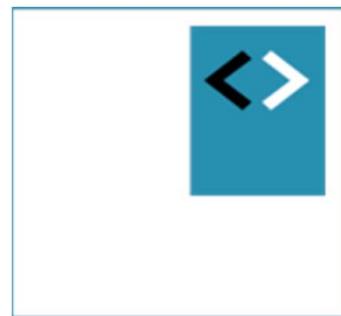




Angular Advanced State management and @ngrx/store



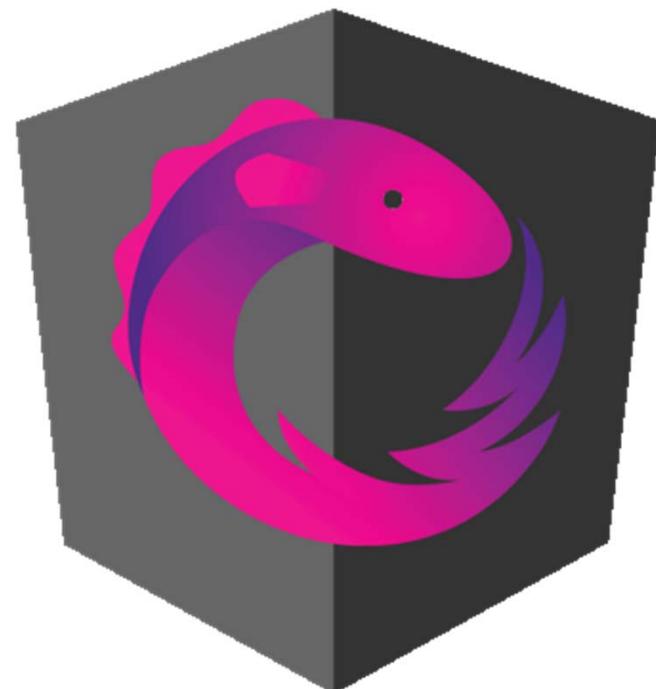
Peter Kassenaar
info@kassenaar.com

What is State Management?

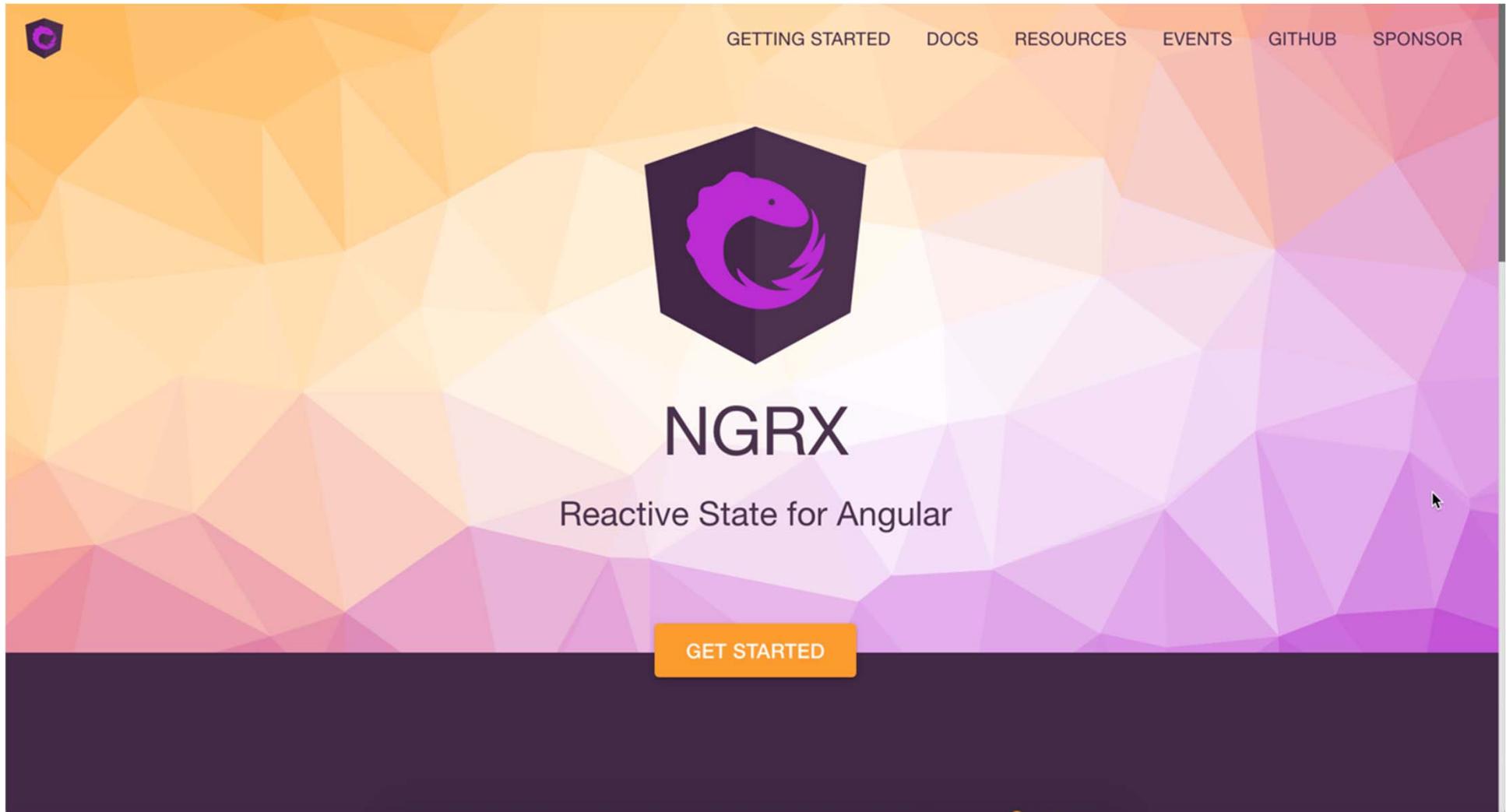
- Various **design patterns**, used for managing *state* (data in its broadest sense!) in your application.
- **Multiple solutions** possible – depends on application & framework



Redux



<https://ngrx.io/>

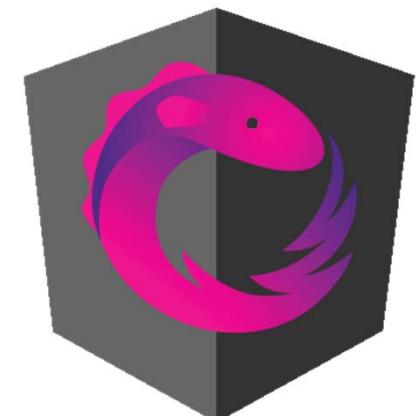


What is ngrx?

*“Ngrx provides **reactive state management** for Angular apps inspired by Redux”*

@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
 - (they try to make it) less complex...
 - ...if you know the principles and where to look



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, there's a navigation bar with the Medium logo, a search icon, a notifications icon (with a '3' notification), 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, it says "Rachel Cavin" with a "Follow" button, and the date "Dec 5, 2018 · 4 min read". The article content starts with a paragraph about managing state in small Angular apps using RxJS Behavior Subjects instead of the standard input/emitter way. At the bottom of the article, there's a small image of a yellow, textured surface and some text: "Behavior subjects are similar to regular subjects in RxJS, except".

Medium | Javascript

My favorite state management technique in Angular — RxJS Behavior Subjects

Rachel Cavin Follow Dec 5, 2018 · 4 min read

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](#)!



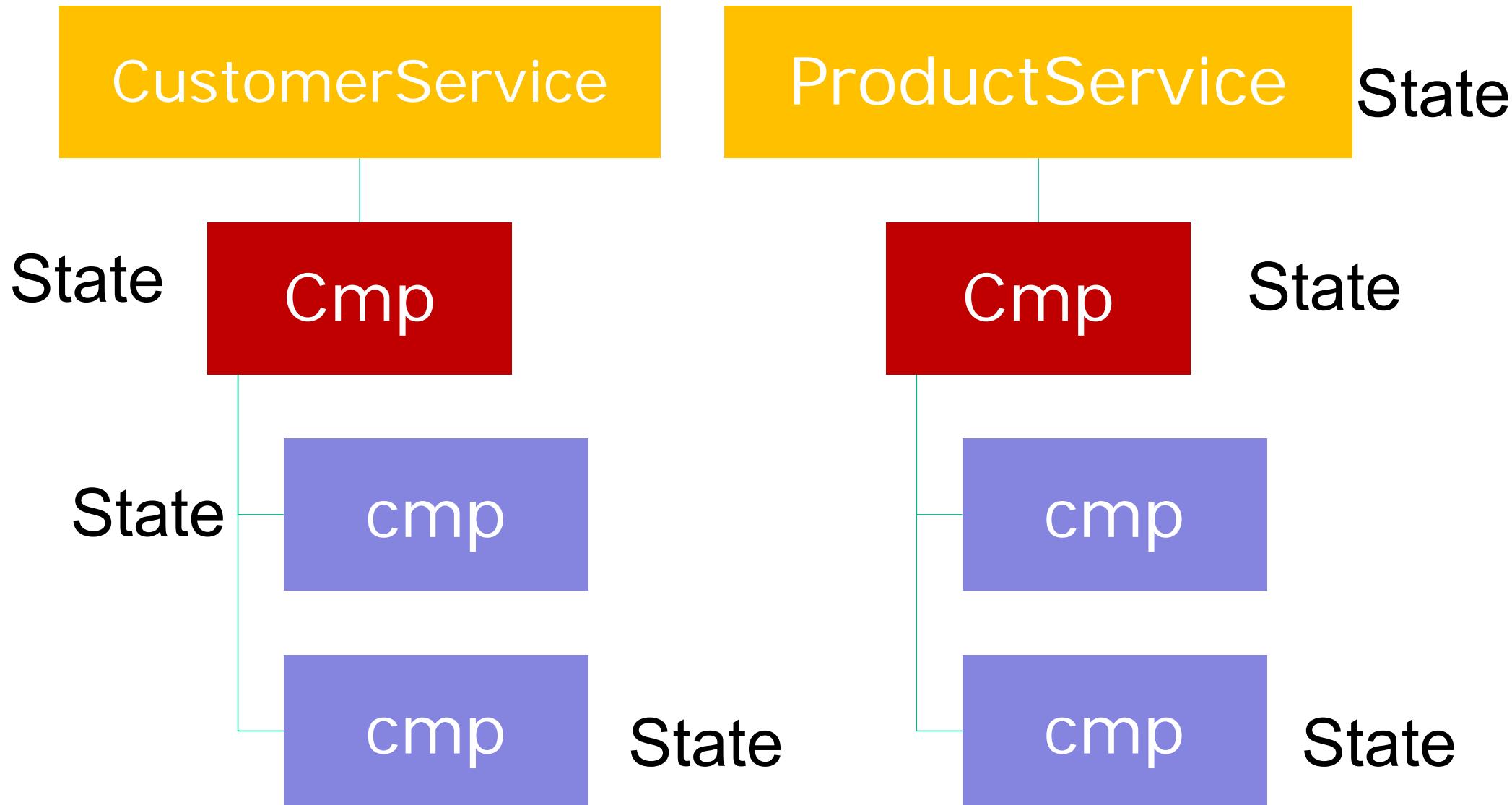
Behavior subjects are similar to regular subjects in RxJS, except



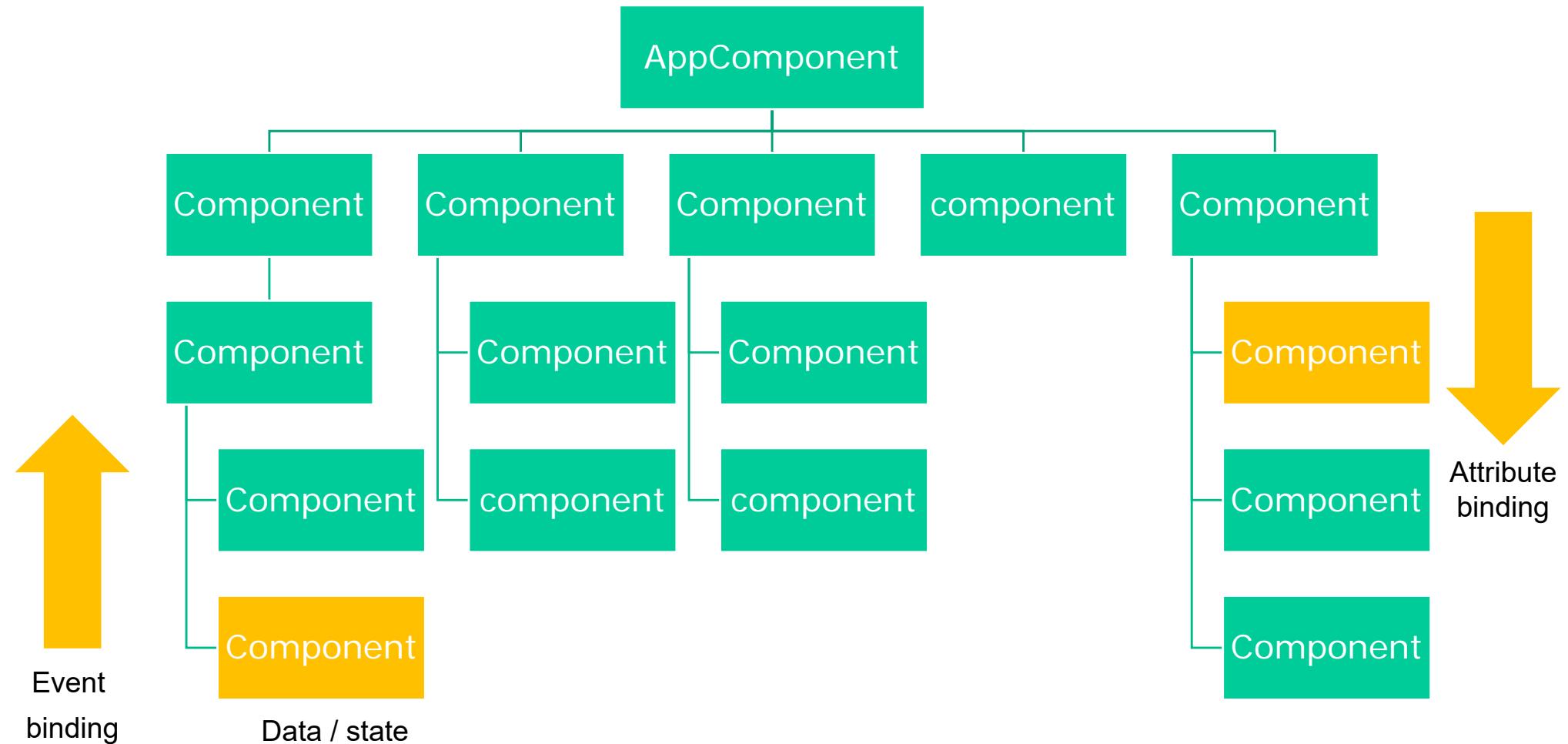
Why state management?

Why on earth would you need/want a Store ?

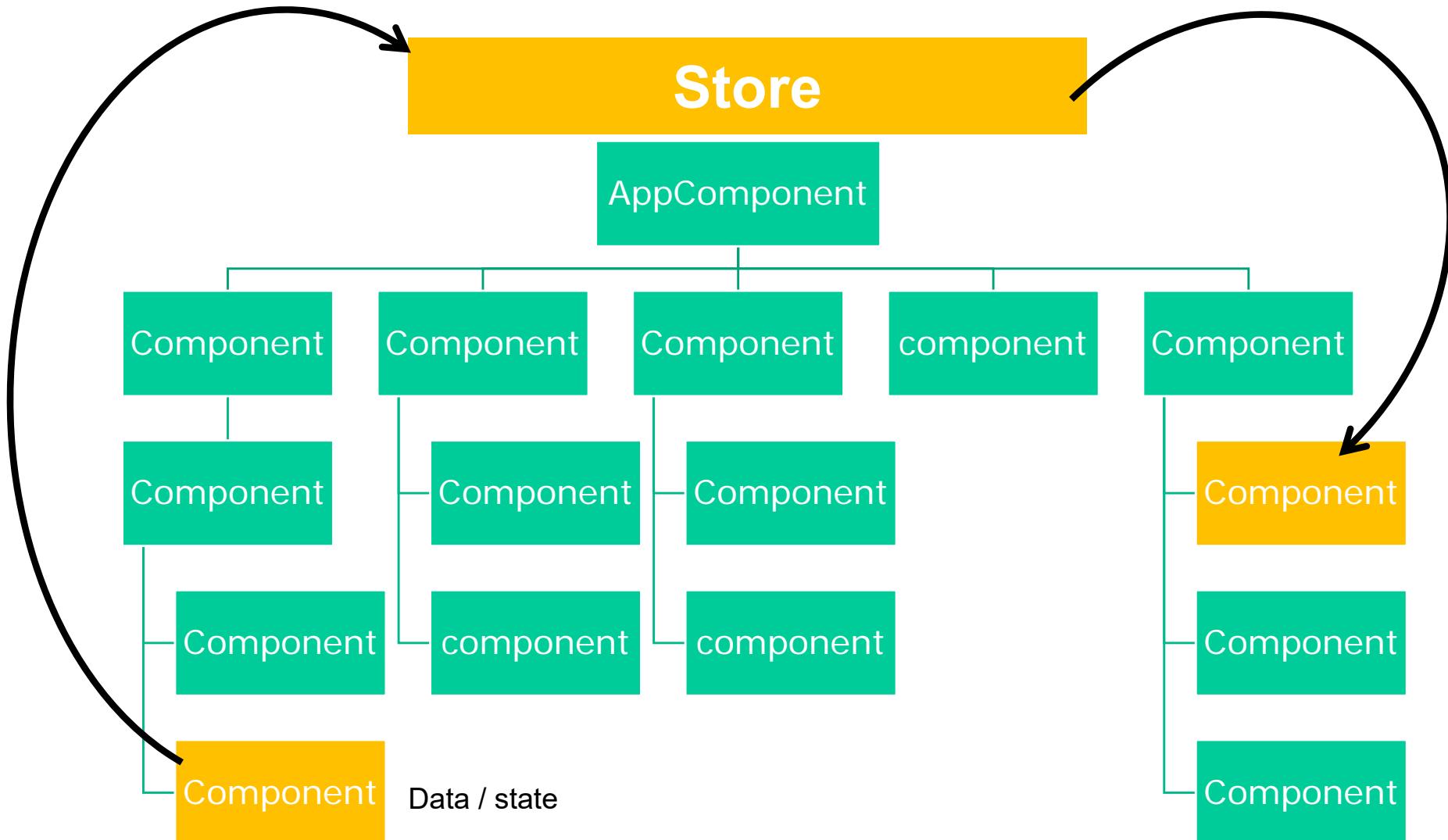
State management **without** a store



Data flow in complex applications



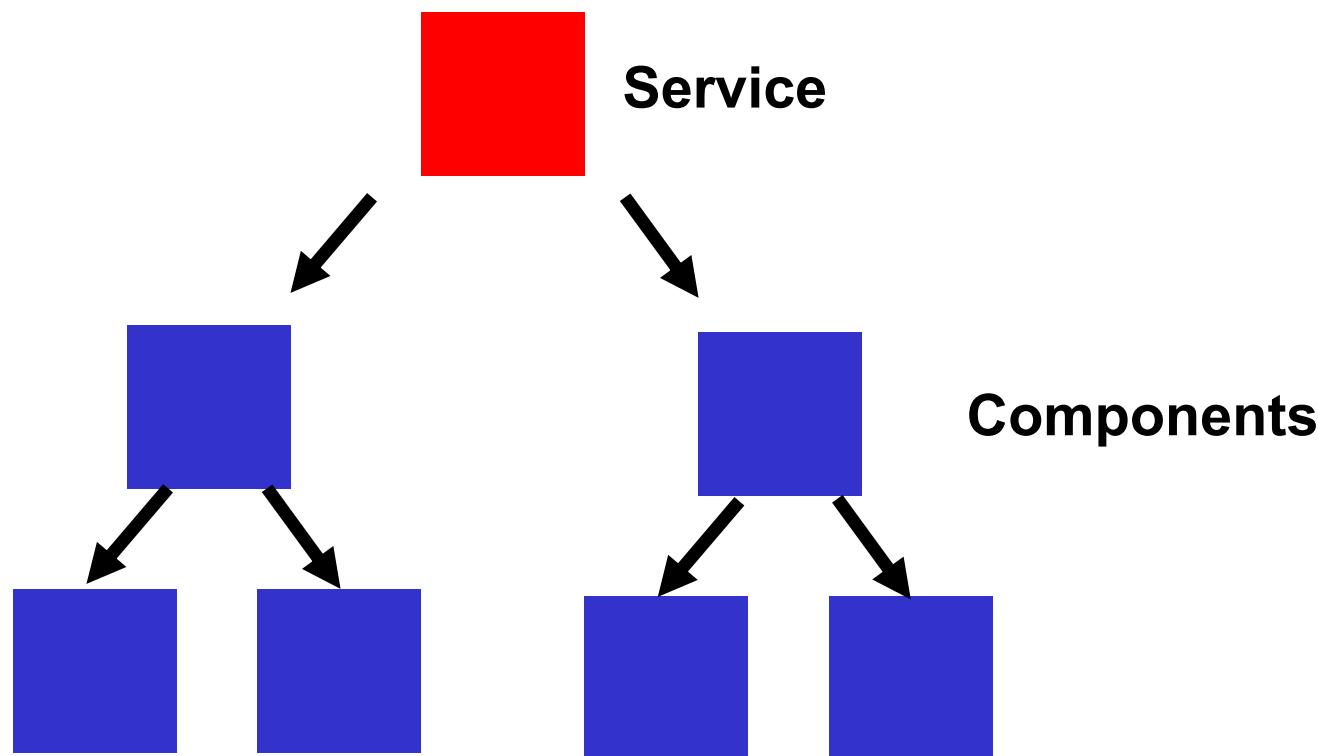
State management *with* a store



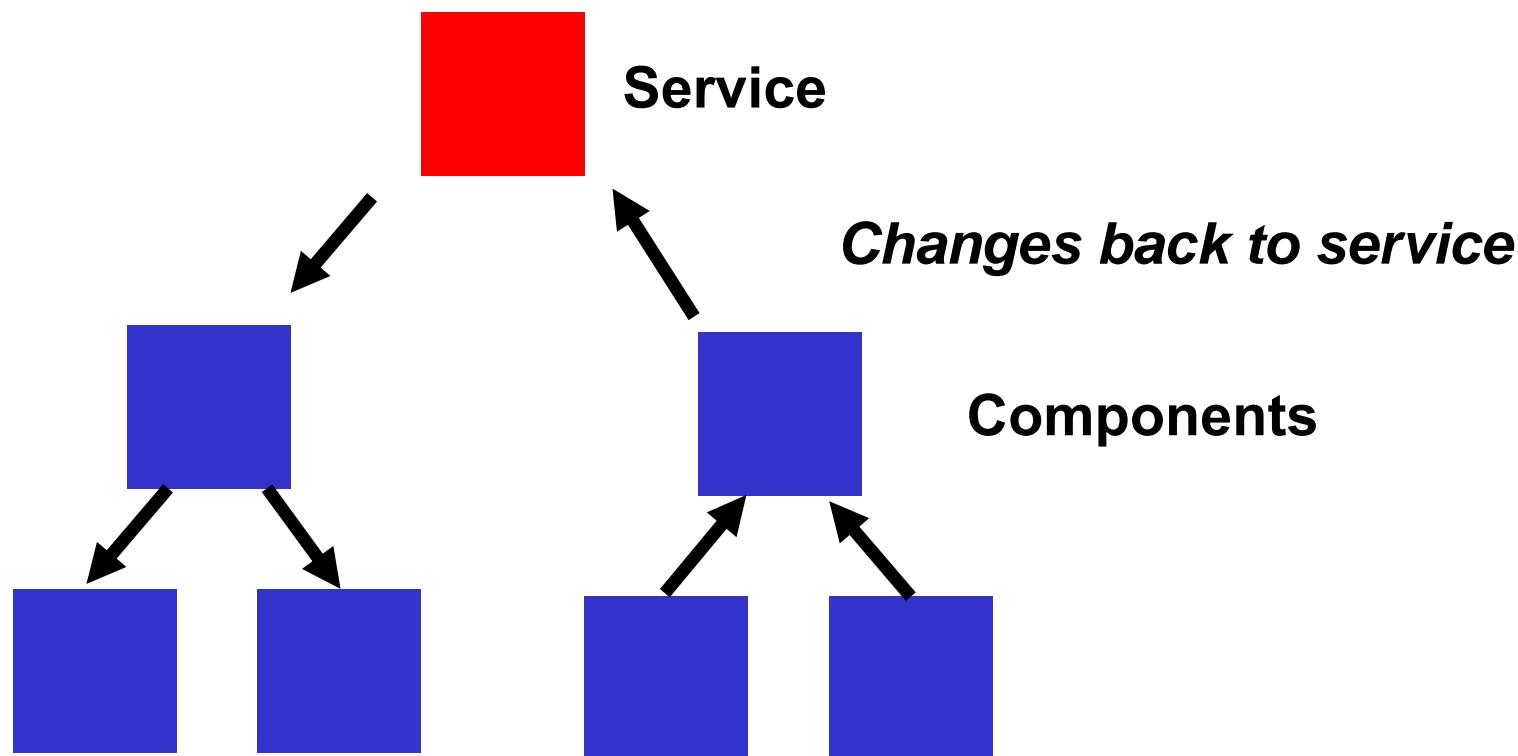
Benefits of using a store

- 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”

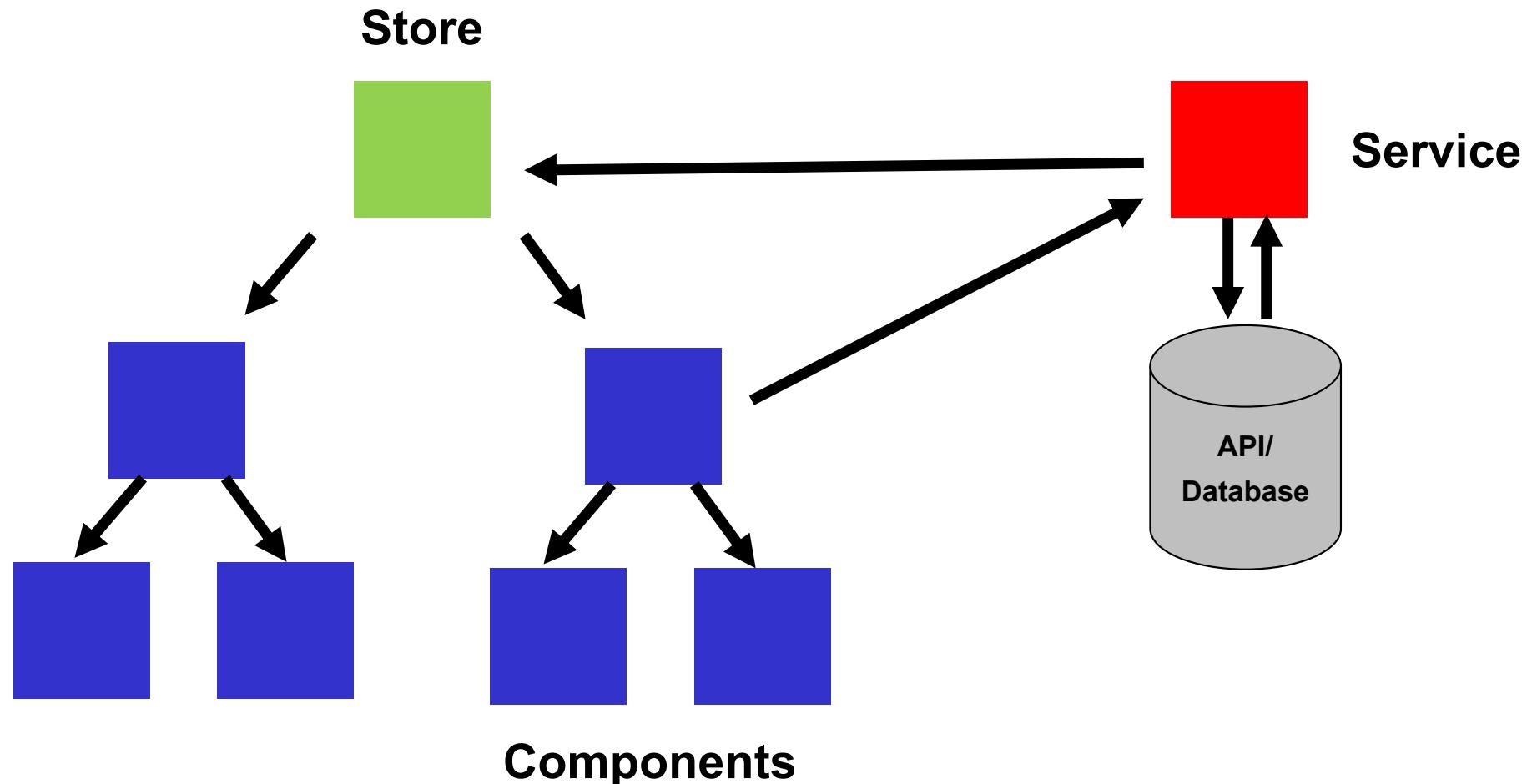
Store architecture - #2 - traditional



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());  
}
```

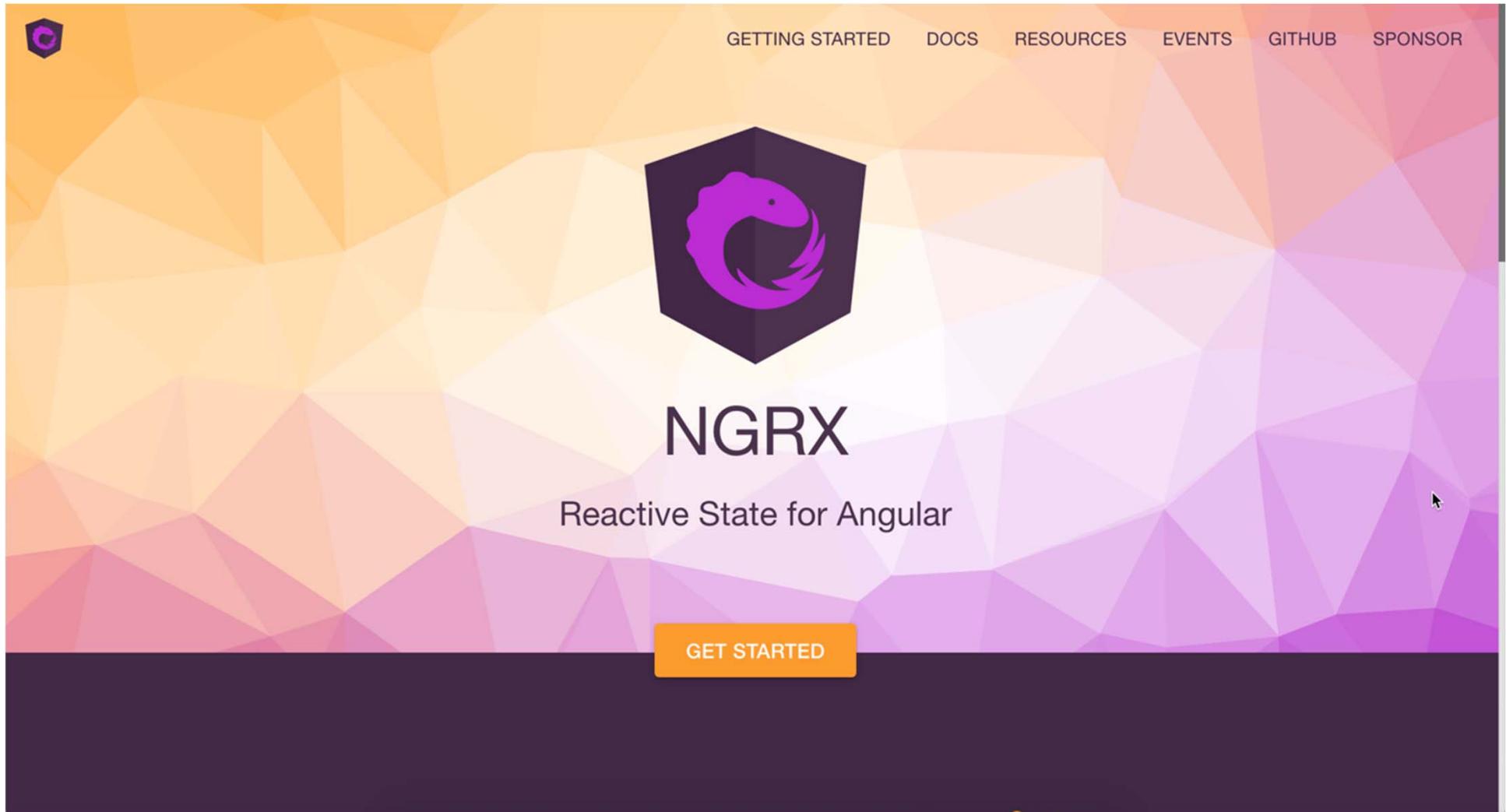


@ngrx/store

Terminology and concepts

Working with @ngrx/store, the officially endorsed state management library for Angular

<https://ngrx.io/>



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 -

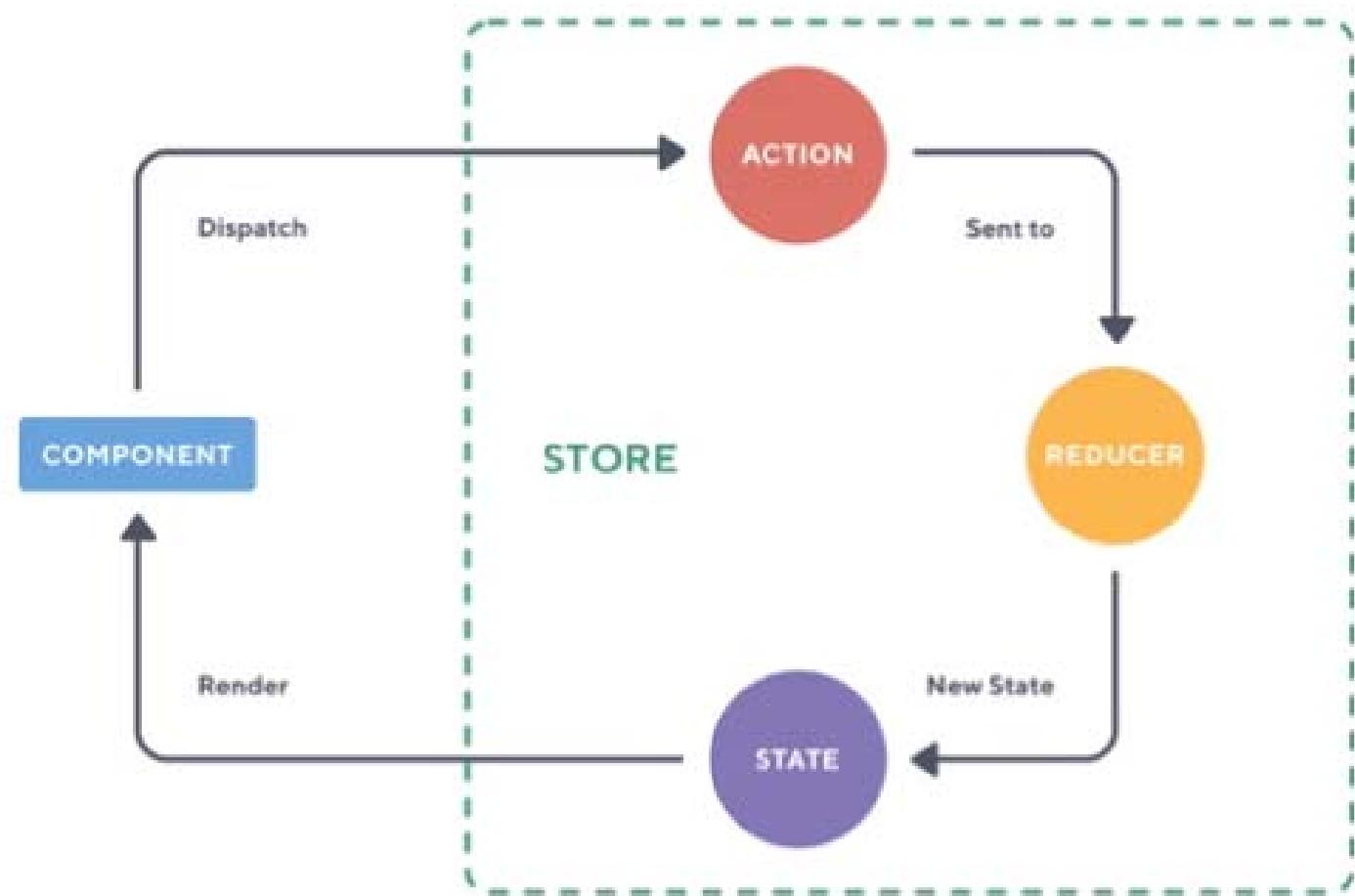
The complete picture

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).

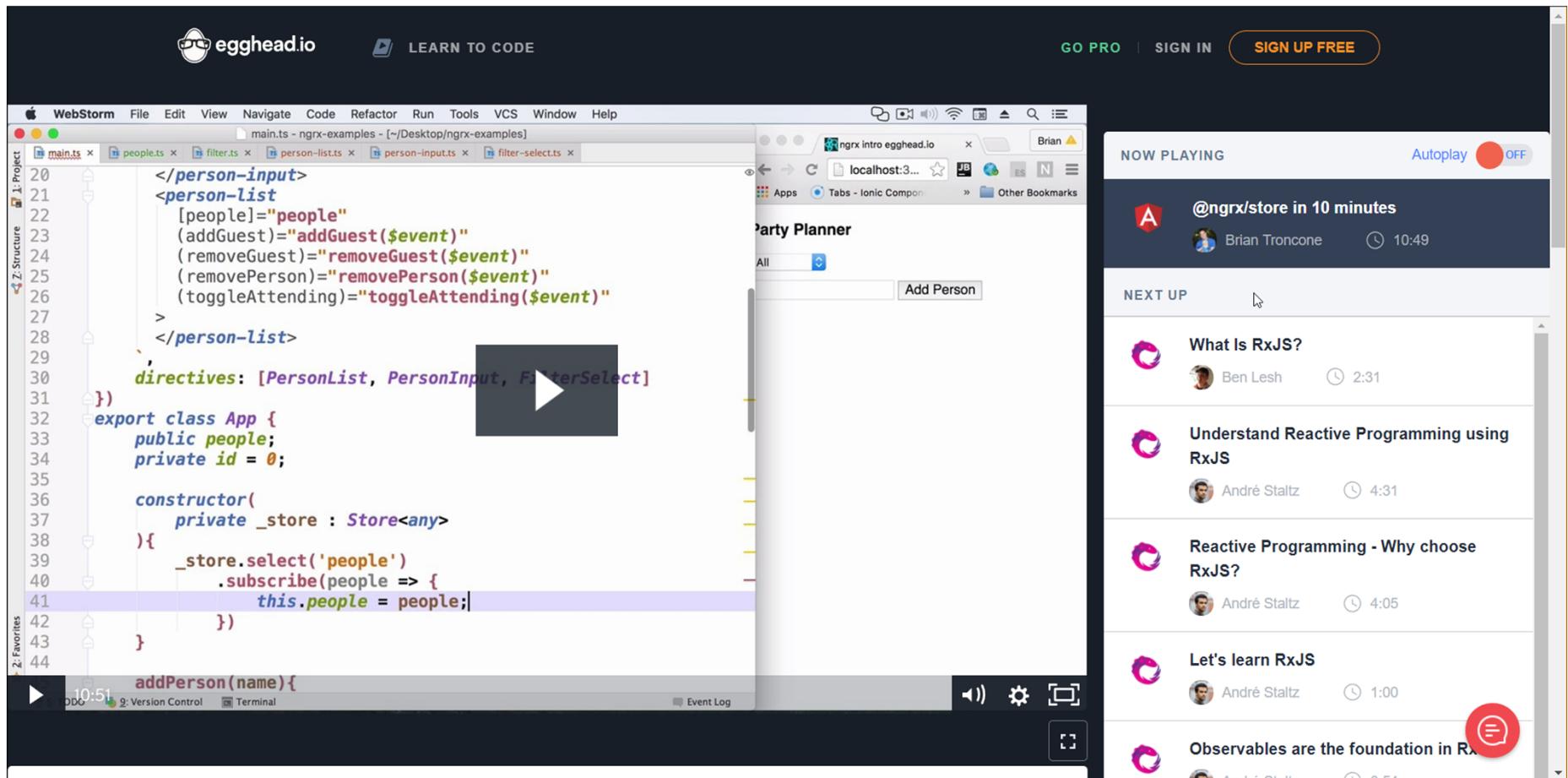
REDUX ARCHITECTURE

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
```

OR

```
ng add @ngrx/store
```

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 navigation 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' and contains sections for 'Installing with npm', 'Installing with yarn', 'Installing with ng add', and 'Optional ng add flags'. 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`
- Optional ng add flags**: A list including 'path' and 'project'.

The sidebar on the left lists various components and tools under the heading '@ngrx/store': Introduction, @ngrx/store, Getting Started, Installation, Architecture, Advanced, Recipes, Configuration, Testing, @ngrx/store-devtools, @ngrx/effects, @ngrx/router-store, @ngrx/entity, @ngrx/data, @ngrx/schematics, Recipes, Migrations, API, Contributing, and stable (v8.4.0). The 'Installation' link in the sidebar is highlighted with a blue dot.

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



Creating your first store

Set up a simple store – explaining all the concepts

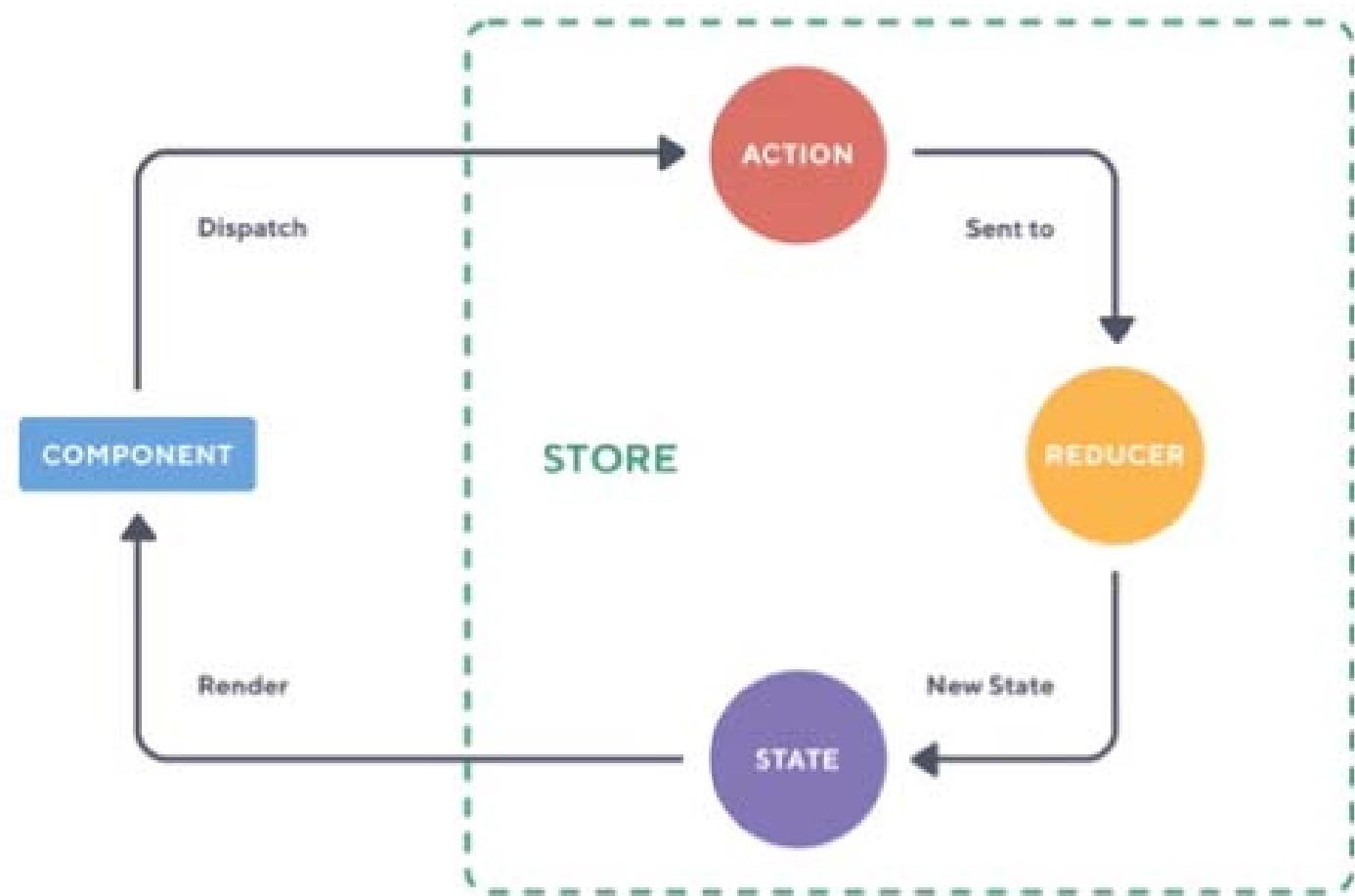
Step 0 – install core files

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

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

REDUX ARCHITECTURE

One-way dataflow



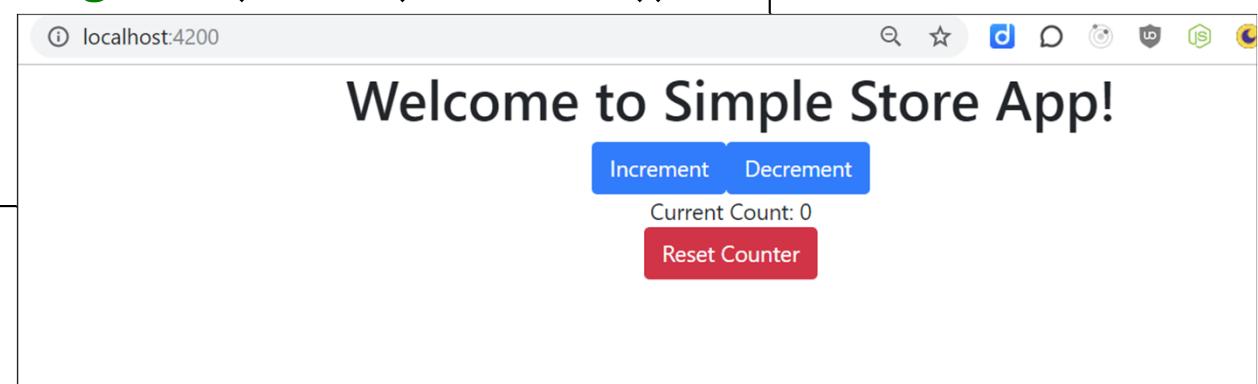
<https://platform.ultimateangular.com/courses/ngrx-store-effects/lectures/3788532>

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
 - metaReducer: <https://ngrx.io/guide/store/metareducers>

```
...
// 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';

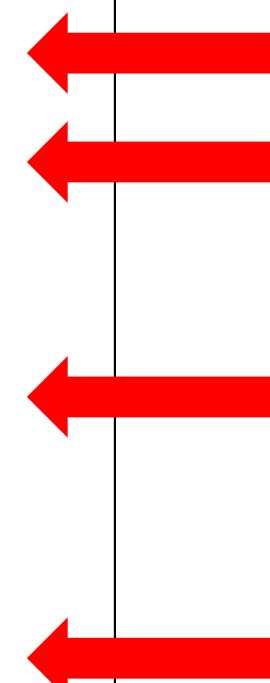
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements OnInit {
  title = 'Simple Store App';
  count$: Observable<number>;

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

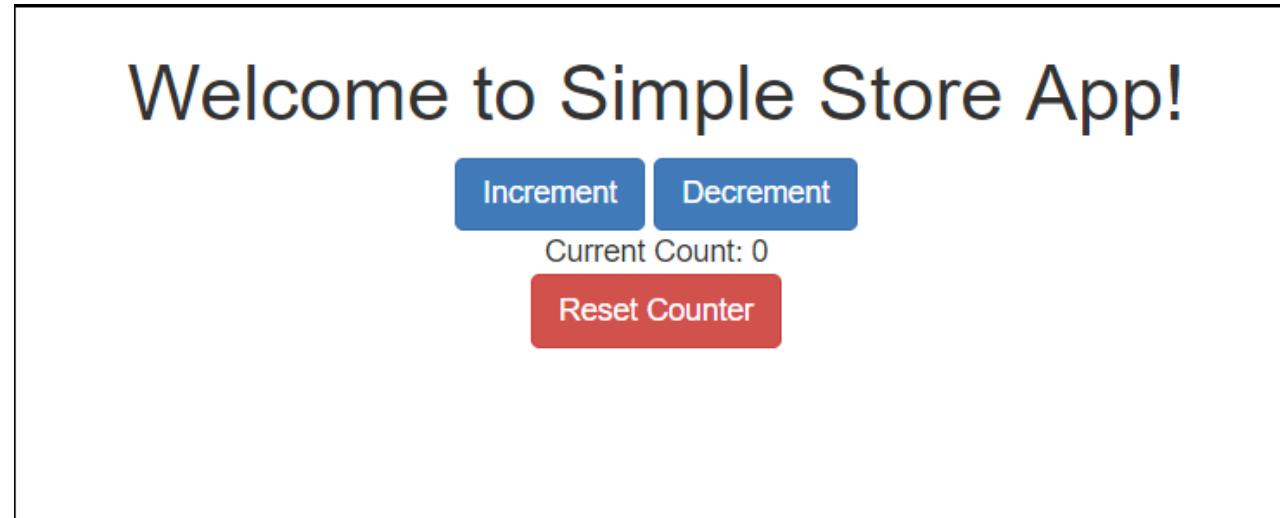
  ngOnInit() {
    // Select the 'count' property from the store and
    // assign it to count$ variable.
    this.count$ = this.store.pipe(
      select('count')
    );
  }

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


```



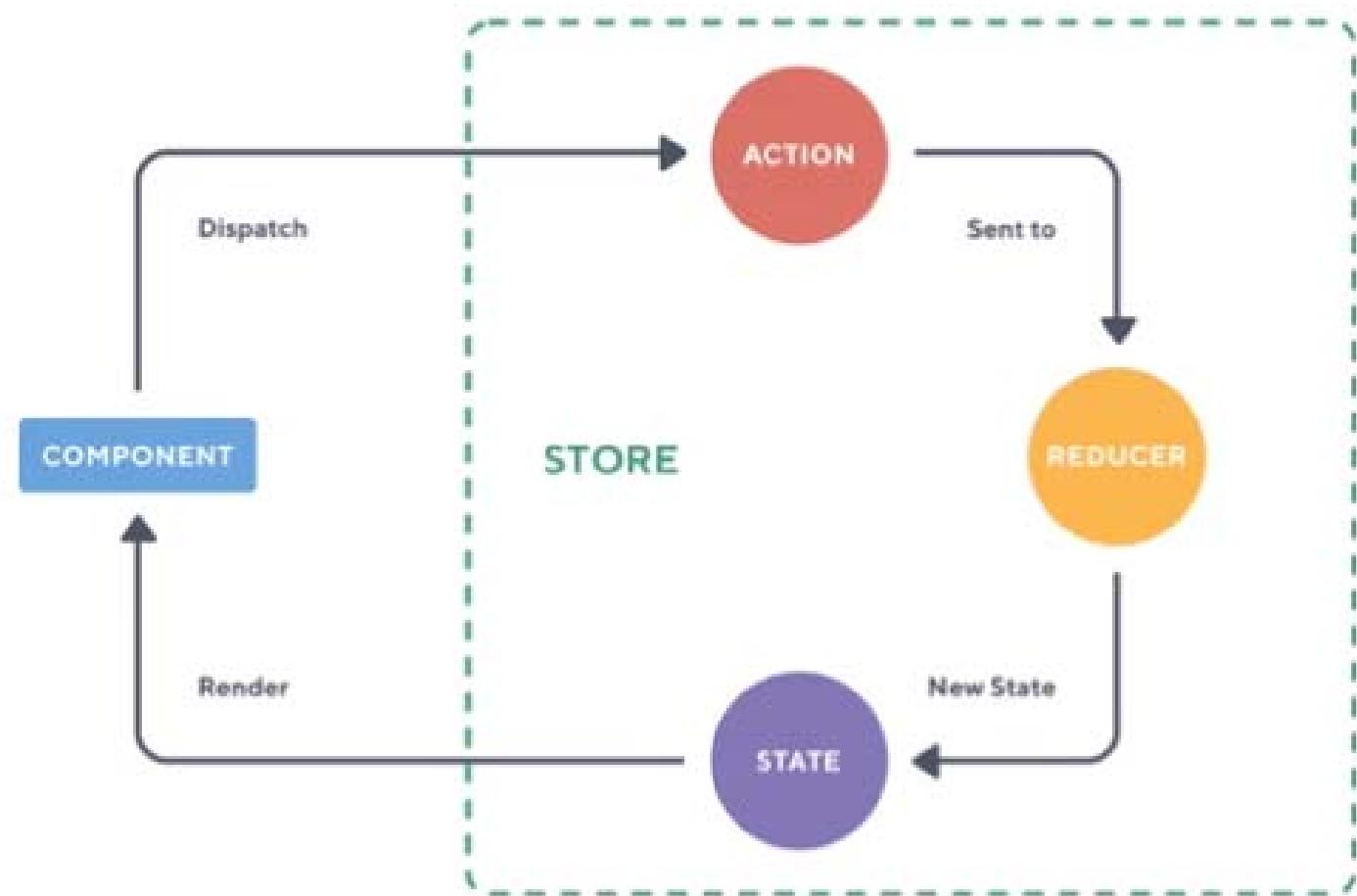
Run the app



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

REDUX ARCHITECTURE

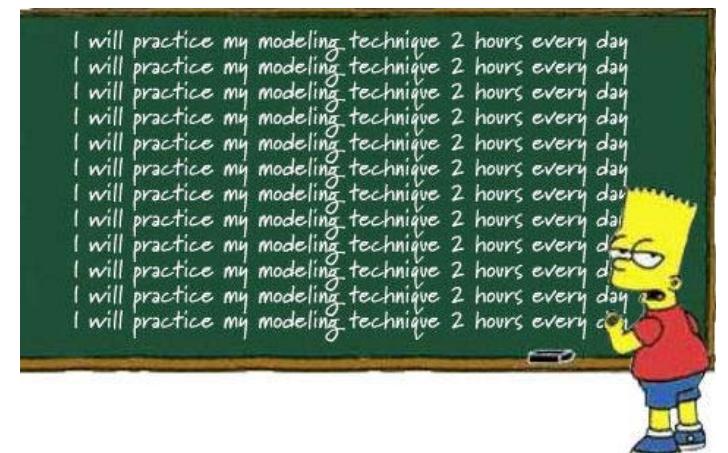
One-way dataflow



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

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 ≥ 25 ;



Official site



<https://ngrx.io/>

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, with a 'Follow' button, and the post was published on Dec 13 · 10 min read. The main title of the article is 'The ugly side of Redux'. The first paragraph discusses Redux at a high level and promotes an alternative called Controllerim. A section titled 'What is Redux' follows, with a quote from the Redux repo defining it as a predictable state container for JavaScript apps. The article ends with a call to action to read the next story, 'React-Native Tutorial #3— Int...'. The URL of the article is <https://medium.com/@niryo/the-ugly-side-of-redux-6591fde68200>.

Applause from John Papa and 26 others

Nir Yosef [Follow](#)
Dec 13 · 10 min read

The ugly side of Redux

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](#).

What is Redux

Let's start from the very beginning and try to define Redux. In the Redux repo you can find this definition:

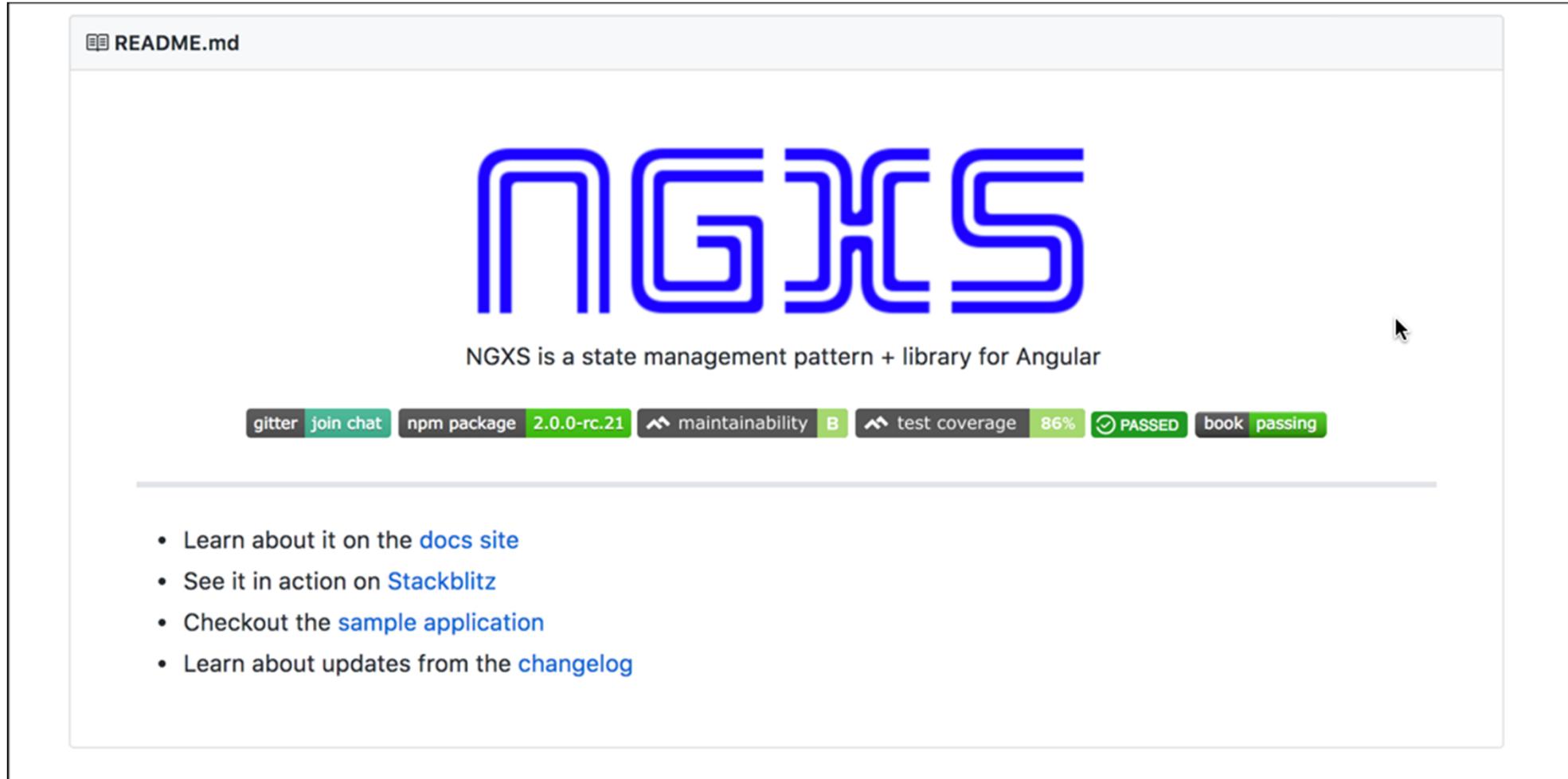
Redux is a predictable state container for JavaScript apps.

Simple isn't it? You just take Redux and put a state inside it, and everything

186 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 green buttons with white text: 'gitter' (with a person icon), 'join chat' (with a speech bubble icon), 'npm package' (with a package icon), '2.0.0-rc.21' (with a checkmark icon), 'maintainability' (with a chart icon and a 'B' grade), 'test coverage' (with a chart icon and '86%'), 'PASSED' (with a checkmark icon), and 'book' (with a book icon) and 'passing' (with a checkmark icon). 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

The screenshot shows a blog post on the netbasal.com website. At the top left are user profile icons for 'M' and 'Netanel Basal'. A 'Follow' button is also present. On the right side are search, notification, and user profile icons. The main title of the post is '🚀 Introducing Akita: A New State Management Pattern for Angular Applications'. Below the title is a small circular profile picture of Netanel Basal and the author's name. The date 'Jun 12, 2018 · 4 min read' is also visible. The central image of the post is a chalkboard with the text 'HELLO MY NAME IS' written in white chalk.

<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

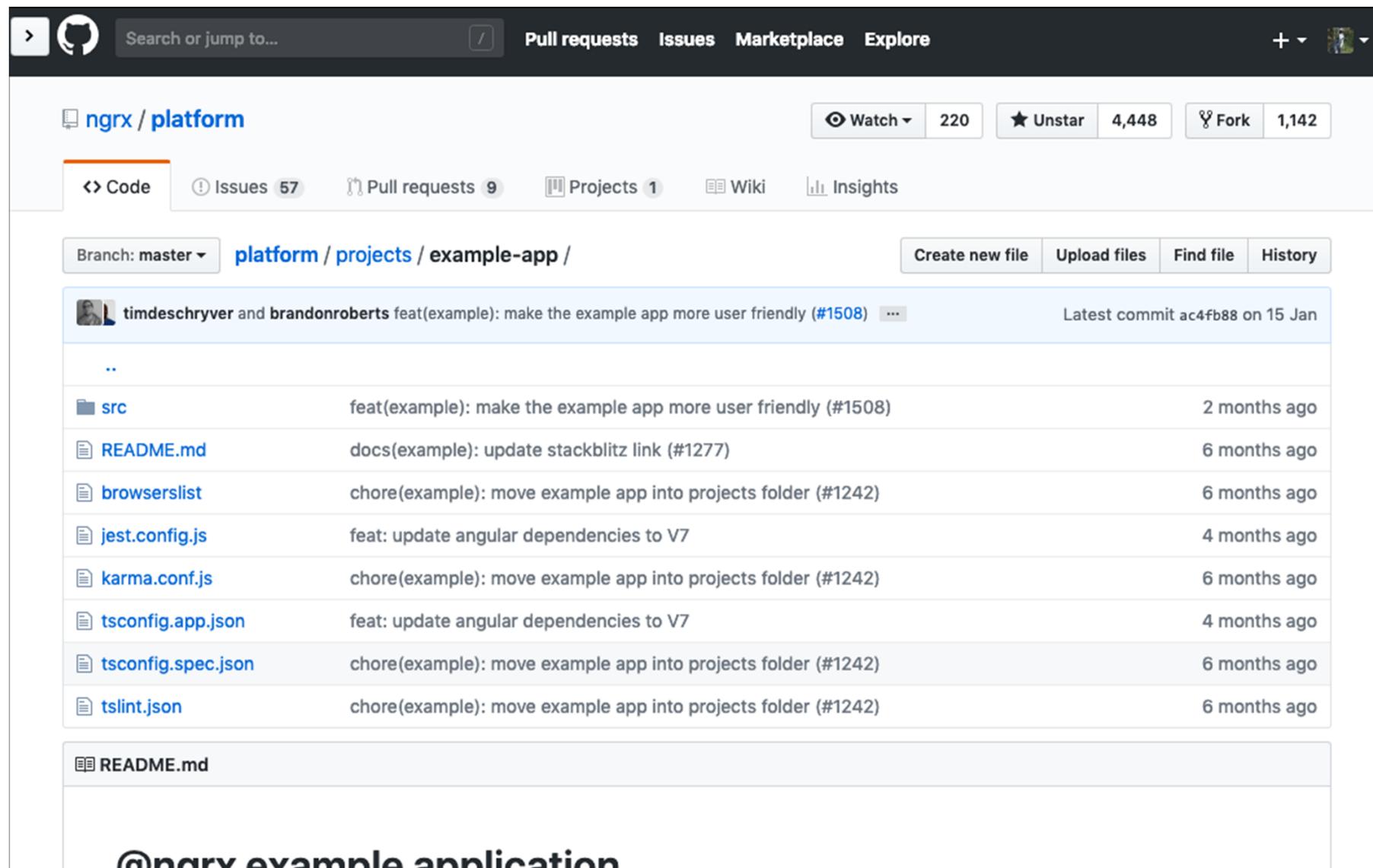
<https://ngrx.io/docs>



Sample Store apps

Some study material

Ngrx store platform sample app



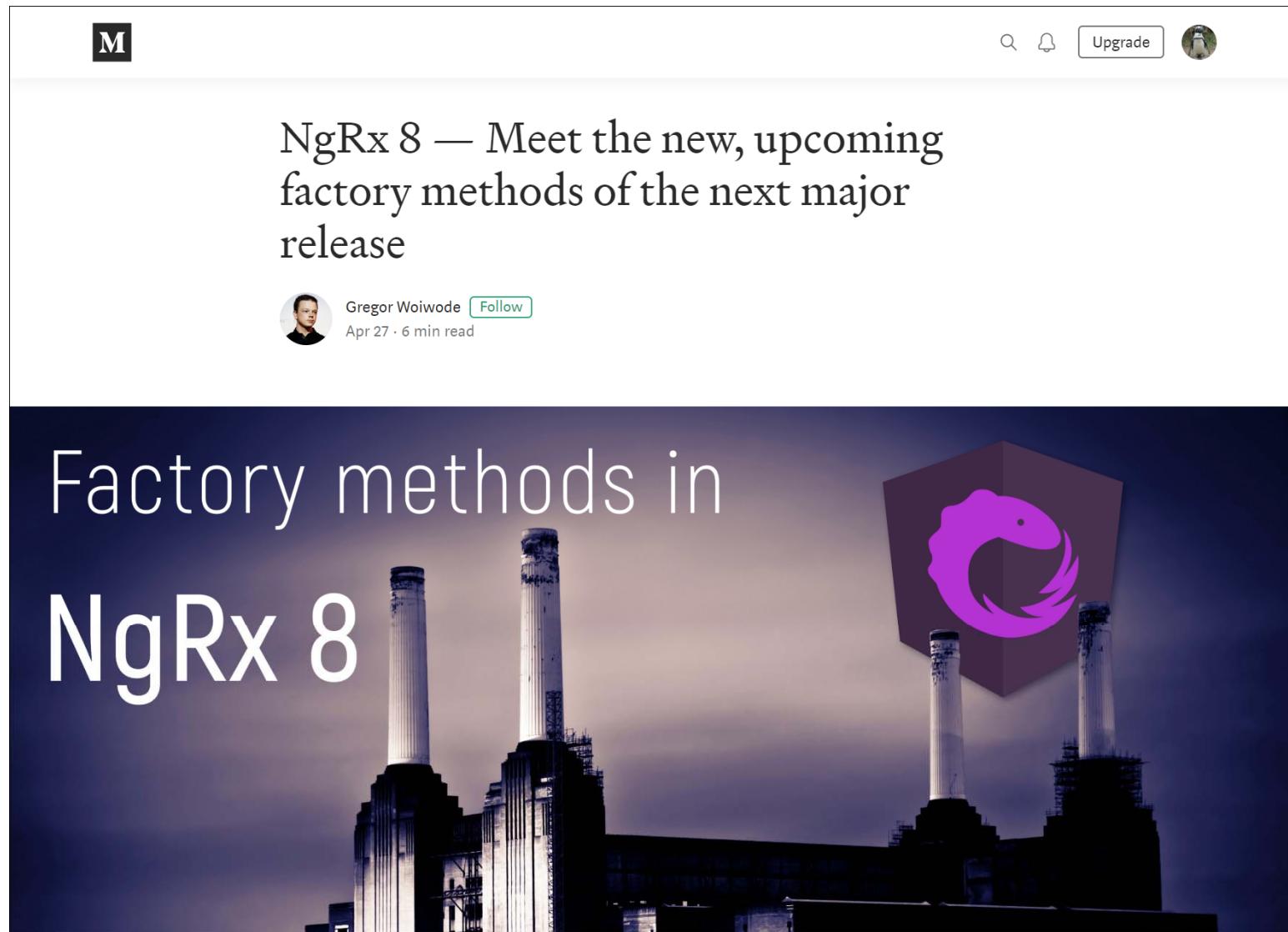
The screenshot shows the GitHub repository page for `ngrx/platform`. The repository has 220 stars and 1,142 forks. The `Code` tab is selected, showing the `platform/projects/example-app` directory. The commit history for this directory is displayed, all made by `timdeschryver` and `brandonroberts`. The commits are:

- feat(example): make the example app more user friendly (#1508) - 2 months ago
- docs(example): update stackblitz link (#1277) - 6 months ago
- chore(example): move example app into projects folder (#1242) - 6 months ago
- feat: update angular dependencies to V7 - 4 months ago
- chore(example): move example app into projects folder (#1242) - 6 months ago
- feat: update angular dependencies to V7 - 4 months ago
- chore(example): move example app into projects folder (#1242) - 6 months ago
- chore(example): move example app into projects folder (#1242) - 6 months ago

At the bottom of the commit list, there is a note: `@ngrx example application`.

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

More info



A screenshot of a Medium article page. At the top left is a black square icon with a white letter 'M'. To its right are search, notification, and upgrade buttons, followed by 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: a circular 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. A purple cube with the NgRx logo (a white circular swirl) is positioned in front of the chimneys.

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