



ManfredSteyer



State Management with Redux und @ngrx/store

Manfred Steyer

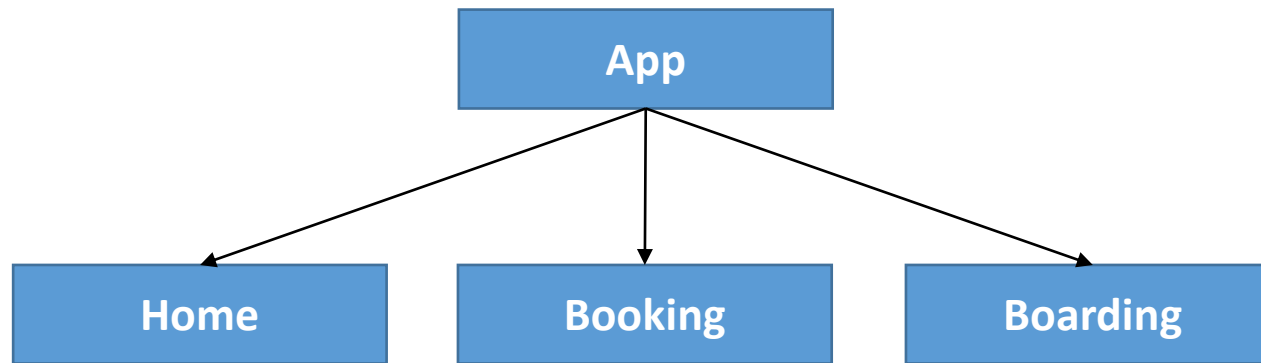
ANGULARarchitects.io

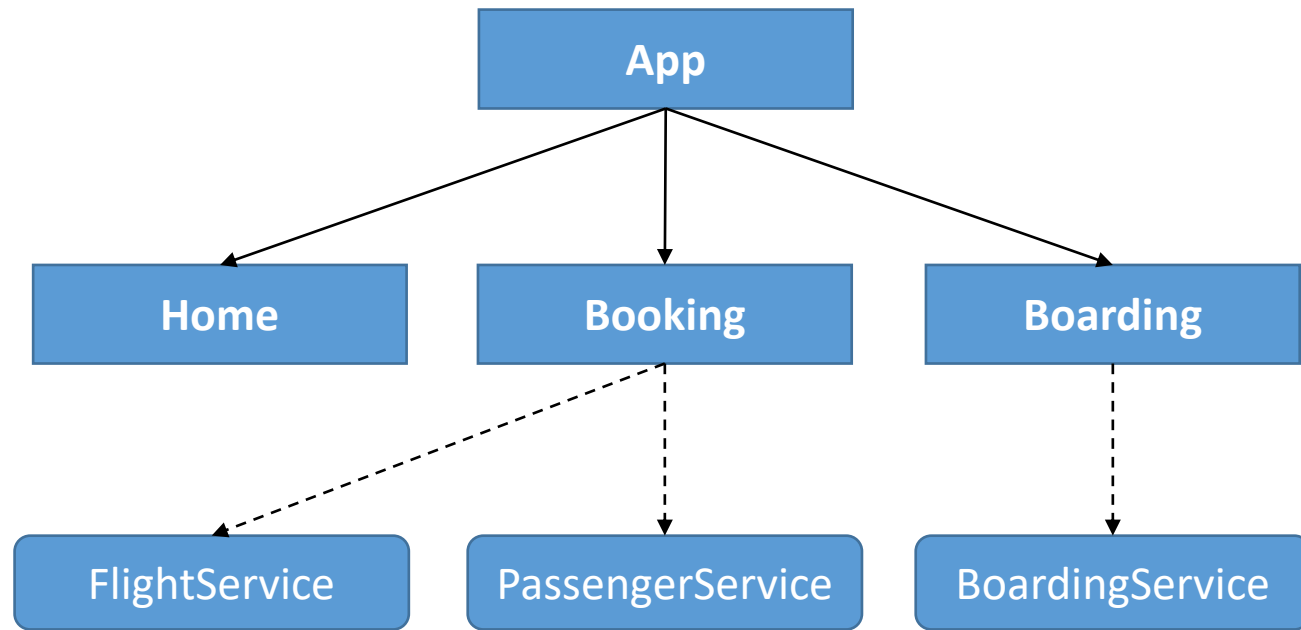
Contents

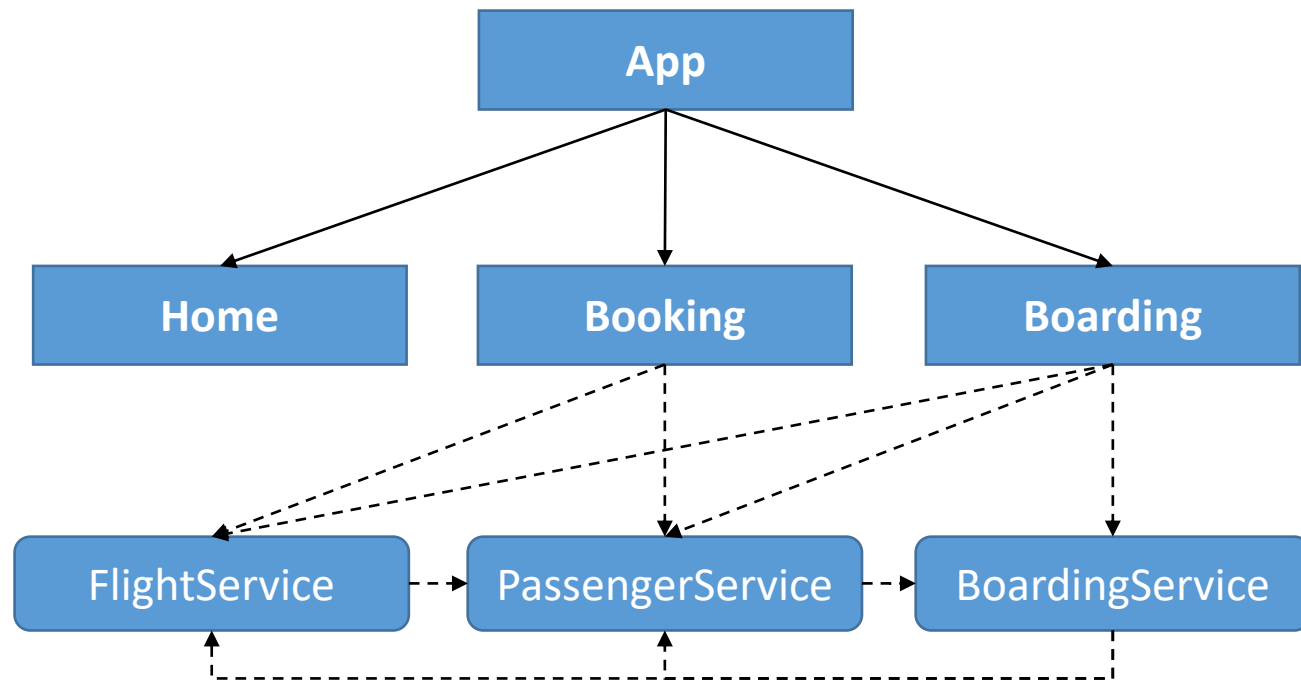
- Motivation
- State
- Actions
- Reducer
- Store
- Immutables
- Effects
- DEMO



Motivation





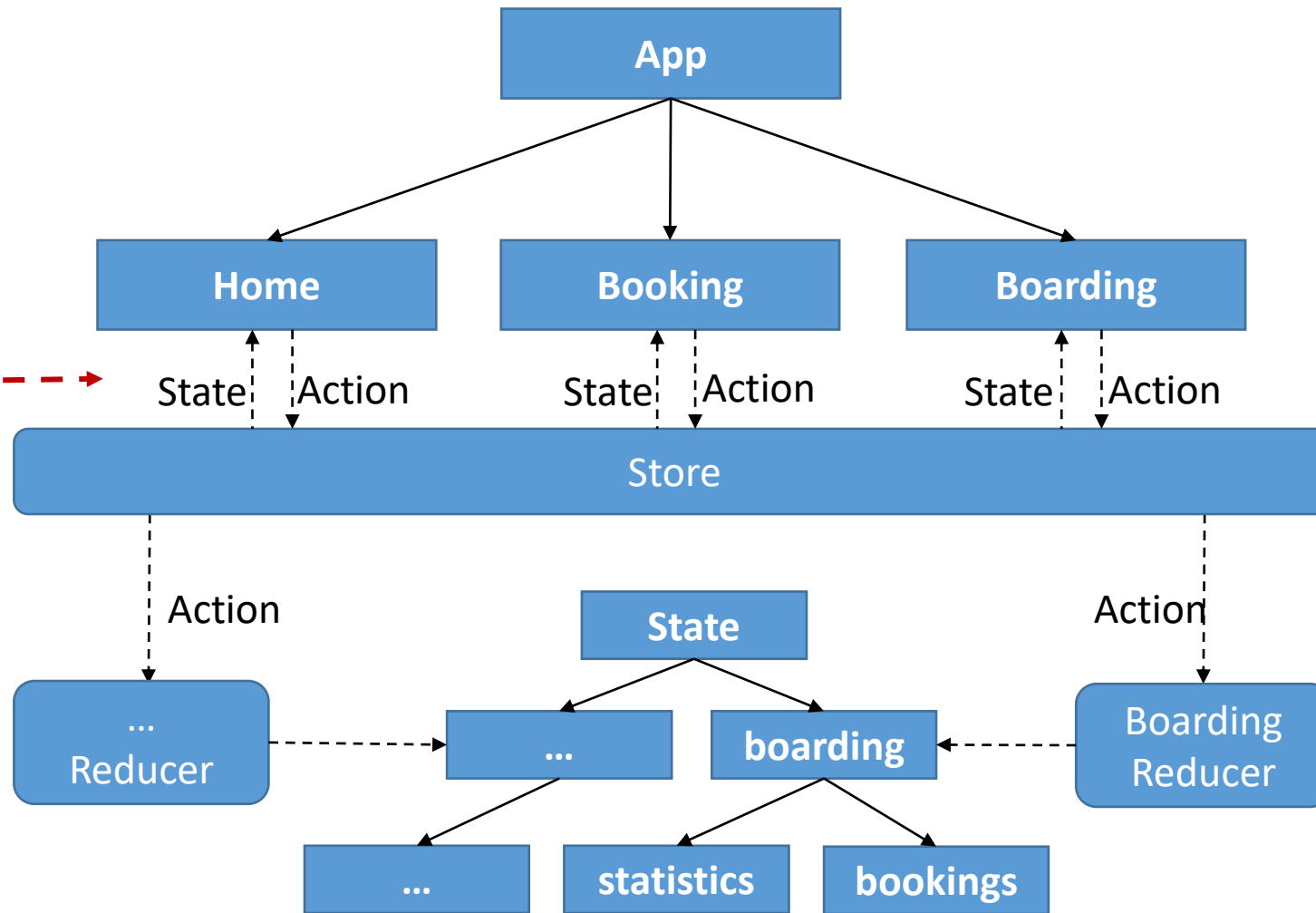


Redux

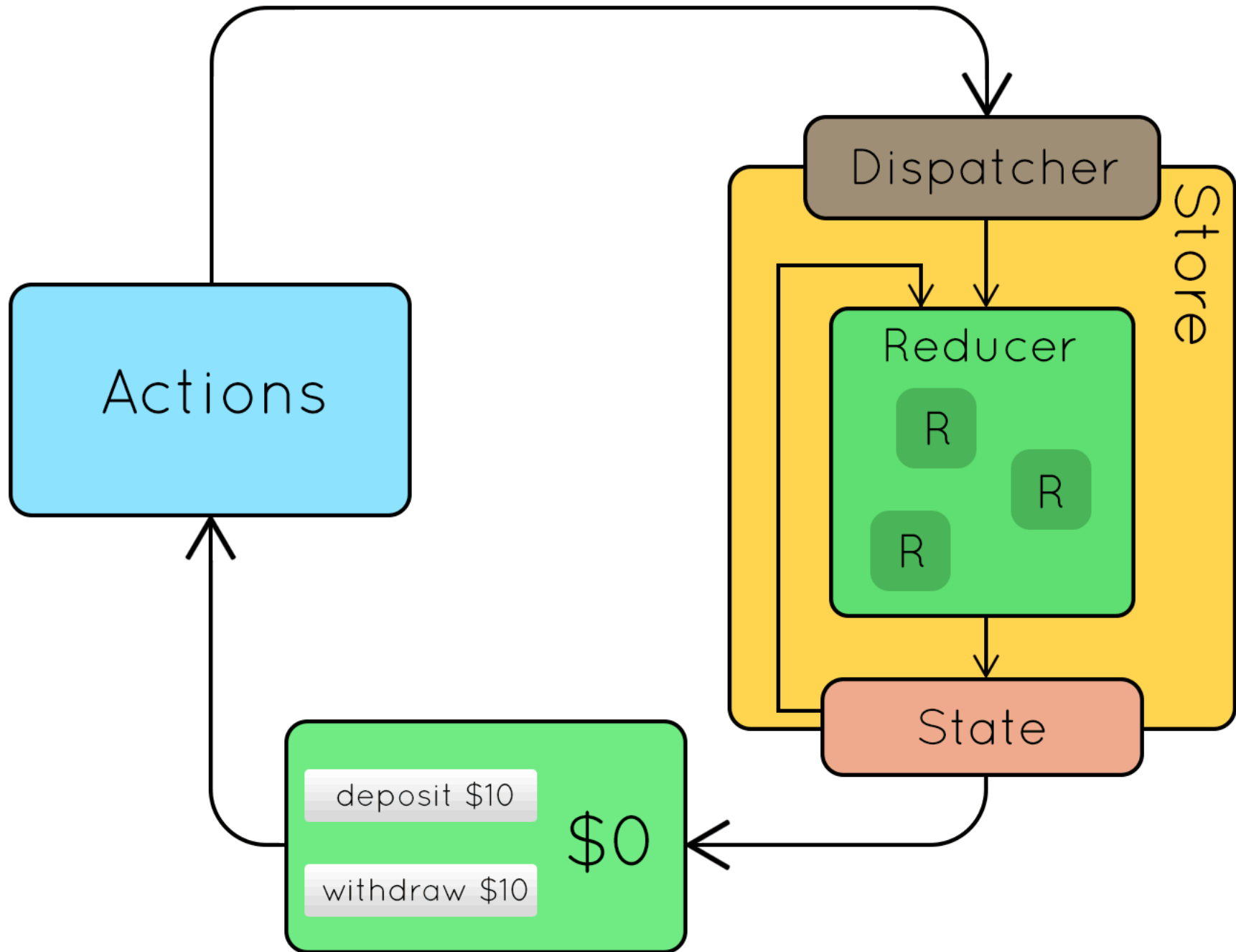
- Redux makes complex UI manageable
- Origin: React Ecosystem
- Implementation used here: `@ngrx/store`

`npm install @ngrx/store --save`

*Publish/Subscribe
via Observables*



Single Immutable State Tree





Building Blocks

State

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

AppState

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

Actions

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

Reducer

```
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

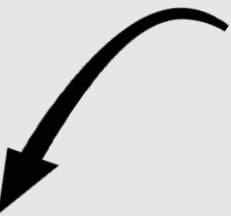
- `select(tree => tree.flightBooking.flights): Observable<Flight[]>`
- `dispatch(flightsLoaded({ flights })))`

Registering @ngrx/Store in AppModule

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

Registering @ngrx/Store in Feature Module

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



State branch for feature

DEMO

LABS

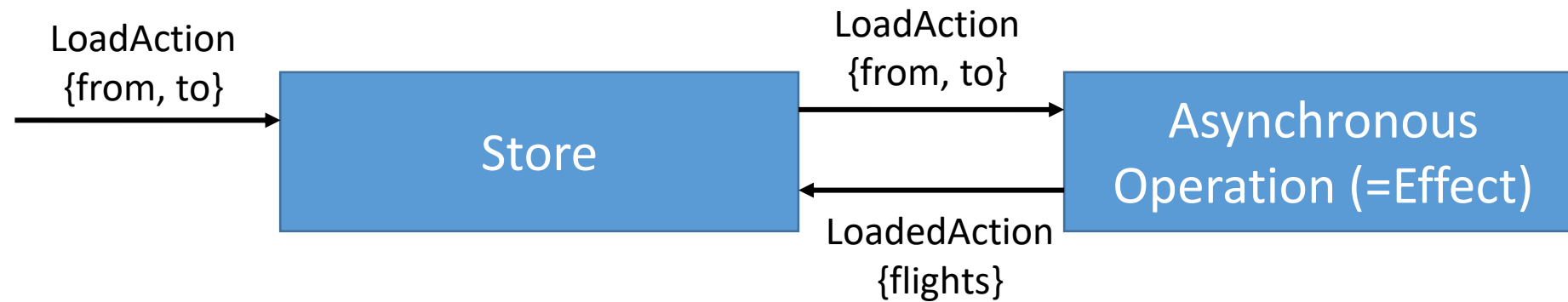
Effects



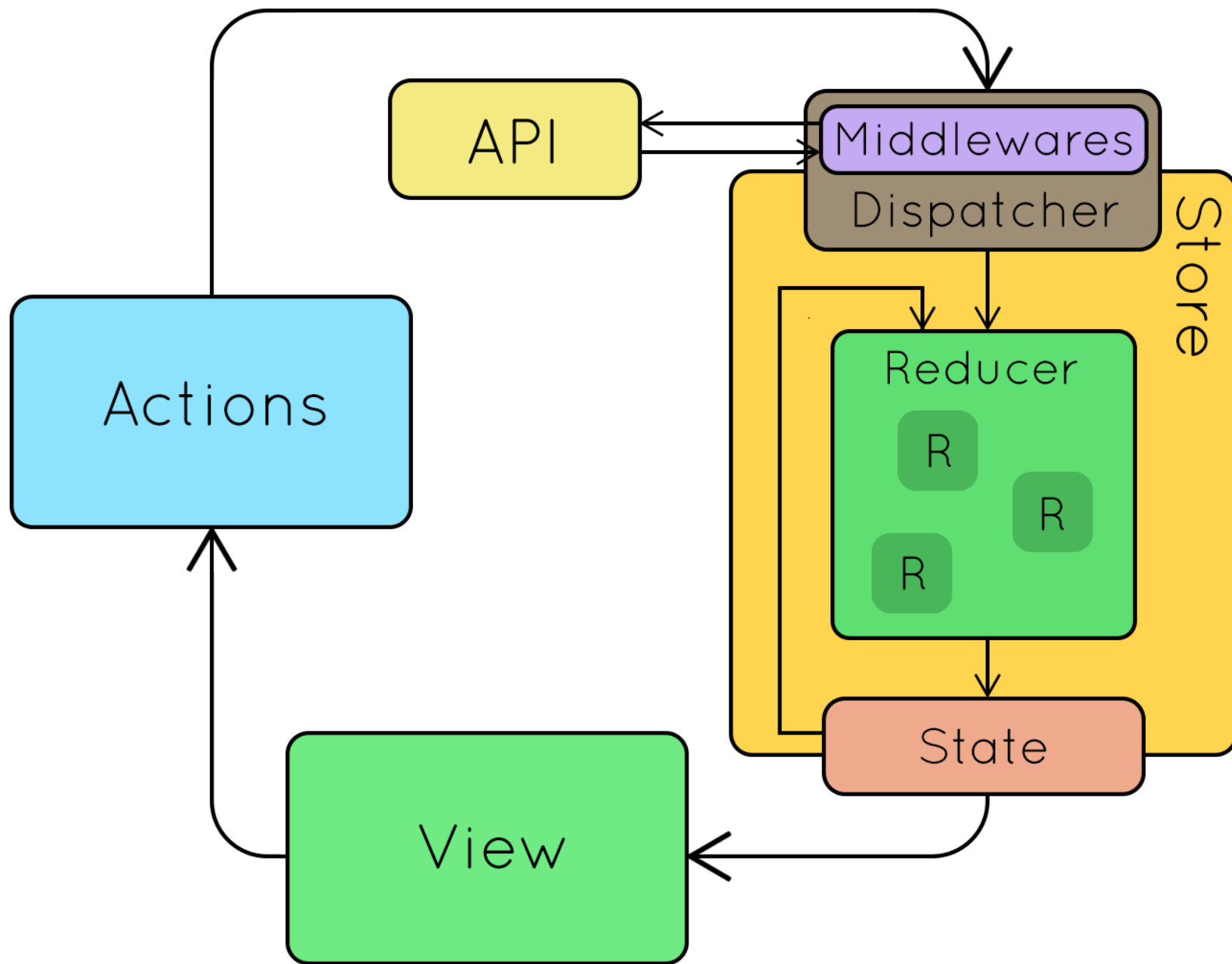
Challenge

- 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})))));

}
```

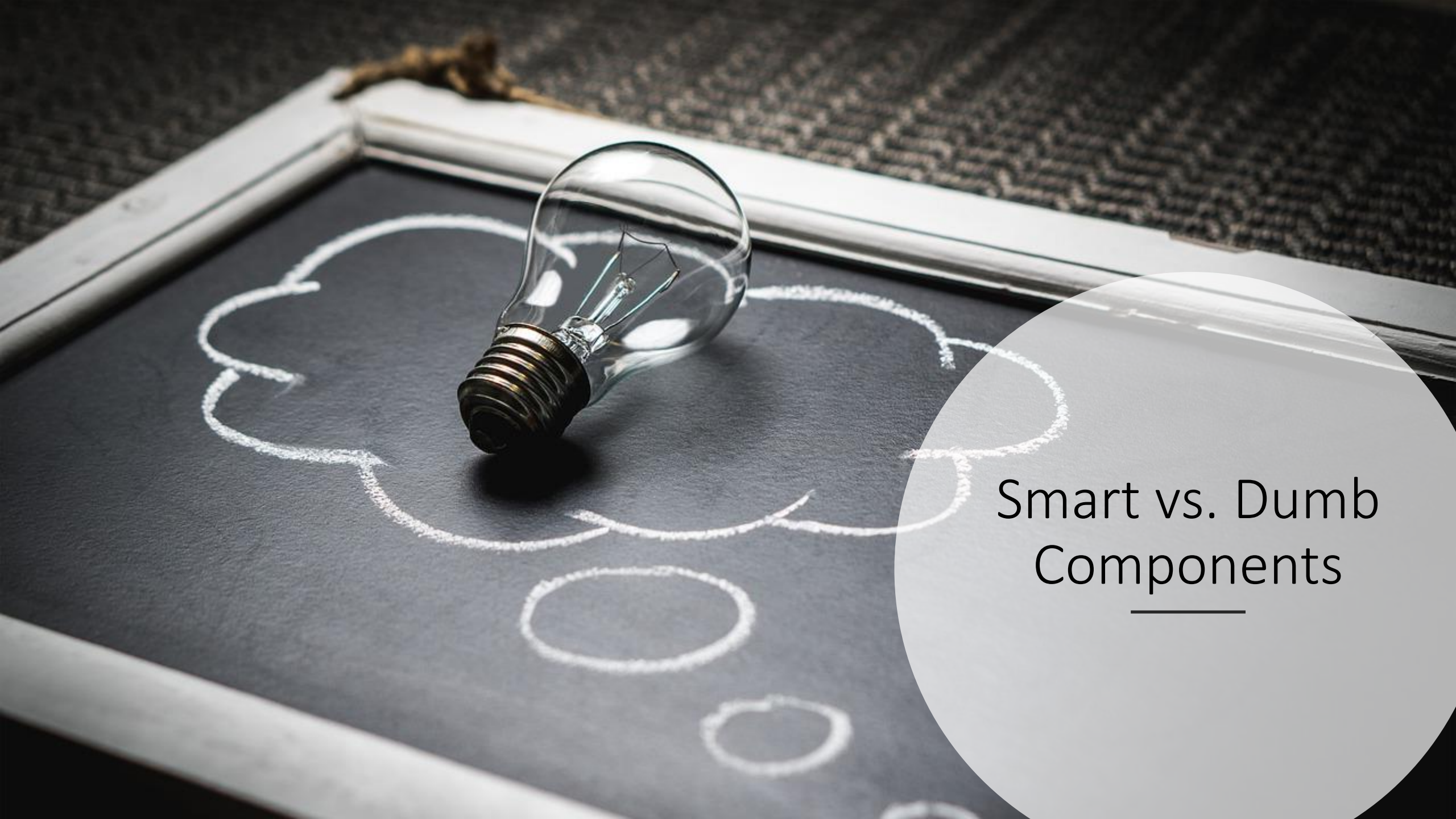
DEMO

LAB

@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



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"