

State Management with Redux und @ngrx/store

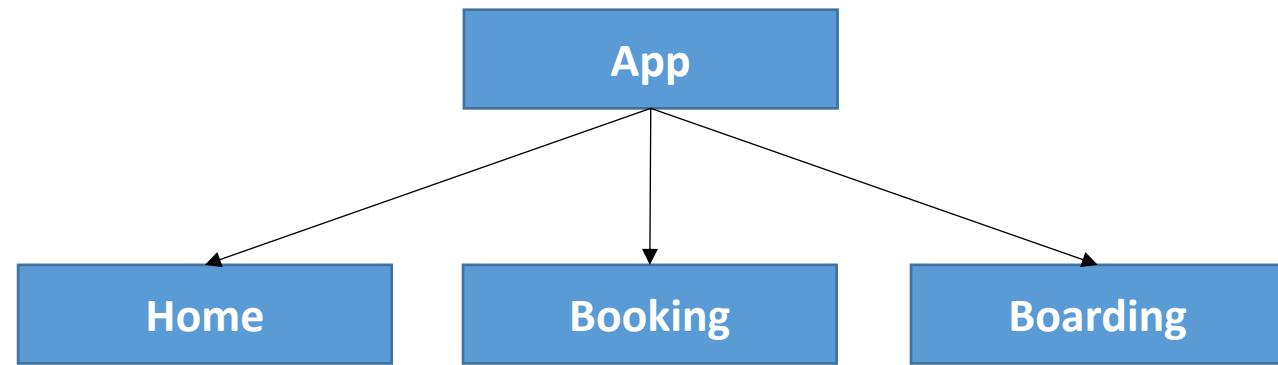
Alex Thalhammer

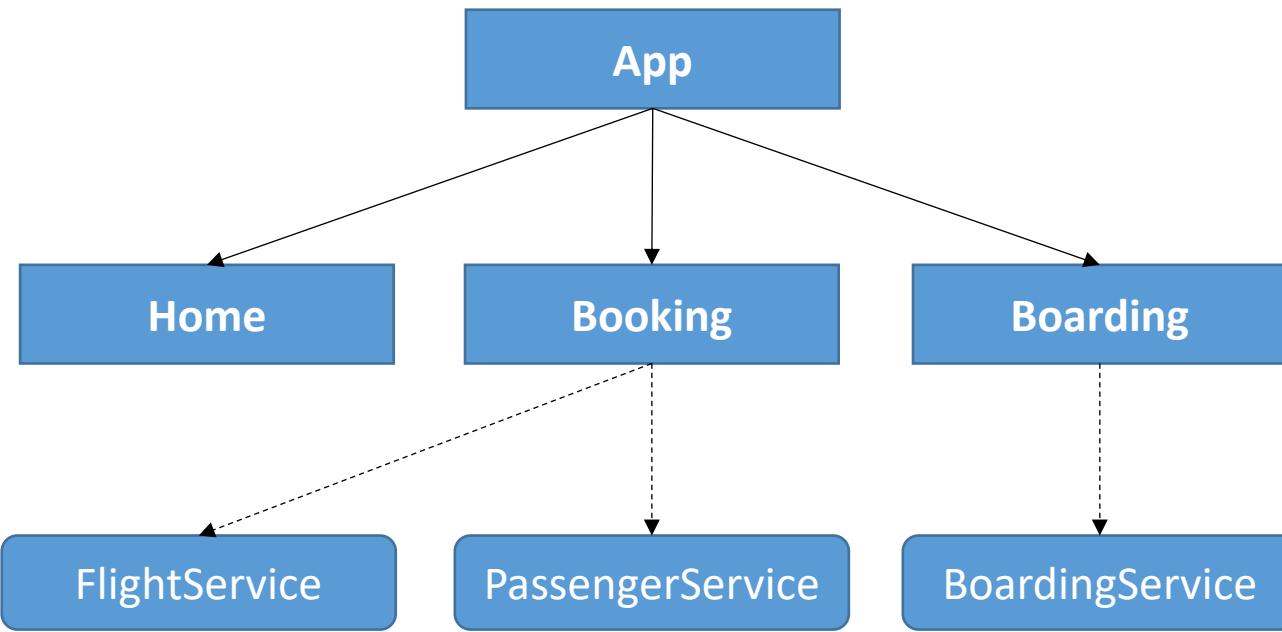
Contents

- Motivation
- State
- Actions
- Reducer
- Store
- Immutables
- Effects
- Labs / Demo

Motivation



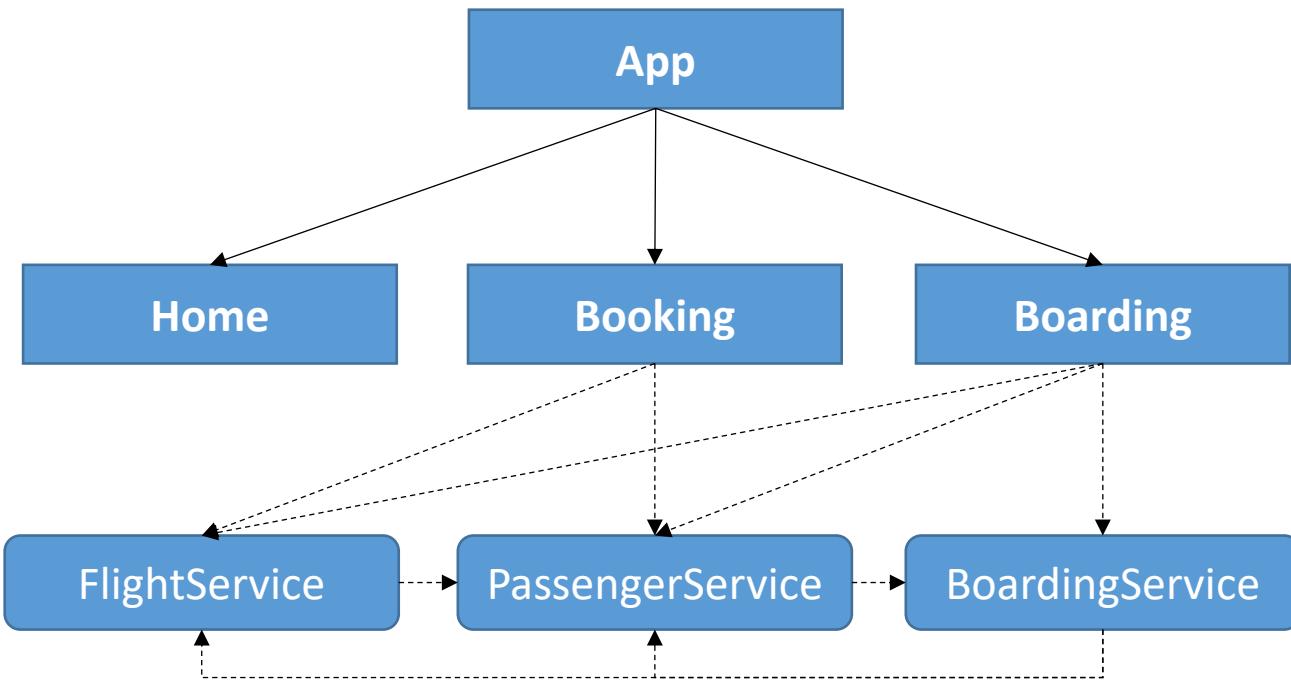




ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



SOFTWARE
ARCHITECT



ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



SOFTWARE
ARCHITECT

Redux

- Redux makes complex UI manageable
- Origin: React Ecosystem
- Implementation used here: `@ngrx/store`

npm install @ngrx/store --save

Alternatives

 npm trends

@ngrx/store vs @ngxs/store

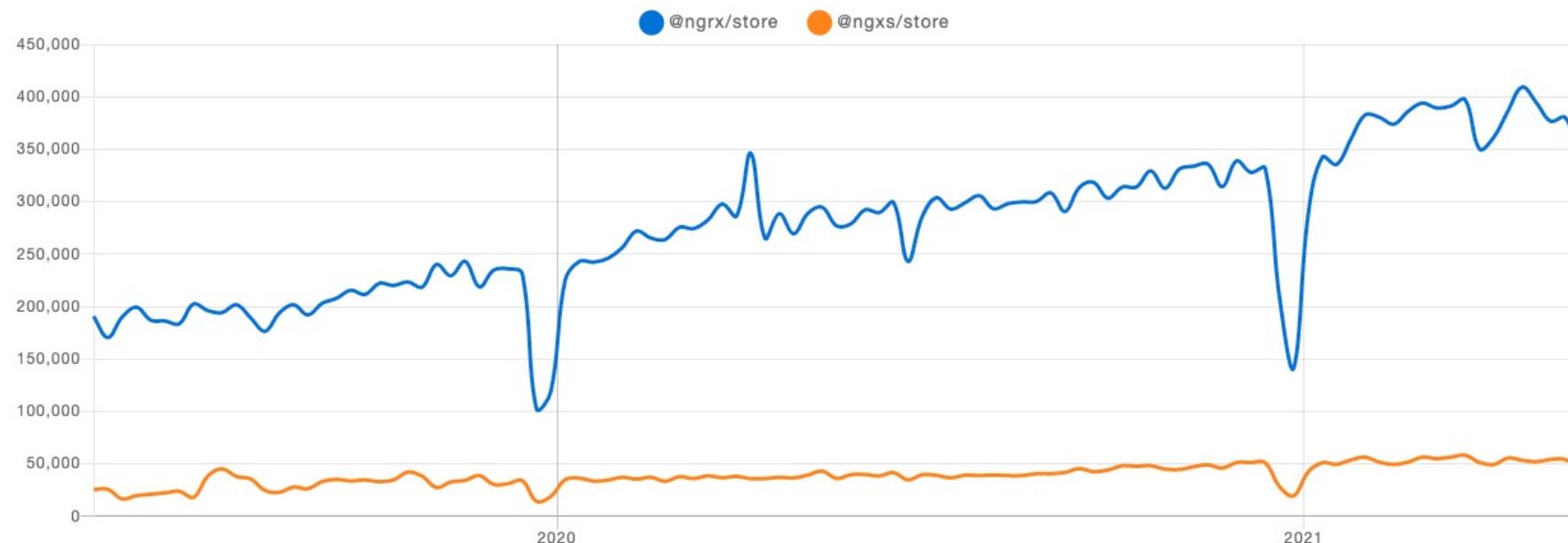
Enter an npm package...

@ngrx/store x

@ngxs/store x

+ @angular-redux/store + @datorama/akita + ngxs + akita + mobx

Downloads in past 2 Years ▾



Alternatives



@ngrx/store vs @ngxs/store vs mobx

Enter an npm package...

@ngrx/store x

@ngxs/store x

mobx x

+ redux

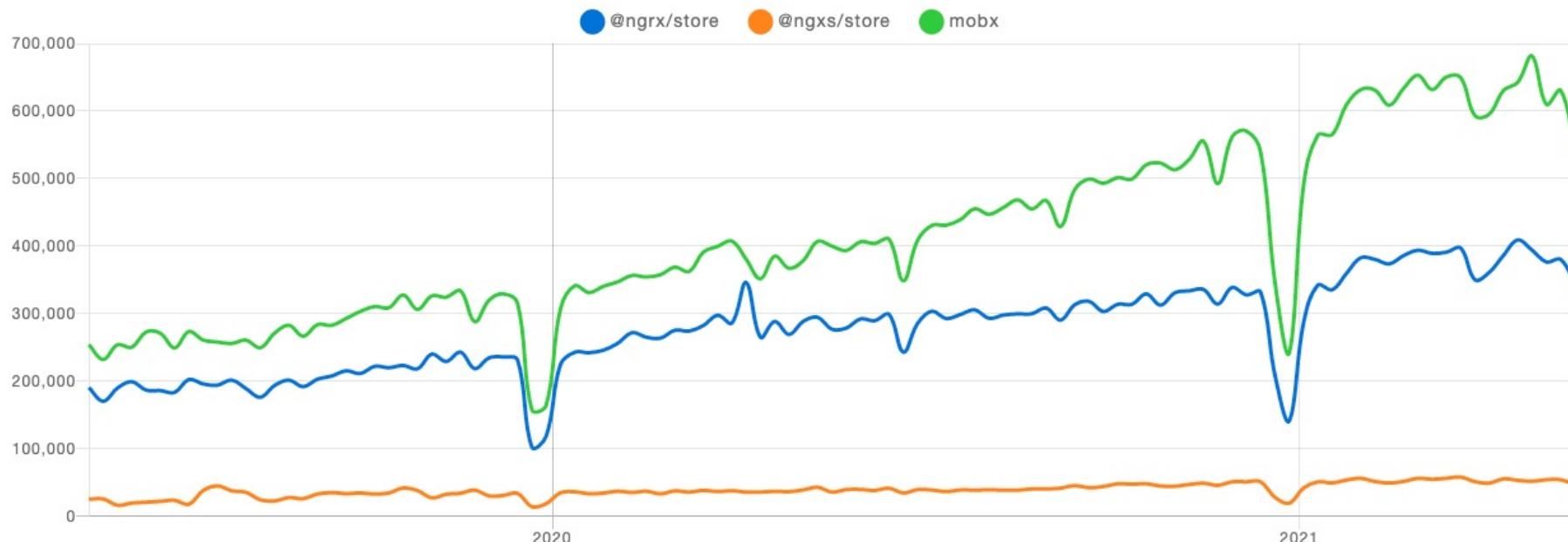
+ @angular-redux/store

+ @datorama/akita

+ react-redux

+ ngxs

Downloads in past 2 Years ▾



Alternatives

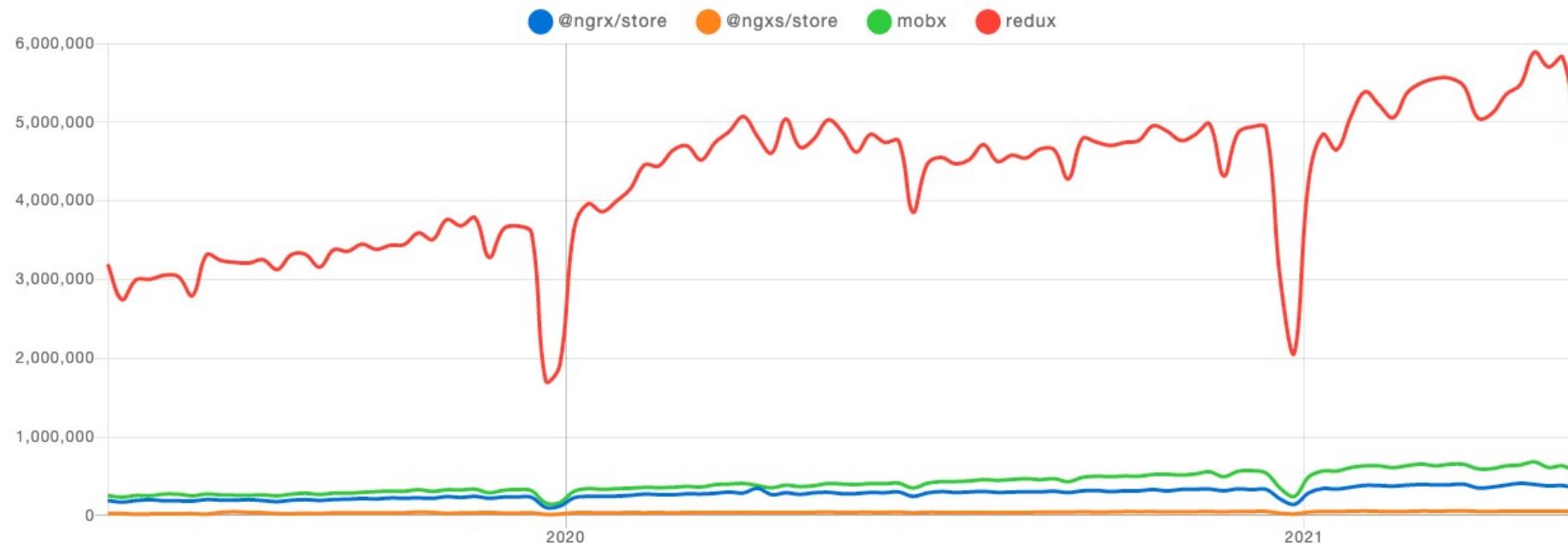


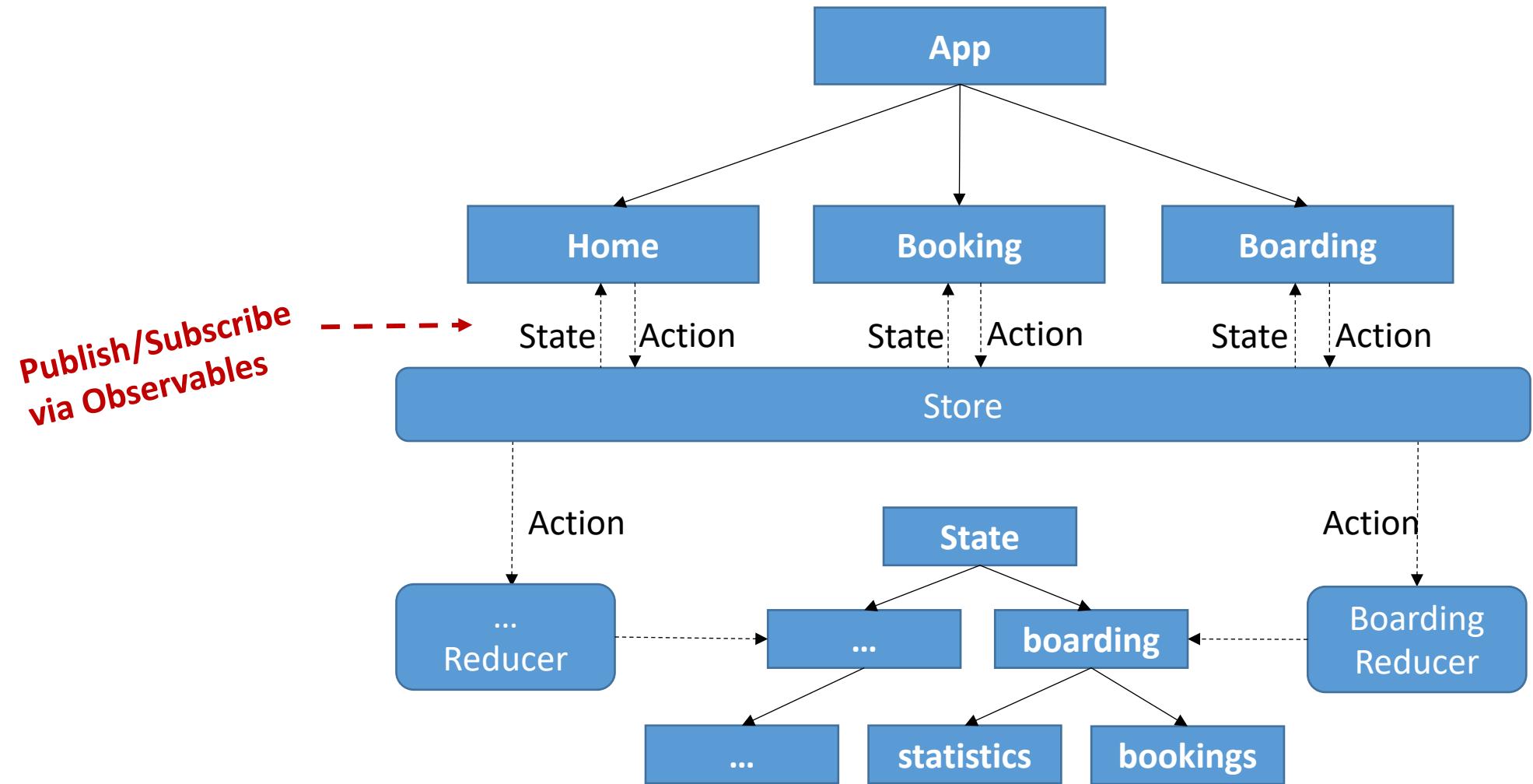
@ngrx/store vs @ngxs/store vs mobx vs redux

Enter an npm package...

@ngrx/store × @ngxs/store × mobx × redux × + @angular-redux/store + @datorama/akita + react-redux + react + ngxs

Downloads in past 2 Years ▾





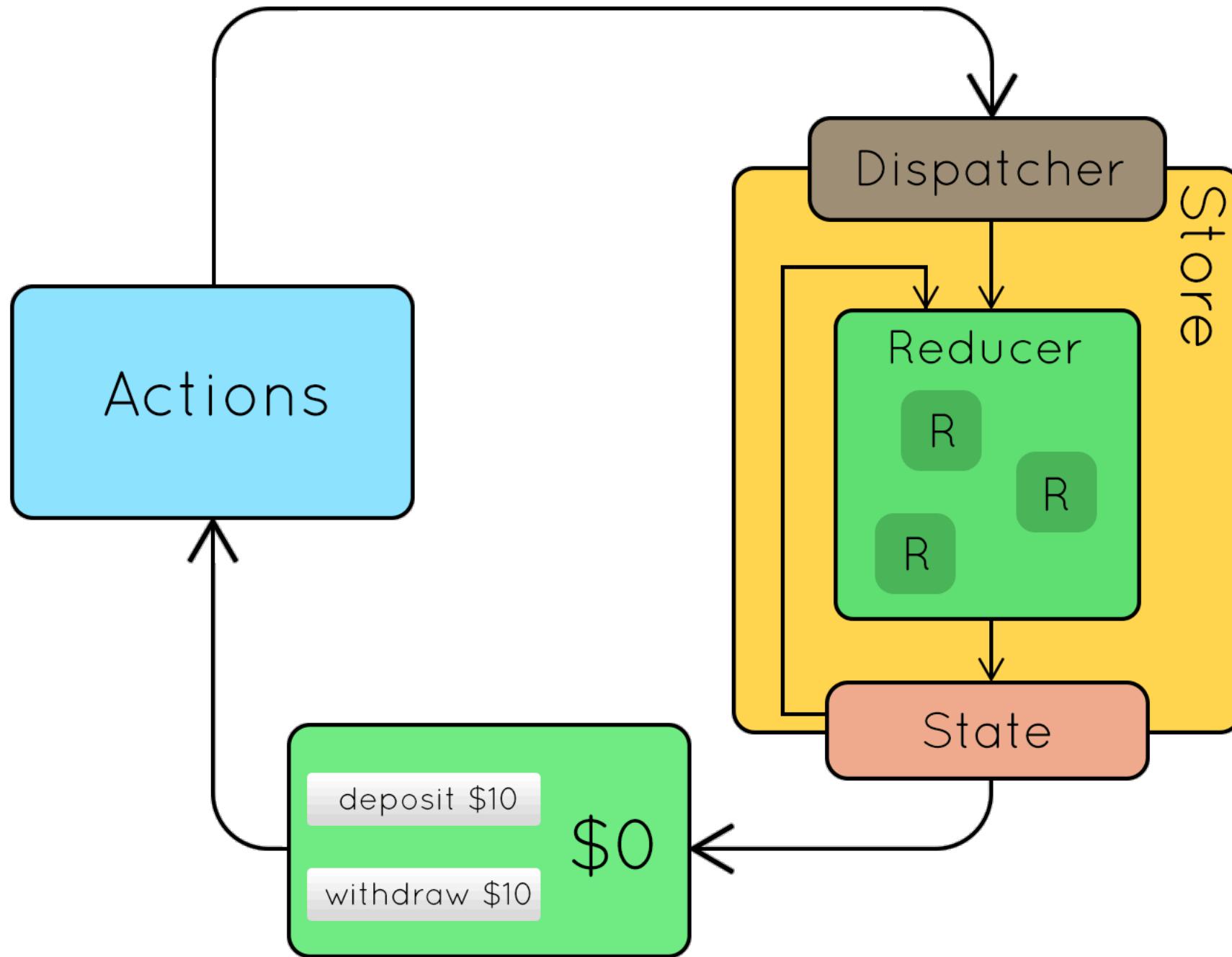
Single Immutable State Tree



ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



SOFTWARE
ARCHITECT



State



State

```
export interface FlightBookingState {  
  flights: Flight[];  
  statistics: FlightStatistics;  
  basket: object;  
}
```

State

```
export interface FlightBookingState {  
  flights: Flight[];  
  statistics: FlightStatistics;  
}  
  
export interface FlightStatistics {  
  countDelayed: number;  
  countInTime: number;  
}
```

AppState

```
export interface AppState {  
  flightBooking: FlightBookingState;  
  currentUser: UserState;  
}
```



Actions

Parts of an Action

- Type
- Payload

Defining an Action

```
export const flightsLoaded = createAction(  
  '[FlightBooking] FlightsLoaded',  
  props<{flights: Flight[]}>()  
);
```

Reducer



Reducer

- Function that executes Action
- Pure function (stateless, etc.)
- Each Reducer gets each Action
 - Check whether Action is relevant
 - This prevents cycles

Reducer

(currentState, action) => newState

Reducer for FlightBookingState

```
export const flightBookingReducer = createReducer(  
    initialState,  
  
    on(flightsLoaded, (state, action) => {  
        const flights = action.flights;  
        return { ...state, flights };  
    })  
)
```

Map Reducers to State Tree

```
const reducers = {
  "flightBooking": flightBookingReducer,
  "currentUser": authReducer
}
```

A close-up photograph of several wooden barrels stacked together. The barrels are made of dark wood and have metal bands around them. Some have circular holes. The lighting is dramatic, highlighting the texture of the wood and the metallic sheen of the bands.

Store

Store

- Manages state tree
- Allows to read state (via Selectors / Observables)
- Allows to modify state by dispatching actions

Selectors

- Selectors are pure functions used for obtaining slices of store state
- `select(tree => tree.flightBooking.flights): Observable<Flight[]>`

Actions

- Actions express *events* that happen throughout your application
- `dispatch(flightsLoaded({ flights }))`

Registering @ngrx/store



Registering @ngrx/Store

```
@NgModule({
  imports: [
    ...
    StoreModule.forRoot(reducers)
  ],
  ...
})
export class AppModule { }
```

Registering @ngrx/Store

```
@NgModule({
  imports: [
    [...]
    StoreModule.forRoot(reducers),
    !environment.production ? StoreDevtoolsModule.instrument() : []
  ],
  [...]
})
export class AppModule { }
```

@ngrx/store-devtools

A Curiosity rover is shown on the surface of Mars, performing a wheelie maneuver. The rover's body is tilted, with its front wheels lifted off the ground. A bright yellow glow is visible at the point where the front wheels were in contact with the reddish-brown Martian soil. The background shows the vast, arid landscape of Mars under a hazy orange sky.

ngrx and Feature Modules

Reducers

- Reducers are responsible for handling transitions from one state to the next state in your application
- Receive Actions

Reducers for Shared State

```
const reducers = {
  flightBooking: flightBookingReducer,
  currentUser: authReducer
}
```

Reducers for Shared State

```
const reducers = {
  flightBooking: flightBookingReducer,
  currentUser: authReducer
}
```

Registering @ngrx/Store

```
@NgModule({
  imports: [
    ...
    StoreModule.forFeature('flightBooking', flightBookingReducer)
  ],
  ...
})
export class FlightBookingModule { }
```

Lab

NgRx Store & Selectors

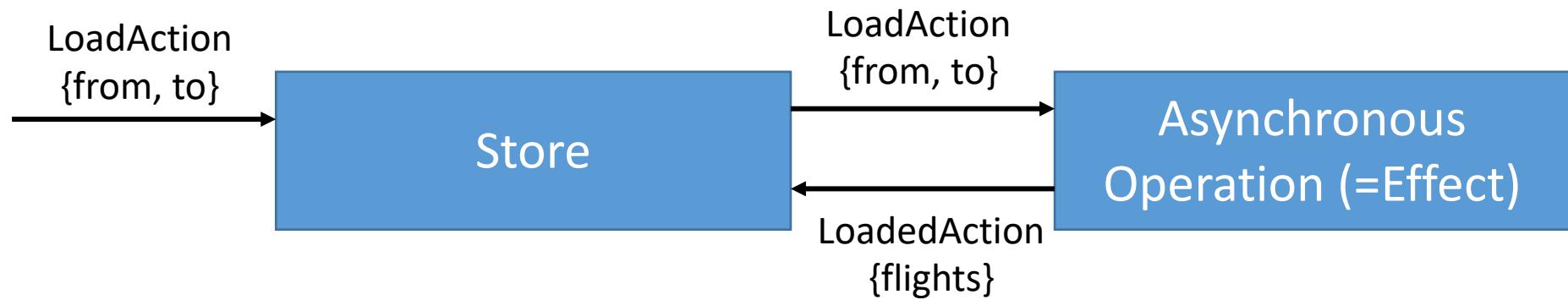
Effects



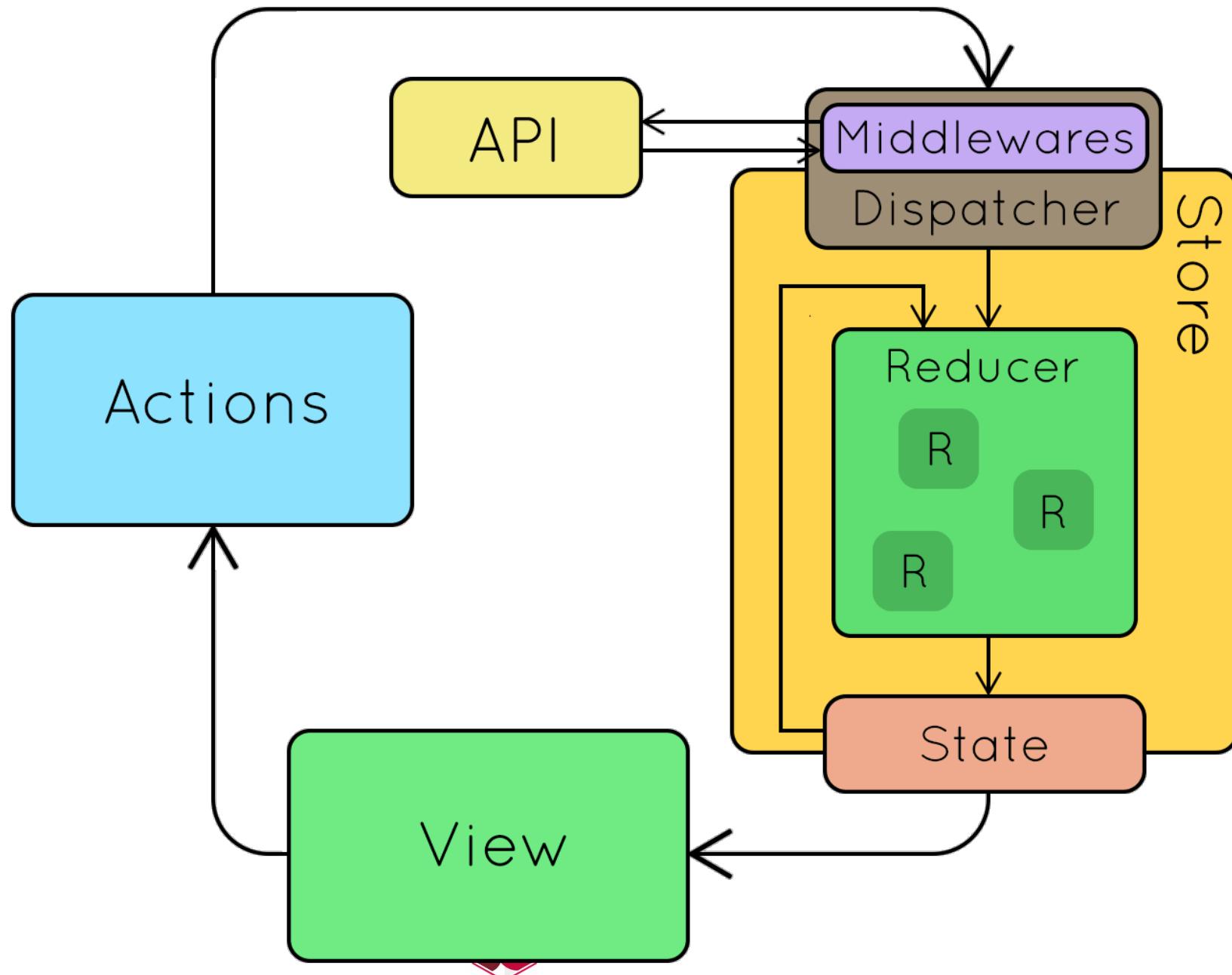
Challenge

- Reducers are synchronous by definition
- What to do with asynchronous operations?

Solution: Effects



ng add @ngrx/effects



Effects are Observables



Implementing Effects

```
@Injectable()  
export class FlightBookingEffects {  
  
    [...]  
  
}
```

Implementing Effects

```
@Injectable()
export class FlightBookingEffects {

  constructor(
    private flightService: FlightService, private actions$: Actions) {
  }

  [...]

}
```

Implementing Effects

```
@Injectable()
export class FlightBookingEffects {

    constructor(
        private flightService: FlightService, private actions$: Actions) {
    }

    myEffect$ = createEffect(() => this.actions$.pipe(
        ofType(loadFlights));
}
```

Implementing Effects

```
@Injectable()
export class FlightBookingEffects {

    constructor(
        private flightService: FlightService, private actions$: Actions) {
    }

    myEffect$ = createEffect(() => this.actions$.pipe(
        ofType(loadFlights),
        switchMap(a => this.flightService.find(a.from, a.to, a.urgent)));
}
```

Implementing Effects

```
@Injectable()
export class FlightBookingEffects {

  constructor(
    private flightService: FlightService, private actions$: Actions) {
  }

  myEffect$ = createEffect(() => this.actions$.pipe(
    ofType(loadFlights),
    switchMap(a => this.flightService.find(a.from, a.to, a.urgent)),
    map(flights => flightsLoaded({flights})));
}

}
```

Implementing Effects

```
@NgModule({
  imports: [
    StoreModule.provideStore(appReducer, initialState),
    EffectsModule.forRoot([SharedEffects]),
    StoreDevtoolsModule.instrument()
  ],
  [...]
})
export class AppModule { }
```

Implementing Effects

```
@NgModule({
  imports: [
    [...]
    EffectsModule.forFeature([FlightBookingEffects])
  ],
  [...]
})
export class FeatureModule {
```

Lab

NgRx Effects

@ngrx/entity and @ngrx/schematics

- ng add @ngrx/entity
- ng add @ngrx/schematics
- ng g module passengers
- ng g entity Passenger --module passengers.module.ts

DEMO



ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



@ngrx/store-devtools

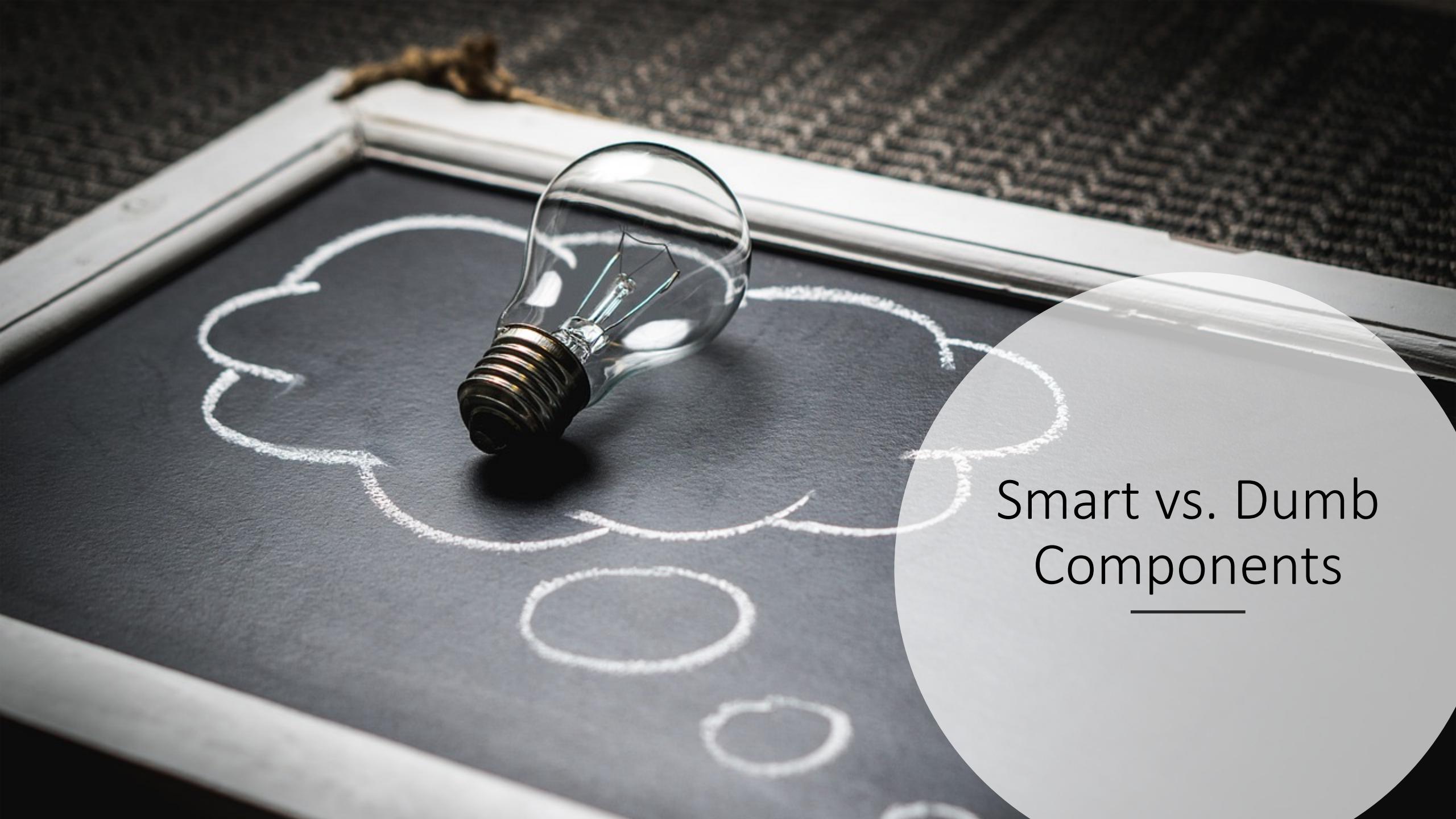
- Add Chrome / Firefox extension to use Store Devtools
 - Works with Redux & NgRx
 - <https://ngrx.io/guide/store-devtools>
- Also available for MobX
 - MobX Developer Tools

DEMO



ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



A photograph of a clear incandescent lightbulb lying on a dark, textured surface. A hand-drawn network diagram in white chalk is visible behind it, consisting of several interconnected circles of varying sizes. A metal key lies horizontally across the top of the board. In the bottom right corner, there is a large, semi-transparent circular overlay containing the text.

Smart vs. Dumb Components

Thought experiment

- What if <flight-card> would directly talk with the store?
 - Querying specific parts of the state
 - Triggering effects
- Traceability?
- Performance?
- Reuse?

Smart vs. Dumb Components

Smart Component

- Drives the "Use Case"
- Usually a "Container"

Dumb

- Independent of Use Case
- Reusable
- Usually a "Leaf"



ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



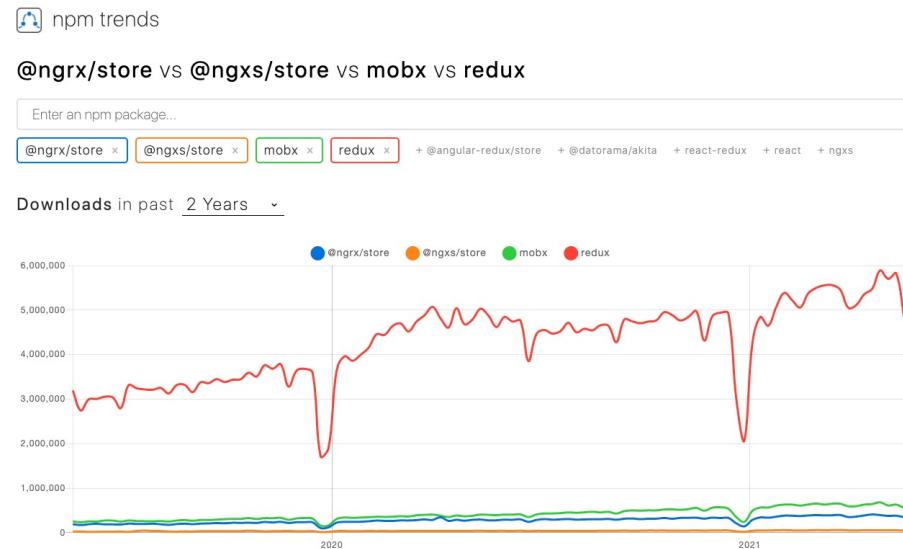
What about Mobx?

I think it's okay to continue using Mobx

What do you think?

Comparison

blog.logrocket.com/redux-vs-mobx/



ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



SOFTWARE
ARCHITECT

Like this topic?

Check out the NgRx Guide

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