

Basic State Management with Redux and @ngrx/store

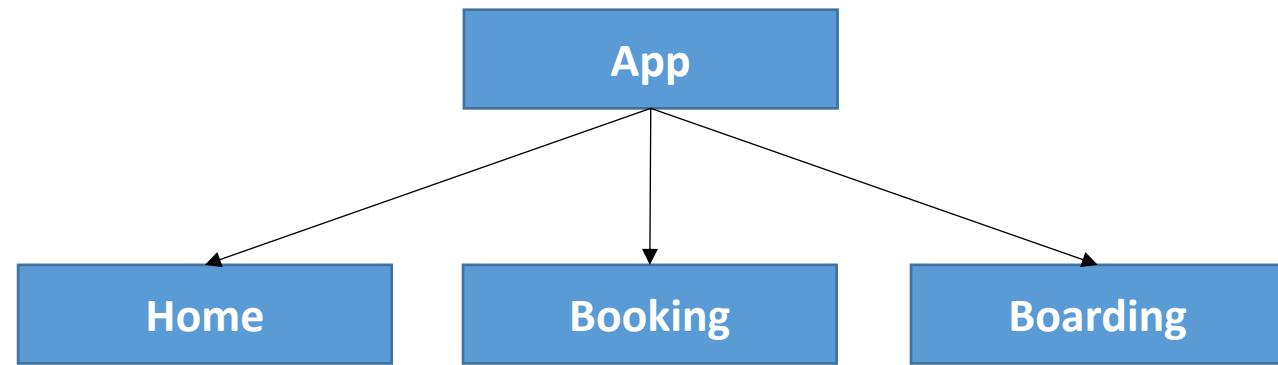
Alex Thalhammer

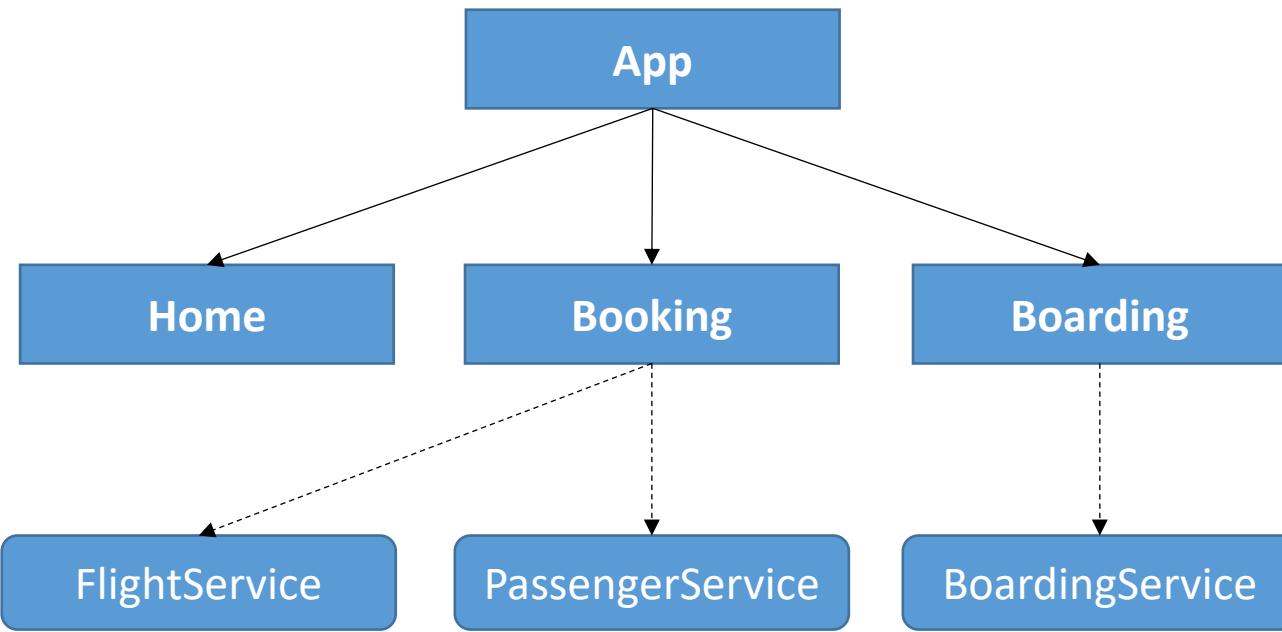
Outline

- Motivation
- State
- Actions
- Reducer
- Store
- Selectors
- Effects

Motivation



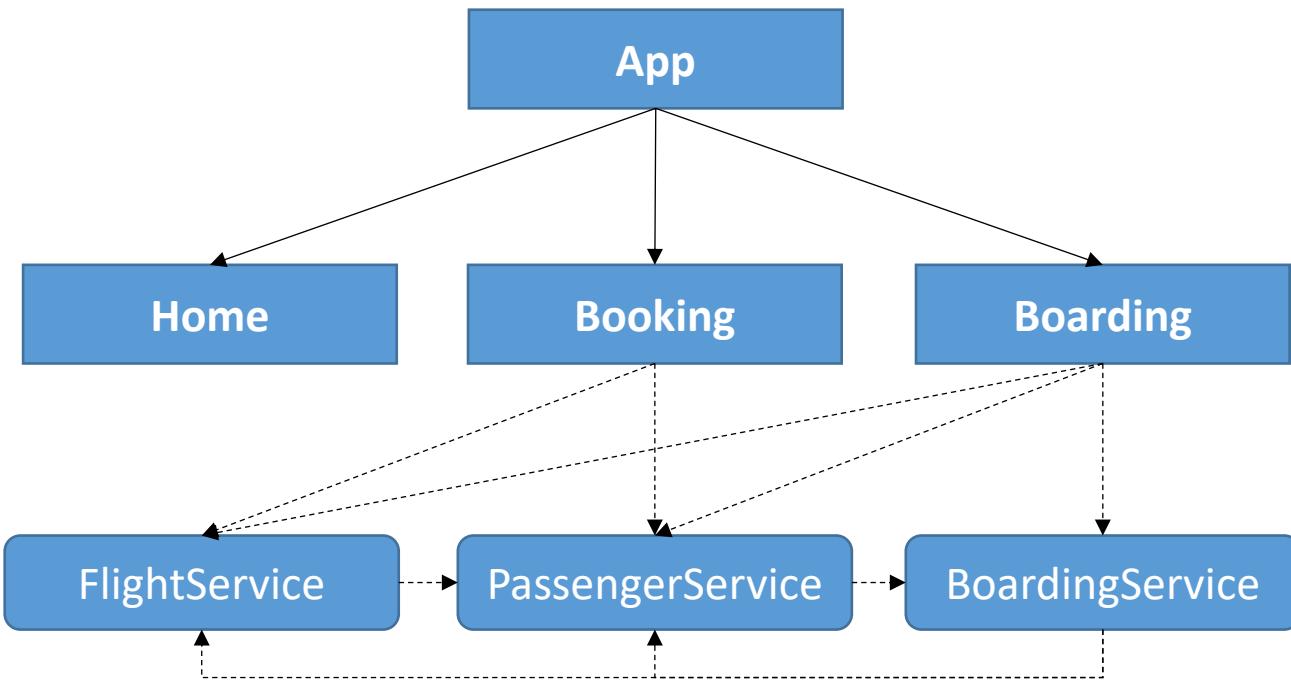




ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



SOFTWARE
ARCHITECT



ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



SOFTWARE
ARCHITECT

Redux

- Redux makes complex UI manageable
- Origin: React Ecosystem
- Implementation used here: `@ngrx/store`
- Alternative: `@ngxs/store`
- Or: `@dataroma/akita`



Alternatives

Alternatives

- NGXS
 - More Object-Oriented
 - Built-In Immutability
 - Reducer/Store also works with Asynchrony – no need for Effects
- Akita
 - Not based on Redux, only RxJS
 - Works very similar to @ngrx/entity
 - Can be seen as a simplified subset of ngrx
 - Very limited in terms of customisation

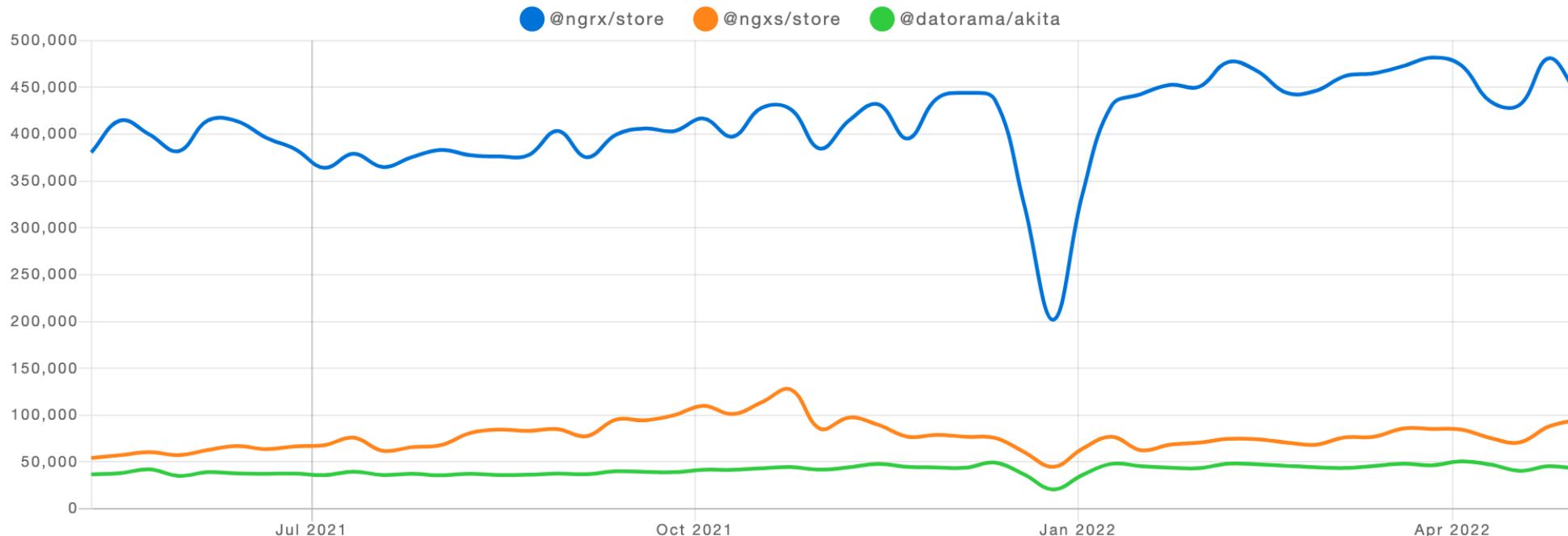
Alternatives

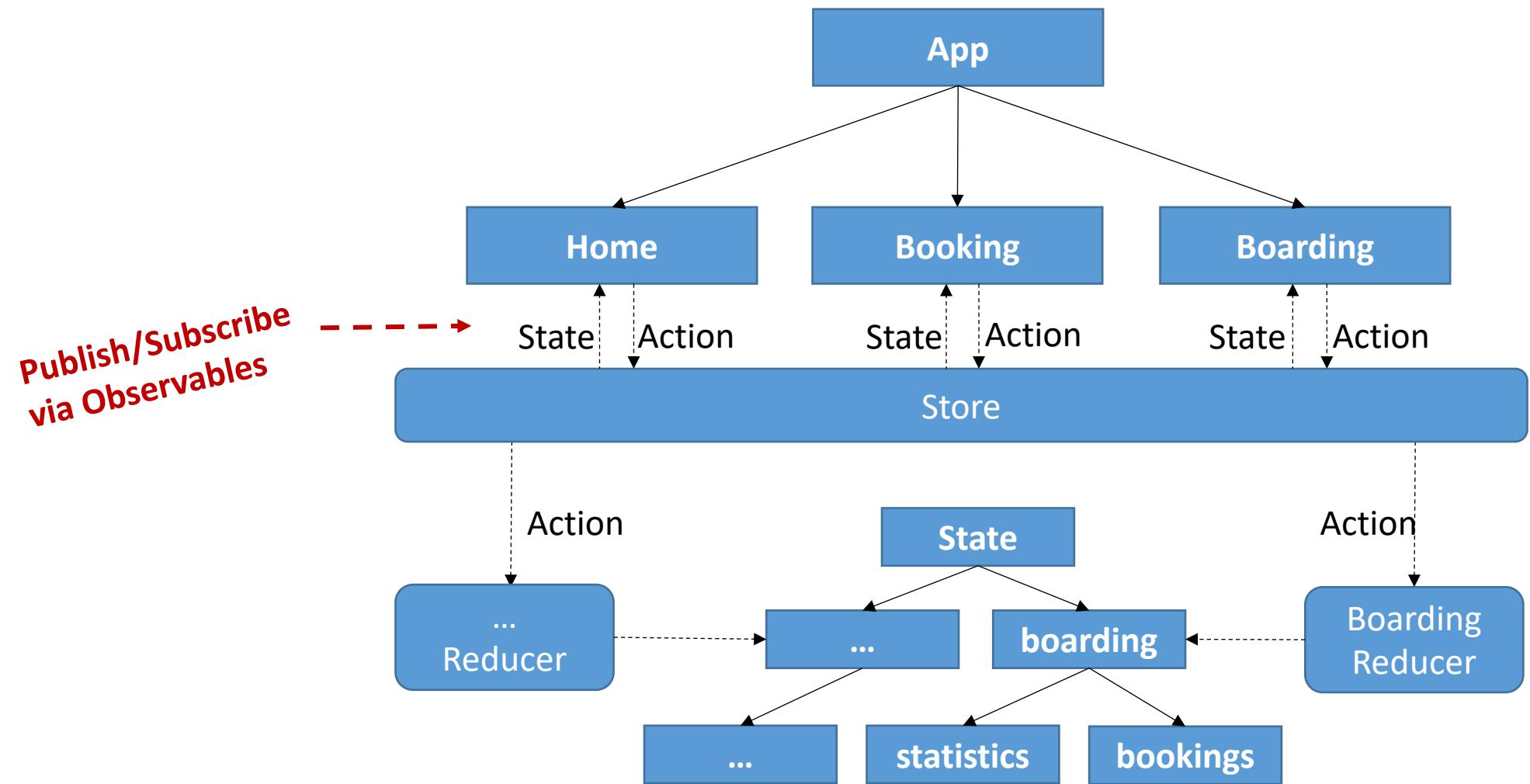
@datorama/akita vs @ngrx/store vs @ngxs/store

Enter an npm package...

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

Downloads in past 1 Year ▾





Single Immutable State Tree

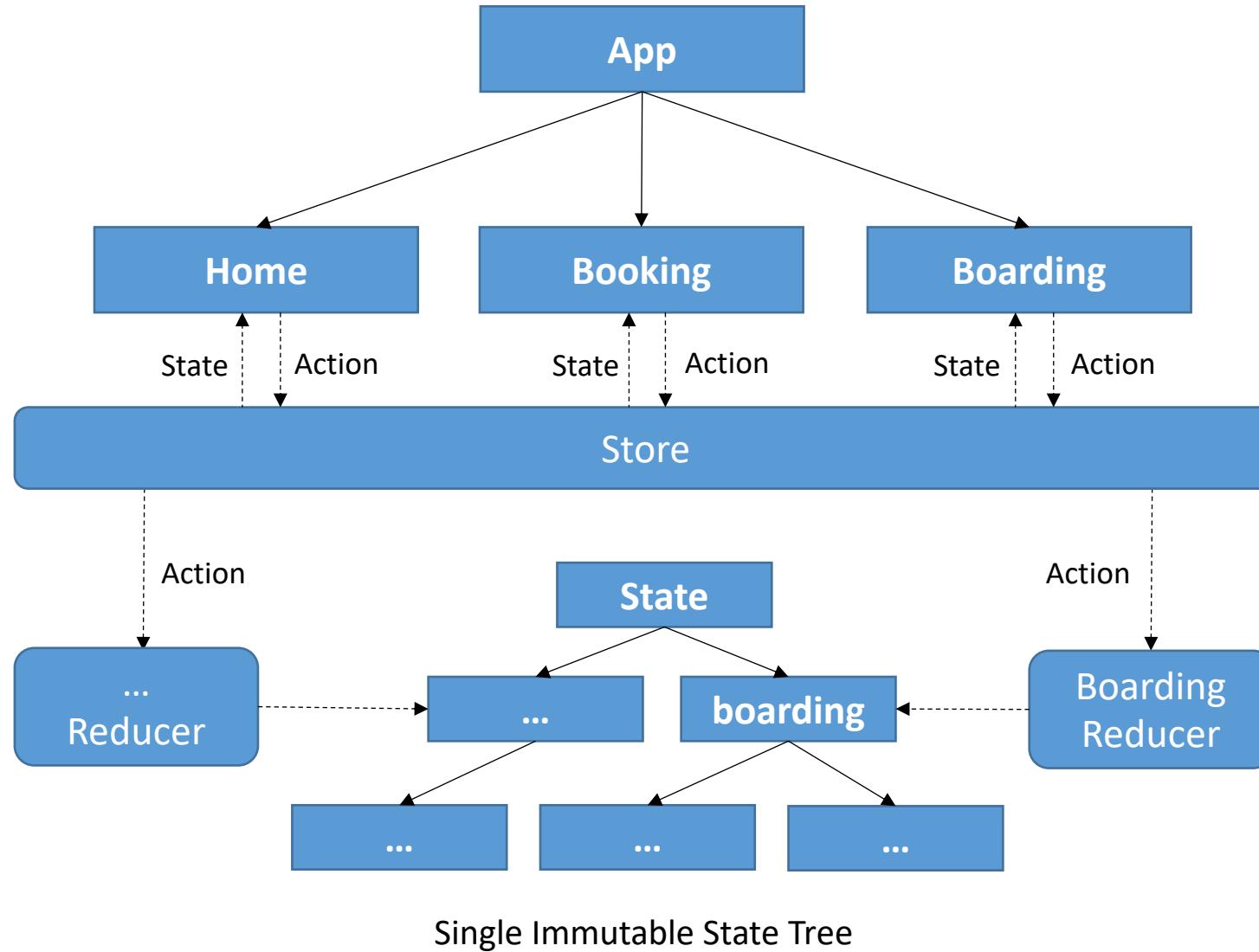


ANGULAR
ARCHITECTS

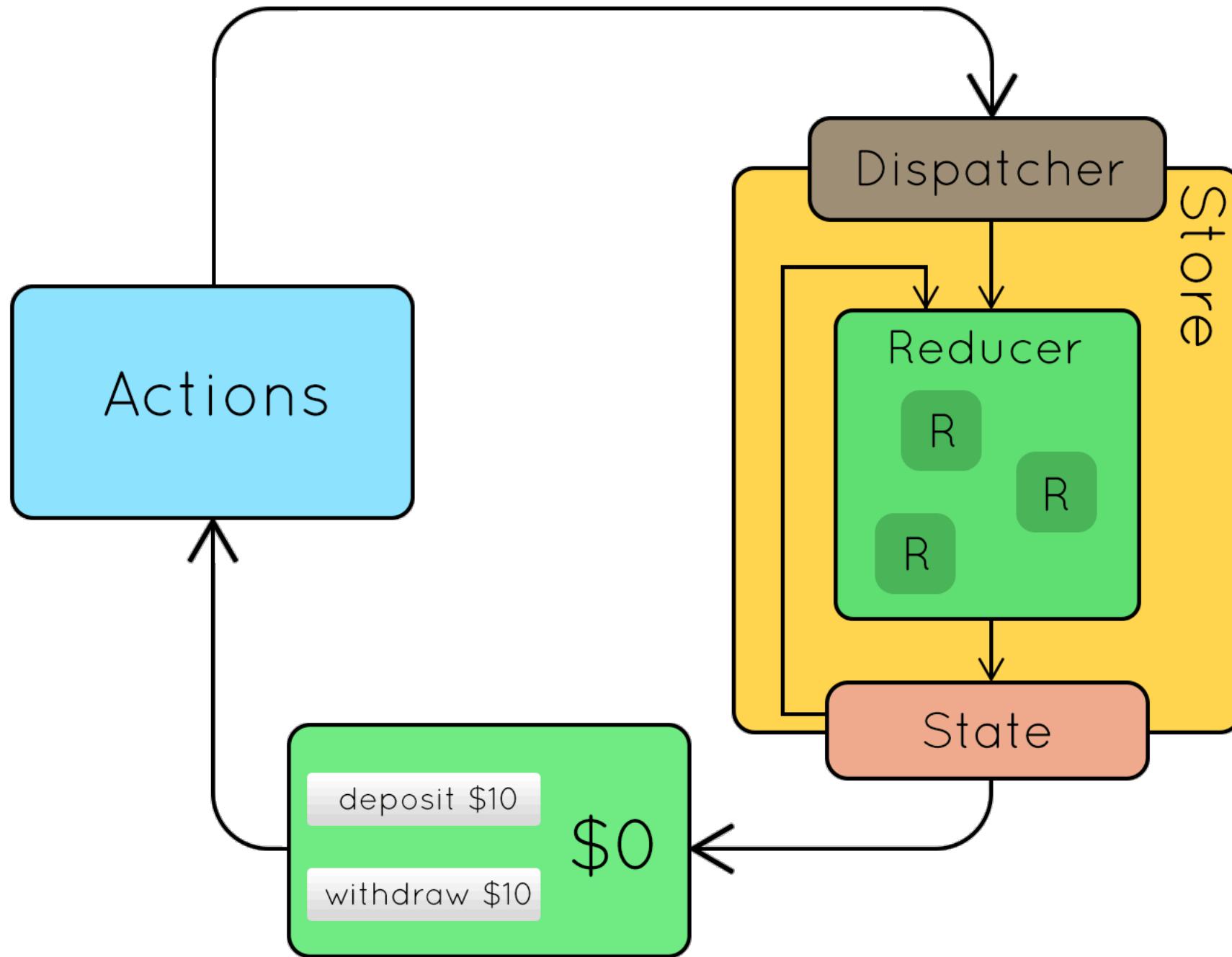
INSIDE KNOWLEDGE



SOFTWARE
ARCHITECT



- One "source of truth"
- Prevents Cycles
- Easy to debug
- Structured
- Performance
 - Observables
 - Immutables



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

Actions

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

Actions consist of

- Type (must have)
- Payload (optional)

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

- Reducers are responsible for handling transitions from one state to the next state in your application
- Signature of "on"

`(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 appReducers = {
  "flightBooking": flightBookingReducer,
  "currentUser": authReducer
}
```

A close-up photograph of several dark wooden barrels stacked together. The barrels have metal bands around them and some have circular holes. The wood shows signs of age and wear.

Store

Store

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

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

@ngrx/store-devtools

- Add Chrome / Firefox extension to use Store Devtools
 - Works with Redux & NgRx
 - <https://ngrx.io/guide/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, kicking up a cloud of dust. The background shows the reddish-brown terrain of Mars under a hazy sky.

ngrx and Feature Modules

Registering @ngrx/Store

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

Demo & Lab

NgRx Store

Selectors

Selectors

- Selectors are pure functions used for obtaining slices of store state (also called state streams)
- `select(tree => tree.flightBooking.flights): Observable<Flight[]>`
- We can use [createSelector](#) or [createFeatureSelector](#)

Using selectors for manipulation (filtering)

```
export const selectFlightBookingState =  
  createFeatureSelector<fromFlightBooking.State>  
  (fromFlightBooking.flightBookingFeatureKey);
```

```
export const selectFlights =  
  createSelector(selectFlightBookingState, (s) => s.flights);
```

Defining selectors with Payload

```
export const selectFlightsWithProps =  
  (props: { blackList: number[] }) =>  
    createSelector(selectFlights, (flights) =>  
      flights.filter((f) => !props.blackList.includes(f.id)));
```

Demo & Lab

NgRx Selectors

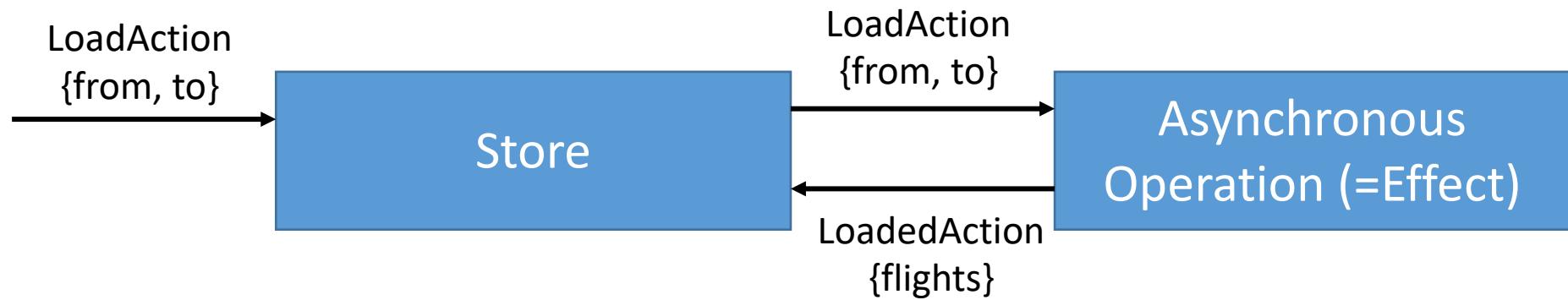
Effects



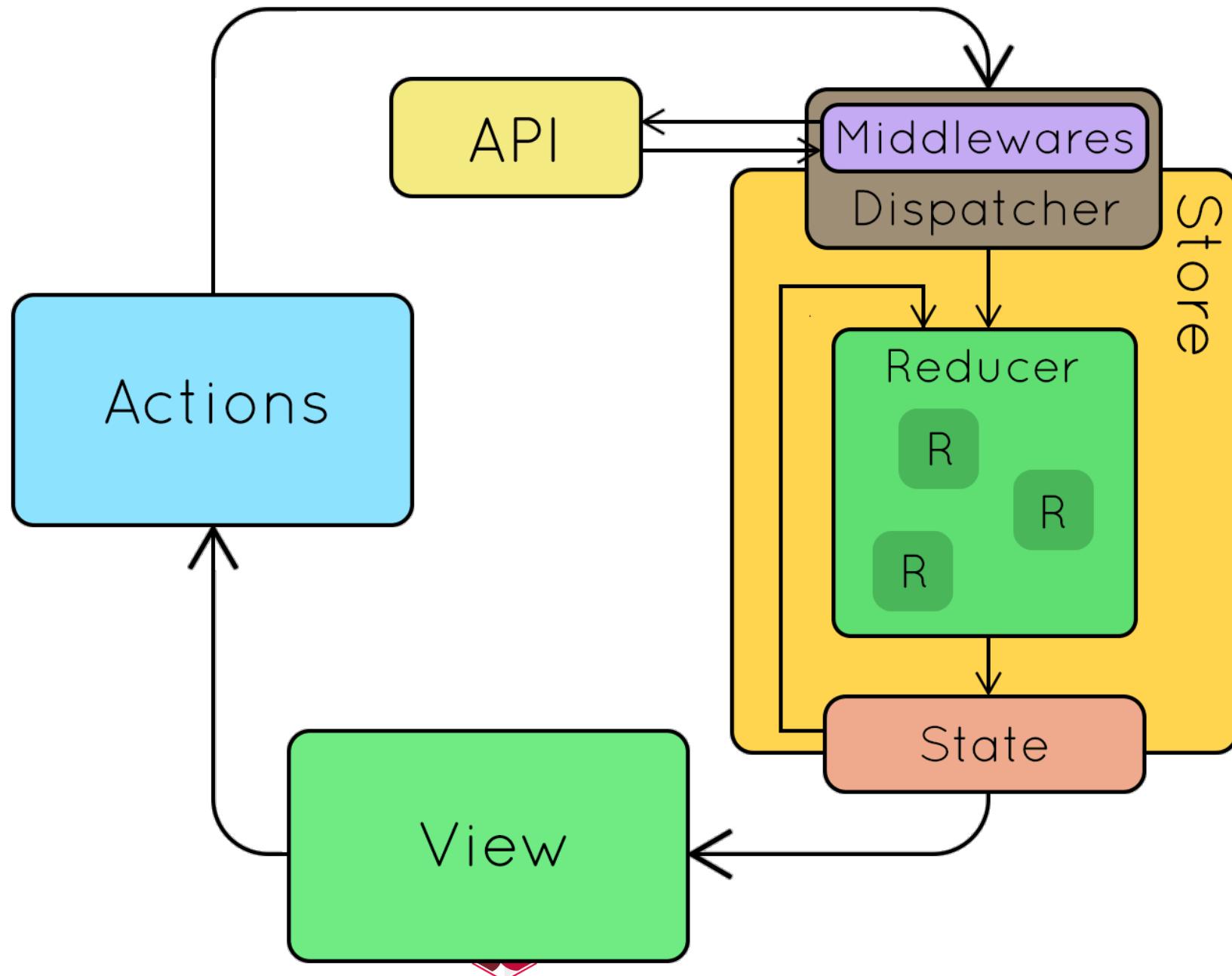
Challenge

- 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

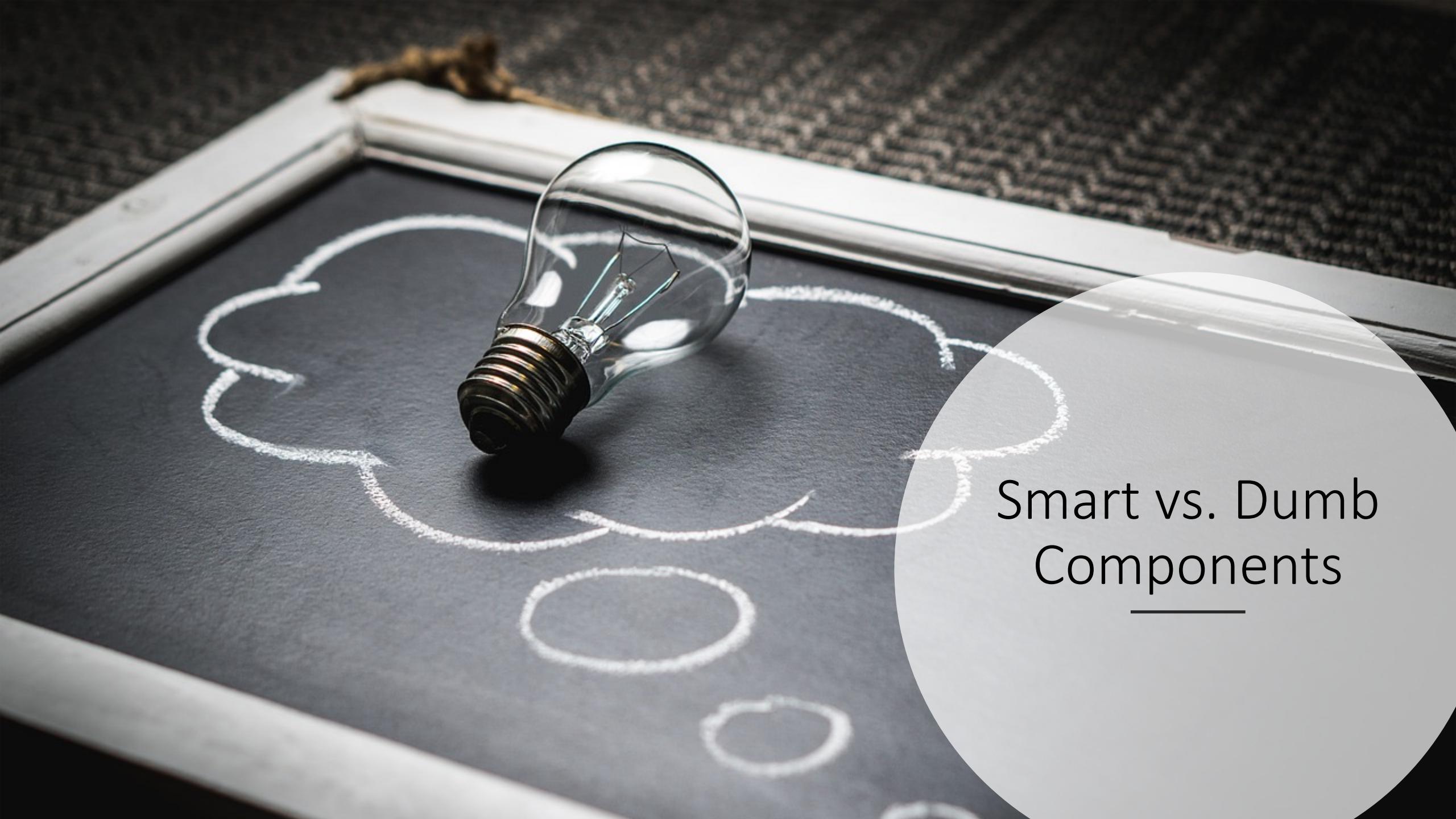
```
@NgModule({
  imports: [
    StoreModule.provideStore(appReducer, initialState),
    EffectsModule.forRoot([SharedEffects]),
    StoreDevtoolsModule.instrument()
  ],
  [...]
})
export class AppModule { }
```

Implementing Effects

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

Demo & Lab

NgRx Effects

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



Like this topic?

- Check out the NgRx Guide
- <https://ngrx.io/guide/store> and
- <https://ngrx.io/guide/data/architecture-overview>