ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

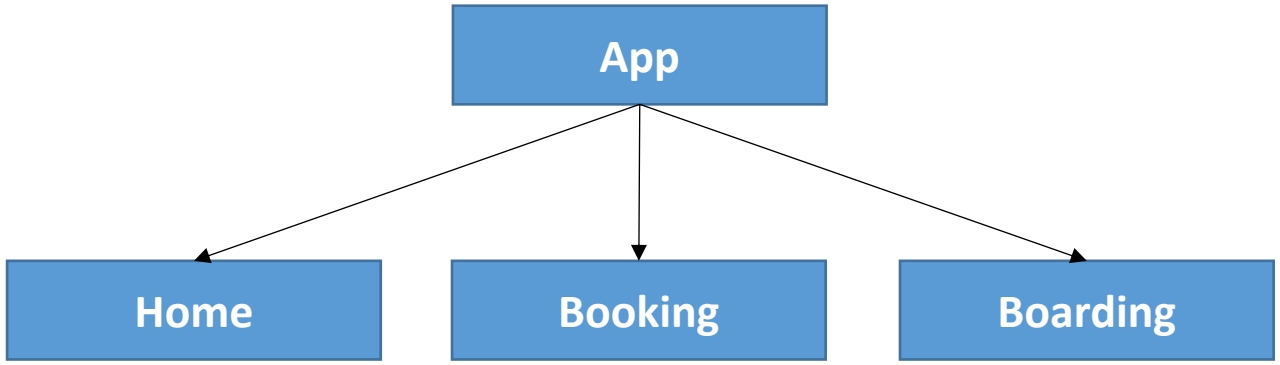
State Management
With NgRx

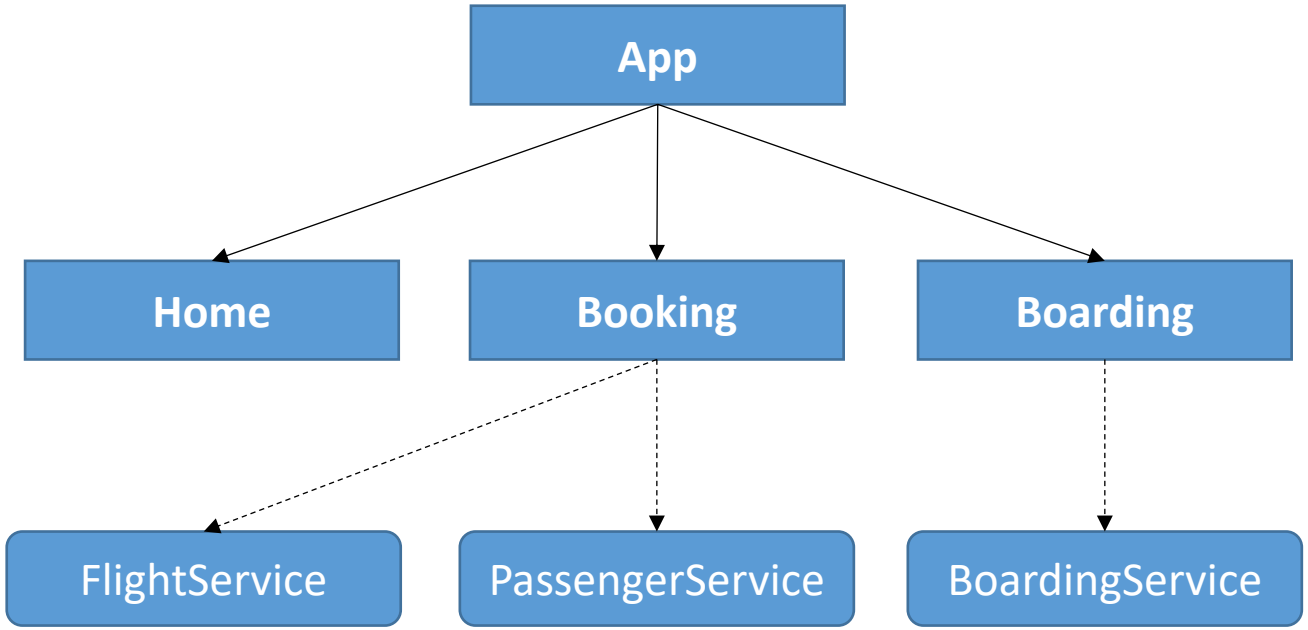
Alex Thalhammer

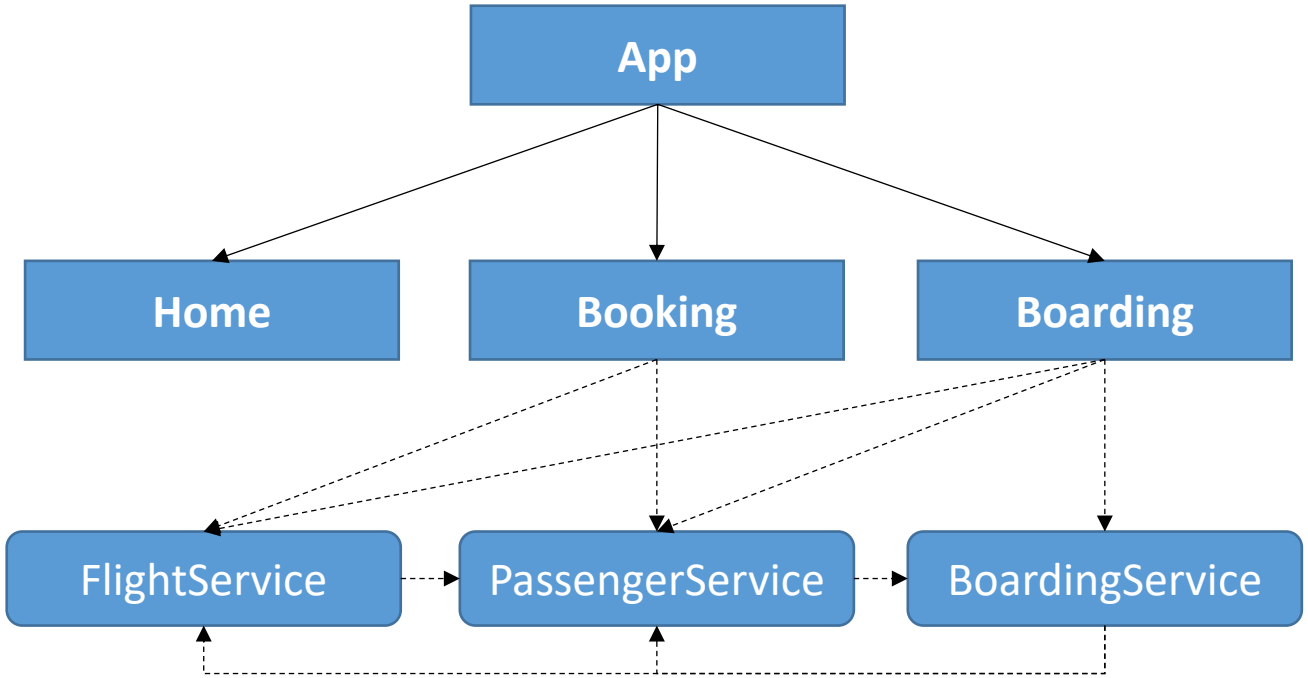
Outline

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

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

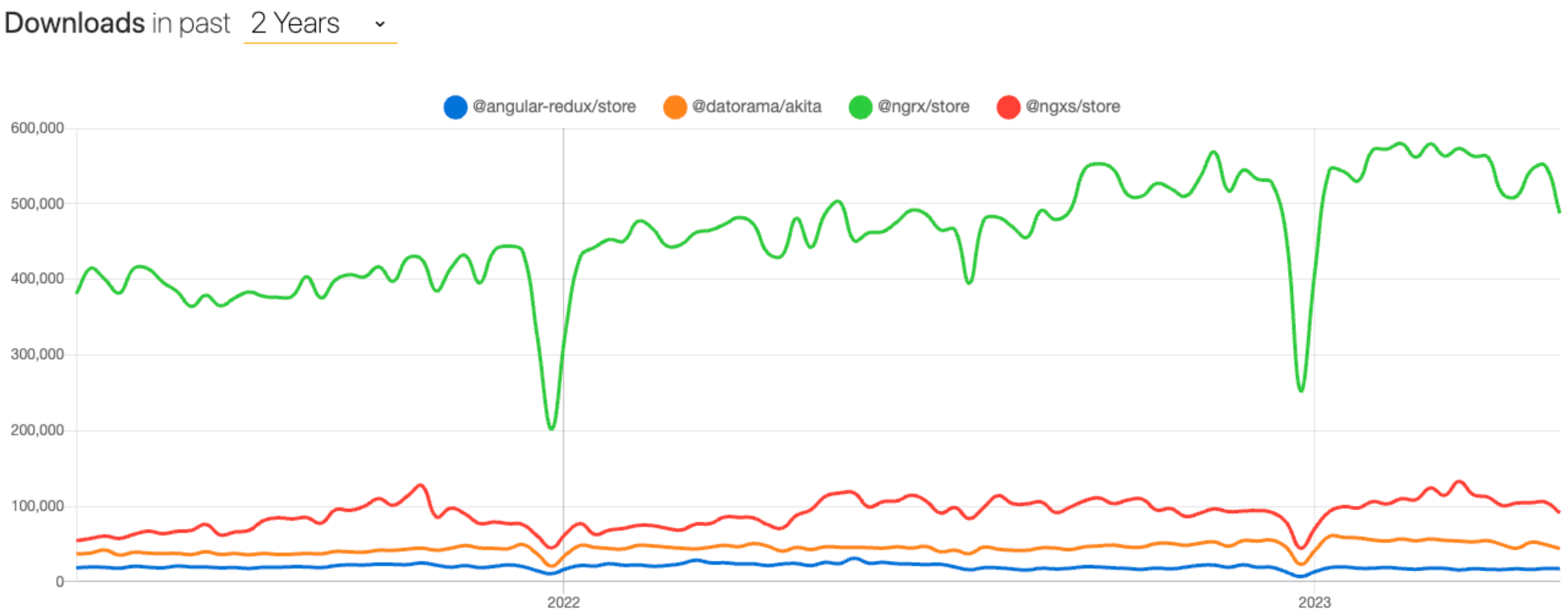
- Implementation used here: @ngrx/store
- Alternative: @ngxs/store
- Or: @dataroma/akita

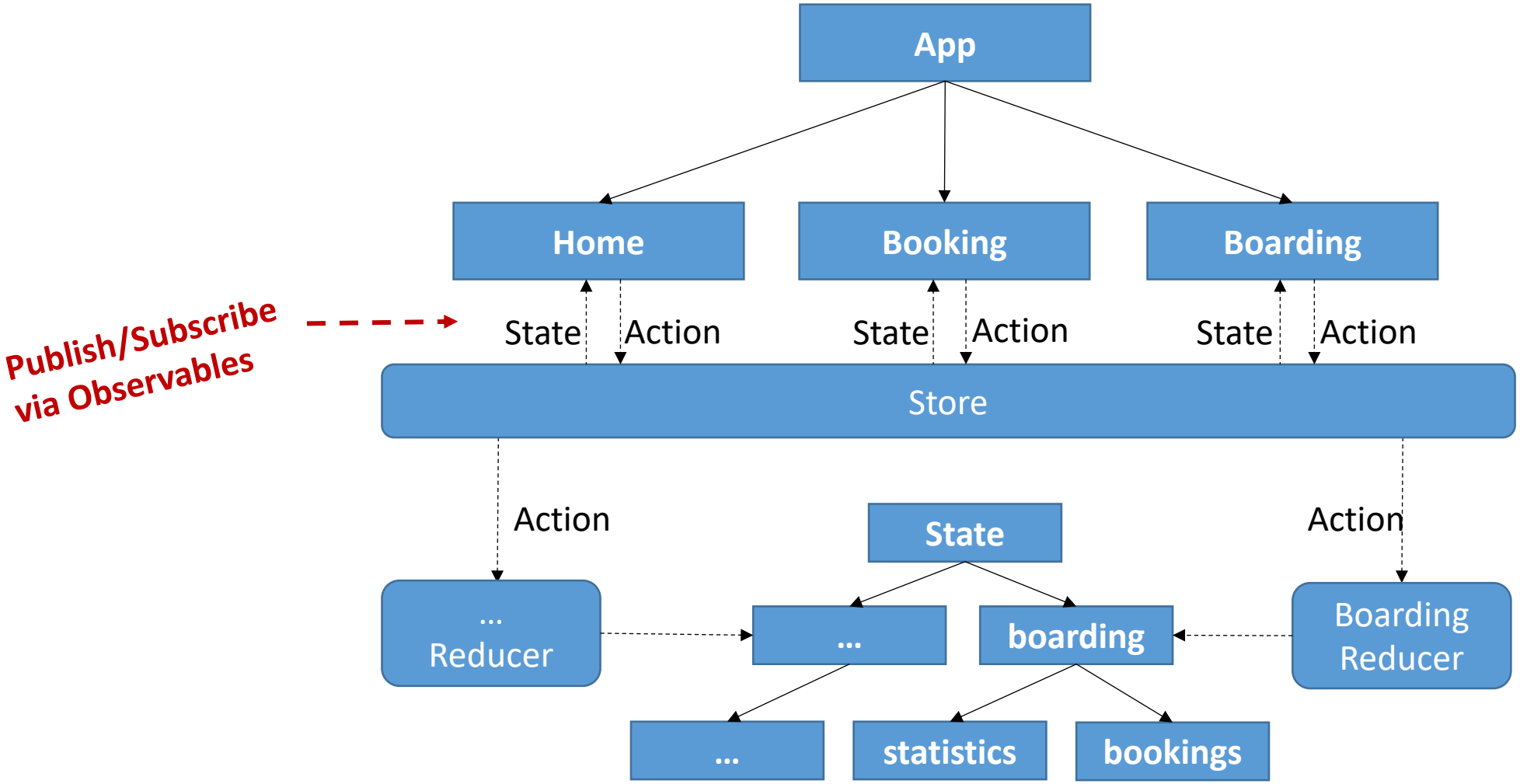
Alternatives

Alternatives

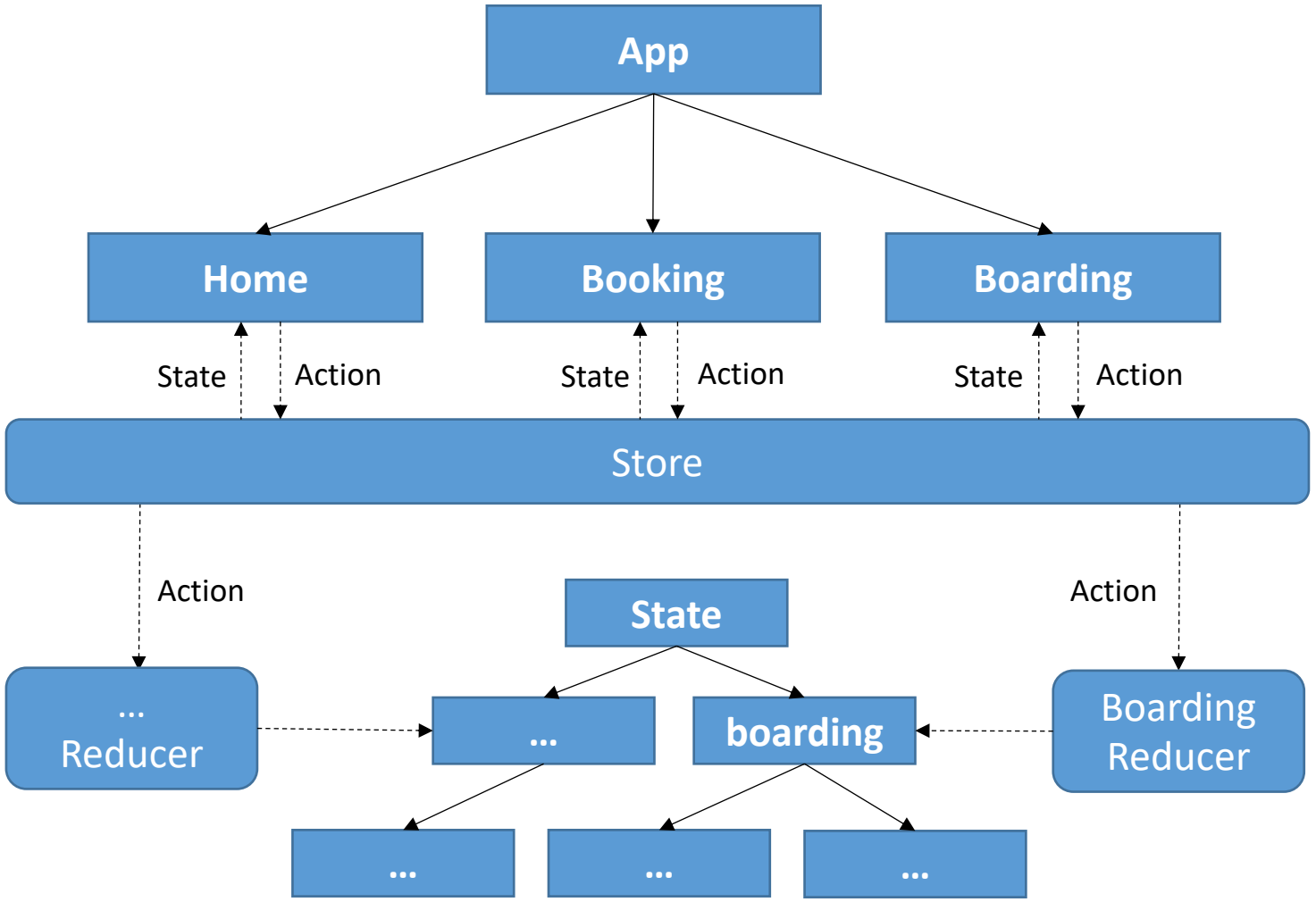
- NGXS
  - More Object-Oriented
  - Built-In Immutability
  - Reducer/Store also works with Asynchonity – no need for Effects
- Akita
  - Not based on Redux, only RxJS
  - Works very similar to @ngrx/entity
  - Can be seen as a simplified subset of ngrx
  - Very limited in terms of customisation

Alternatives



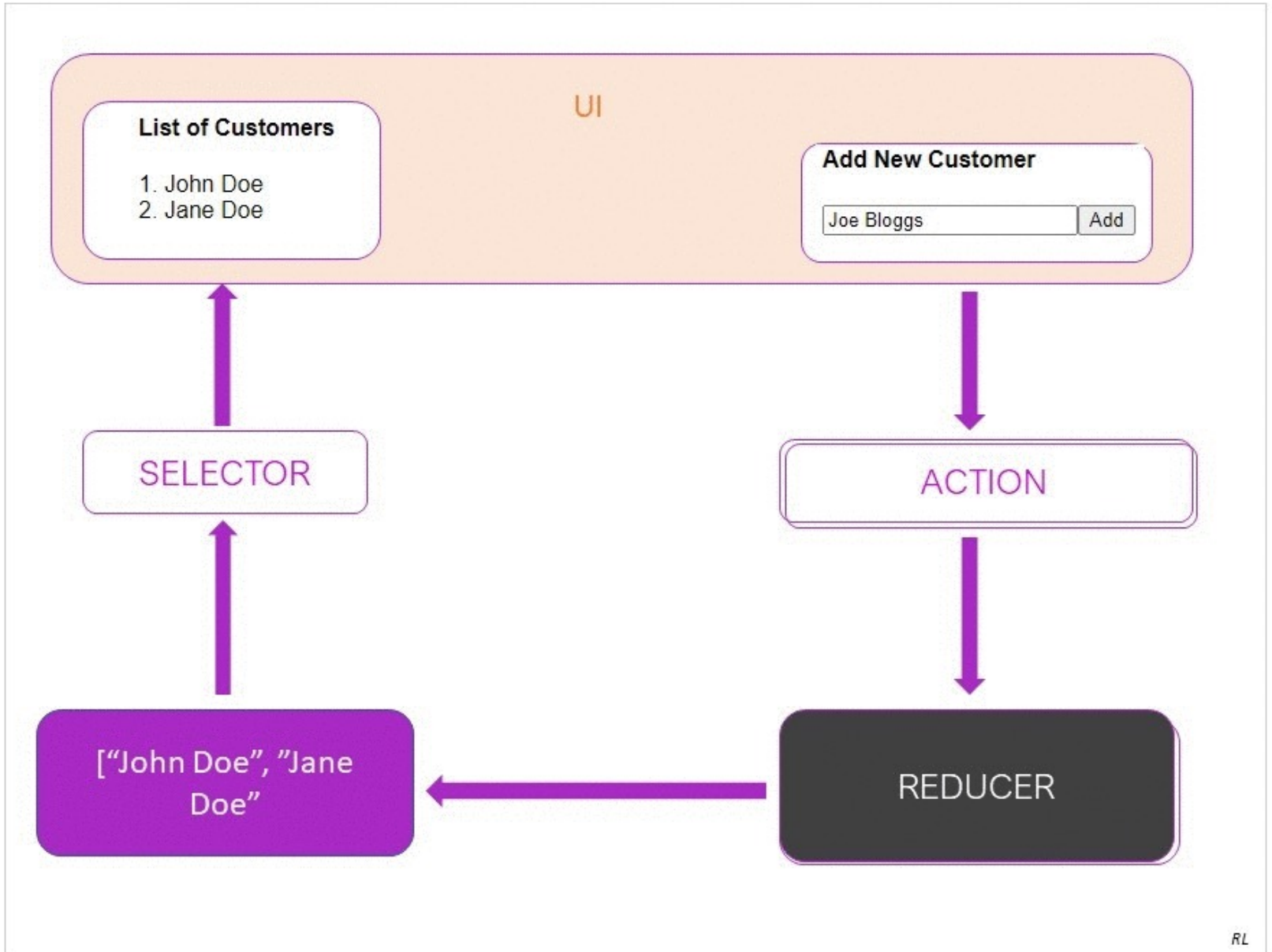


Single Immutable State Tree



Single Immutable State Tree

- Single "source of truth"
- Easy to understand & debug
- Easy onboarding of devs
- Easier testing
- Well structured
- Prevents cycles
- Performance
  - Reactivity (Selects)
  - Immutability (State)



https://dzone.com/articles/angular-app-state-management-with-ngrx


Store

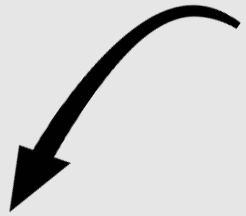
Store

- 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 { }
```



Just reducers for shared state

Registering @ngrx/store

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

needs to be inserted after using schematic/generator in the lab!

@ngrx/store-devtools

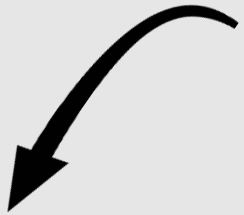
@ngrx/store-devtools

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

ngrx and Feature Modules

Registering @ngrx/Store

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



State branch for feature

State

FeatureState

```
export const flightBookingFeatureKey = 'flightBooking';

export interface FlightBookingAppState {
  [flightBookingFeatureKey]: State;
}

export interface State {
  flights: Flight[];
}
```

FeatureState

```
export const flightBookingFeatureKey = 'flightBooking';

export interface FlightBookingAppState {
  [flightBookingFeatureKey]: State;
}

export interface State {
  flights: Flight[];
}

export const initialState: State = {
  flights: []
};
```

AppState

```
export interface State {
  // empty
}
```

AppState

```
export interface State {
  flightBooking: FlightBookingState; // feature state
  currentUser: UserState; // shared state
}
```


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

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

Reducer

- responsible for transitions in your application

  from current state -> next state

- Signature of "on"

  (currentState, action) => newState

Reducer for FlightBookingState

```
export const reducer = createReducer(
  initialState,

  on(FlightBookingActions.flightsLoaded, (state, { flights }): State => {
    return { ...state, flights };
  }),
);
```

Demo & Lab

NgRx Store

ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE ARCHITECT

Selectors

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE
ARCHITECT

Selectors

- pure functions for obtaining slices of store state (also "state streams")
- select((tree) => tree.flightBooking.flights): Observable<Flight[]>
- We can use createSelector or createFeatureSelector
- Can be used for filtering and mapping
- Built-in memoization

Using selectors

```
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

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE
ARCHITECT

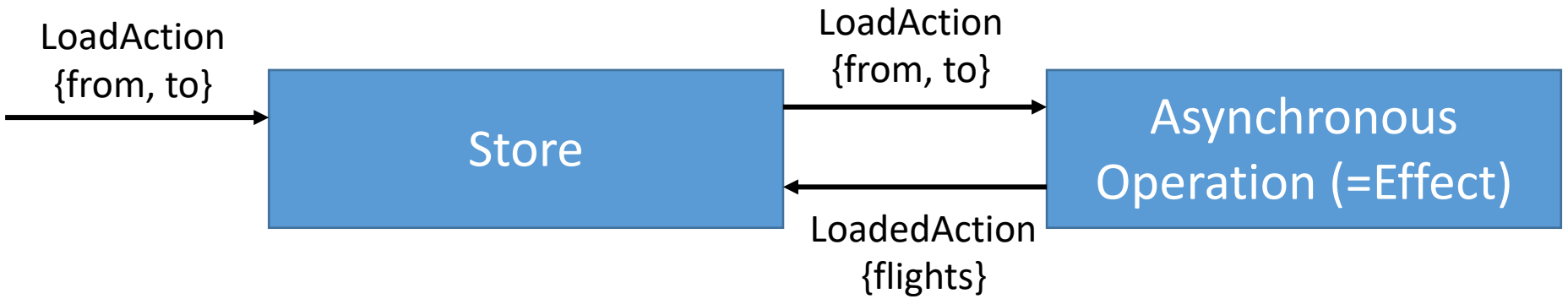
Effects

Challenge

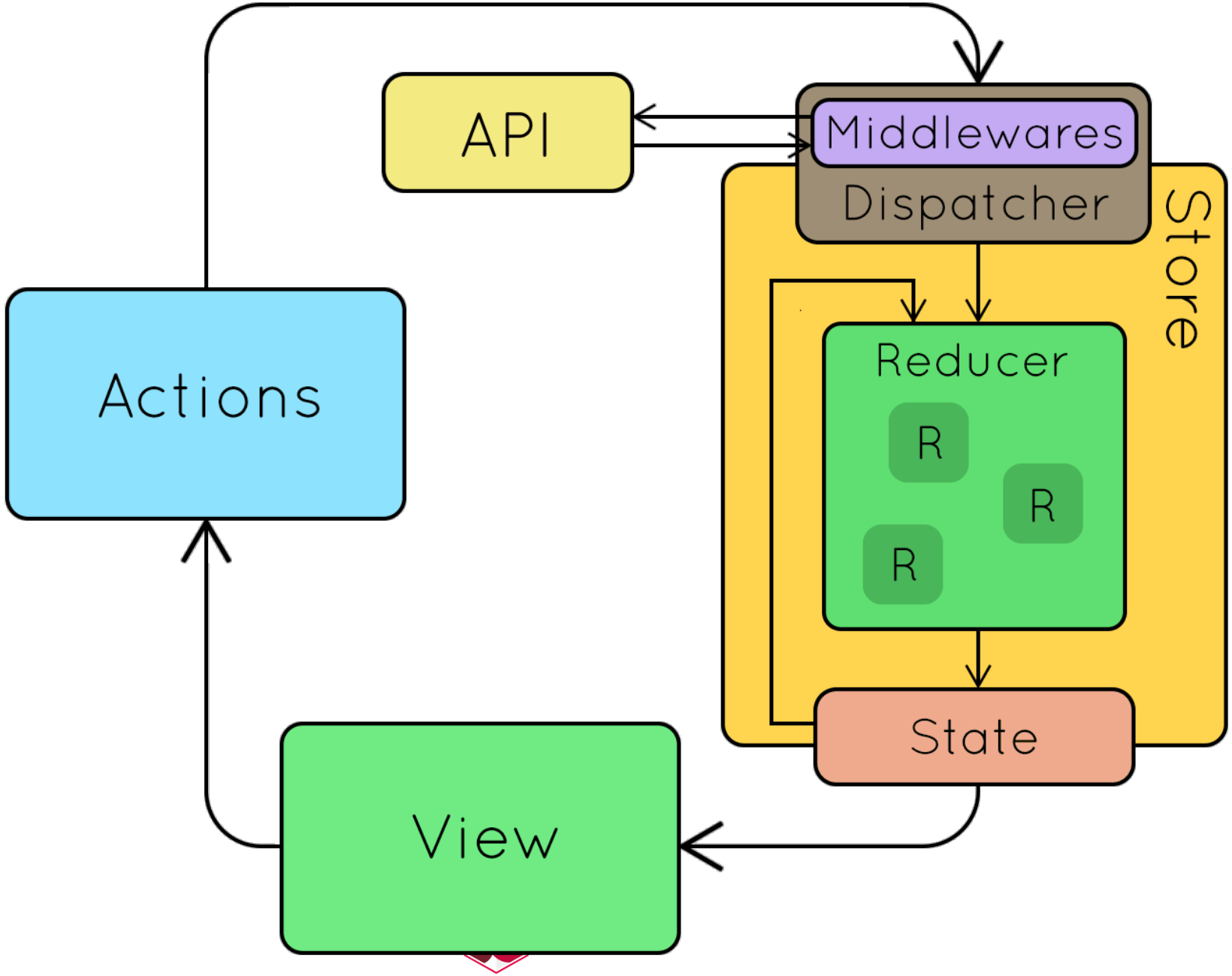
- Reducers are synchronous by definition

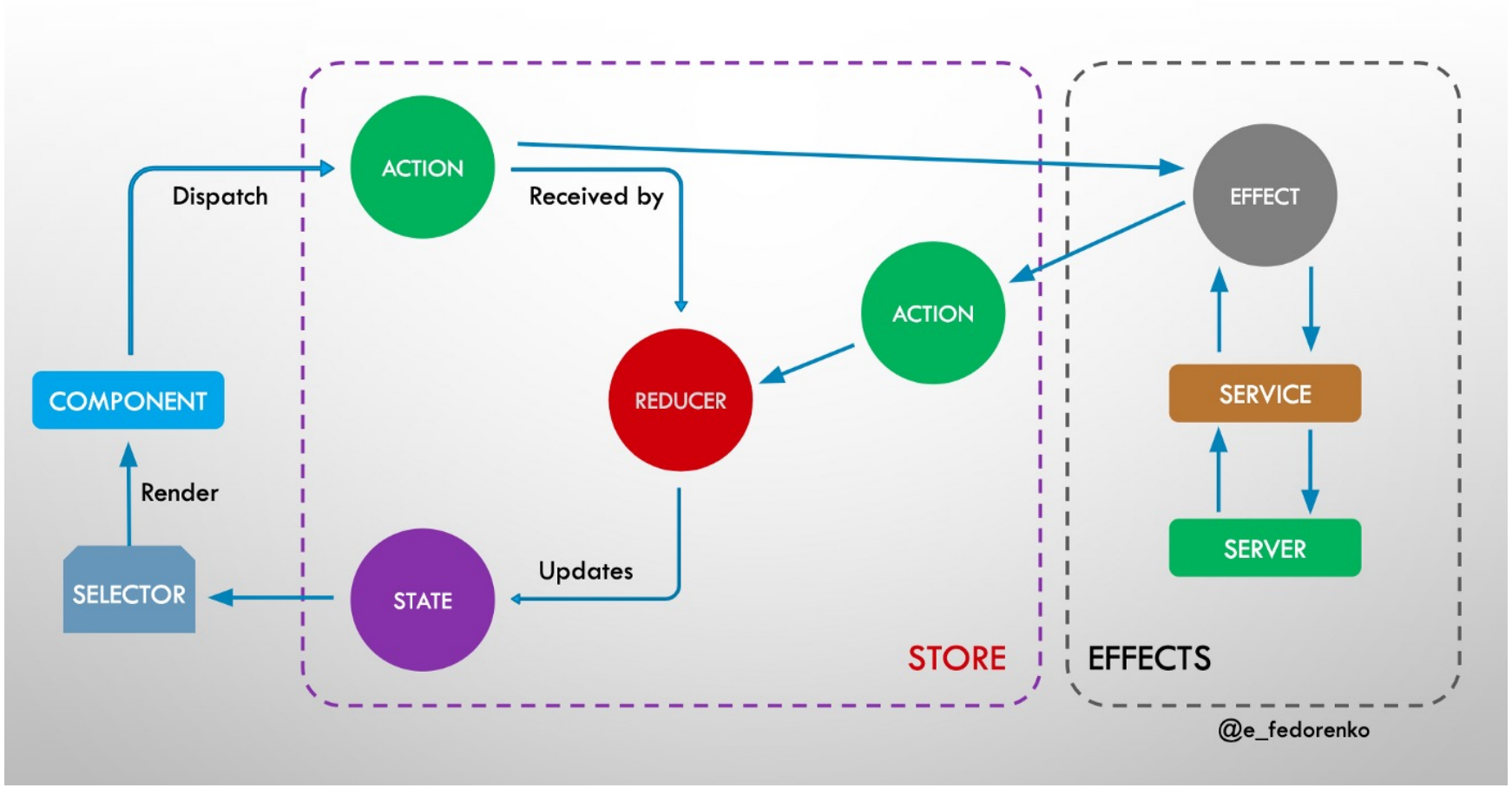
- What to do with **asynchronous** operations?

Solution: Effects



ng add @ngrx/effects


API
Middlewares
Dispatcher
Store
Reducer
R
R
R
Actions
State
View



https://medium.com/angular-in-depth/how-i-wrote-ngrx-store-in-63-lines-of-code-dfe925fe979b

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 {
  loadFlights$ = createEffect(
    (): Observable<any> =>
      this.actions$.pipe(
        ofType(FlightBookingActions.loadFlights),
        […]
      )
  );

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

Implementing Effects

```
@Injectable()
export class FlightBookingEffects {
  loadFlights$ = createEffect(
    (): Observable<any> =>
      this.actions$.pipe(
        ofType(FlightBookingActions.loadFlights),
        switchMap((a) => this.flightService.find(a.from, a.to, a.urgent)),
        map((flights) => FlightBookingActions.flightsLoaded({ flights }))
      )
  );

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

Implementing Effects - AppModule

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

Implementing Effects - FeatureModule

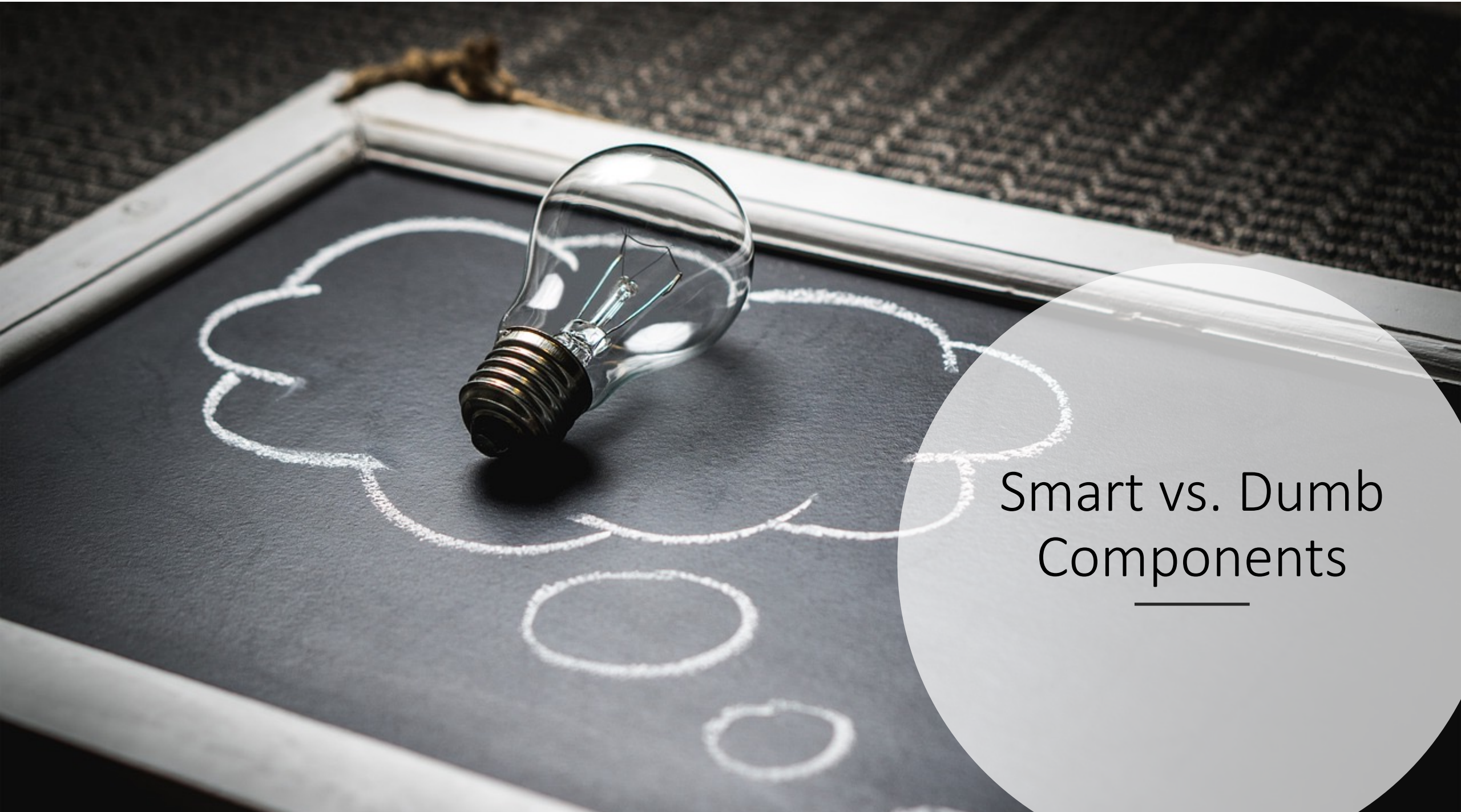
```
@NgModule({
  imports: [
    […]
     StoreModule.forFeature('flightBooking', flightBookingReducer),
     EffectsModule.forFeature([FlightBookingEffects])
  ],
  […]
})
export class FeatureModule {
}
```

Demo & Lab

NgRx Effects

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	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/store and
- https://ngrx.io/guide/component-store/comparison