



ManfredSteyer



# State Management with Redux und @ngrx/store

Manfred Steyer

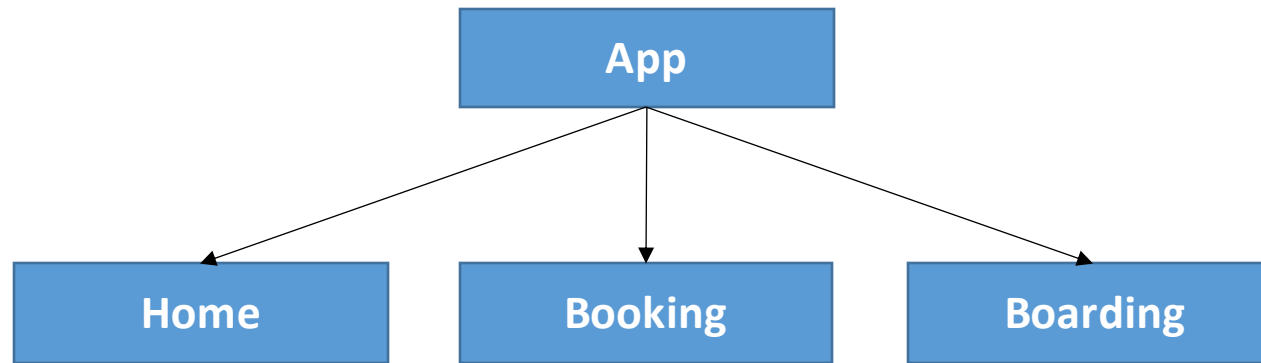
**ANGULAR**architects.io

# Contents

- Motivation
- State
- Actions
- Reducer
- Store
- Selectors
- Effects
- Labs / Demos



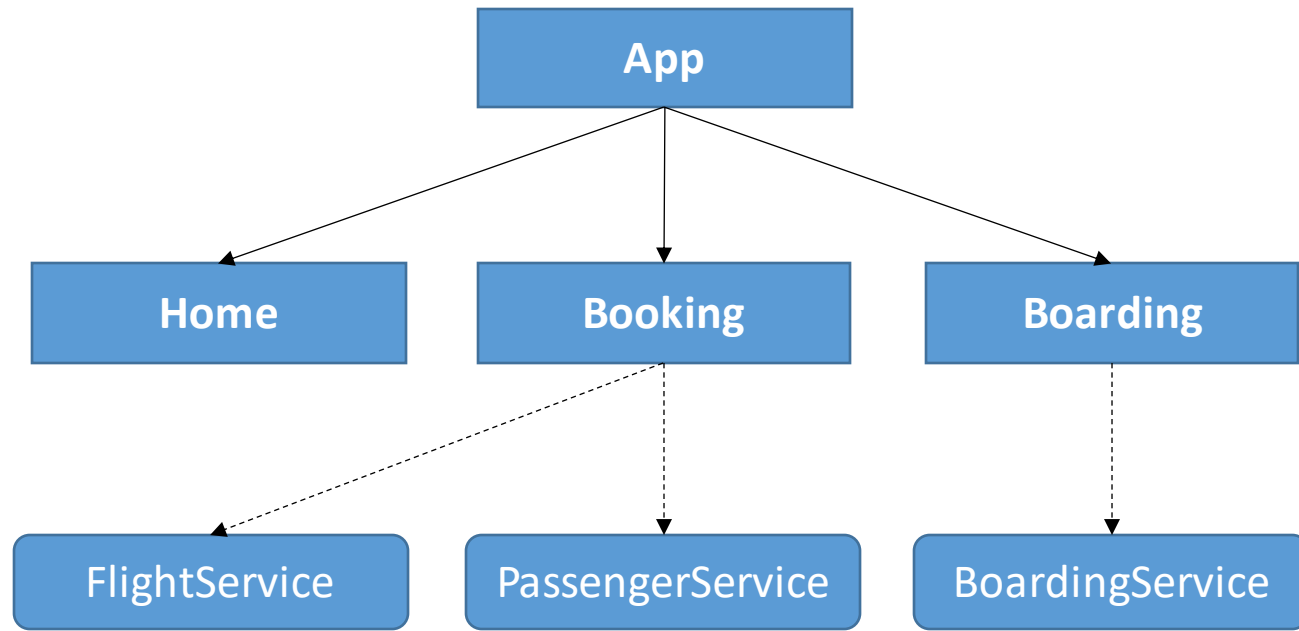
Motivation

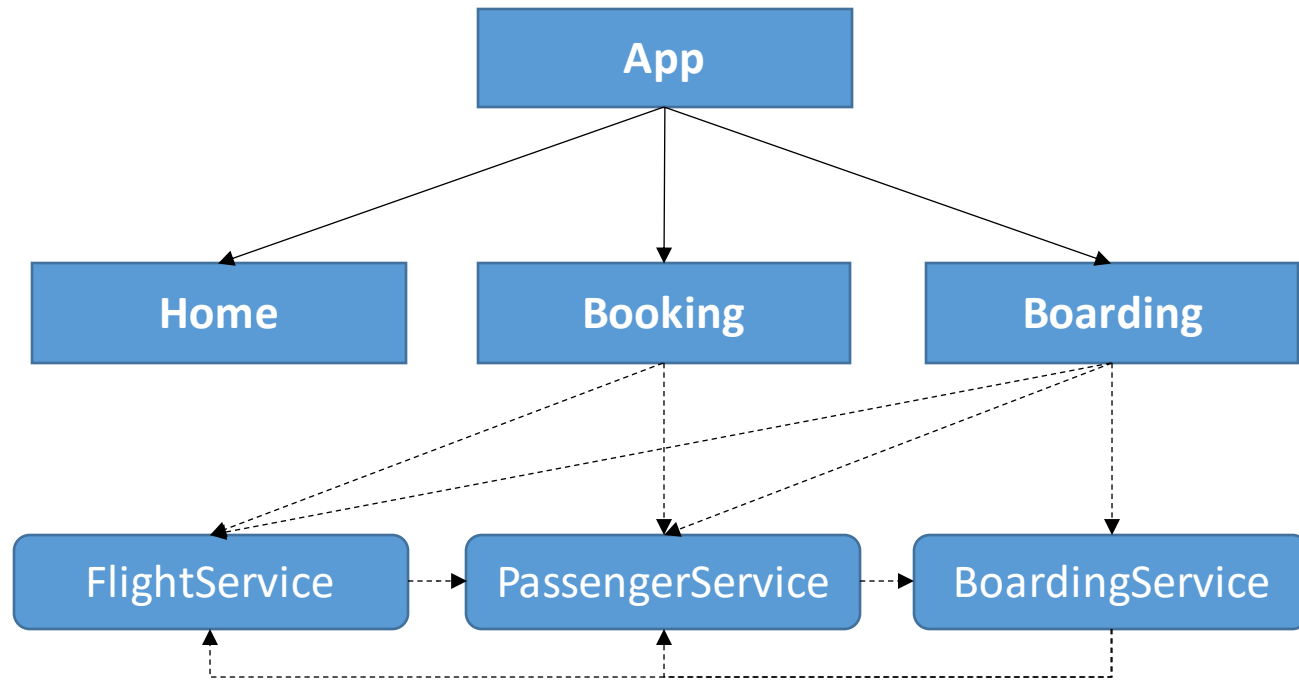


ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



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

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



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT

# Alternatives

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

Enter an npm package...

@datorama/akita x

@ngrx/store x

@ngxs/store x

+ @angular-redux/store

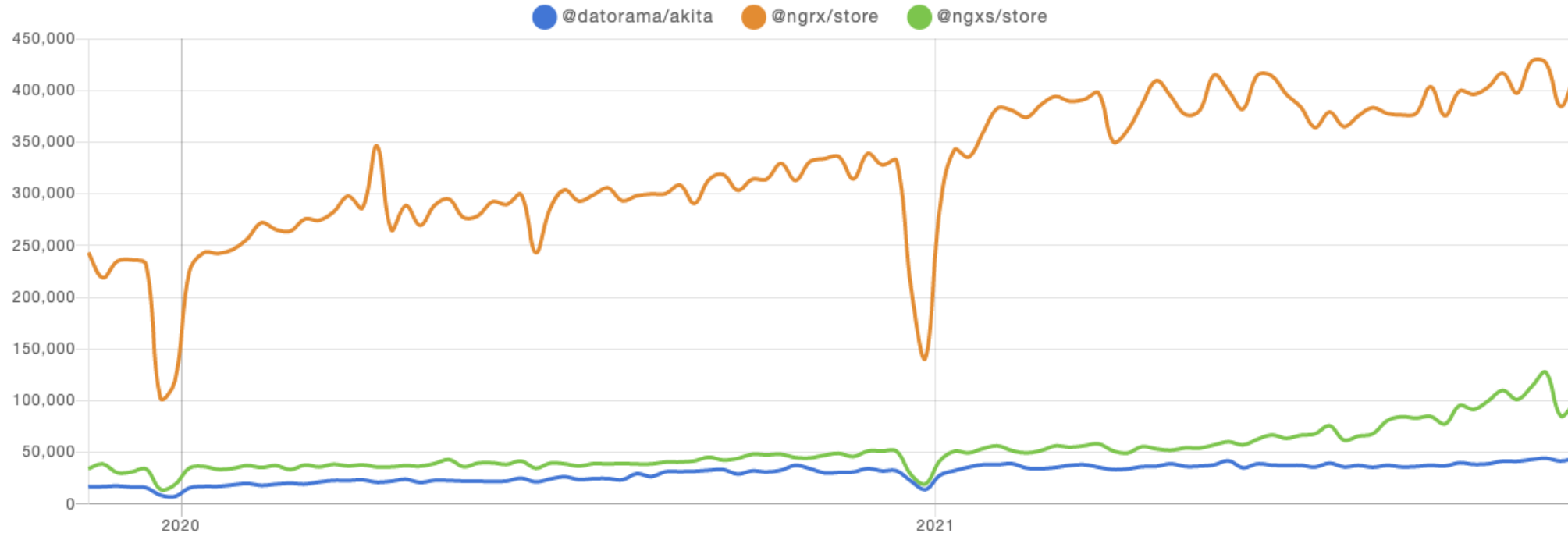
+ ngxs

+ akita

+ mobx

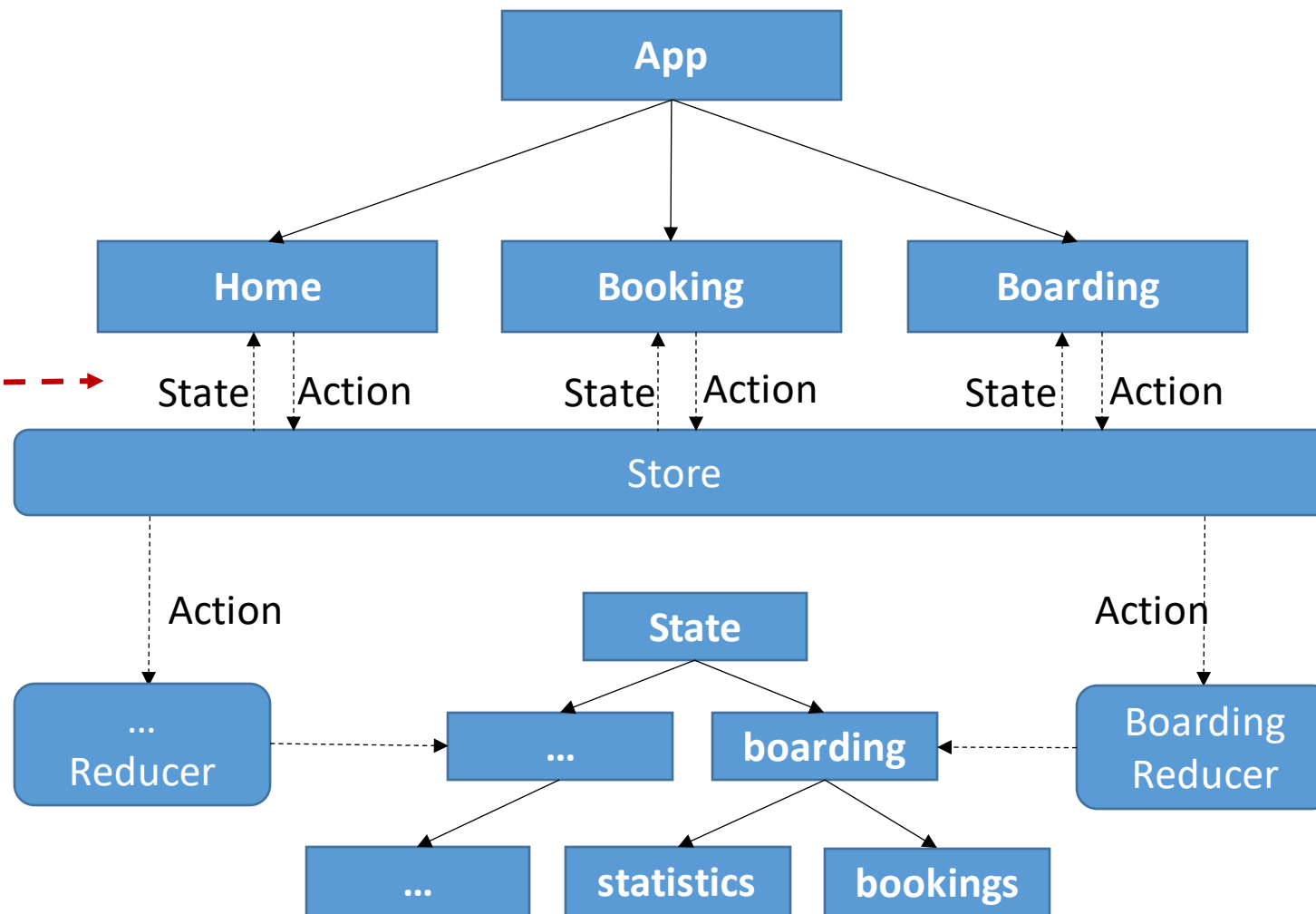
+ store2

Downloads in past 2 Years v





*Publish/Subscribe  
via Observables*



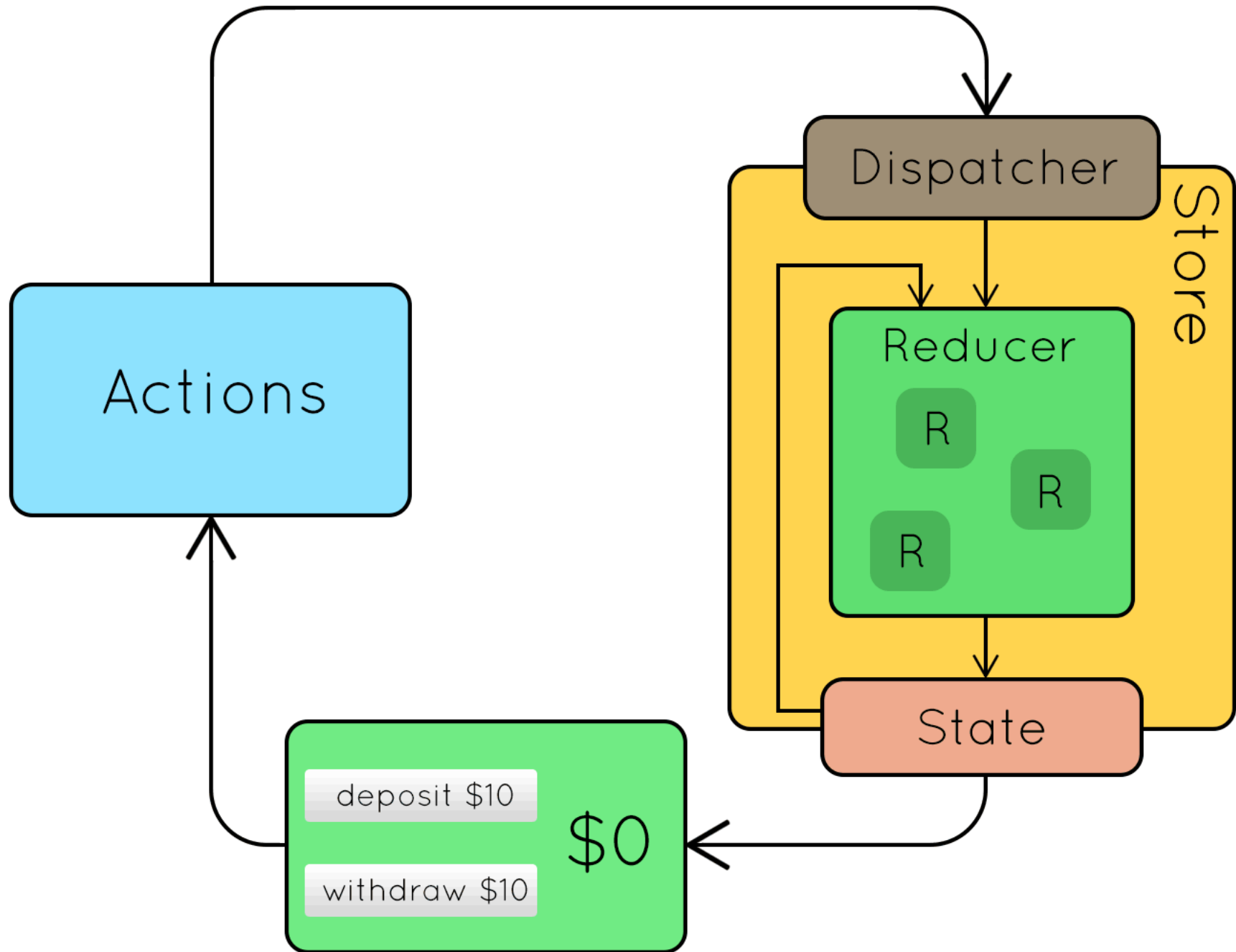
Single Immutable State Tree



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**



# State

# State

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



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# State

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

```
export interface FlightStatistics {  
  countDelayed: number;  
  countInTime: number;  
}
```



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# AppState

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



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**



Actions

# Actions

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



# Parts of an Action

- Type
- Payload



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Defining an Action

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



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

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
- Using on

**(currentState, action) => newState**



# Reducer for FlightBookingState

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



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT



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**



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**



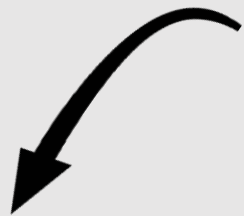
# ngrx and Feature Modules

---

# Registering @ngrx/Store

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

*State branch for feature*



# DEMO



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Lab

NgRx Store



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# 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](#)



# Defining selectors

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



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT

# Using selectors for manipulation (filtering)

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

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



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT

# DEMO



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Lab

NgRx Store & Selectors



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

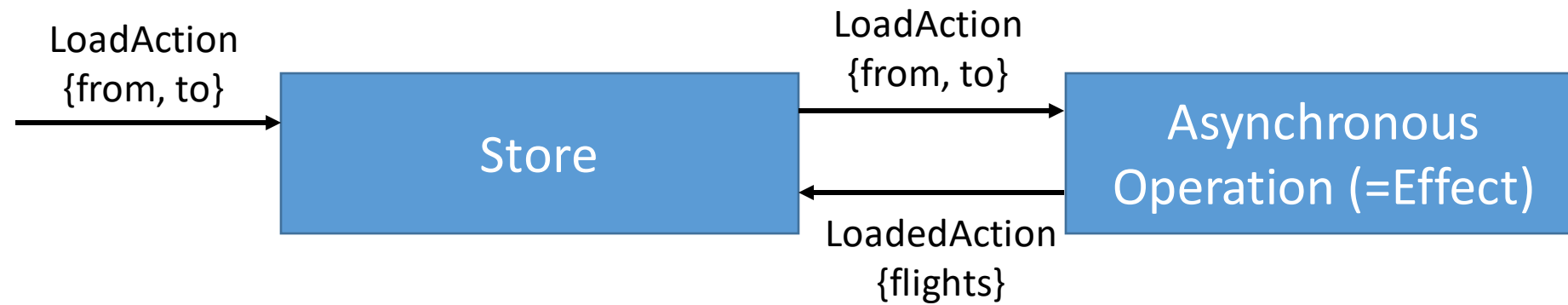
Effects



# Challenge

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

# Solution: Effects



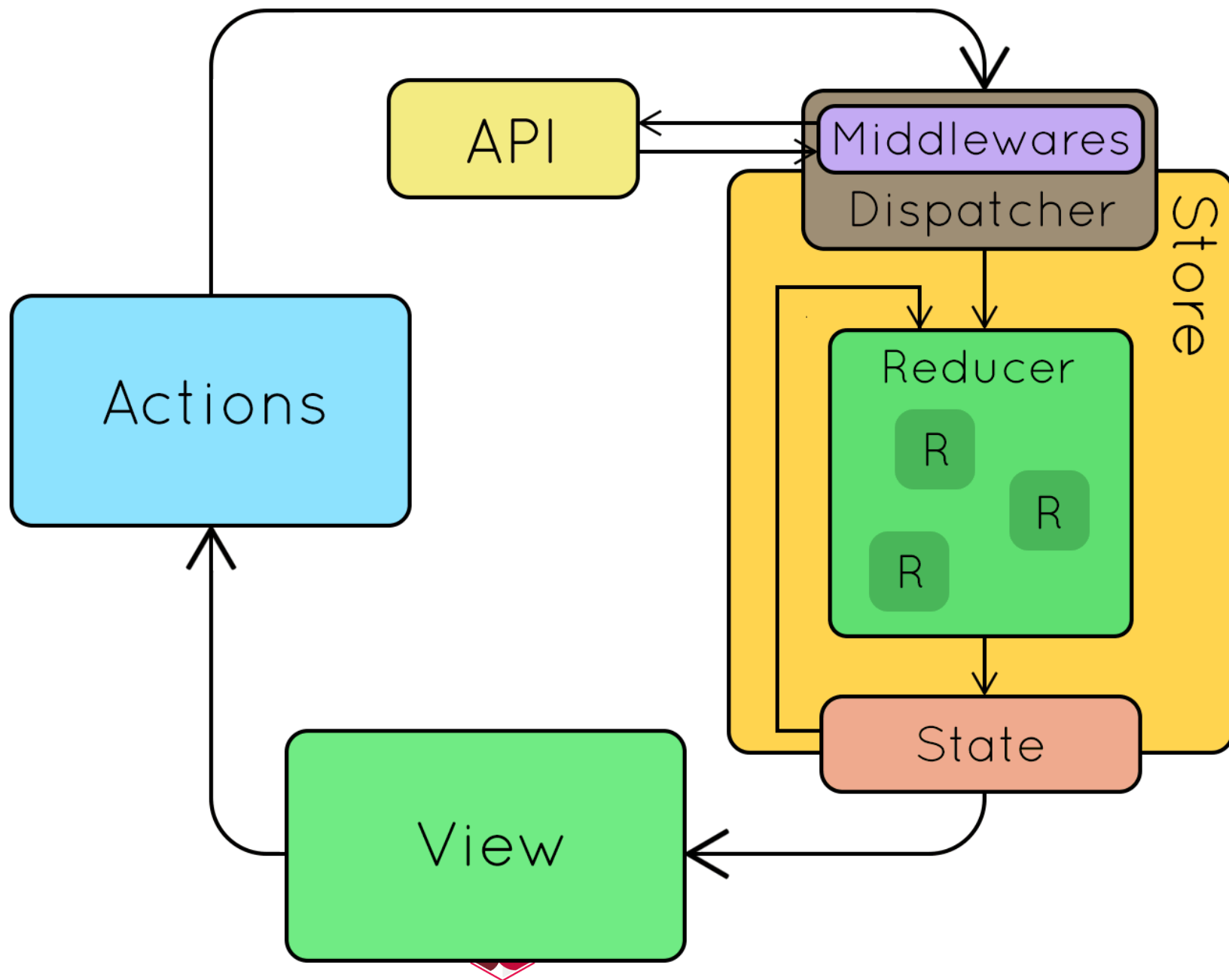
**ng add @ngrx/effects**



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE

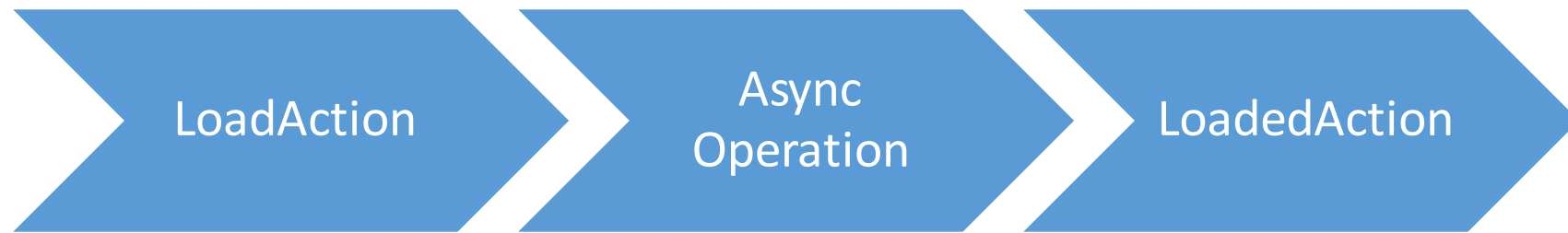


SOFTWARE  
**ARCHITECT**





# Effects are Observables



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Implementing Effects

```
@Injectable()
export class FlightBookingEffects {

  [...]

}
```



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Implementing Effects

```
@Injectable()
export class FlightBookingEffects {

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

  [...]

}
```



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Implementing Effects

```
@Injectable()
export class FlightBookingEffects {

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

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



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# 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)))));
}
```



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT

# 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})))));
}
```



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT

# Implementing Effects

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



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT

# Implementing Effects

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



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT



# DEMO



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Lab

NgRx Effects



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# @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



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# DEMO



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# @ngrx/store-devtools

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

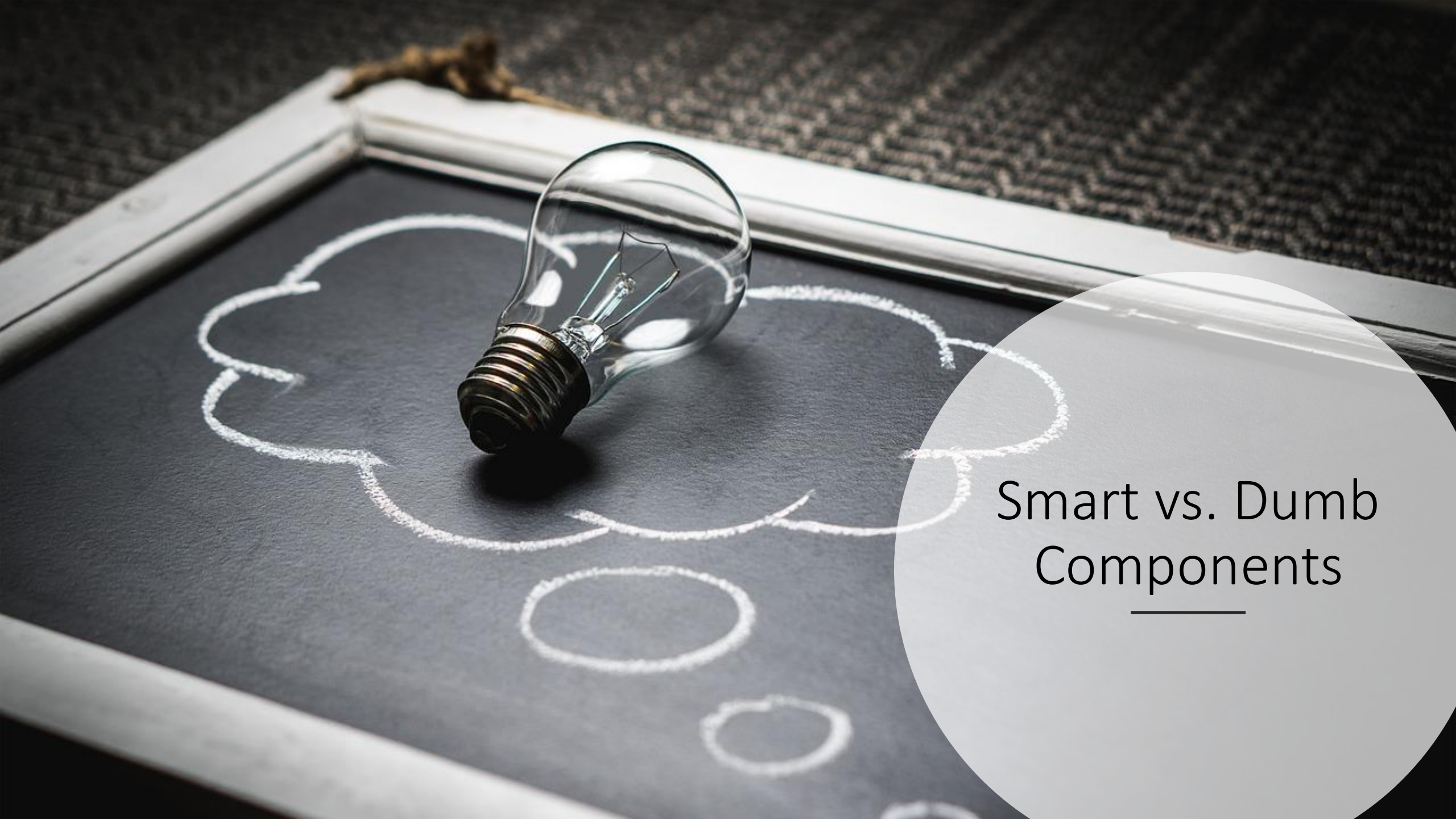
# DEMO



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**



## 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"



# Like this topic?

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