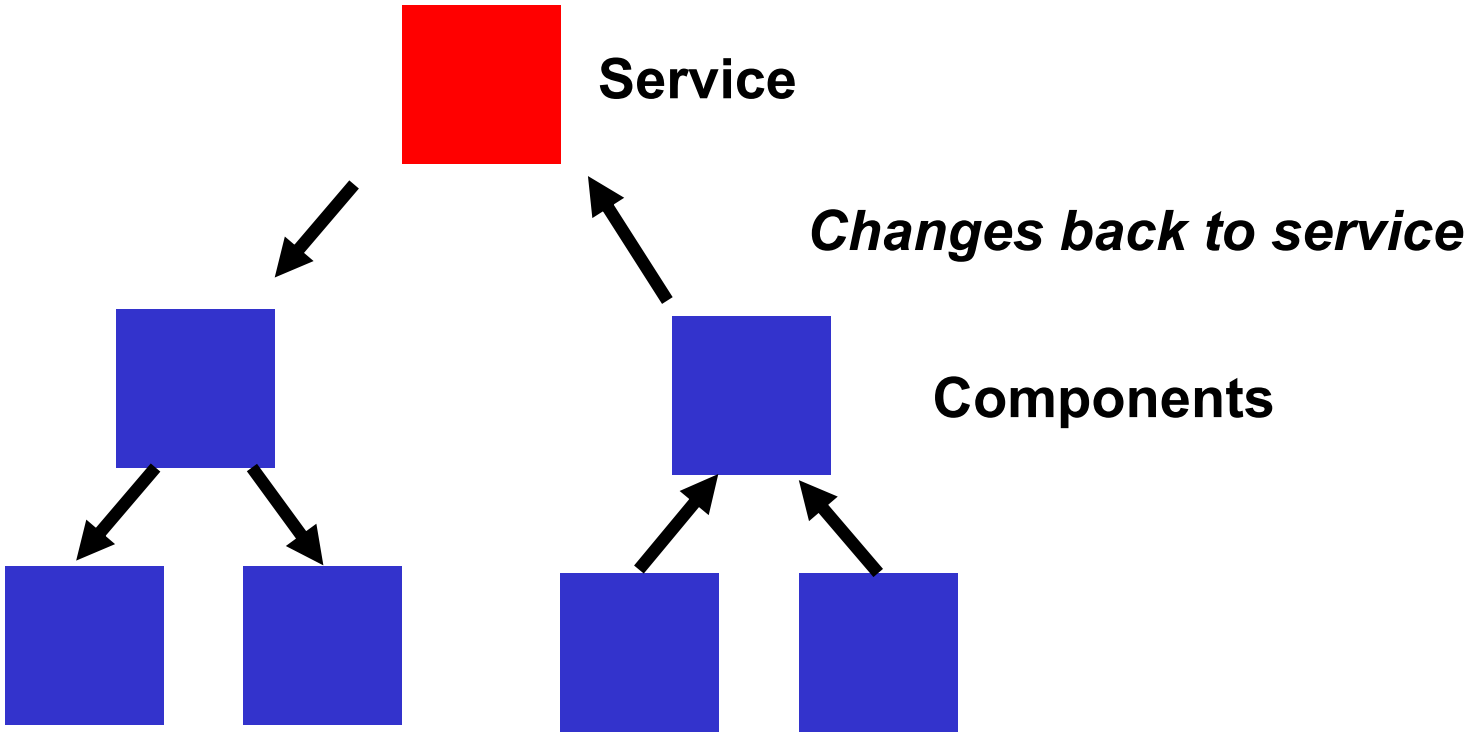
https://gist.github.com/btroncone/a6e4347326749f938510

Store architecture - #2 - traditional

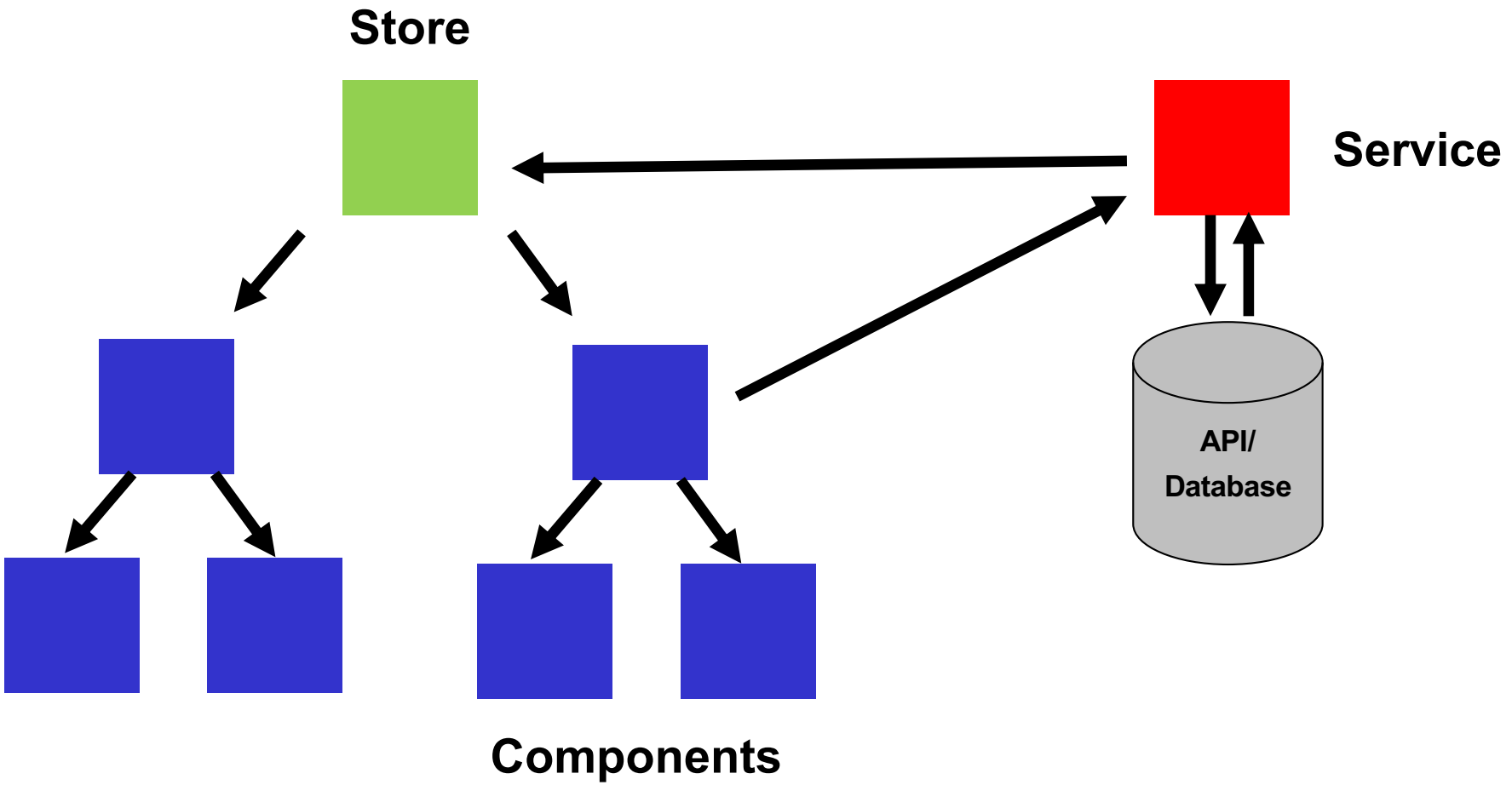


https://ultimateangular.com/angular-pro

Store architecture - #2



Store architecture - #2 with a store



Angular State Management

- Simple applications - In the component
  - `counter : number = 0;`
  - `this.counter += 1;`
- Intermediate applications - In a service
  - `counter : number = 0;`
  - `this.counter = this.counterService.increment(1);`
  - Cache `counter` value in the service

- Larger applications - In a *data store* – all based on *observables*

```
counter$: Observable<number>;

constructor(private store: Store<AppState>){
        this.counter$ = store.select('counter');
}

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

Store Terminology and concepts

Important Store terminology / concepts

Store

"The store can be seen as your client side database. But more importantly, it reflects the state of your application. You can see it as the single source of truth."

"The store holds all the data. You modify it by dispatching ***actions*** *to it."*

Reducer

"Reducers are functions that know what to do with a given action and the previous state of your app.

Reducers will take the previous state from your store and apply a pure function to it. From the result of that pure function, you will have a new state. The new state is put in the store."

Actions

*"Actions are the payload that contains needed information to alter your store. Basically, an action has a **type** and a **payload** that your reducer function will take to alter the state."*

Dispatcher

"Dispatchers are simply an entry point for you to dispatch your action. In Ngrx, there is a dispatch method directly on the store. I.e., you call `this.store.dispatch({…})`*"*

Source: https://www.toptal.com/angular-js/ngrx-angular-reaction-application

Reducers, Store and Components - The complete picture

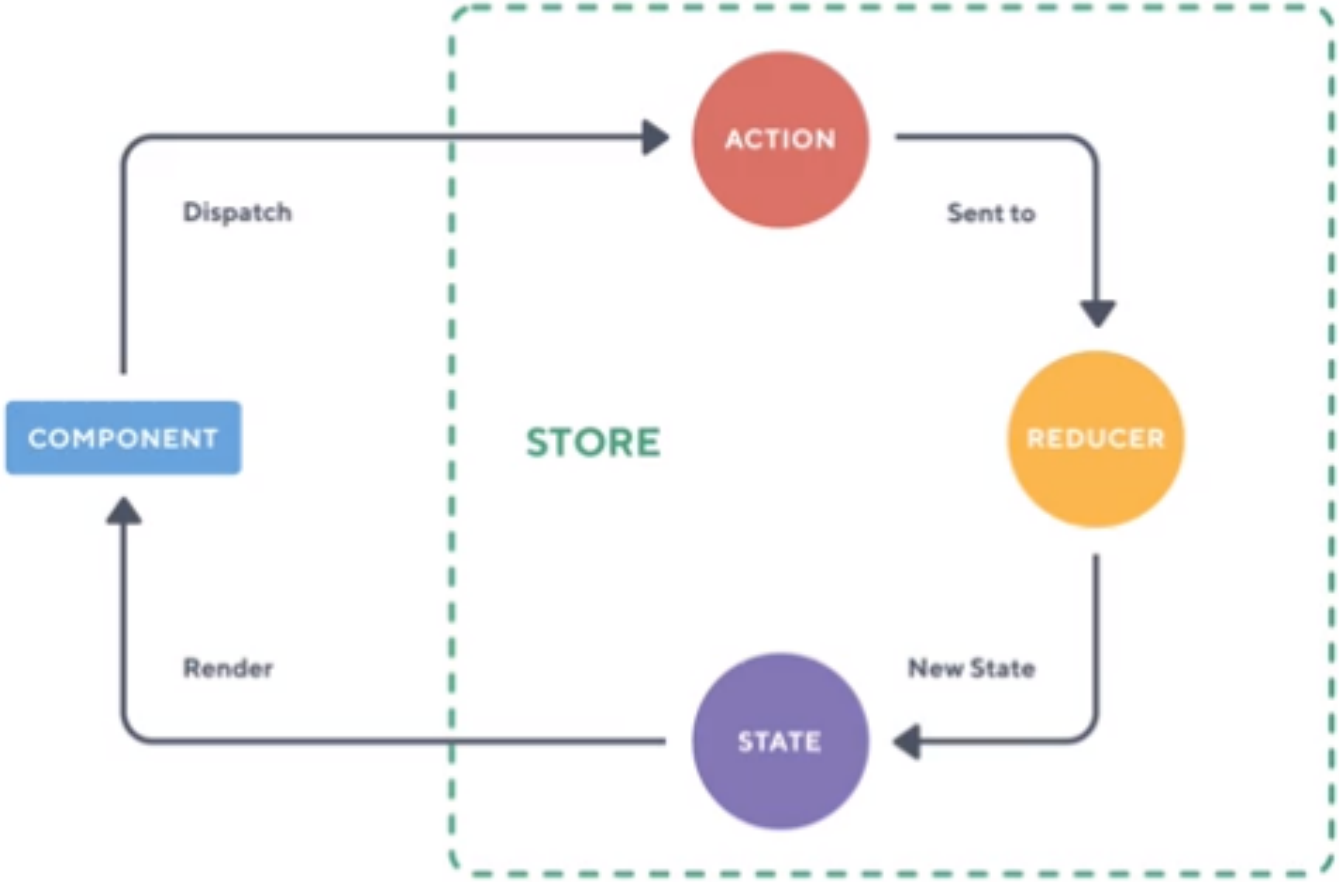
The **Component** first dispatches an Action. When the **Reducer** gets the Action, it will update the state(s) in the **Store**.

The Store has been injected to the Component, so the View will update based on the store state change (it is `subscribed`).

https://hongbo-miao.gitbooks.io/angular2-server/content/part4/add-ngrx-store.html

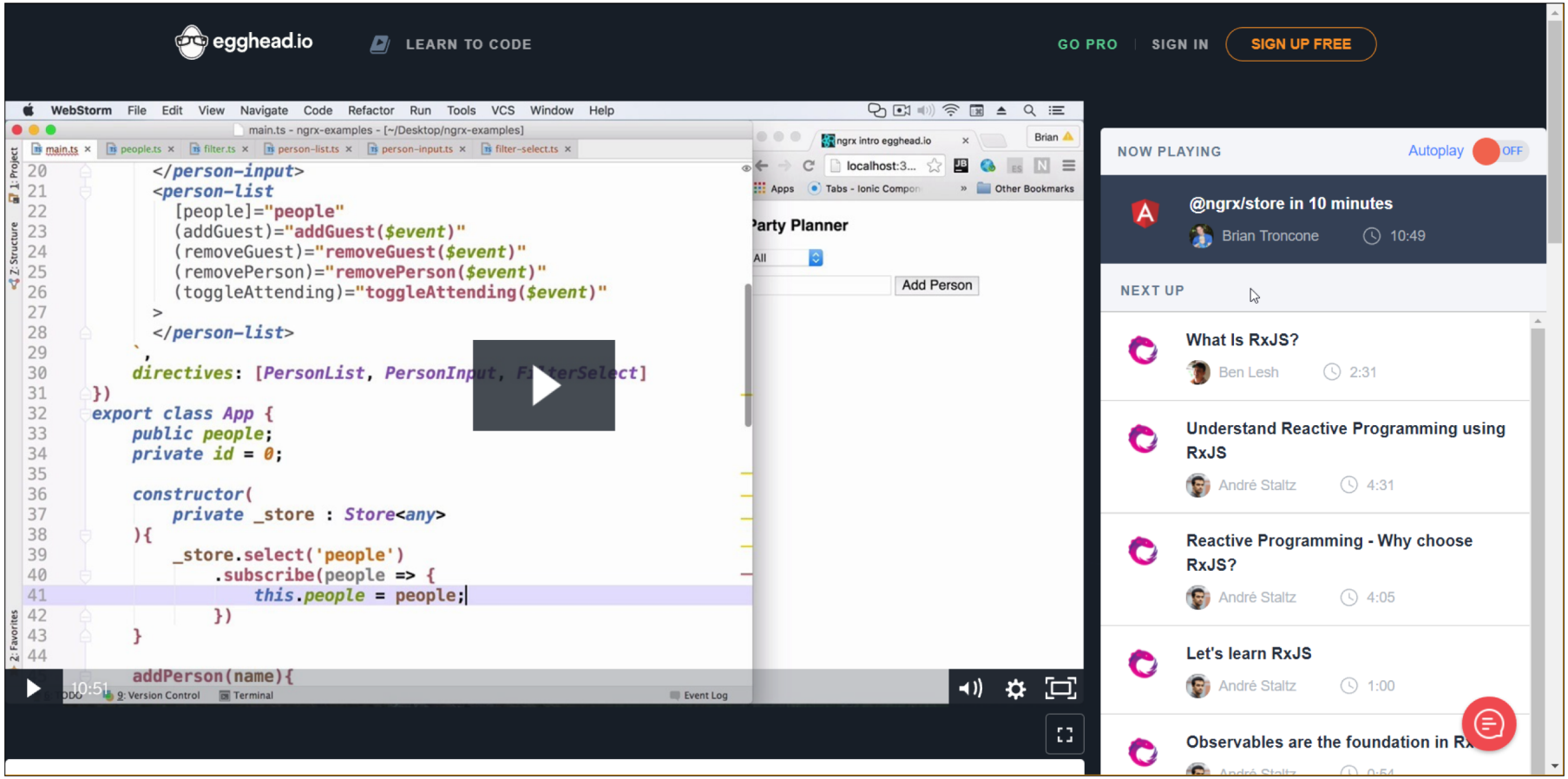
REDUX ARCHITECTURE

One-way dataflow



https://platform.ultimateangular.com/courses/ngrx-store-effects/lectures/3788532

Store concepts in a video



https://egghead.io/lessons/angular-2-ngrx-store-in-10-minutes

Setting up @ngrx/store

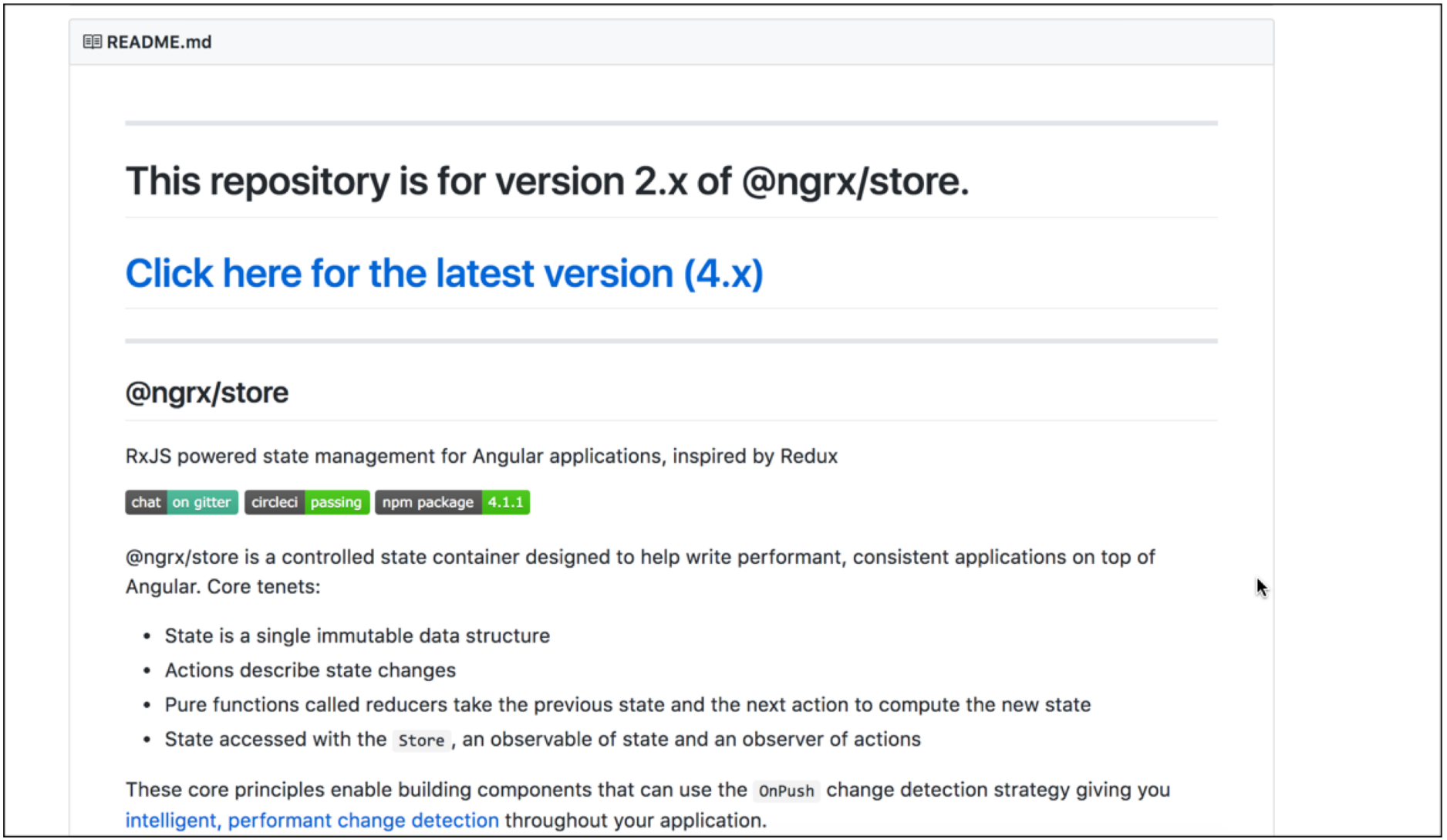
- Install core files & store files
- Create new project or add to existing project
- v2.x:

```
npm install @ngrx/core @ngrx/store --save
```

- V4.x +:

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

Different versions, different docs...


README.md

This repository is for version 2.x of @ngrx/store.

Click here for the latest version (4.x)

@ngrx/store

RxJS powered state management for Angular applications, inspired by Redux

chat on gitter | circleci passing | npm package 4.1.1

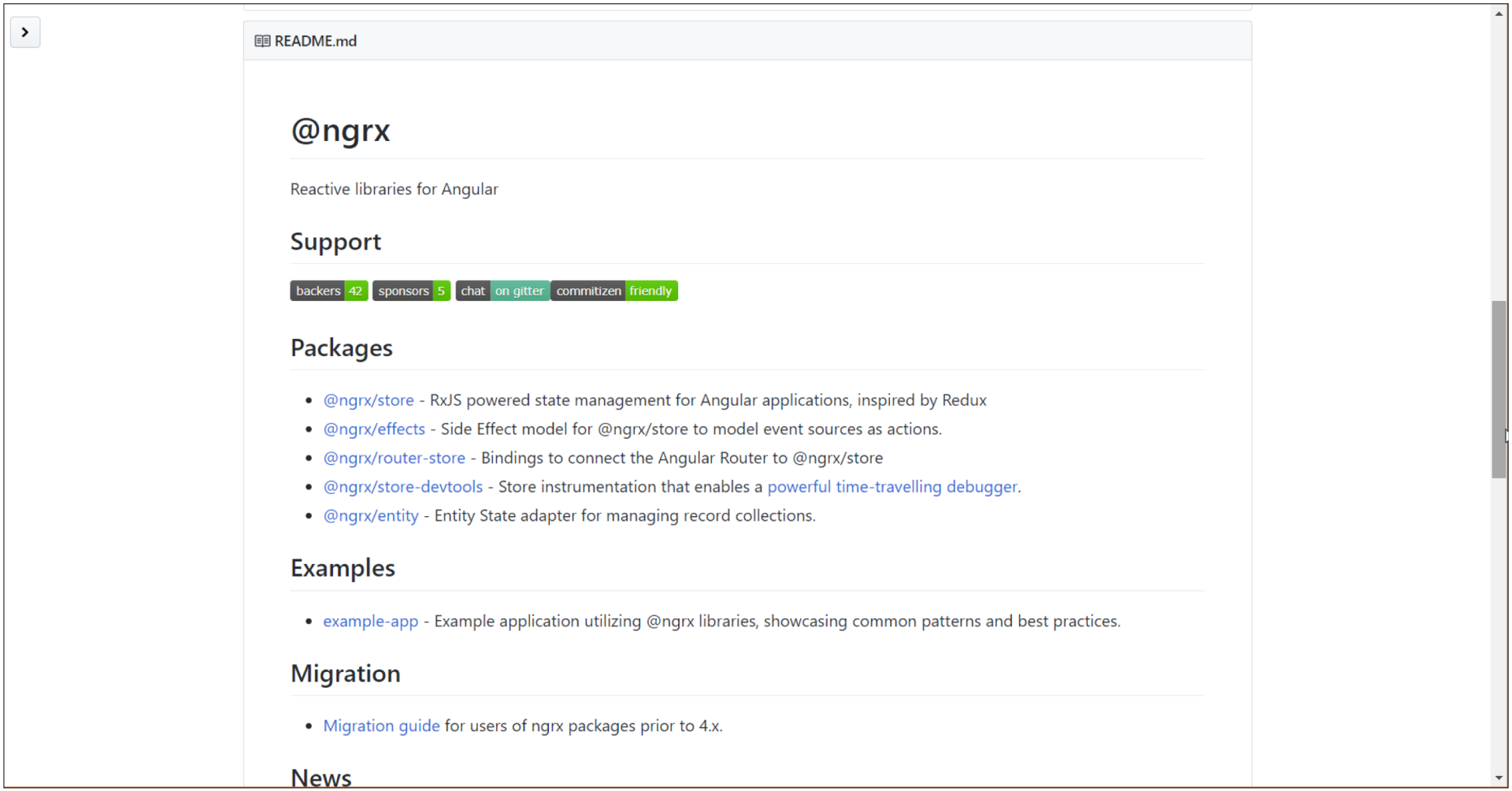
@ngrx/store is a controlled state container designed to help write performant, consistent applications on top of Angular. Core tenets:

- State is a single immutable data structure
- Actions describe state changes
- Pure functions called reducers take the previous state and the next action to compute the new state
- State accessed with the `Store`, an observable of state and an observer of actions

These core principles enable building components that can use the `OnPush` change detection strategy giving you intelligent, performant change detection throughout your application.

https://github.com/ngrx/store

Different versions, different docs...



README.md

@ngrx

Reactive libraries for Angular

Support

backers 42 sponsors 5 chat on gitter commitizen friendly

Packages

- @ngrx/store - RxJS powered state management for Angular applications, inspired by Redux
- @ngrx/effects - Side Effect model for @ngrx/store to model event sources as actions.
- @ngrx/router-store - Bindings to connect the Angular Router to @ngrx/store
- @ngrx/store-devtools - Store instrumentation that enables a powerful time-travelling debugger.
- @ngrx/entity - Entity State adapter for managing record collections.

Examples

- example-app - Example application utilizing @ngrx libraries, showcasing common patterns and best practices.

Migration

- Migration guide for users of ngrx packages prior to 4.x.

News

https://github.com/ngrx/platform

Migration guide (but Google is more useful):

Dependencies

You need to have the latest versions of TypeScript and RxJS to use ngrx V4 libraries.

TypeScript 2.4.x
RxJS 5.4.x

@ngrx/core

@ngrx/core is no longer needed, and can conflict with @ngrx/store. You should remove it from your project.

BEFORE:

```
import { compose } from '@ngrx/core/compose';
```

AFTER:

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

@ngrx/store

Action interface

The `payload` property has been removed from the `Action` interface. It was a source of type-safety issues, especially when used with `@ngrx/effects`. If your interface/class has a payload, you need to provide the type.

https://github.com/ngrx/platform/blob/master/MIGRATION.md

1. Create your first reducer

- A reducer is simply an exported function with a name.
- It takes two parameters:
  - Current `state`, or otherwise empty object/initial state
  - `action`, of type `Action`
- `action` is of shape `{type : string , payload? : any}`.
- NO DEFAULT `payload` in `@ngrx/store >4.0.0`
- Switch on `action.type`
  - Later: use abstraction. For now – a simple string
  - Always provide a `default` action which returns the unaltered `state`.

New file – counter.ts

```
// counter.ts - a simple reducer
import {Action} from '@ngrx/store';

export function counterReducer(state: number = 0, action: Action) {
   switch (action.type) {
      case 'INCREMENT':
         return state + 1;

      case 'DECREMENT':
         return state - 1;

      case 'RESET':
         return 0;

      default:
         return state;
   }

}
```

2. Adding store and reducer

- Register the state container with your application.
- Import reducers
- V.4.0.0: Use `StoreModule.forRoot()` to add it to the module

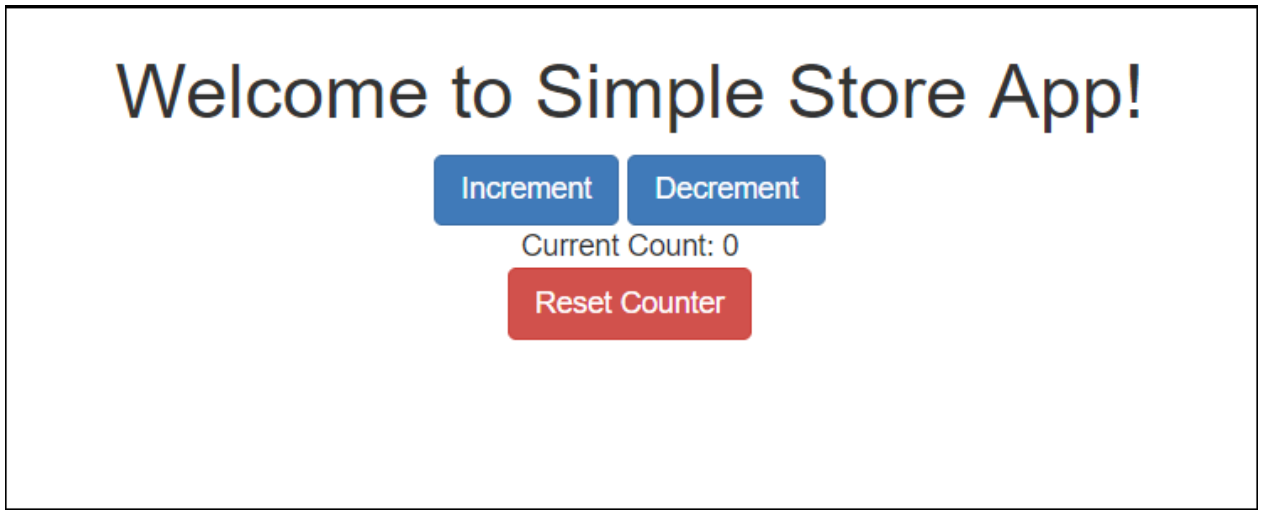
```
…
// Store stuff
import {StoreModule} from '@ngrx/store';
import {counterReducer} from './reducers/counter';

@NgModule({
   …,
   imports      : [
      BrowserModule,
      StoreModule.forRoot({counter: counterReducer})
   ],
   …
})
export class AppModule {}
```

3. Using the Store Service

- Import and inject the `Store` service to components
- Create an interface (or class) for your `AppState`
- Initialize the store with `AppState` Type
- Use `store.select()` to select slice(s) of the state
- Add methods to call reducer functions
  - `increment()`
  - `decrement()`
  - `etc..`

```
import {Component} from '@angular/core';
import {Store} from '@ngrx/store';
import {Observable} from 'rxjs/Observable';

interface AppState {
   counter: number;
}

@Component({
   selector   : 'app-root',
   templateUrl: './app.component.html'
})
export class AppComponent {
   counter$: Observable<number>;

   constructor(private  store: Store<AppState>) {}

   ngOnInit() {
      this.counter$ = this.store.select('counter')
   }

   increment() {
      this.store.dispatch({type: 'INCREMENT'})
   }
   …
}
```

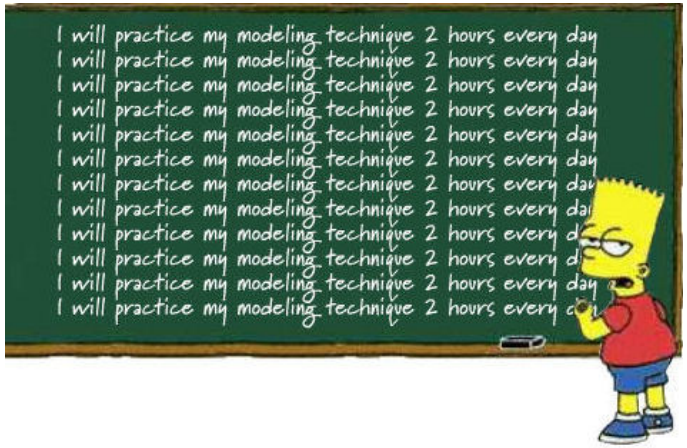
Next steps



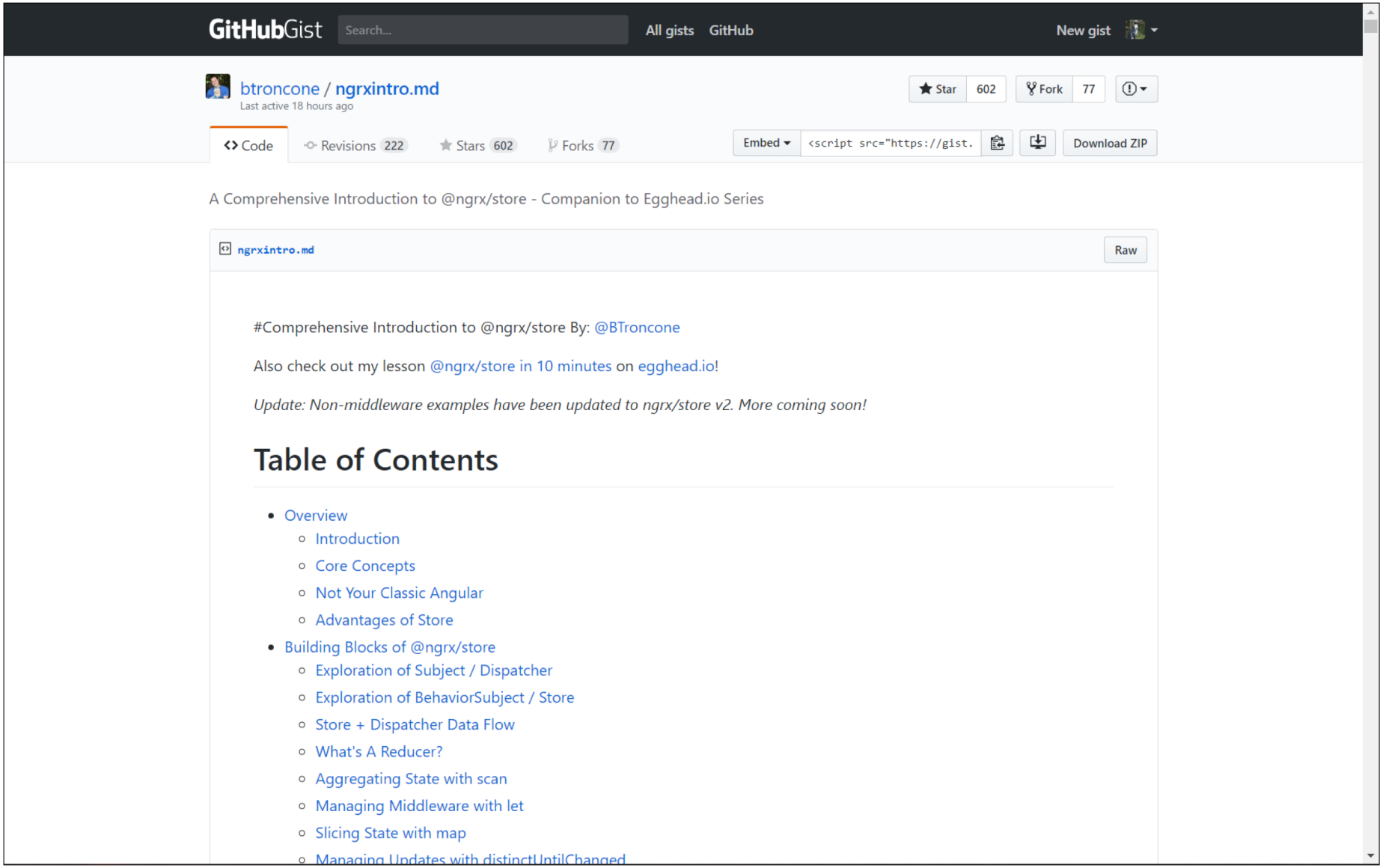
Add new components, subscribe to store, enhance store, etc.

Workshop

- Create a new app, follow the previous steps to add a Store
- OR: Start from `../200-ngrx-simple-store`
- Make yourself familiar with the store concepts and data flow. Study the example code.
- Create some extra actions on the reducer. For example:
  - Add `+5` with one click
  - Subtract `-5` with one click
  - Reset counter to 0 if `counter >= 10`;

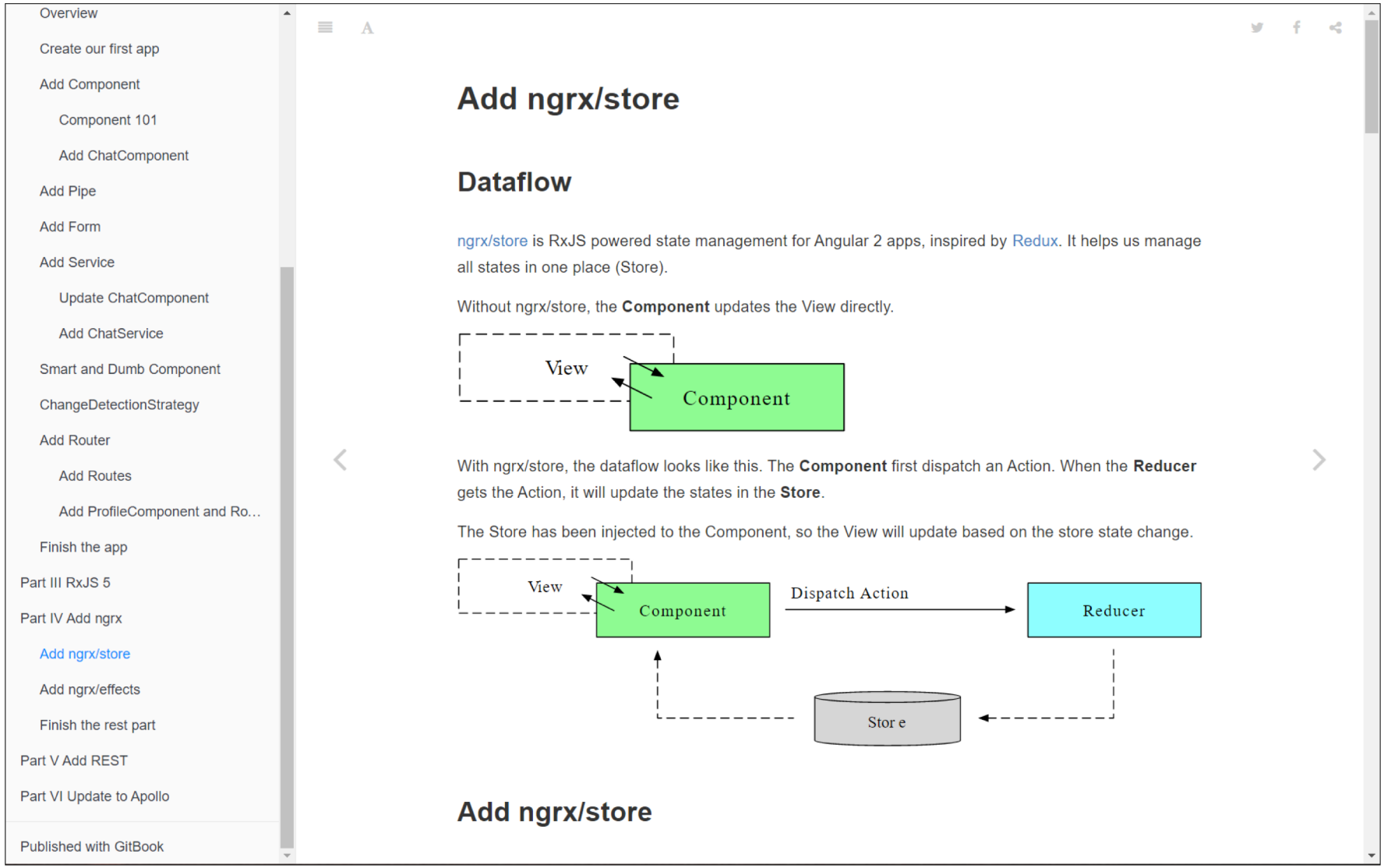


More info on @ngrx/store



https://gist.github.com/btroncone/a6e4347326749f938510

Gitbooks



https://hongbo-miao.gitbooks.io/angular2-server/content/part4/add-ngrx-store.html

OneHungryMind – Lukas Ruebbelke

ONEHUNGRYMIND ABOUT

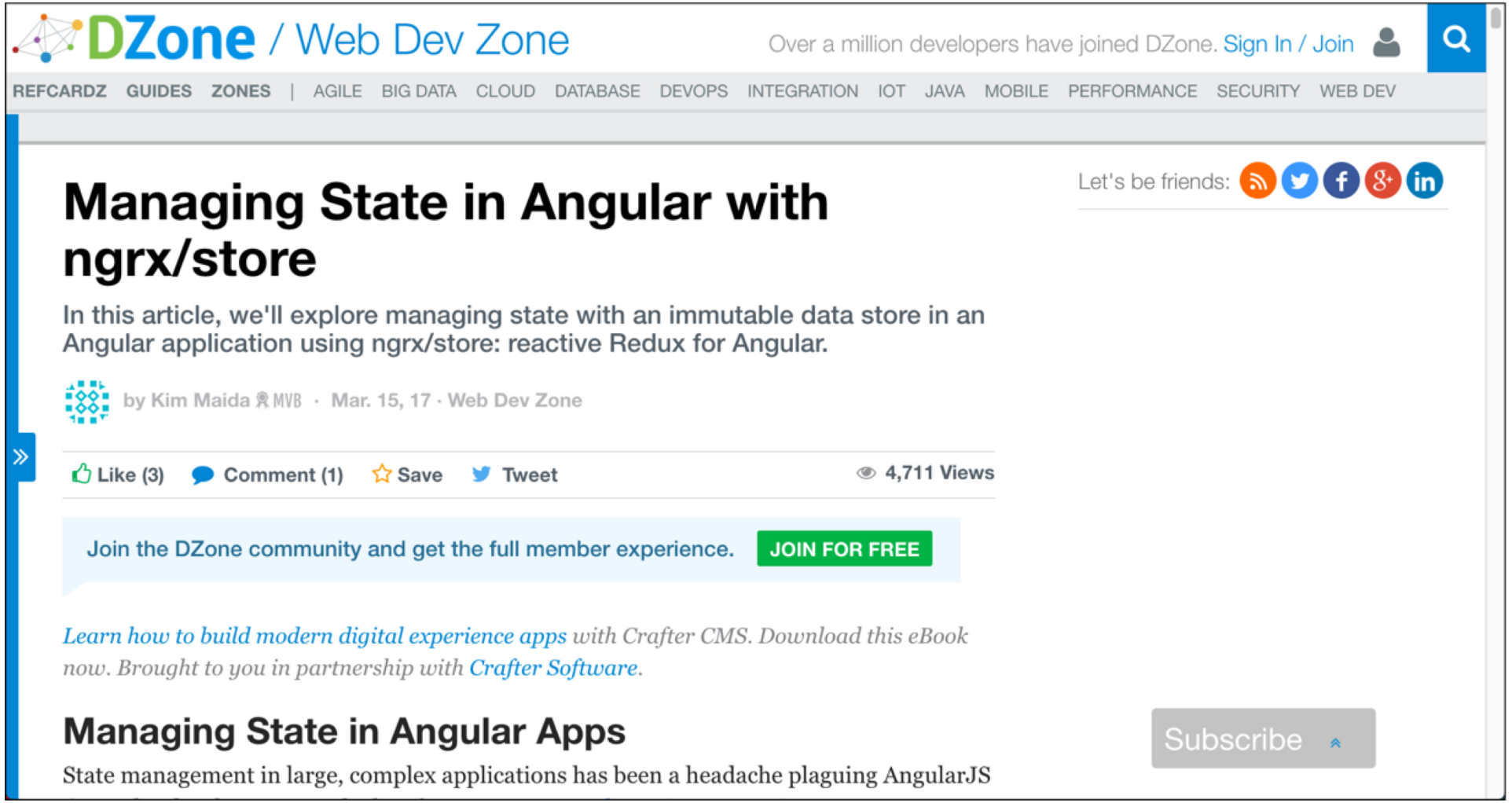
Build a Better Angular 2 Application with Redux and ngrx

Lukas Ruebbelke

The Evolution of Angular State Management

http://onehungrymind.com/build-better-angular-2-application-redux-ngrx/

Dzone article

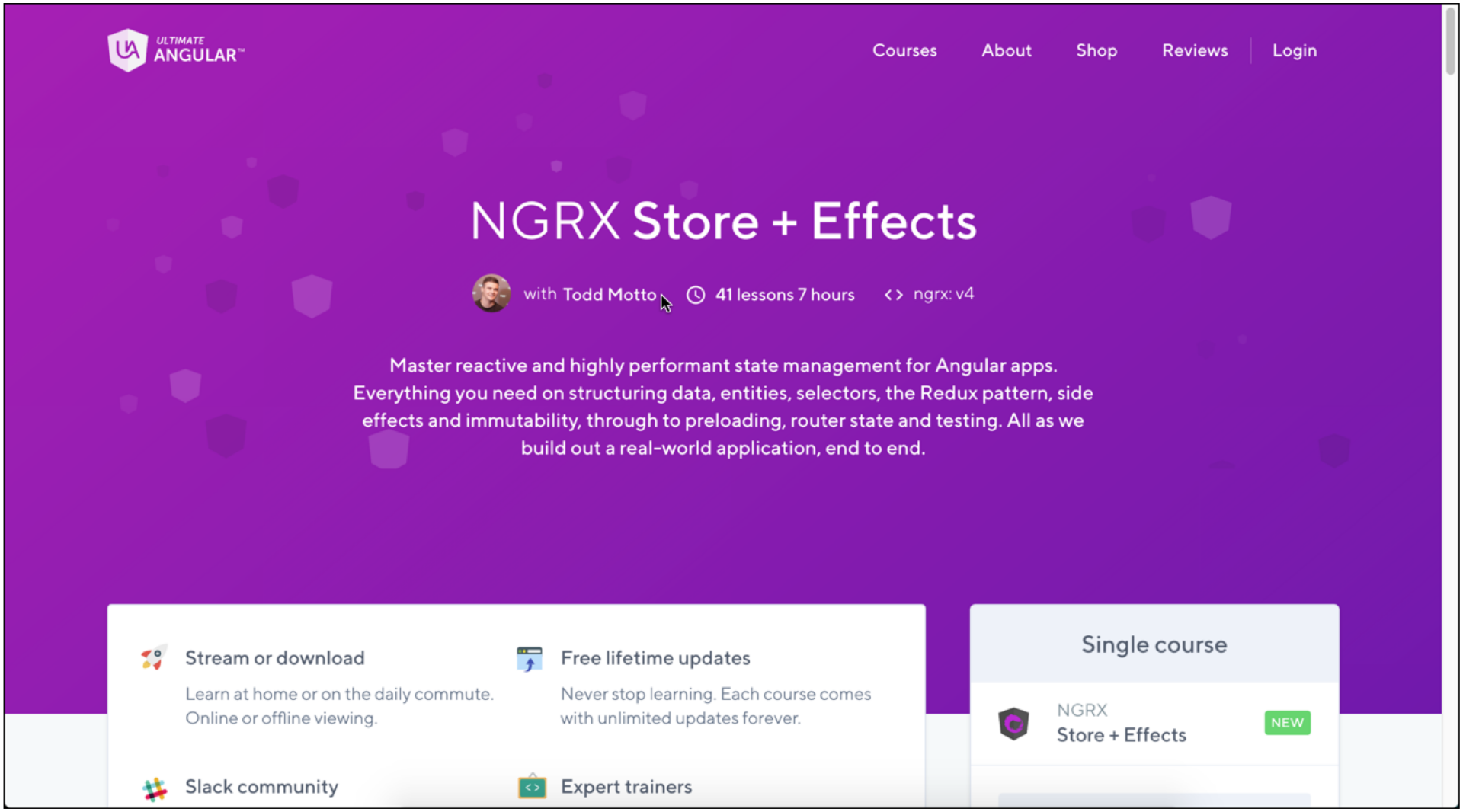


https://dzone.com/articles/managing-state-in-angular-with-ngrxstore

Rob Wormald - co-created @ngrx/store

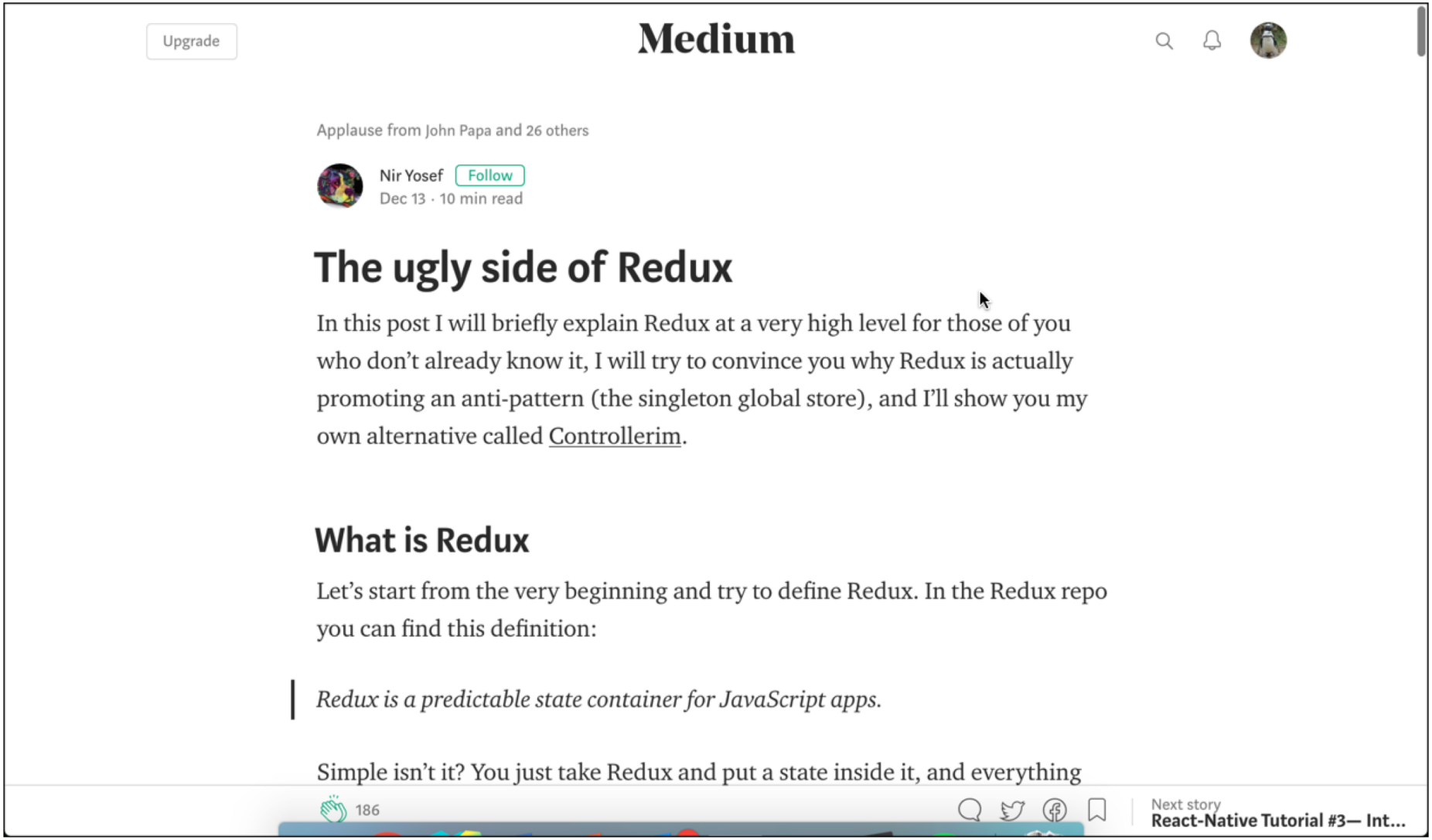
https://www.youtube.com/watch?v=mhA7zZ23Odw - and more

Online Training by Todd Motto



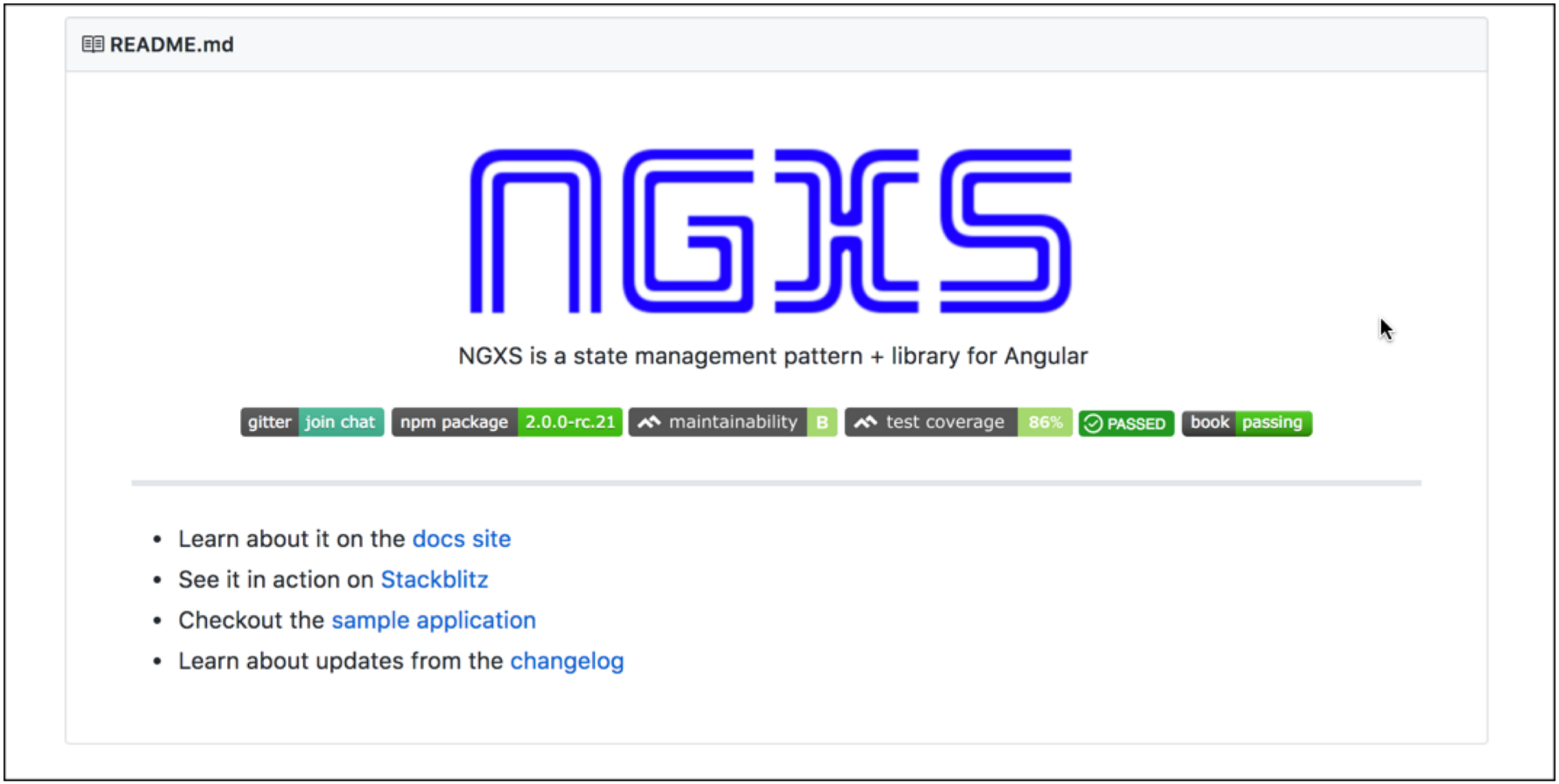
https://ultimateangular.com/ngrx-store-effects

Think about this – “The Ugly side of Redux”



https://medium.com/@niryo/the-ugly-side-of-redux-6591fde68200

Alternative State Management solution



https://github.com/amcdnl/ngxs