Reactive Angular met RxJS @ngrx/store – Feature Modules

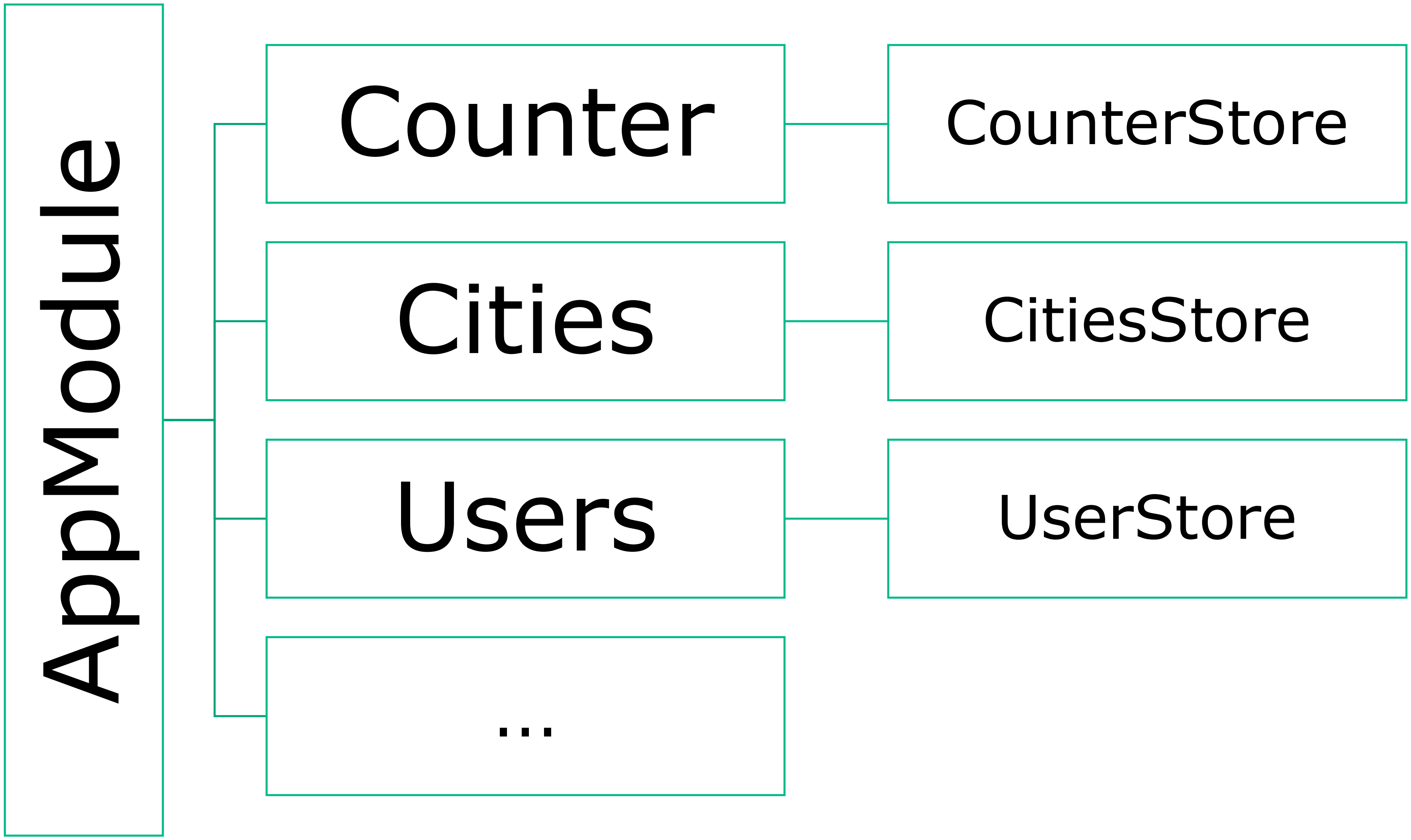
Peter Kassenaar – info@kassenaar.com

Using feature modules

- As we know, it is better practice to separate the logic of the app into feature modules
- Eeach module is responsible for *it’s own slice* of the store
- Use `StoreModule.forFeature(‘name’, reducer)` in the modules
- In `app.module.ts` use `StoreModule.forRoot({}).`
  - Initialize the root store with an empty object
- We can use lazy loading with feature modules!
- The module store is added to the main store as soon as the module gets loaded

Boilerplate / setup

1. Create the main application as usual
2. Set up / initialize :
   a) Feature modules,
   b) Routing,
   c) Lazy loading – as usual
3. In `app.module.ts` initialize the store with an empty reducers object
4. Initialize the store with an empty Effects array (if you're using `@ngrx/effects`)
5. Set up `StoreDevTools` module as usual


AppModule
Counter
Cities
Users
...
CounterStore
CitiesStore
UserStore

Example `app.module.ts` – empty store!

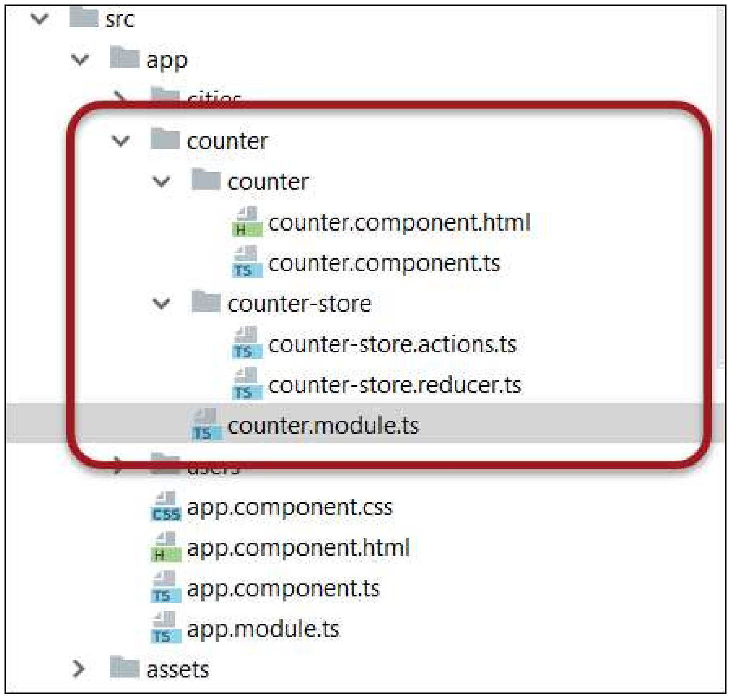
```
// app.module.ts
import …;

export const routes: Routes = [
  { path: '', redirectTo: 'counter', pathMatch: 'full' },
  { path: 'counter', loadChildren: './counter/counter.module#CounterModule' },
   …
];

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    StoreModule.forRoot({}),
    EffectsModule.forRoot([]),
    // disable StoreDevTools in production
    !environment.production
      ? StoreDevtoolsModule.instrument({ maxAge: 20 })
      : [],
    RouterModule.forRoot(routes)
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Define store-stuff in each module

- Start simple – `CounterModule` (=no external stuff)
- Possible architecture
    - `\counter-store` – holding all actions and reducer files
    - `\counter` – holding the counter component
    - `\<components>` - holding the other components
- `counter.module.ts`
    - configuring the module with child routes
    - Use to add specific slice to the store



Counter.module.ts

```
// counter.module.ts
…
import { CounterComponent } from './counter/counter.component';
import { RouterModule, Routes } from '@angular/router';
import { StoreModule } from '@ngrx/store';
import { counterReducer } from './counter-store/counter-store.reducer';

const routes: Routes = [{ path: '', component: CounterComponent }];

@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild(routes),
    StoreModule.forFeature('counter', counterReducer)
  ],
  declarations: [CounterComponen
})
export class CounterModule {}
```

Define the *name* of the slice for this module on the complete store. In this case `counter`

Actions and Reducer

- Nothing special – as in previous examples

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

// *** Action constants
// These are the strings for the action
export const INCREMENT = '[COUNTER] - increment';
export const DECREMENT = '[COUNTER] - decrement';
export const RESET = '[COUN

// *** Action Creators.
export class CounterIncreme
  readonly type = INCREMENT
  constructor(public payloa
}
```

```
// counter-store.reducer.ts
import * as fromActions from './counter-store.actions';

// create initial State.
export const initialState = 0;

export interface CounterState {
  counter: number;
}

export function counterReducer(state = initialState,
  action: fromActions.CounterAction
) {
  switch (action.type) {
    case fromActions.INCREMENT:
```

The Cities Feature Store

- This is a store, much as the previous examples.
- Only the State interface is now more complex, as it has multiple properties.
  - Also the complete, composed state has potentially *a lot* of levels
- This means later on we have the need for so called *State Selectors*

```
export interface CityState {
  cities: City[];
  loading: boolean;
  loaded: boolean; // and so on…
}
```

Creating State selectors

```
// city-store.reducer.ts

// Here, we create a *state selector*. Otherwise, the
// complete state (including the 'loading' flag)
// would be returned upon selection. So we're creating a
// function here that takes an
// object of type CityState and returns the .cities property
// from that object.
// The function is called in cities.store.ts. Look up that file!
export const getCitiesEntities = (state: CityState) => state.cities;
```

- Followed by: use the `store.createFeatureSelector()` to return specific slices of the state

```
// cities.store.ts
// Create a very specific selector that tells ngrx
// how to get a hold of this particular feature.

import * as fromCityStore from './cities-store.reducer';
import { createFeatureSelector, createSelector } from '@ngrx/store';

// 1. What this line tells the store, is that it should find a
// State of type CityState on the property 'cities',
export const getCityFeatureState = createFeatureSelector<
   fromCityStore.CityState>('cities');

// 2. Now, for the actual selector that will return our cities
// we can use this featureSelector, to drill into our CityState
// (my head exploded when first trying to comprehend this, PK):
export const getCityEntities = createSelector(
   getCityFeatureState,
   fromCityStore.getCitiesEntities
);
```

Now look in the cities.component.ts and see how this getCityEntities is used.

cities.component.ts

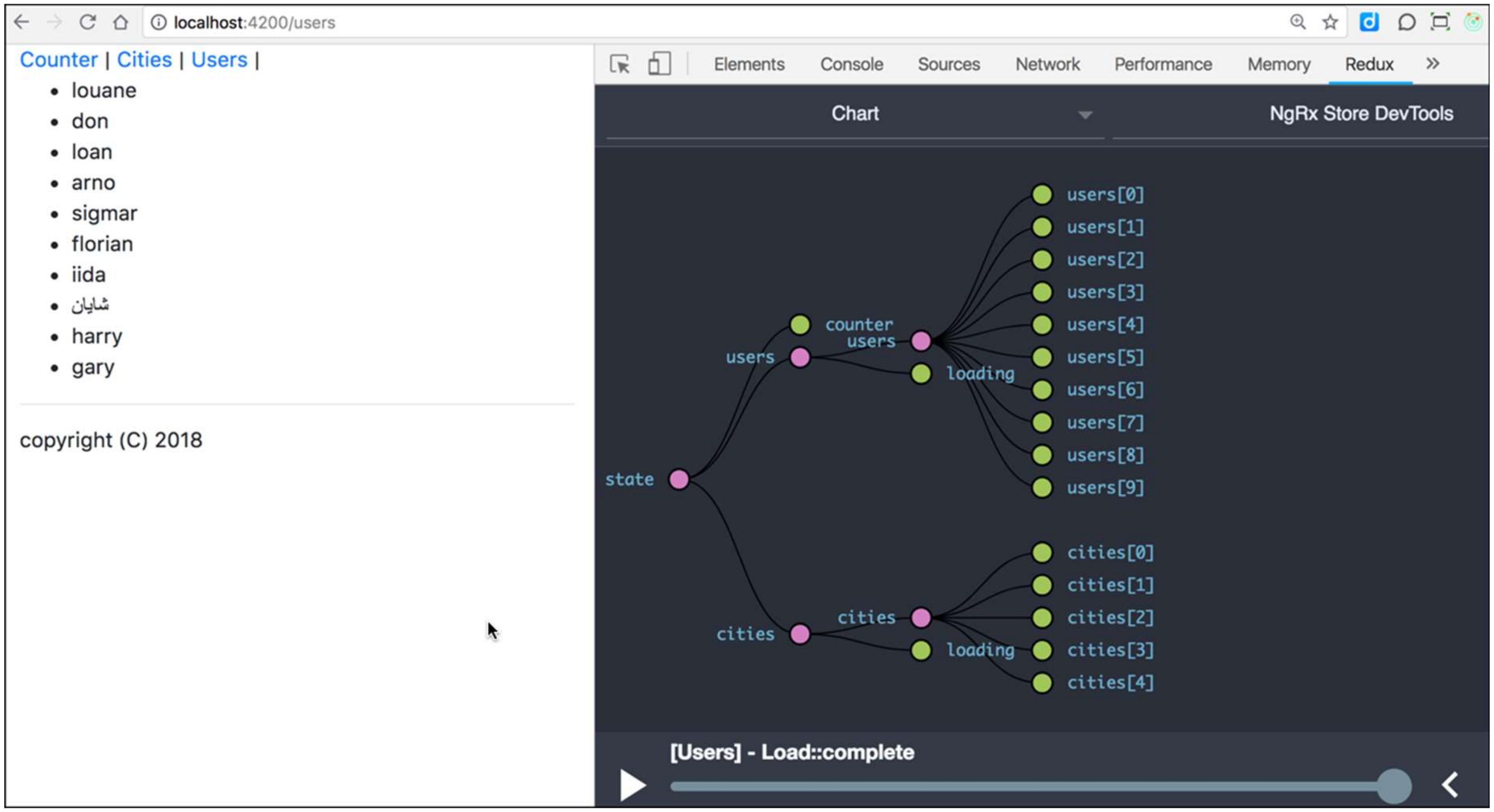
```
ngOnInit() {
  this.store.dispatch(new fromCityActions.LoadCities());
  // Here, we use the feature selector to select specific
  // slices of the complete State.
  this.cities$ =
    this.store.select(fromCityStore.getCityEntities);
}
```

Remember, you only have to do this if your store/state is a complex object with possible multi-level deep nesting of properties.

Use the Redux DevTools to inspect the Store/State

The Users Feature Store

- Same procedure
- TBC…



localhost:4200/users

Counter | Cities | Users |

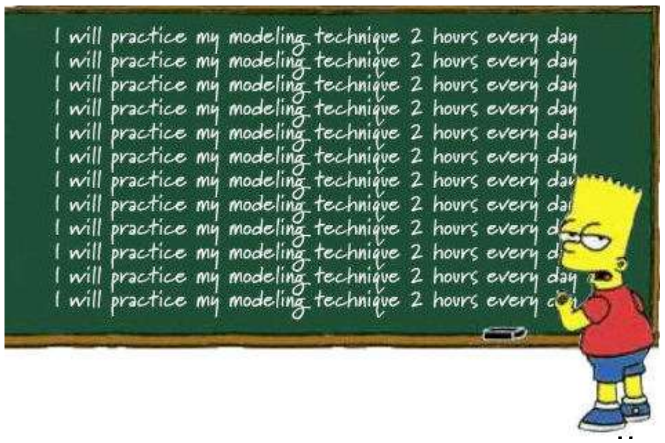
- louane
- don
- loan
- arno
- sigmar
- florian
- iida
- شایان
- harry
- gary

copyright (C) 2018

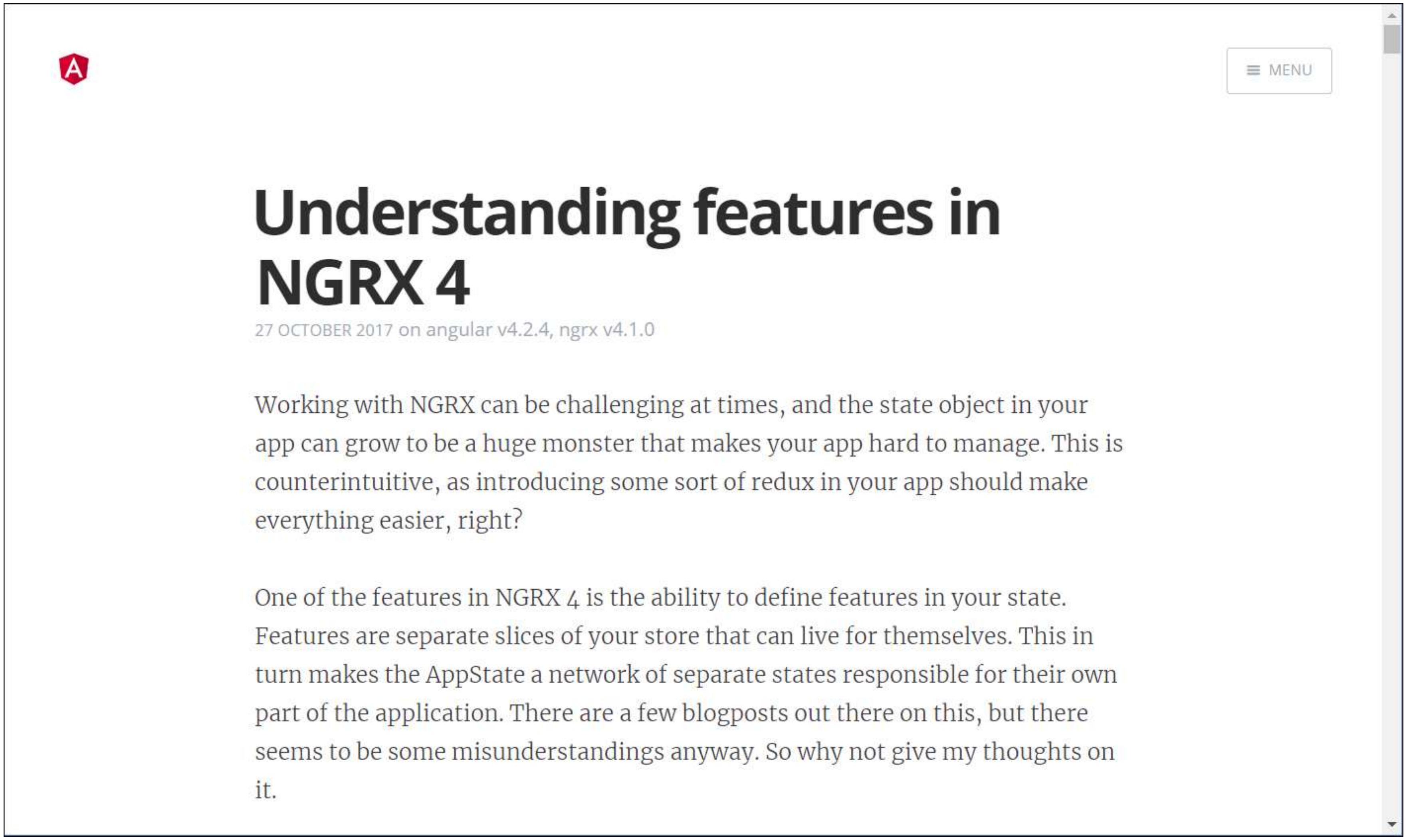
Workshop

- Use your own project, or start from the Store project `/240-ngrx-store-feature-modules`
- Add a new module to the store
- Create a basic Feature store for this module, displaying some date (for instance a number or a name). See `/counter-store` as example

- OR – study the example project using the Store DevTools and see the data flow in the example. Can you:
  - Create a service and communicate thus with `http`?
  - Call only `http` if the store/cache has no data yet?
  - Add additional `http`-endpoints to put data in (one of) the stores?



More info



Understanding features in NGRX 4

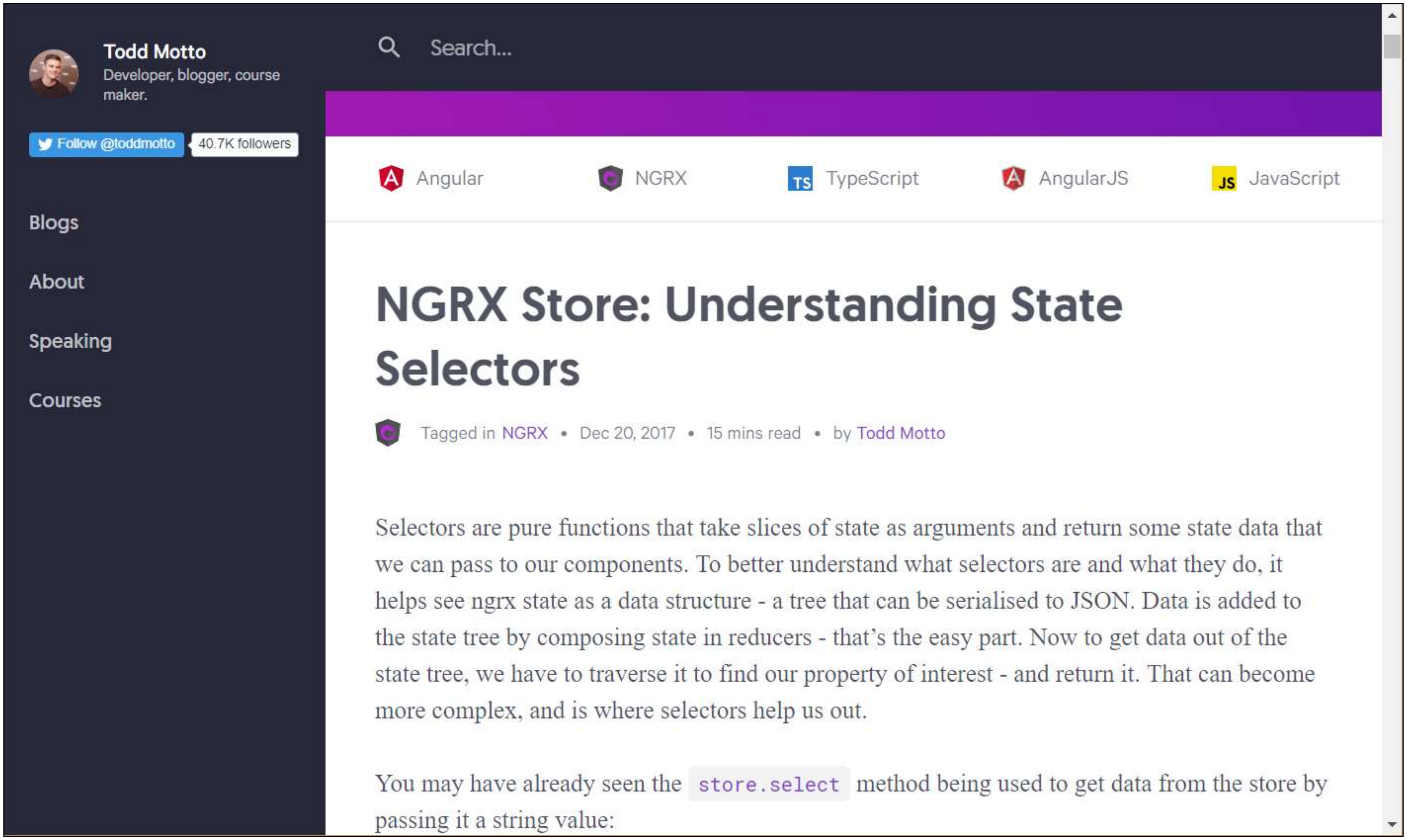
27 OCTOBER 2017 on angular v4.2.4, ngrx v4.1.0

Working with NGRX can be challenging at times, and the state object in your app can grow to be a huge monster that makes your app hard to manage. This is counterintuitive, as introducing some sort of redux in your app should make everything easier, right?

One of the features in NGRX 4 is the ability to define features in your state. Features are separate slices of your store that can live for themselves. This in turn makes the AppState a network of separate states responsible for their own part of the application. There are a few blogposts out there on this, but there seems to be some misunderstandings anyway. So why not give my thoughts on it.

http://ngxsolutions.azurewebsites.net/understanding-features-in-ngrx-4/

NGRX Store: Understanding State Selectors



NGRX Store: Understanding State Selectors

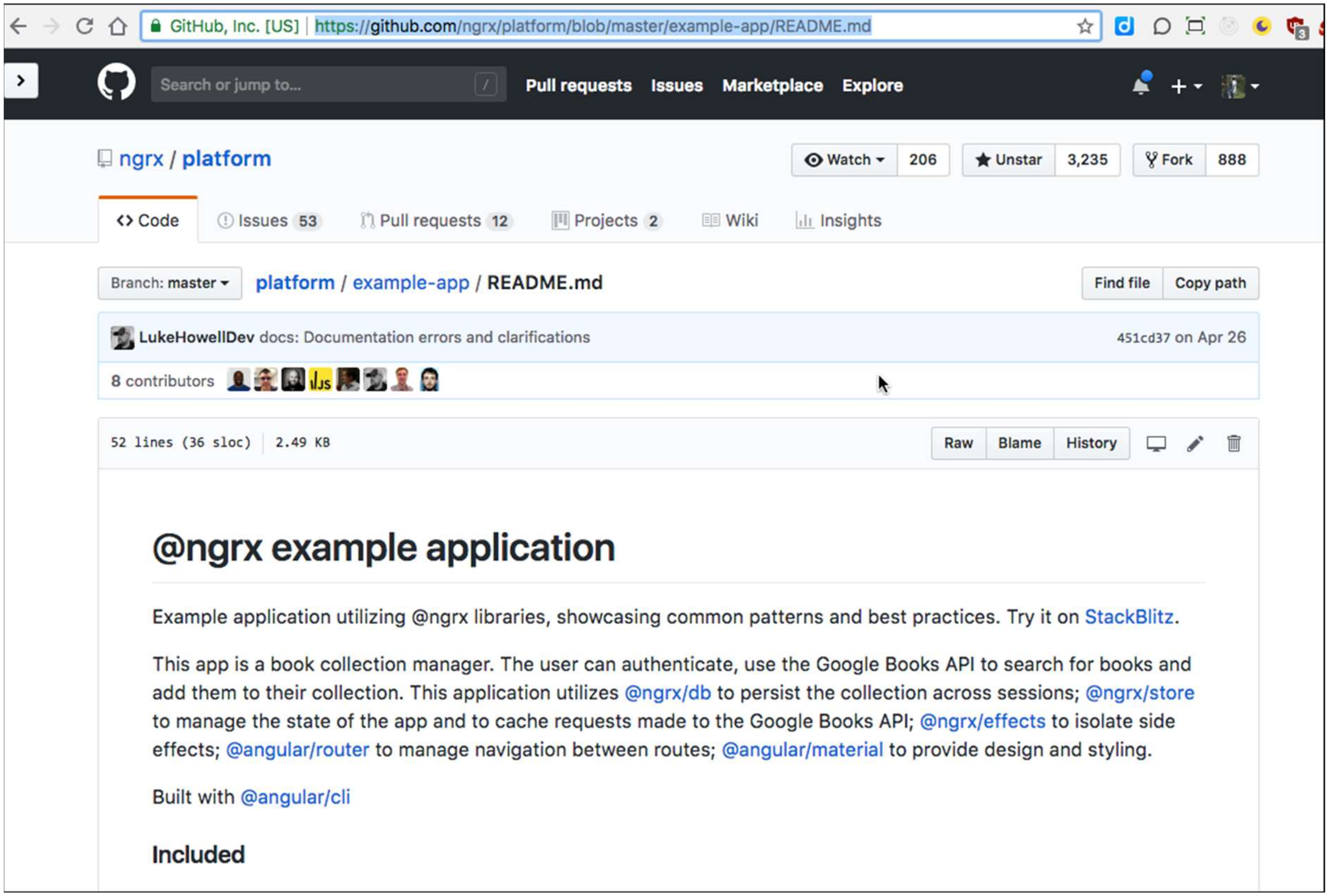
Tagged in NGRX • Dec 20, 2017 • 15 mins read • by Todd Motto

Selectors are pure functions that take slices of state as arguments and return some state data that we can pass to our components. To better understand what selectors are and what they do, it helps see ngrx state as a data structure - a tree that can be serialised to JSON. Data is added to the state tree by composing state in reducers - that's the easy part. Now to get data out of the state tree, we have to traverse it to find our property of interest - and return it. That can become more complex, and is where selectors help us out.

You may have already seen the `store.select` method being used to get data from the store by passing it a string value:

https://toddmotto.com/ngrx-store-understanding-state-selectors

Example Application



platform / example-app / README.md

LukeHowellDev docs: Documentation errors and clarifications 451cd37 on Apr 26

8 contributors

52 lines (36 sloc) | 2.49 KB

@ngrx example application

Example application utilizing @ngrx libraries, showcasing common patterns and best practices. Try it on StackBlitz.

This app is a book collection manager. The user can authenticate, use the Google Books API to search for books and add them to their collection. This application utilizes @ngrx/db to persist the collection across sessions; @ngrx/store to manage the state of the app and to cache requests made to the Google Books API; @ngrx/effects to isolate side effects; @angular/router to manage navigation between routes; @angular/material to provide design and styling.

Built with @angular/cli

Included

https://github.com/ngrx/platform/blob/master/example-app/README.md