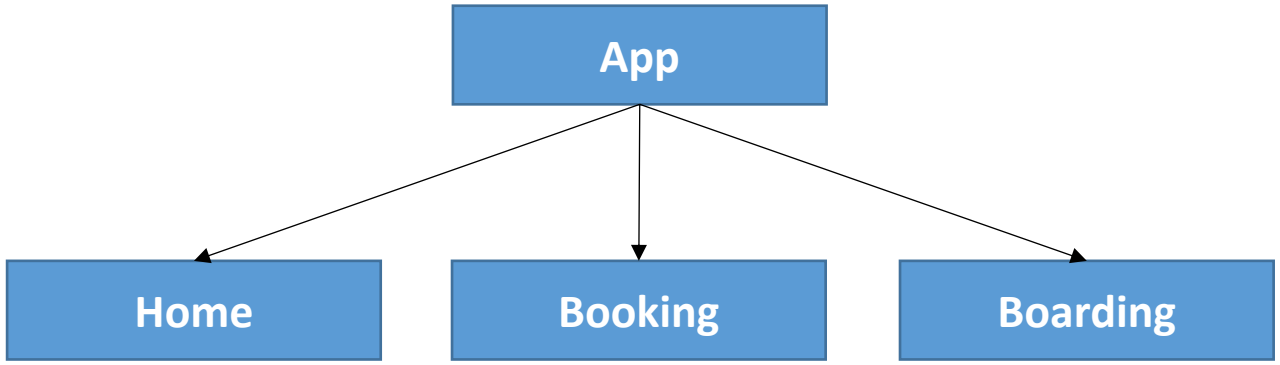
State Management with Redux und @ngrx/store

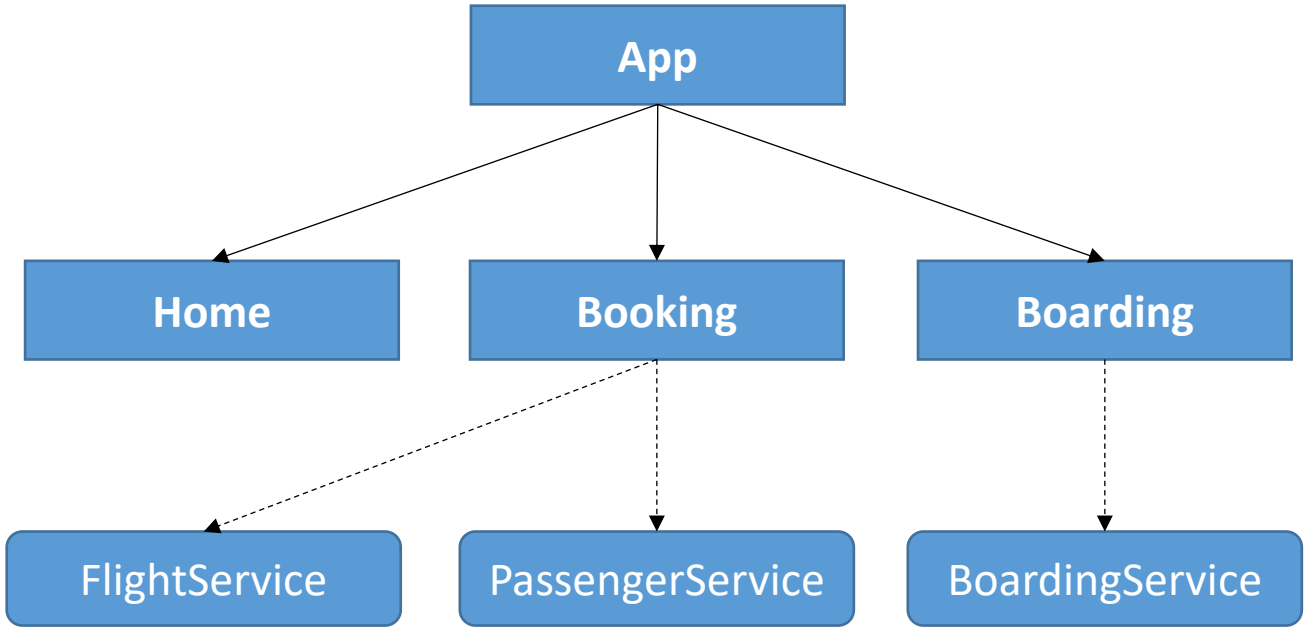
Alex Thalhammer

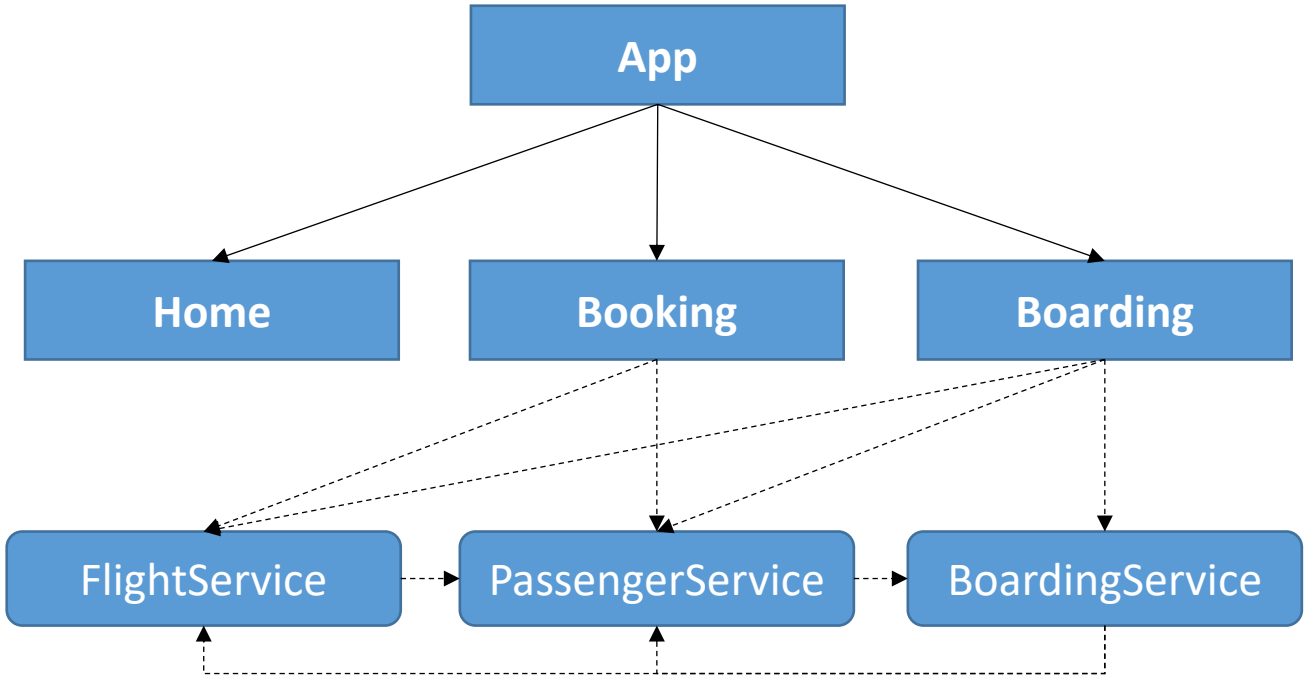
Contents

- Motivation
- State
- Actions
- Reducer
- Store
- Immutables
- Effects
- Labs!

Motivation


App
Home
Booking
Boarding


App
Home
Booking
Boarding
FlightService
PassengerService
BoardingService

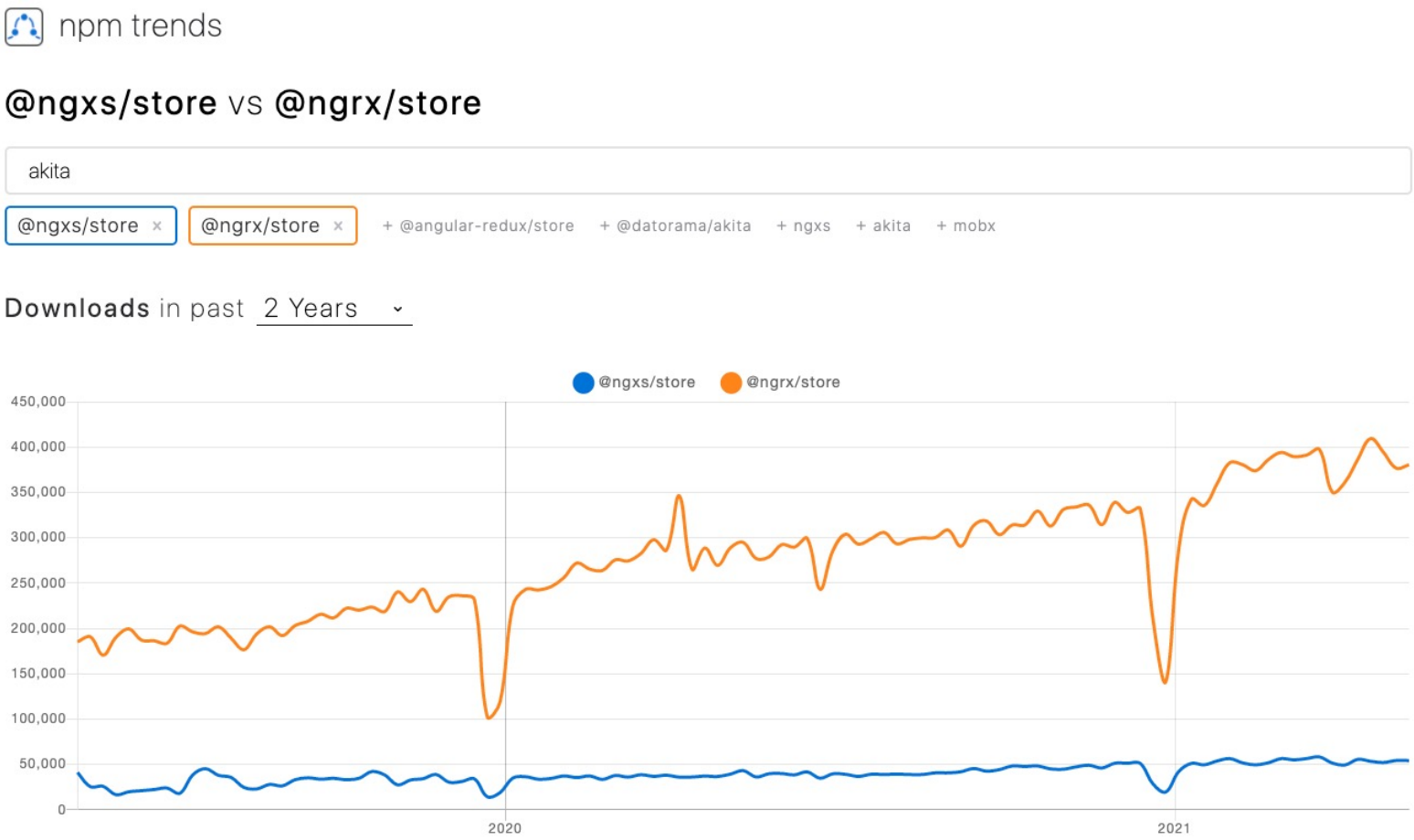

App
Home
Booking
Boarding
FlightService
PassengerService
BoardingService

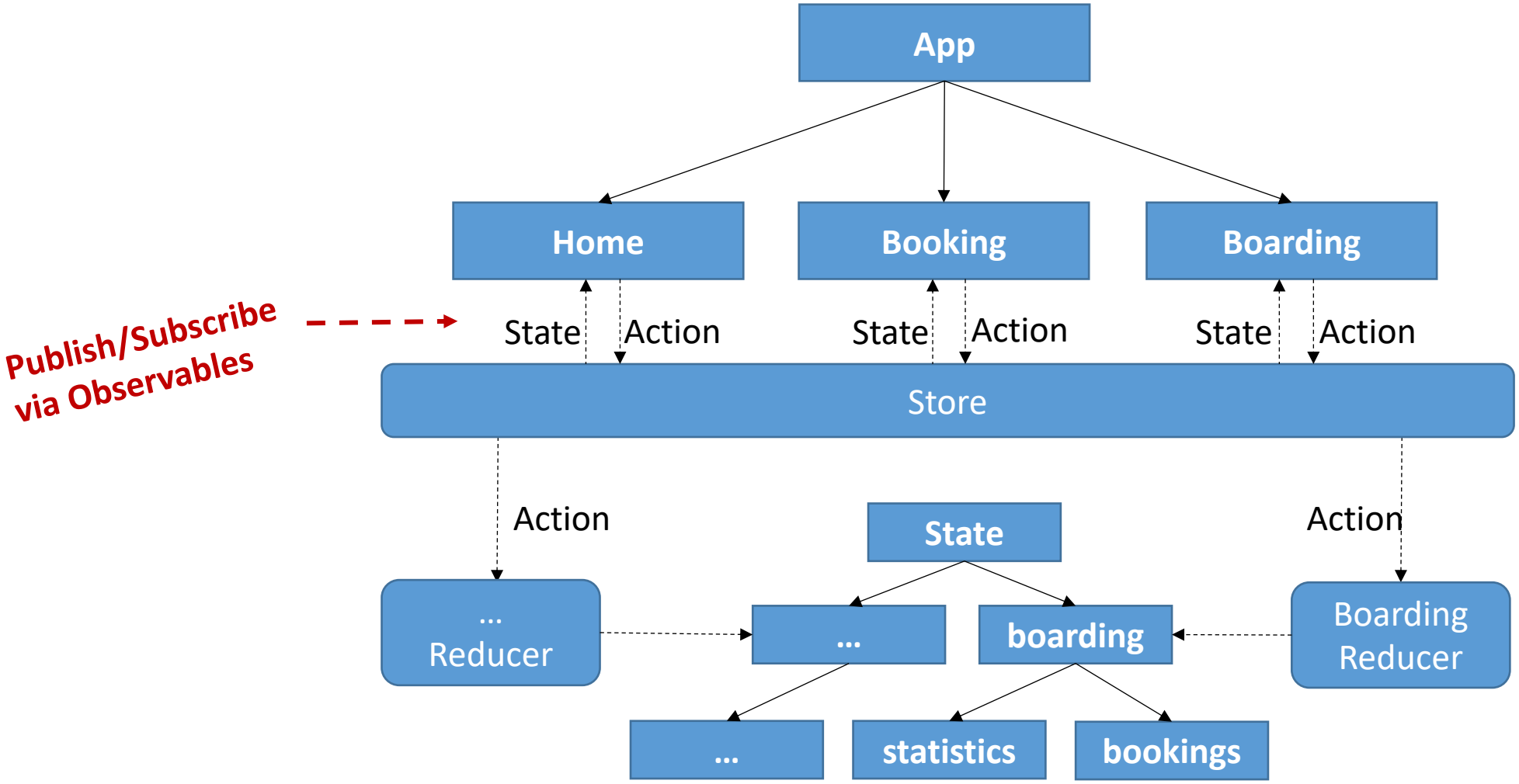
Redux

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

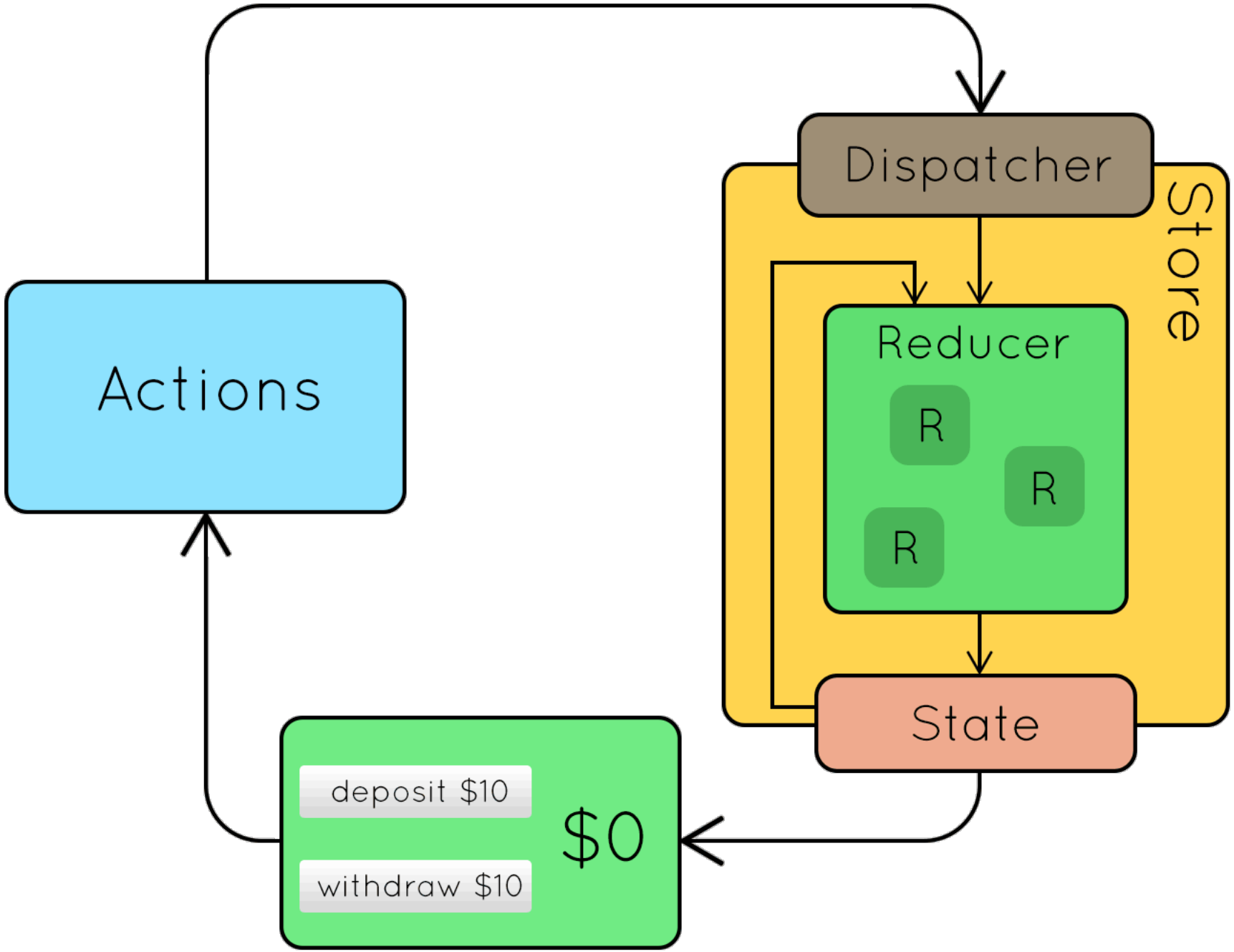
npm install @ngrx/store --save

Alternatives





Single Immutable State Tree


Actions
Dispatcher
Store
Reducer
R
R
R
State
deposit $10
withdraw $10
$0

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;
}
```


7DAYZ
Allianz
Allianz
Allianz
Title:
Direction:
Scene:
Camera:

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
}
```

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

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 { }
```

State branch for feature

Lab

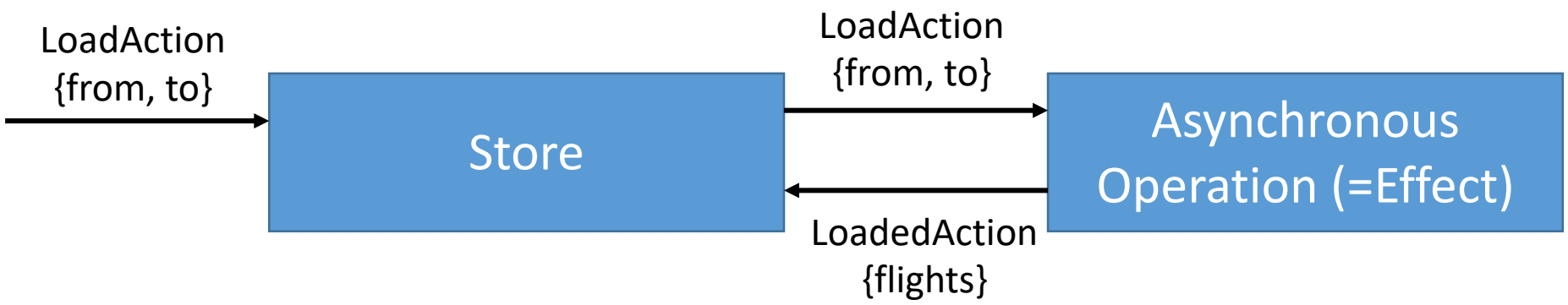
NgRx Store & Selectors

Effects

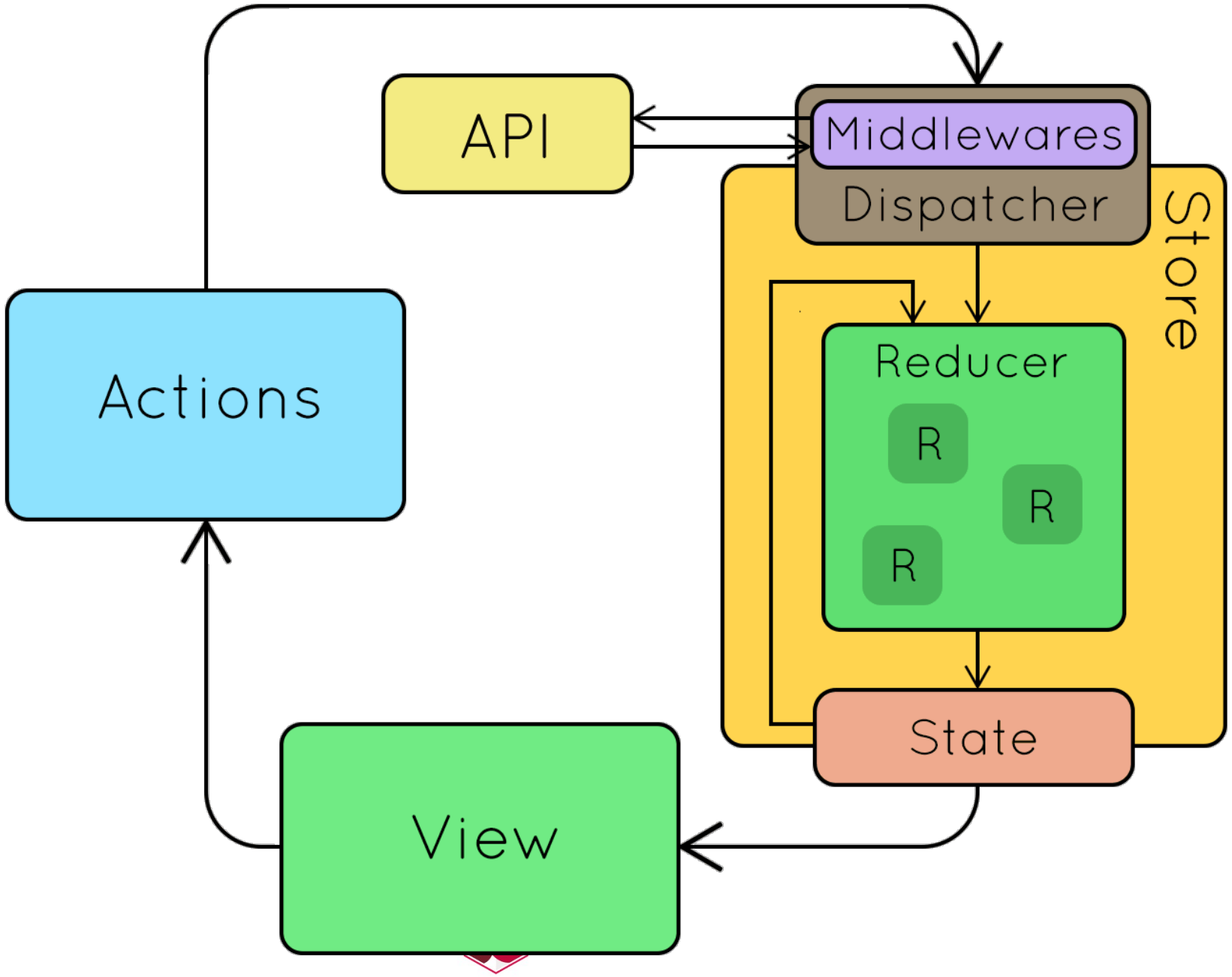
Challenge

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

Solution: Effects



ng add @ngrx/effects


API
Middlewares
Dispatcher
Store
Actions
Reducer
R
R
R
State
View

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

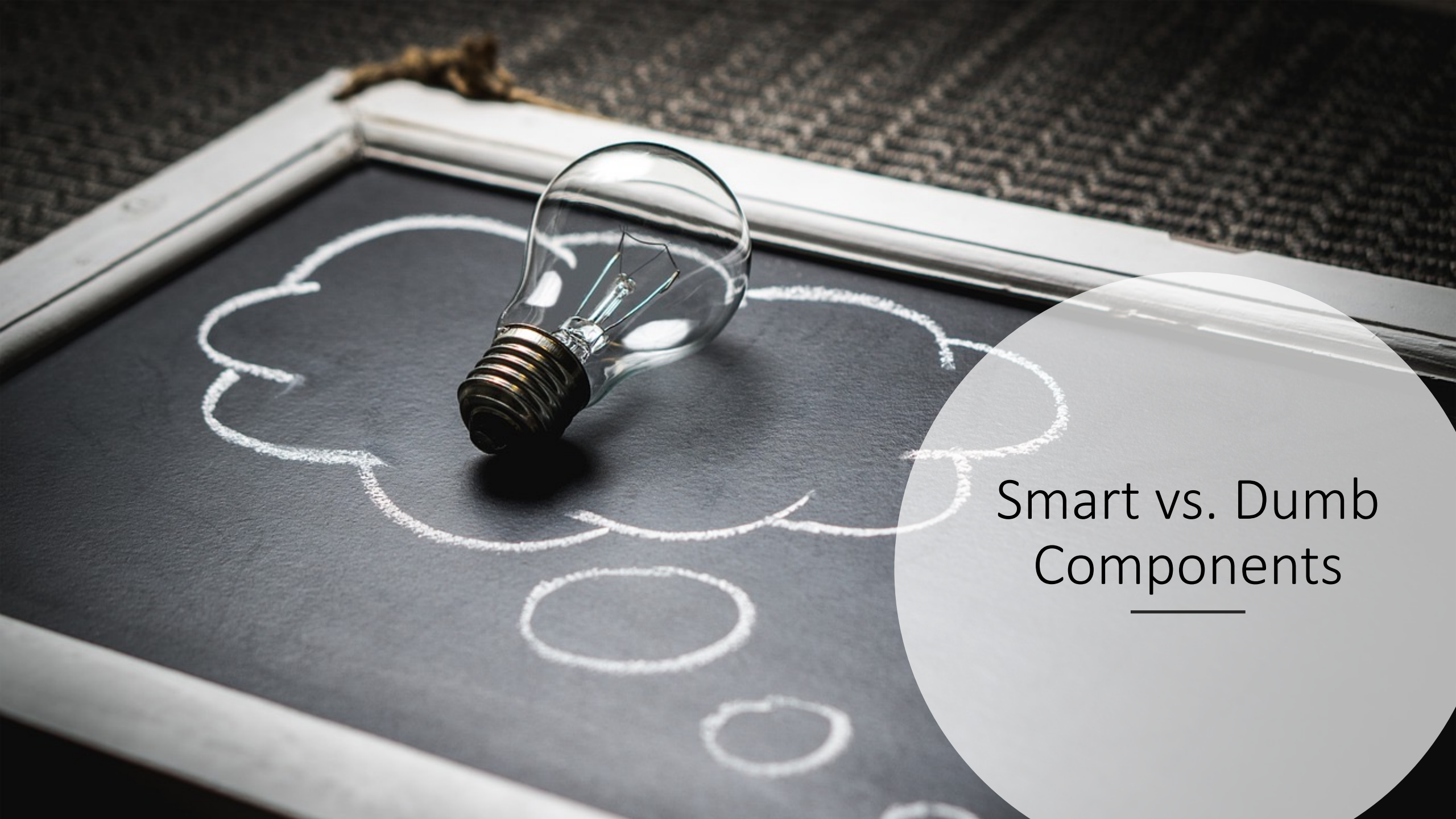
```
@NgModule({
  imports: [
    StoreModule.provideStore(appReducer, initialAppState),
    EffectsModule.forRoot([SharedEffects]),
    StoreDevtoolsModule.instrument()
  ],
  [...]
})
export class AppModule { }
```

Implementing Effects

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

Lab

NgRx Effects



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	Dumb
• Drives the "Use Case" • Usually a "Container"	• Independent of Use Case • Reusable • Usually a "Leaf"

Like this topic?

Check out the NgRx Guide

https://ngrx.io/guide/