

illionx



Global Knowledge.®

Angular Advanced State Management with @ngrx/store

Peter Kassenaar –
info@kassenaar.com

WORLDWIDE LOCATIONS

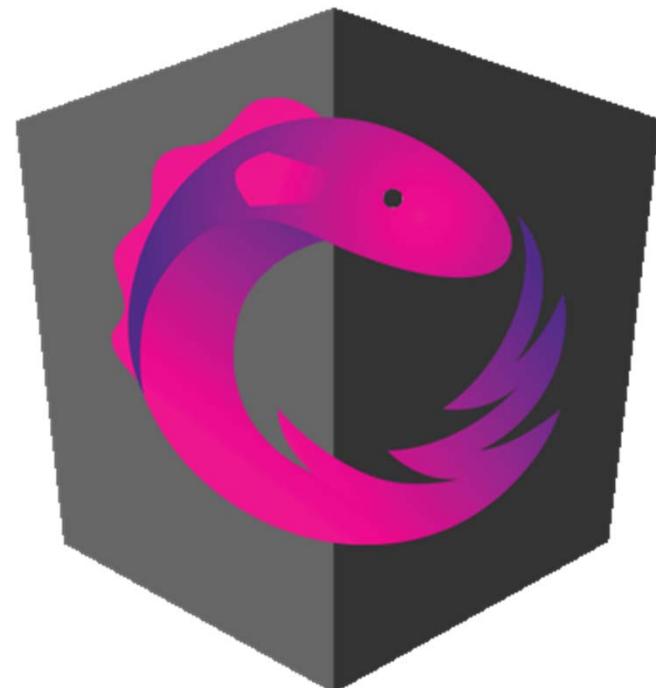
BELGIUM CANADA COLOMBIA DENMARK EGYPT FRANCE IRELAND JAPAN KOREA MALAYSIA MEXICO NETHERLANDS NORWAY QATAR
SAUDI ARABIA SINGAPORE SPAIN SWEDEN UNITED ARAB EMIRATES UNITED KINGDOM UNITED STATES OF AMERICA

What is State Management?

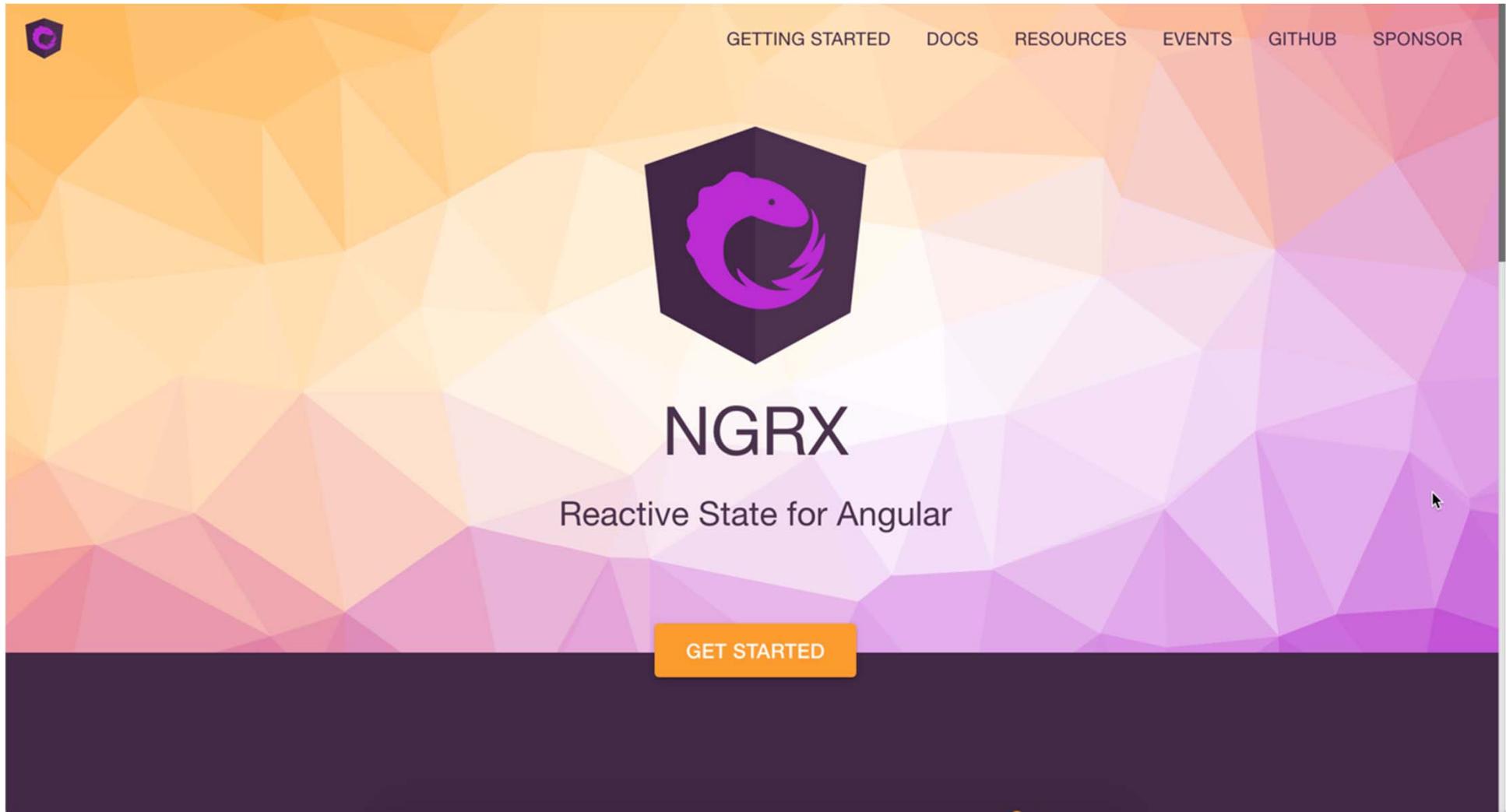
- Various design patterns, used for managing *state* (data in its broadest sense!) in your application.
- Multiple solutions possible – depends on application & framework



Redux

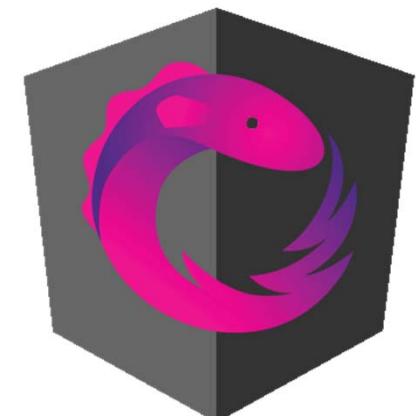


<https://ngrx.io/>



@ngrx/store – 3 generations

- Generation #1 – Angular 2
 - Creator: Rob Wormald
 - Simple implementation, (almost) all hand coded
- Generation #2 – Angular 4-7
 - Action Creators, custom payload
 - @Effects
- Generation #3 – Angular 8+
 - createAction(), createReducer() and more
 - (even) more complex...



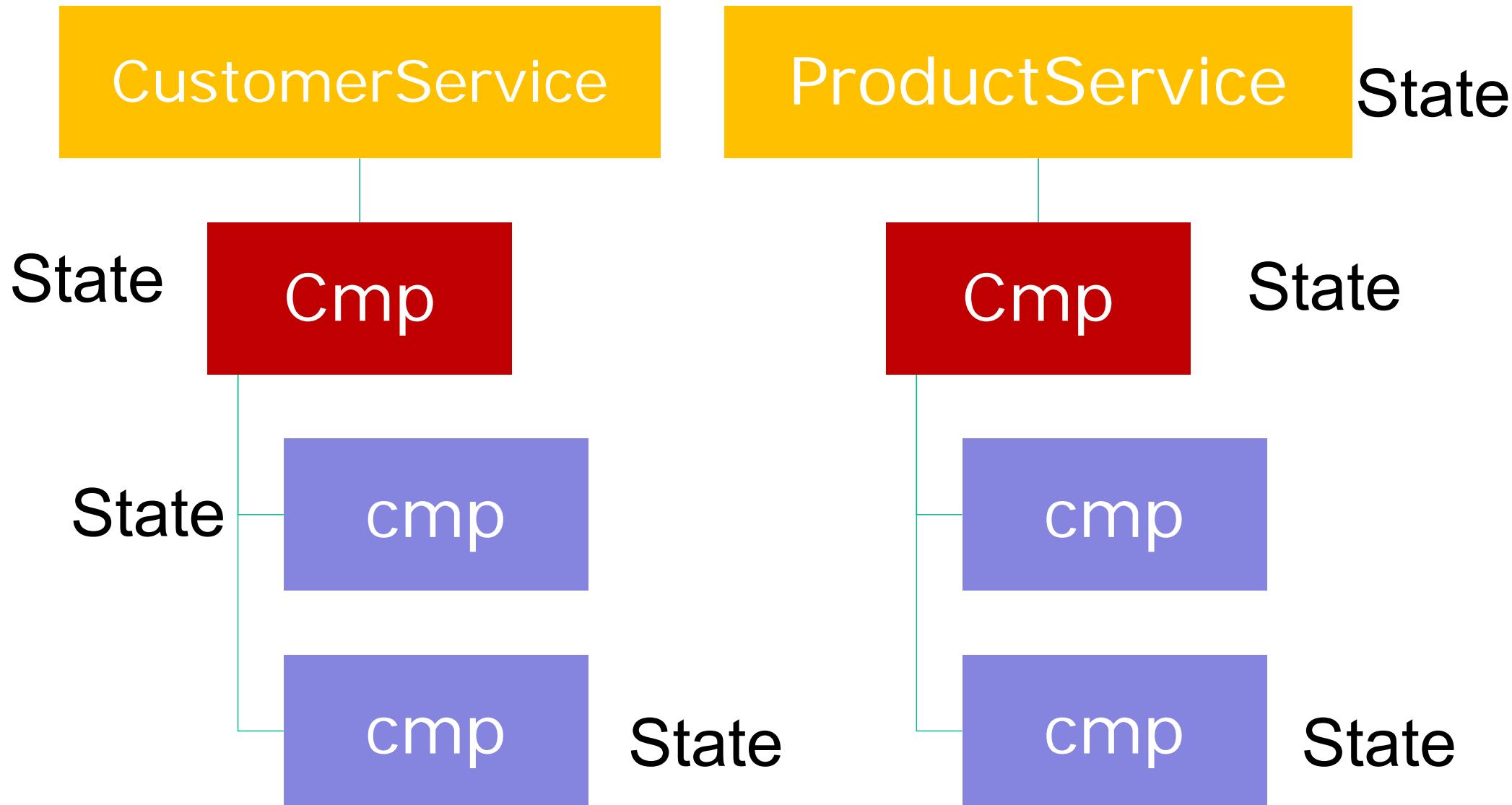
Maybe you don't need a store...

- <https://medium.com/@rmcavin/my-favorite-state-management-technique-in-angular-rxjs-behavior-subjects-49f18daa31a7>

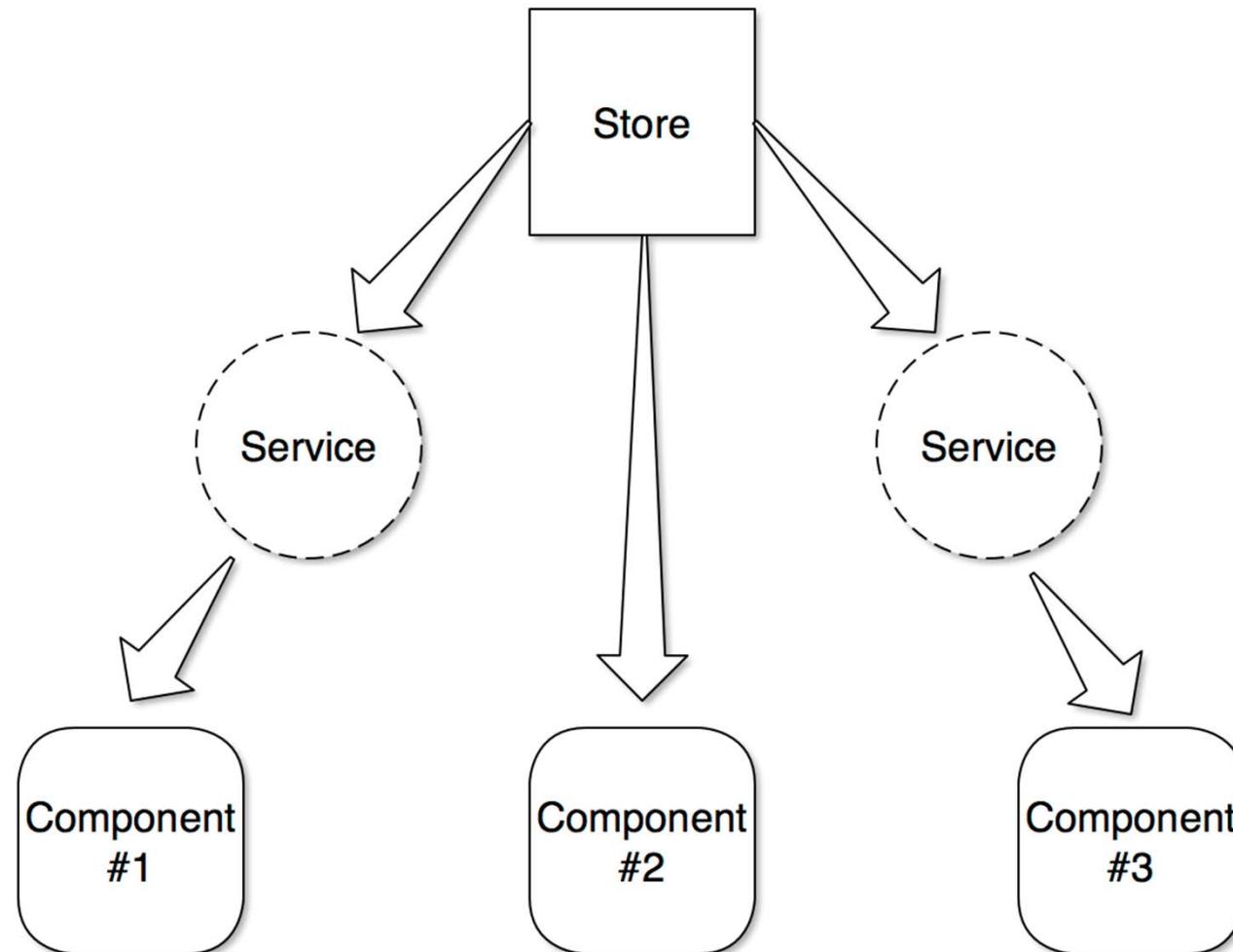
The screenshot shows a Medium article page. At the top left is the Medium logo and a sidebar with the word "Javascript". On the right are a search icon, a green circle with the number "3", an "Upgrade" button, and a user profile picture. The main title of the article is "My favorite state management technique in Angular — RxJS Behavior Subjects". Below the title is a photo of the author, Rachel Cavin, followed by her name and a "Follow" button, and the date "Dec 5, 2018 · 4 min read". The article content starts with: "Most of the apps I build in Angular are fairly small, we build many small front end apps instead of a few larger ones. Historically, my team and I had always just relied on the standard input/emitter Angular way of [component interaction](#), which worked well most of the time but could lead to the occasional excessive passing between sibling components. We had looked into NgRx and other flux implementations but they felt a bit overkill for the size of our applications. Recently, we discovered the solution to our state management needs—the [RxJS Behavior Subject!](#)".

Behavior subjects are similar to regular subjects in RxJS, except

State management without a store



Store architecture - #1

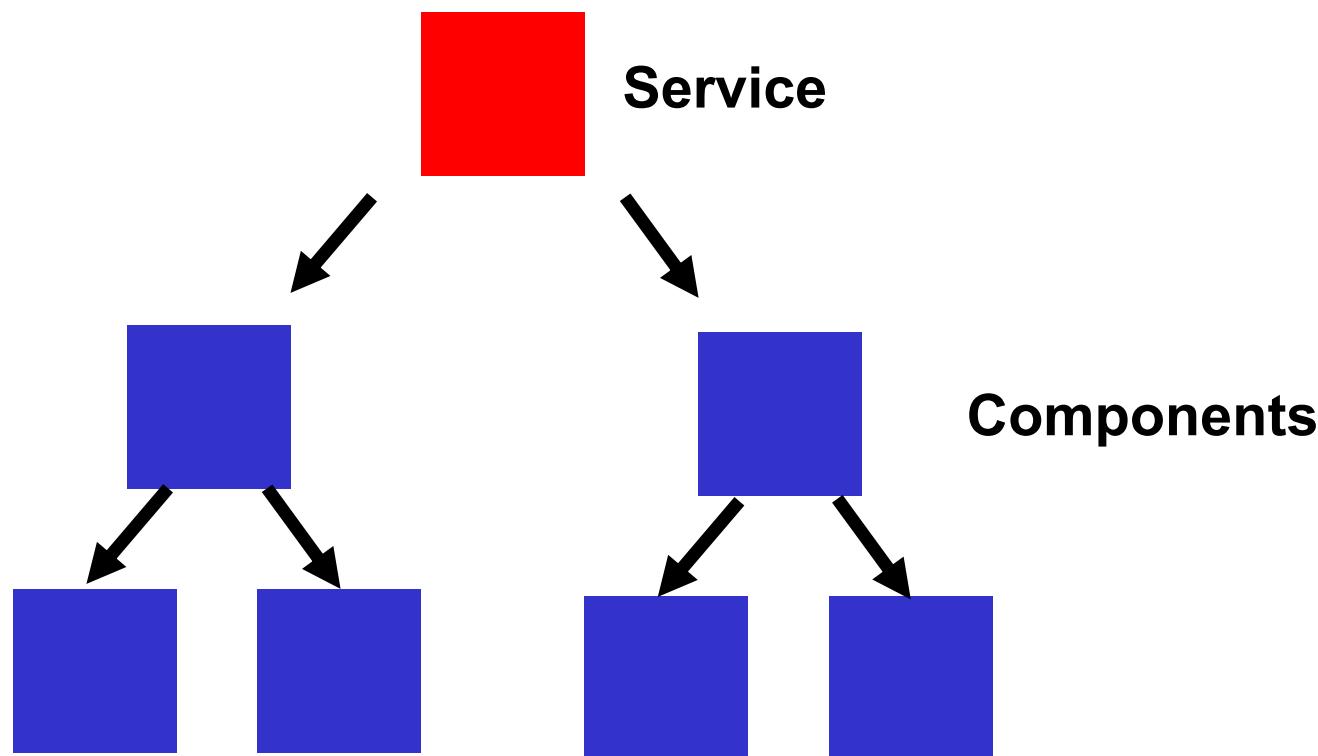


<https://gist.github.com/btroncone/a6e4347326749f938510>

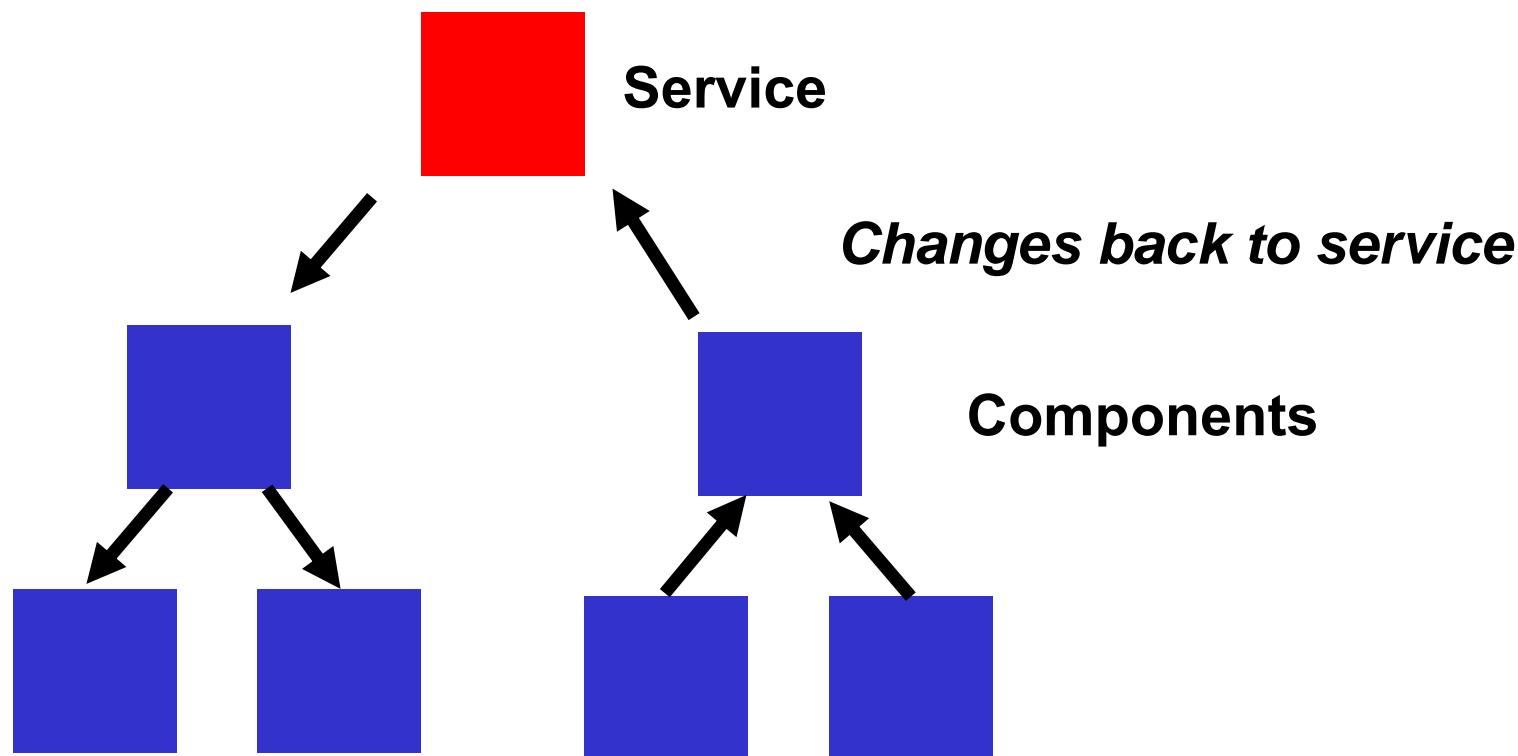
Benefits of using a store

- 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”

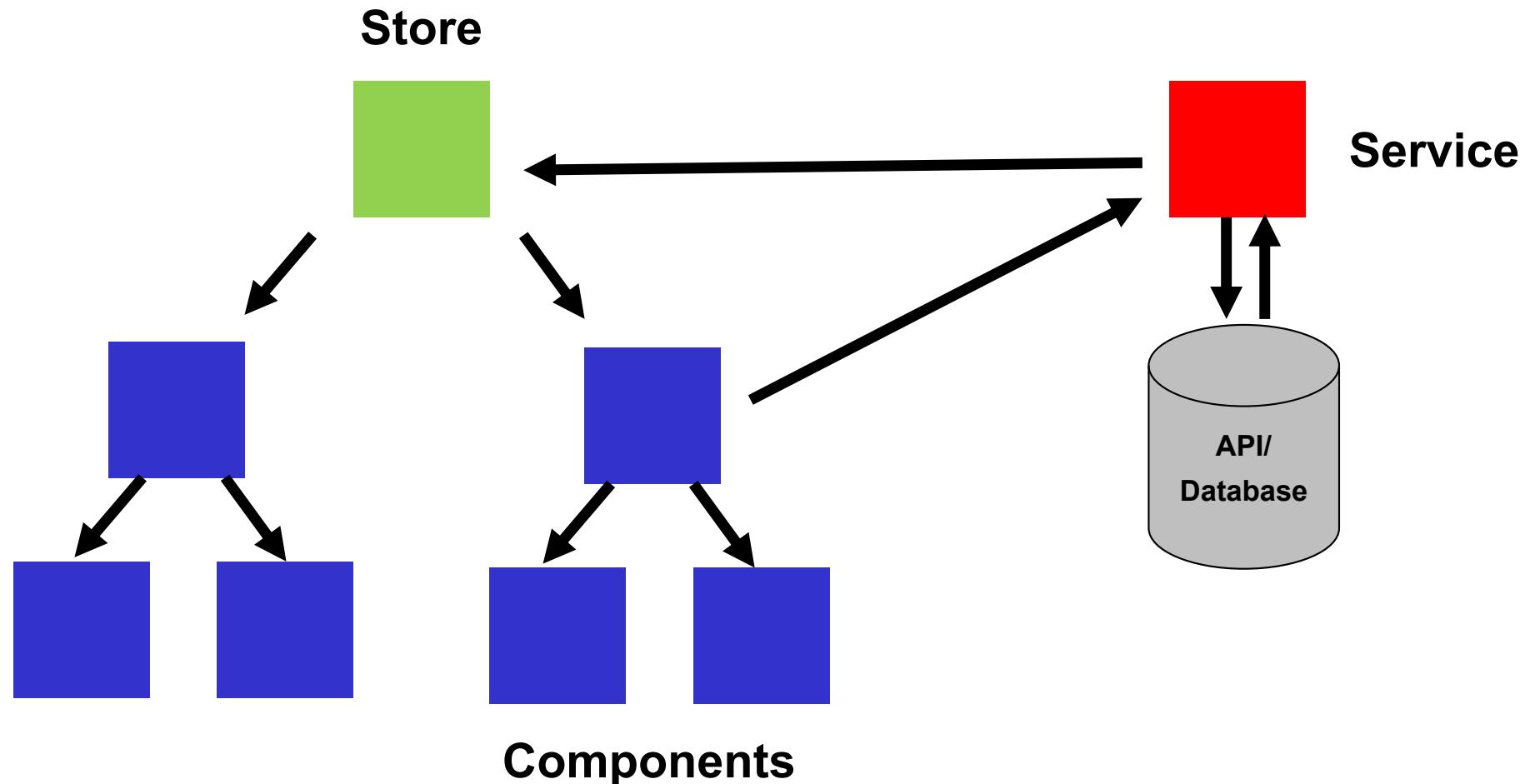
Store architecture - #2 - traditional



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<State>){  
    this.counter$ = store.pipe(select('counter'));  
}  
  
increment() {  
    this.store.dispatch(counterIncrement());  
}
```



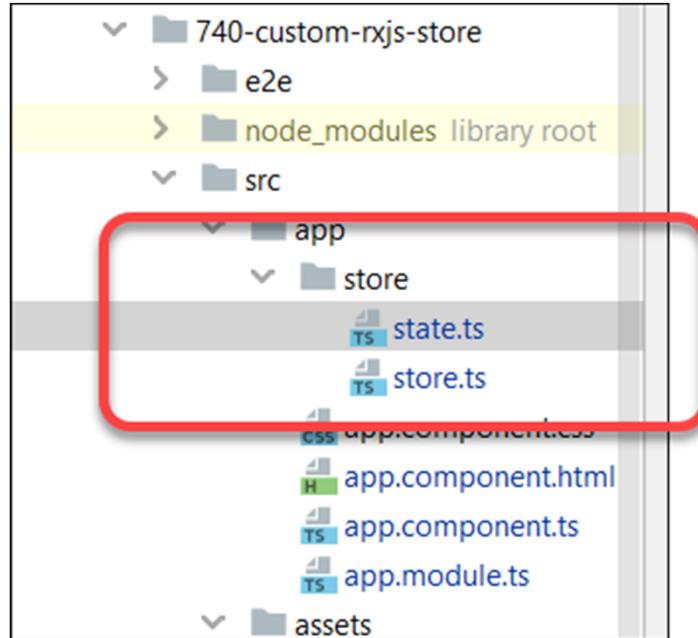
Building a store from scratch

Using observables, standard RxJS techniques and custom code, without a library

Steps in creating a simple store

1. Create a `../store` folder and `store.ts` file, add it to the module
2. Create interface `State` for the data you want to 'store' (duh...)
3. Create a constant `state` of type `State`
4. Create a subject of type `BehaviorSubject` with type `State`, initialize it with initial state.
5. Expose the subject as an observable
6. Create `.set()` method and `.select()` methods

Step 1 – create store, and Step 2) state



We are now creating a *simple store*, for one application, with one (1) module.

With @ngrx/store things can get – way – more complex. This example is to demonstrate how store concept works.

```
// state.ts
export interface Todo {
  id: number;
  name: string;
  done: boolean;
}
export interface State {
  todos: Todo[];
  // other slices of the store
}
```

Step 3 – create state, and Step 4) BehaviorSubject

```
// store.ts
import {State} from './state';

const state: State = {
  todos: undefined
};

export class Store {
  // use behaviorsubject to create a subject with initial state
  // the last value is also passed to new subscribers.
  // The behaviorsubject holds the data (i.e. state)
  private subject = new BehaviorSubject<State>(state);
}
```

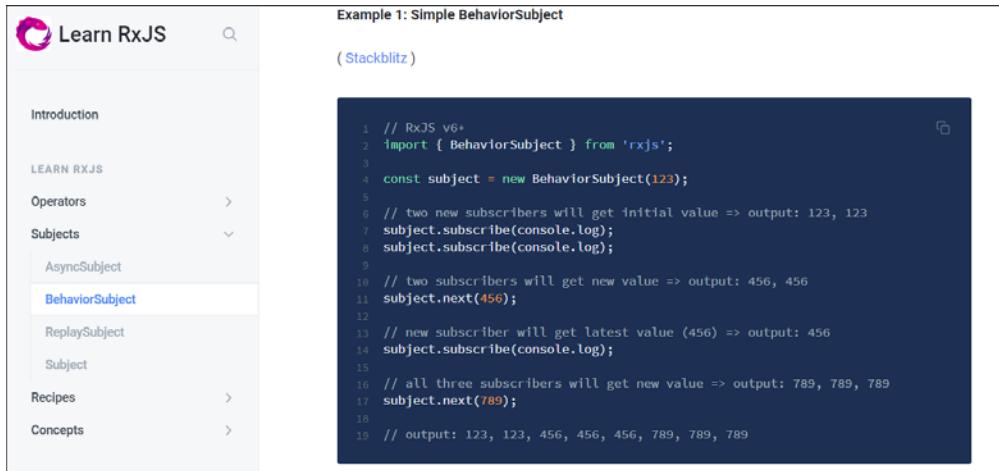
So the state is initially a list of undefined Todo's.

We're going to set them later from the code. Of course you can fetch them from a backend and so on.

We use BehaviorSubject to create **initial state**. A Subject cannot do that.

On BehaviorSubject<Type>()

- BehaviorSubject can hold a variable (i.e. state), where a Subject can not.
- New subscribers get a copy of that data, i.e. the last emitted state, which of course is very useful in a store scenario.
- You pass a new piece of data to the BehaviorSubject with the `.next()` method.



The screenshot shows the Learn RxJS website with the URL <https://www.learnrxjs.io/learn-rxjs/subjects/behaviorsubject>. The page title is "Example 1: Simple BehaviorSubject". It includes a Stackblitz link and a code editor window containing the following TypeScript code:

```
// RxJS v6+
import { BehaviorSubject } from 'rxjs';

const subject = new BehaviorSubject(123);

// two new subscribers will get initial value => output: 123, 123
subject.subscribe(console.log);
subject.subscribe(console.log);

// two subscribers will get new value => output: 456, 456
subject.next(456);

// new subscriber will get latest value (456) => output: 456
subject.subscribe(console.log);

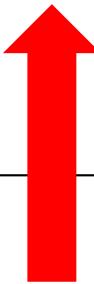
// all three subscribers will get new value => output: 789, 789, 789
subject.next(789);

// output: 123, 123, 456, 456, 456, 789, 789, 789
```

<https://www.learnrxjs.io/learn-rxjs/subjects/behaviorsubject>

Step 5 - Expose the subject as an observable

```
// store.ts
export class Store {
  // use behaviorsubject to create a subject with initial state
  // the last value is also passed to new subscribers.
  // The behaviorsubject holds the data (i.e. state)
  private subject = new BehaviorSubject<State>(state);
  private store = this.subject.asObservable()
    .pipe(
      distinctUntilChanged() // make it a little bit smoother, don't overnotify the subscribers
    );
}
```



The `store` is the variable we expose to the outer world later on, so components and services deal with an observable instead of a subject directly

Step 6 – create .set() and .select() methods

- Also create a helper get property that returns the current value of the state

```
// store.ts
export class Store {
    ...
    // internal helper function, return the current
    // value of the subject.
    get value(): any {
        return this.subject.value;
    }

    // set a new piece in the store. Update the
    // current store, using the spread operator (favor immutability)
    set(name: string, payload: any): void {
        this.subject.next({
            ...this.value, [name]: payload
        });
    }

    // select a slice from the store, use pluck to only fetch the
    // requested branch of the json-tree from the store
    select<T>(name: string): Observable<T> {
        return this.store.pipe(
            pluck(name),
        );
    }
}
```

Getter (internal)

Setter, using .next()

Selector, using .pluck()

Dynamically set the name of the property in the store. If it doesn't exist, it creates it

Access our store, only return the selected slice

This is all we need to do to create a reactive store!

7 – Done. Use the store in the component

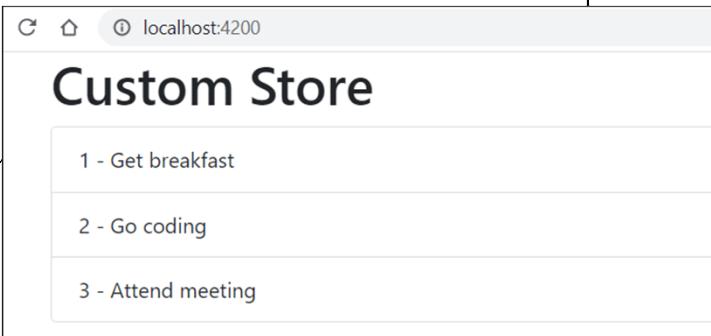
- Import the store in the component
 - set data, retrieve that data and bind it to the UI

```
// app.component.ts
export class AppComponent implements OnInit {
  todo$: Observable<any>;
  constructor(private store: Store) { ← Import store
  }
  ngOnInit(): void {
    // just some dummy data in the store, you can fetch this from a backend of course
    const someTodos: Todo[] = [
      {id: 1, name: 'Get breakfast', done: false},
      {id: 2, name: 'Go coding', done: false},
      {id: 3, name: 'Attend meeting', done: false},
    ];
    this.store.set('todos', someTodos); // 1. don't set a component property directly! Instead, set data in the store
    this.todo$ = this.store.select<Todo[]>('todos'); // 2. Fetch data from the store, assign it to local property
    console.log(this.store); // 3. Just some logging, to see if the store works
  }
}
```

Note: no callbacks here. Everything is reactive (as one would expect from a reactive store)

Result

```
<div class="container">  
  <h1>Custom Store</h1>  
  <ul class="list-group">  
    <li class="list-group-item"  
      *ngFor="let todo of todos | async">  
        {{ todo.id }} - {{ todo.name }}  
    </li>  
  </ul>  
  <hr>  
</div>
```



```
current store: ▶ {todos: Array(3)}  
store.ts:35  
app.component.ts:34  
  ▶ Store {subject: BehaviorSubject, store: Observable} ⓘ  
    ▶ value: (...)  
    ▶ subject: BehaviorSubject {_isScalar: false, observers: Array(1), closed: false, isStop...  
    ▶ store: Observable {_isScalar: false, source: Observable, operator: DistinctUntilChange...  
    ▶ _proto: ...  
    ▶ value: Object  
      ▶ todos: Array(3)  
        ▶ 0: {id: 1, name: "Get breakfast", done: false}  
        ▶ 1: {id: 2, name: "Go coding", done: false}  
        ▶ 2: {id: 3, name: "Attend meeting", done: false}  
        ▶ length: 3  
        ▶ __proto__: Array(0)  
      ▶ __proto__: Object  
    ▶ constructor: class Store  
    ▶ set: f set(name, payload)  
    ▶ select: f select(name)  
    ▶ get value: f value()  
    ▶ __proto__: Object  
Angular is running in the development mode. Call enableProdMode() to enable core.js:40480  
the production mode.  
[WDS] Live Reloading enabled.  
client:52
```



Updating the store

Writing new values in the store by writing a custom
.update() method

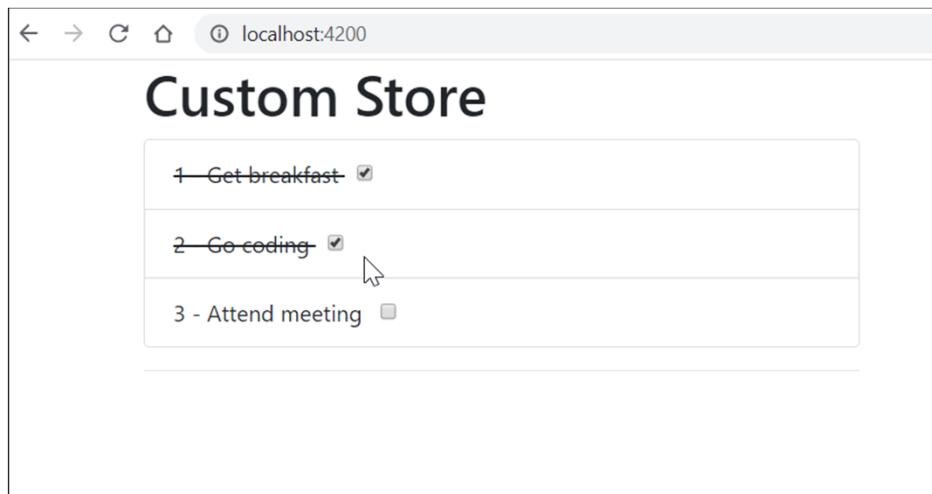
Update the list of Todo's in the store

```
// Update the store, in this case a list of todos
updateTodo(name: string, payload: Todo): void {
  // 1. fetch the correct slice from the store (even if we only have one)
  const value = this.value[name];    ← Get correct slice
  const newTodos: Todo[] = value.map((todo: Todo) => {
    // 2. Loop over our todos and update the given item
    if (payload.id === todo.id) {
      return {...todo, ...payload};   ← Return updated item...
    } else {
      return todo;                  ← Or simply return if not applicable
    }
  });
  // 3. Set the store with the new value of newTodos
  this.set(name, newTodos);         ← Write new array in the store
  // 4. Optional - write state to LocalStorage, save todos in backend, etc.
}
```

Update the UI and logic for component

```
<ul class="list-group">
  <li class="list-group-item"
    *ngFor="let todo of todos$ | async">
    <span [ngClass]="{'todo-done': todo.done}"> ← custom CSS class
      {{ todo.id }} - {{ todo.name}}
    </span>
    &nbsp;
    <input type="checkbox" [checked]="todo.done" (change)="updateTodo(todo)"> ← Add behavior
  </li>
</ul>
```

```
// update the state of a todo item
updateTodo(todo: Todo) {
  // toggle the state of item
  todo.done = !todo.done;
  this.store.updateTodo('todos', todo);
}
```



Result

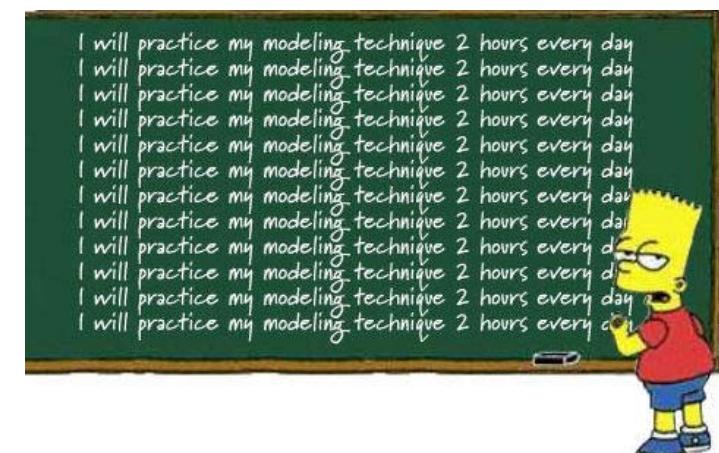
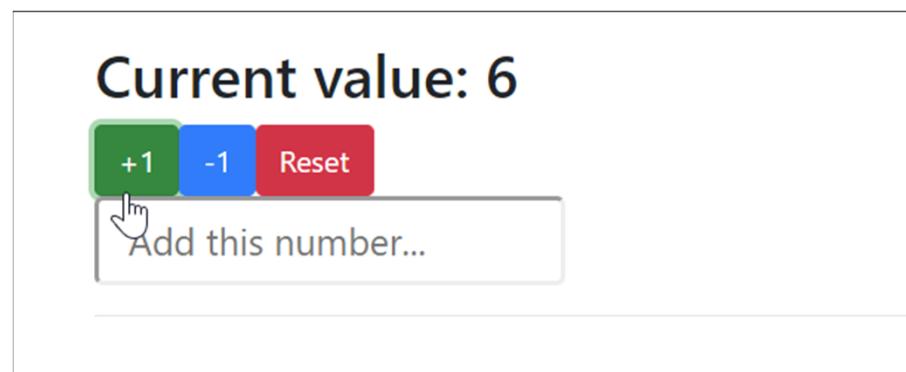
Workshop - 1

- Create a new app, Create a custom store, as described in the slides
- OR: Start from `./740-custom-rxjs-store`
- Create a `counter$` property and add it to the store.
 - In your component: show buttons to `increment()`, `decrement()` and `reset()` the counter in the store
 - Add it –for now – to the *same component, for simplicity*
- Some UI and logic is already available in the example, but first try it yourself!

Current value: 6

+1 -1 Reset

Add this number...



Workshop - 2

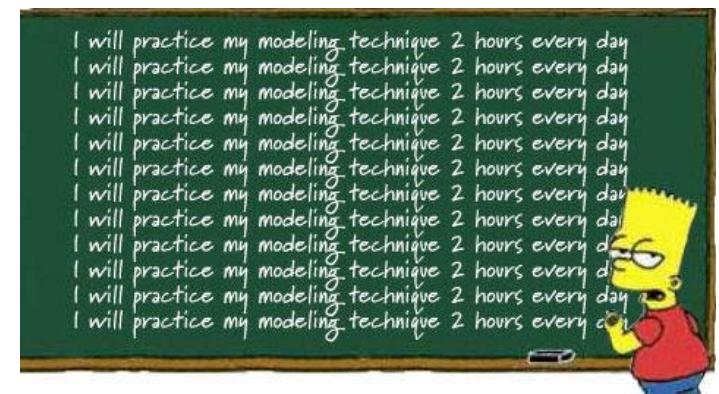
- Create a `movies$` property and add it to the store.
- Add a textbox to search for movies, put movies in the store.
- Search for movie details, based on the `imdbID` which is now available.
- Some UI and logic is already available in the example, but first try it yourself!

Search for movies

Avatar, imdbID: tt0499549
Actors: Sam Worthington, Zoe Saldana, Sigourney Weaver, Stephen Lang
A paraplegic Marine dispatched to the moon Pandora on a unique mission becomes torn between following his orders and protecting the world he feels is his home.

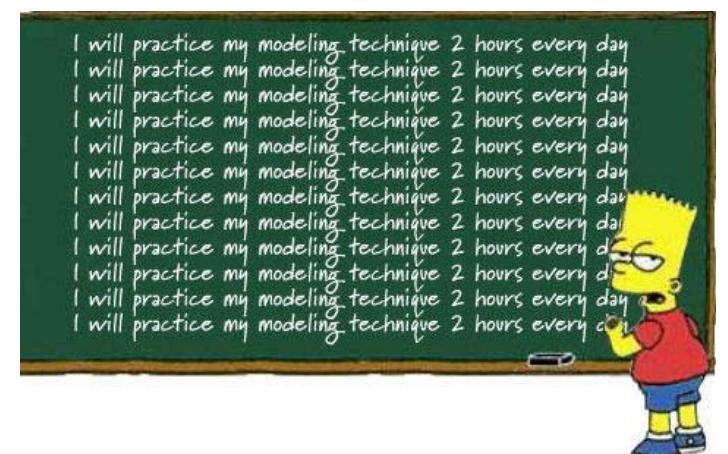
Avatar: The Last Airbender, imdbID: tt0417299
Actors: Dee Bradley Baker, Zach Tyler, Jack De Sena, Mae Whitman
In a war-torn world of elemental magic, a young boy reawakens to undertake a dangerous mystic quest to fulfill his destiny as the Avatar, and bring peace to the world.

Avatar: The Last Airbender, imdbID: tt0959552
Actors: Zach Tyler, Mae Whitman, Jack De Sena, Dante Basco



Optional workshop - 3

- Add the router, (like in `./740-custom-rxjs-store`)
- Make sure that the store contents survive a switch in components.
 - E.g. Movies retain in the store, the counter value is preserved, and so on.
- Tip: *don't reinitialize* the store in the `ngOnInit()` of every component, instead, do it once in `app.component.ts` and work from there.





@ngrx/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."

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."*

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."

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({...})"

Reducers, Store and Components -

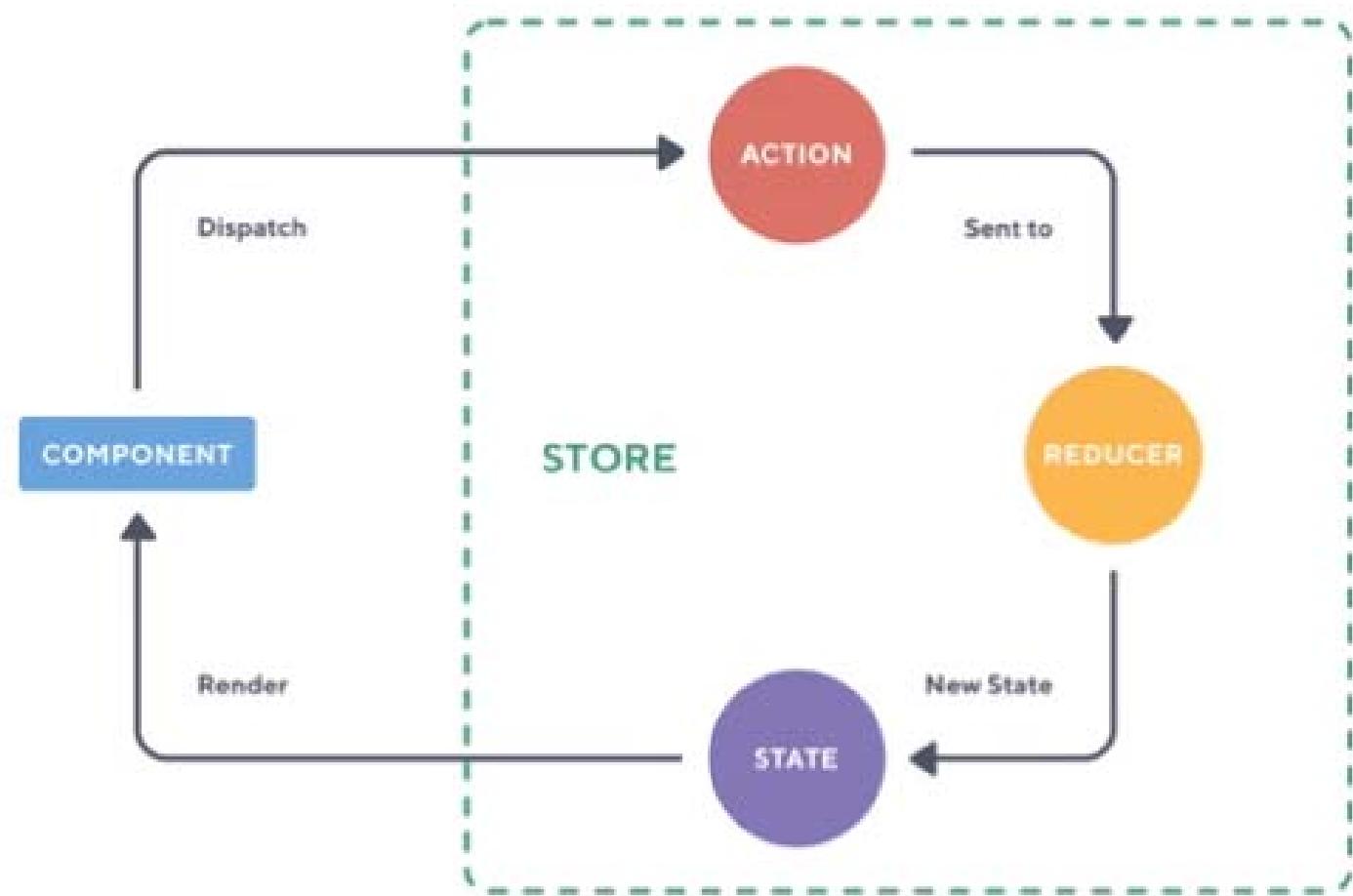
The complete picture

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).

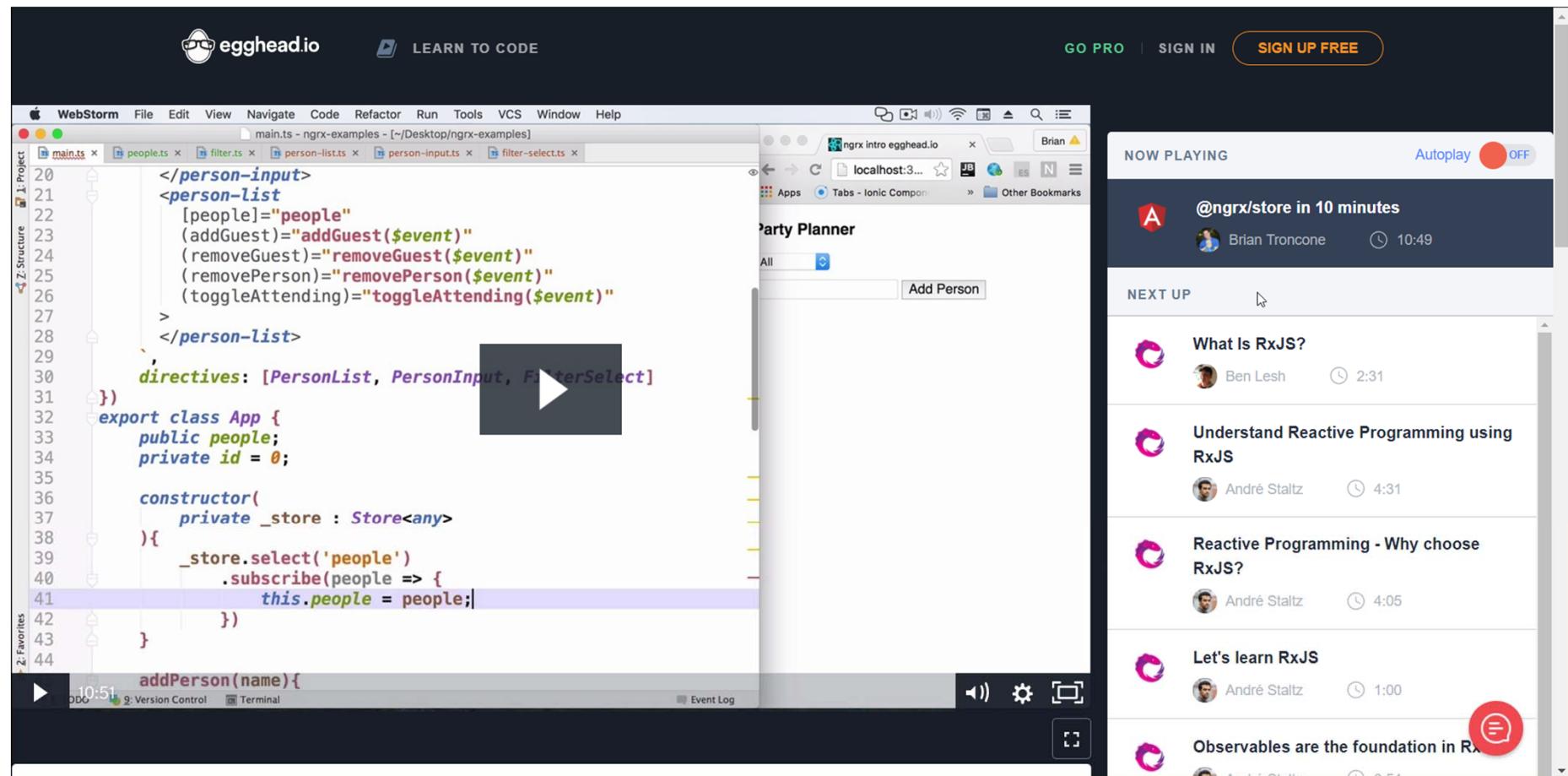
REDUX ARCHITECTURE

One-way dataflow



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

Store concepts in a video (a little bit old now)



<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
- Via npm install or ng add
- Older versions have different installations!

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

Adding via Angular CLI

- **ng add @ngrx/store**
- Option flags, see <https://ngrx.io/guide/store/install>
- Adding via Angular CLI will do the following
 - Update dependencies in package.json and npm install
 - Create src/app/reducers folder.
 - Create src/app/reducers/index.ts file with an empty State interface, an empty reducers map, and an empty metaReducers array.
 - Update src/app/app.module.ts.

Installation docs

The screenshot shows the official @ngrx/store documentation website. The header includes a navigation bar with links for Getting Started, Docs, Blog, Resources, Events, GitHub, and Sponsor, along with a search bar and social media icons.

The main content area is titled "Installation". It contains sections for "Installing with npm", "Installing with yarn", and "Installing with ng add". Each section includes a command-line example in a dark box:

- Installing with npm**: `npm install @ngrx/store --save`
- Installing with yarn**: `yarn add @ngrx/store`
- Installing with ng add**: `ng add @ngrx/store`

Below the "ng add" section, there is a "Optional ng add flags" section with a list of bullet points:

- path - path to the module that you wish to add the import for the StoreModule to.
- project - name of the project defined in your angular.json to help locating the module to add the generated code to

A sidebar on the left lists various documentation categories, and a vertical sidebar on the right provides a detailed navigation for the "Installation" section.

<https://ngrx.io/guide/store/install>



Creating your first store

Set up a simple store – explaining all the concepts

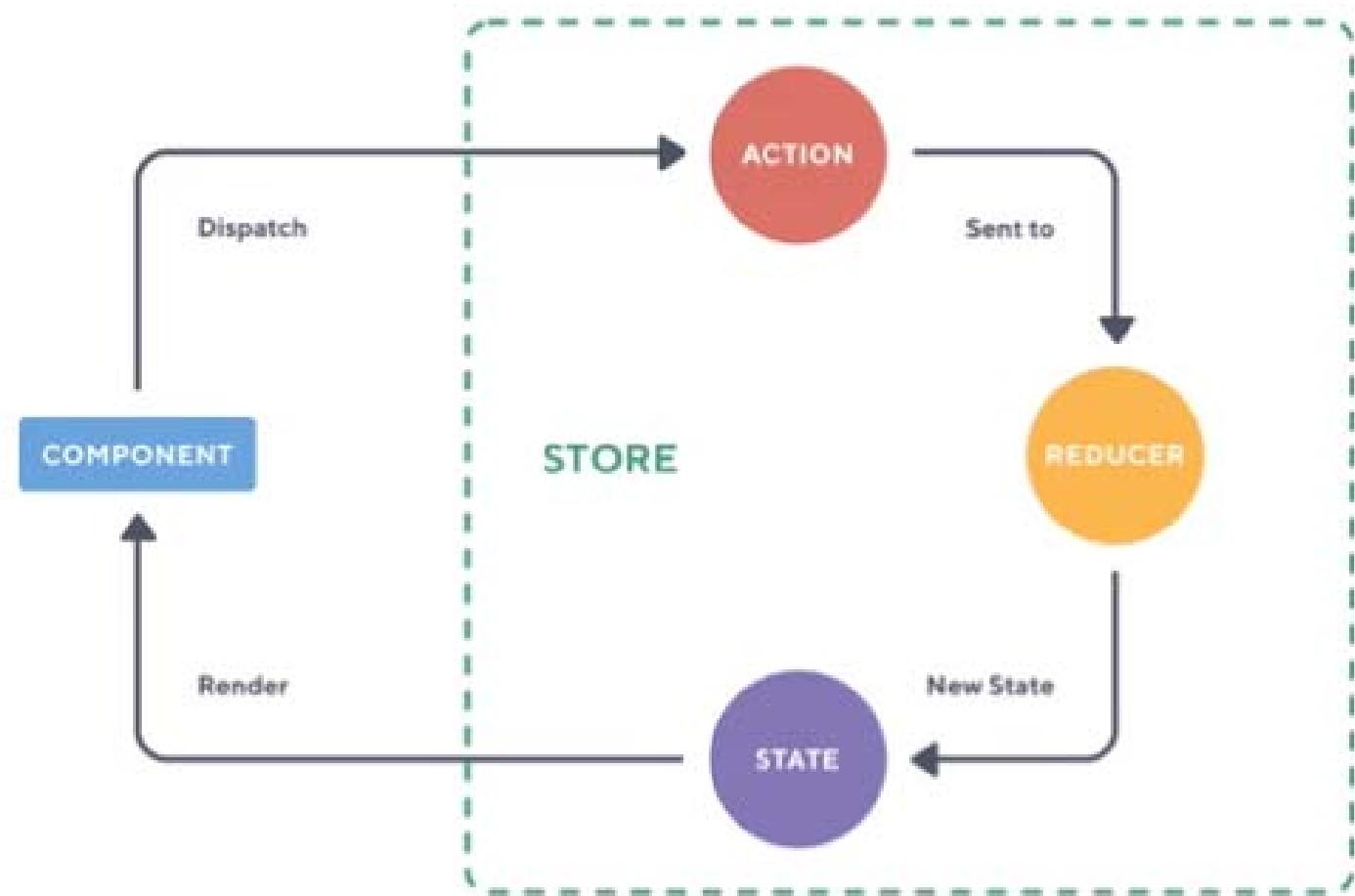
Step 0 – install core files

- We're adding the store manually to explain all concepts

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

REDUX ARCHITECTURE

One-way dataflow



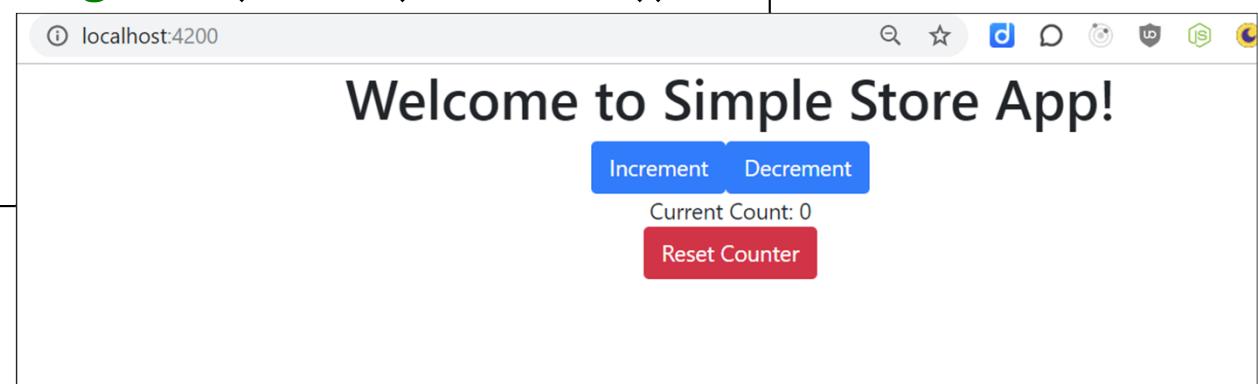
<https://platform.ultimateangular.com/courses/ngrx-store-effects/lectures/3788532>

Start somewhere, then work clockwise

- 1. For instance, first create a **component**

```
<!-- Simple Component, holding a counter store -->
<div>
  <h1>
    Welcome to {{ title }}!
  </h1>
  <button (click)="increment()">Increment</button>
  <button (click)="decrement()">Decrement</button>
  <div>Current Count: {{ count$ | async }}</div>

  <button class="btn btn-danger" (click)="reset()">
    Reset Counter
  </button>
</div>
```



2. Create your actions

- Create a new file, `./store/counter.actions.ts`
- The architecture can be complex, with nested (sub) folders etc, but it doesn't matter for the internals

```
// counter.actions.ts - the Actions for our counter
import {createAction} from '@ngrx/store';

// export our actions as constants
export const increment = createAction('COUNTER - increment');
export const decrement = createAction('COUNTER - decrement');
export const reset = createAction('COUNTER - reset');
```

3. Create your reducers

- 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
- We're going to create more complex actions, with payload later on
- You'll need the exported reducer function to support AOT-compiling
- <https://ngrx.io/guide/store/reducers>

```
// Import store stuff and available actions
import {Action, createReducer, on} from '@ngrx/store';
import {decrement, increment, reset} from './counter.actions';
```



```
// Initial state: counter=0
export const initialState = 0;
```

```
// Internal variable/function with reducers. It receives a state from
// the actual (exported) counterReducer below
```

```
const reducer = createReducer(initialState,
  on(increment, state => state + 1),
  on(decrement, state => state - 1),
  on(reset, state => 0)
);
```



```
// The exported reducer function is necessary
// as function calls are not supported by the AOT compiler.
```

```
export const counterReducer = (state = 0, action: Action) => {
  return reducer(state, action);
};
```



4. Adding store and reducer to module

- Register the state container with your application.
- Import reducers
- Use `StoreModule.forRoot()` to add it to the module
- More complex: we can have a *map* of reducers, or child modules holding their own stores

```
...
// 1. import store stuff
import {StoreModule} from '@ngrx/store';
import {counterReducer} from './store/counter.reducer';

@NgModule({
  declarations: [
    AppComponent,
    ...
  ],
  imports: [
    BrowserModule,
    // 2. Add the StoreModule to the AppModule,
    // to make the store known inside the application
    StoreModule.forRoot({count: counterReducer}),
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
```



5. Using/calling the Store in component

- Import and inject the Store service to components
- Initialize the store with correct Type
 - More complex: create a custom AppState interface
- Use `store.pipe(select())` to select slice(s) of the state
- Add methods to dispatch actions
 - `increment()`
 - `decrement()`
 - etc..

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

// Import all possible actions
import {increment, decrement, reset} from './store/counter.actions';

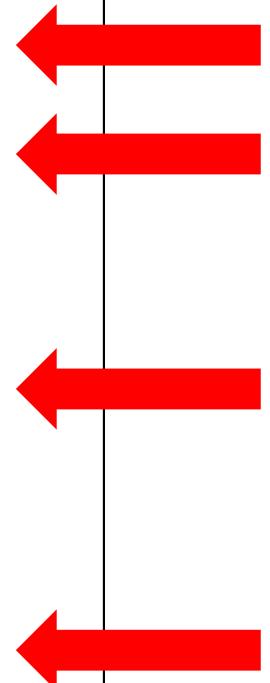
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements OnInit {
  title = 'Simple Store App';
  count$: Observable<number>;

  constructor(private store: Store<{ count: number }>) {}

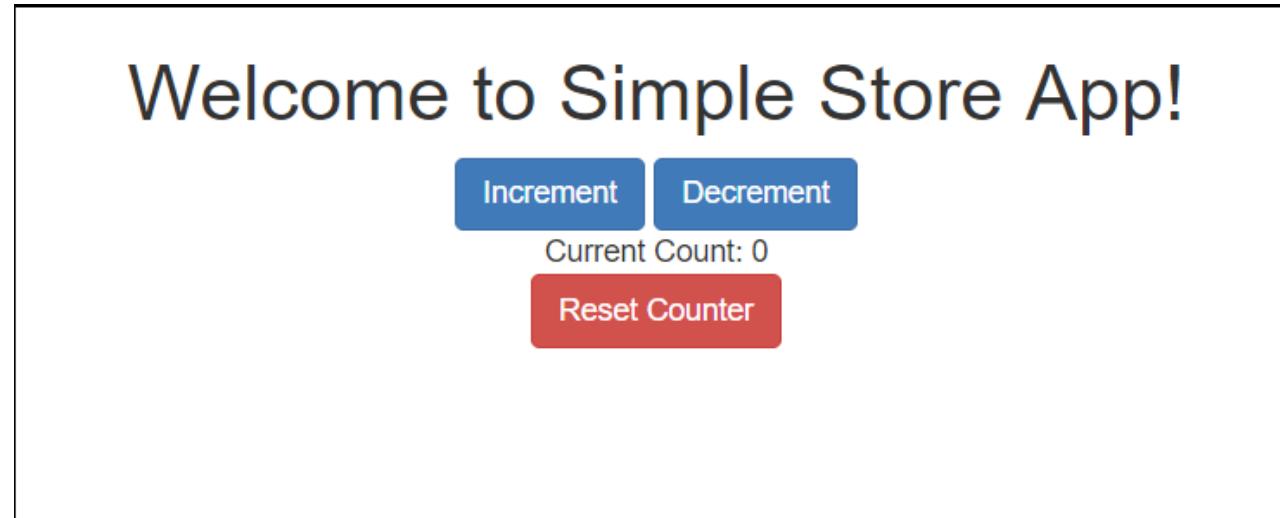
  ngOnInit() {
    // Select the 'count' property from the store and
    // assign it to count$ variable.
    this.count$ = this.store.pipe(
      select('count')
    );
  }

  // dispatch actions for the store. They are imported above
  increment() {
    this.store.dispatch(increment());
  }
  ...
}


```



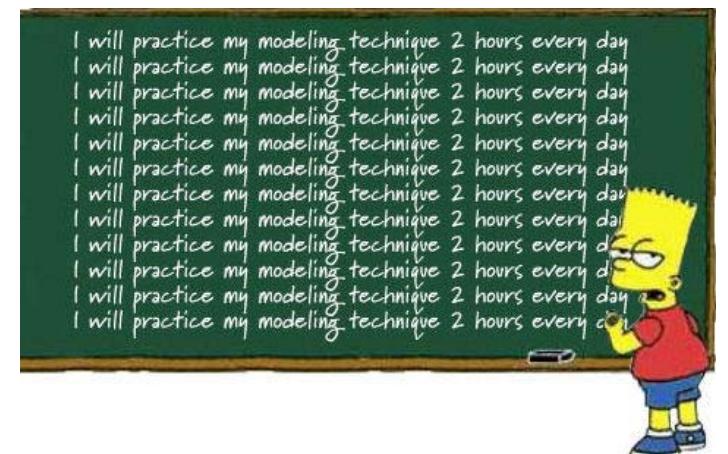
Run the app



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 ;



Official site



<https://ngrx.io/>

Rob Wormald - co-created @ngrx/store (1st gen!)

The image shows a YouTube video player interface. The main video frame displays a presentation slide with a blue background. The text on the slide reads: "Rob Wormald", "Developer Advocate for Angular", and "@robwormald". To the right of the slide, there is a small video thumbnail showing Rob Wormald speaking at a podium. Below the video player, the title "Reactive Angular2 with ngRx - Rob Womald" is visible, along with the "ng-conf" logo and some playback controls.

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

Online Training by Todd Motto (2nd gen!)

The screenshot shows the Ultimate Angular website with a purple header featuring the logo 'ULTIMATE ANGULAR™'. The main navigation menu includes 'Courses', 'About', 'Shop', 'Reviews', and 'Login'. The page title is 'NGRX Store + Effects'.

Course details:

- Instructor: with Todd Motto
- Lessons: 41 lessons
- Duration: 7 hours
- Version: ngrx:v4

Description:

Master reactive and highly performant state management for Angular apps. Everything you need on structuring data, entities, selectors, the Redux pattern, side effects and immutability, through to preloading, router state and testing. All as we build out a real-world application, end to end.

Course features (in boxes):

- Stream or download**: Learn at home or on the daily commute. Online or offline viewing.
- Free lifetime updates**: Never stop learning. Each course comes with unlimited updates forever.
- Single course**: NGRX Store + Effects NEW
- Slack community**
- Expert trainers**

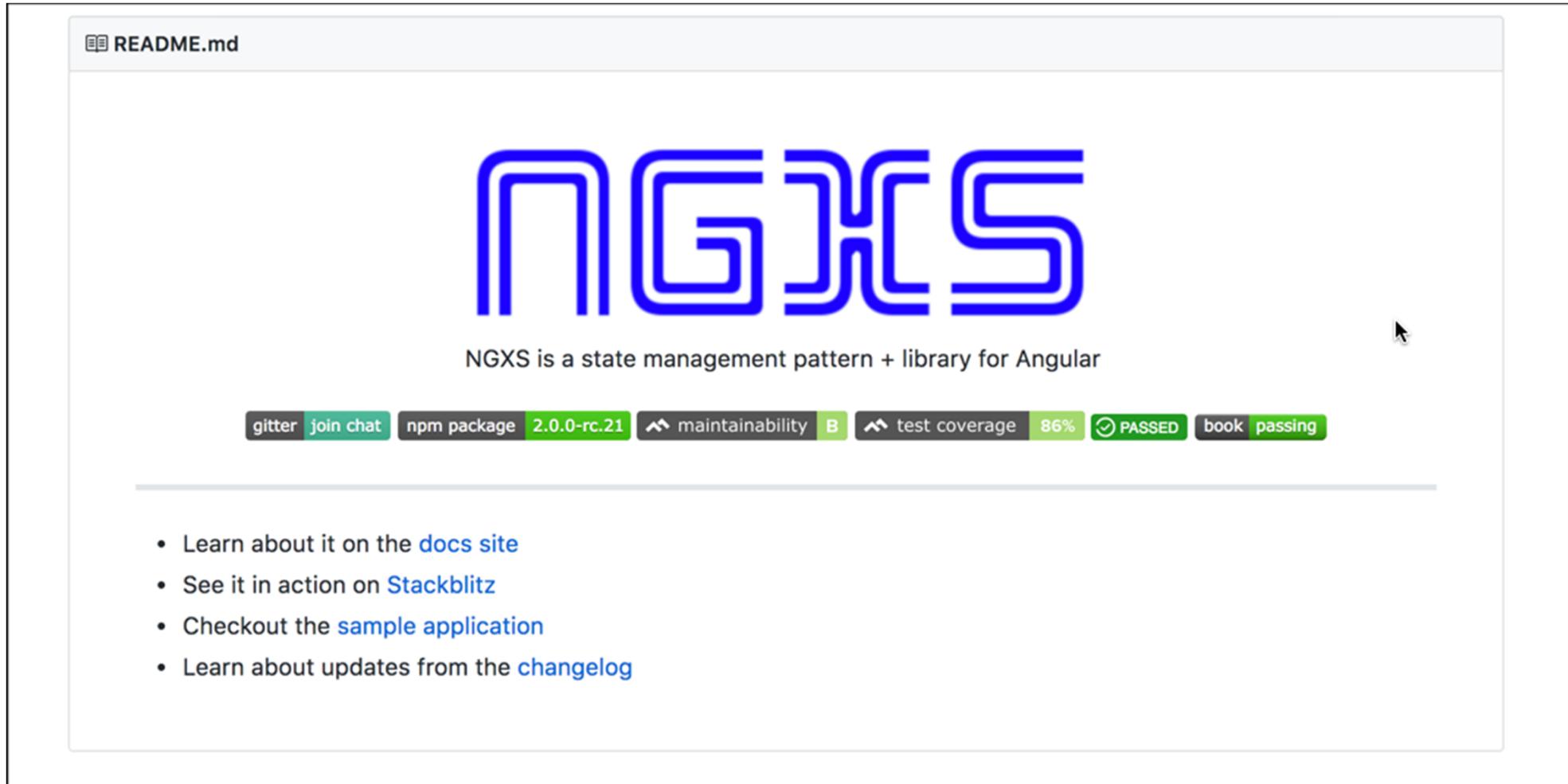
<https://ultimateangular.com/ngrx-store-effects>

Think about this – “The Ugly side of Redux”

The screenshot shows a Medium article page. At the top, there's a navigation bar with a 'Upgrade' button, the 'Medium' logo, a search icon, a notification bell, and a user profile picture. Below the header, a message says 'Applause from John Papa and 26 others'. The author is Nir Yosef, followed by a 'Follow' button, with the date Dec 13 and a note of '10 min read'. The main title of the article is 'The ugly side of Redux'. The first paragraph of the article reads: 'In this post I will briefly explain Redux at a very high level for those of you who don't already know it, I will try to convince you why Redux is actually promoting an anti-pattern (the singleton global store), and I'll show you my own alternative called Controllerim.' Below the article, there's a section titled 'What is Redux' with a definition: 'Redux is a predictable state container for JavaScript apps.' A note below the definition says 'Simple isn't it? You just take Redux and put a state inside it, and everything'. At the bottom of the page, there are social sharing icons (hand, 186, Q, Twitter, Facebook, bookmark) and a link to the next story: 'React-Native Tutorial #3— Int...'

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

Alternative State Management solution



The screenshot shows the GitHub README page for the NGXS project. At the top left is a link to 'README.md'. Below it is the large blue NGXS logo. Underneath the logo is the text 'NGXS is a state management pattern + library for Angular'. A cursor icon is visible on the right side of the page. Below this section are several GitHub badges: 'gitter' (join chat), 'npm package' (2.0.0-rc.21), 'maintainability' (B), 'test coverage' (86%), 'PASSED' (book passing). A horizontal line separates this from a list of links at the bottom. The list includes: 'Learn about it on the [docs site](#)', 'See it in action on [Stackblitz](#)', 'Checkout the [sample application](#)', and 'Learn about updates from the [changelog](#)'.

<https://github.com/amcdnl/ngxs>

Akita – another state management alternative

M |  Follow

 Introducing Akita: A New State Management Pattern for Angular Applications

 Netanel Basal
Jun 12, 2018 · 4 min read



<https://netbasal.com/introducing-akita-a-new-state-management-pattern-for-angular-applications-f2f0fab5a8>

Next Steps

- [`@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.
- [`@ngrx/schematics`](#) - Scaffolding library for Angular applications using NgRx libraries

<https://ngrx.io/docs>



Sample Store apps

Some study material

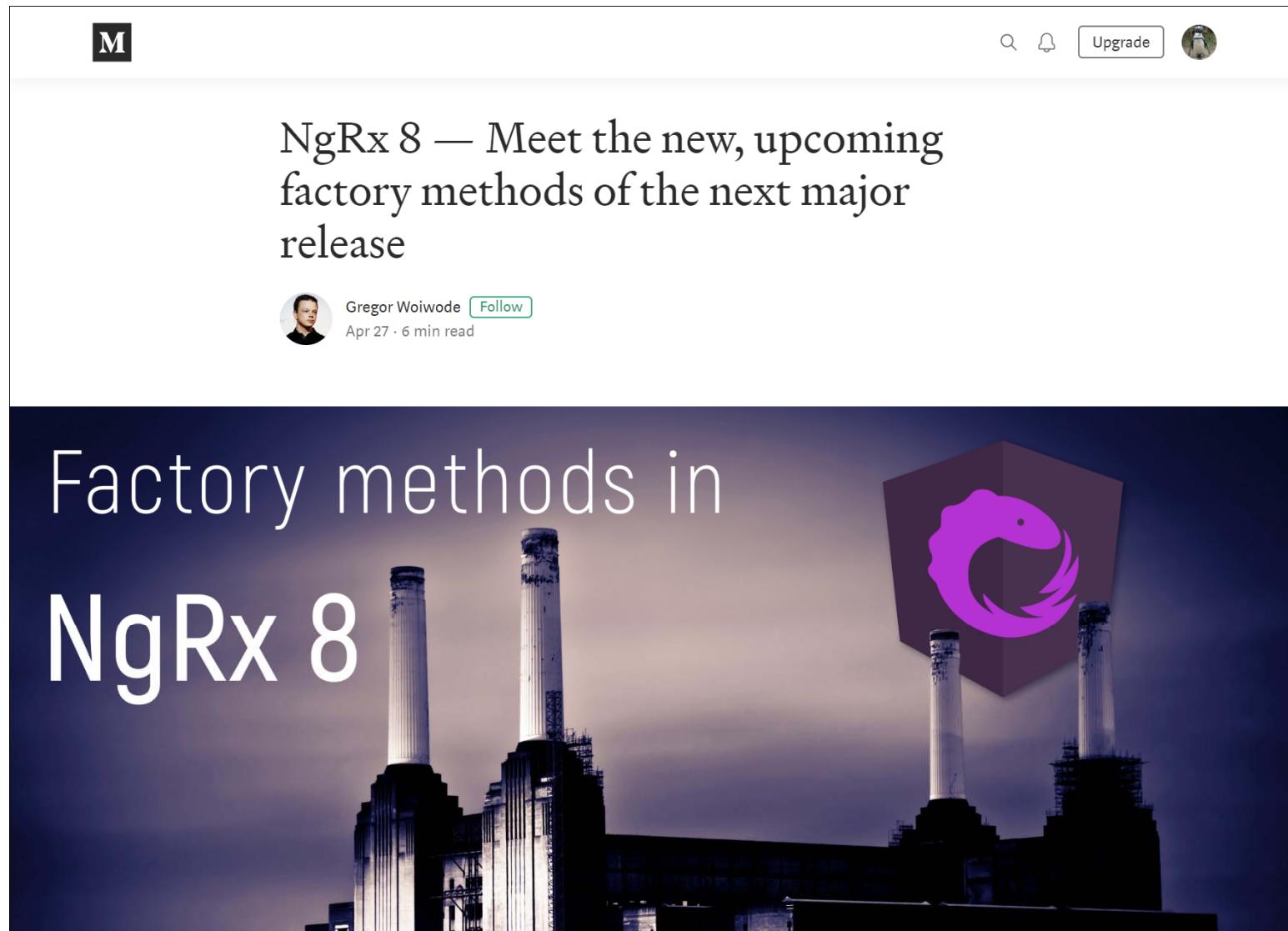
Ngrx store platform sample app

The screenshot shows a GitHub repository page for 'ngrx / platform'. The repository has 220 stars and 1,142 forks. The 'Code' tab is selected, showing the 'platform / projects / example-app /' directory. The commit history for this directory is displayed, showing several commits from 'timdeschryver' and 'brandonroberts' made between 4 and 6 months ago. The commits are related to making the example app more user friendly, updating dependencies, and moving files into the 'projects' folder. A note at the bottom of the commit list reads '@ngrx example application'.

File / Commit	Message	Time Ago
src	feat(example): make the example app more user friendly (#1508)	2 months ago
README.md	docs(example): update stackblitz link (#1277)	6 months ago
browserslist	chore(example): move example app into projects folder (#1242)	6 months ago
jest.config.js	feat: update angular dependencies to V7	4 months ago
karma.conf.js	chore(example): move example app into projects folder (#1242)	6 months ago
tsconfig.app.json	feat: update angular dependencies to V7	4 months ago
tsconfig.spec.json	chore(example): move example app into projects folder (#1242)	6 months ago
tslint.json	chore(example): move example app into projects folder (#1242)	6 months ago
README.md		

<https://github.com/ngrx/platform/tree/master/projects/example-app>

More info

A screenshot of a Medium article page. At the top, there's a dark header bar with a 'M' logo, a search icon, a notification bell, an 'Upgrade' button, and a user profile picture. The main title of the article is "NgRx 8 — Meet the new, upcoming factory methods of the next major release". Below the title is a small author bio showing a profile picture of Gregor Woiwode, the name "Gregor Woiwode", a "Follow" button, and the text "Apr 27 · 6 min read". The main content area features a large image of the Battersea Power Station chimneys at night. Overlaid on the image is the text "Factory methods in NgRx 8" in white. To the right of the text, there's a purple 3D cube with the NgRx logo (a stylized orange 'G' inside a circle) on its front face. A vertical scroll bar is visible on the right side of the content area.

<https://medium.com/@gregor.woiwode/ngrx-8-meet-the-new-upcoming-factory-methods-of-the-next-major-release-a97a079cc089>

Store tutorial from scratch (2nd gen!)

ANGULAR

Angular NgRx Store Tutorial Example From Scratch

Angular 5 ngrx/store Tutorial

By  Krunal — Last updated Oct 23, 2018

 25,859 |  11



Get real time updates directly on your device, subscribe now.



<https://appdividend.com/2018/01/31/angular-ngrx-store-tutorial-example-scratch/>