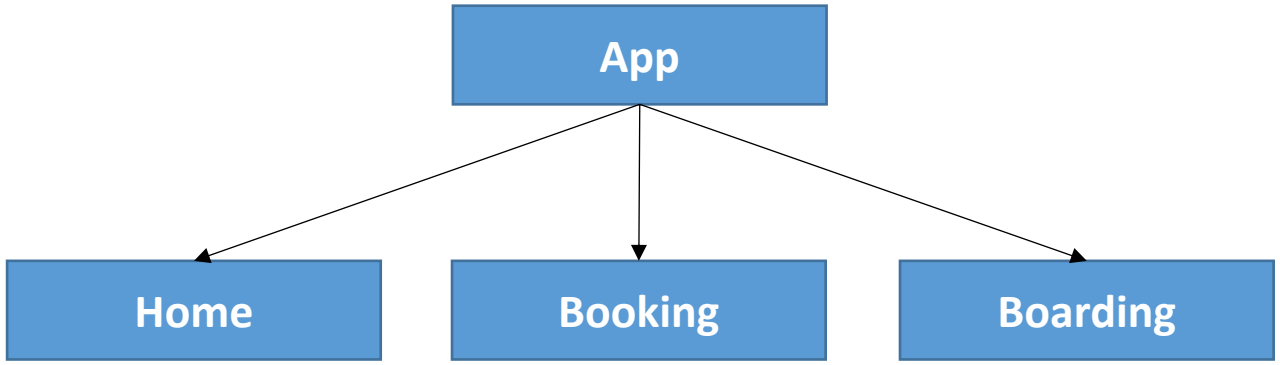
Basic State Management with Redux and @ngrx/store

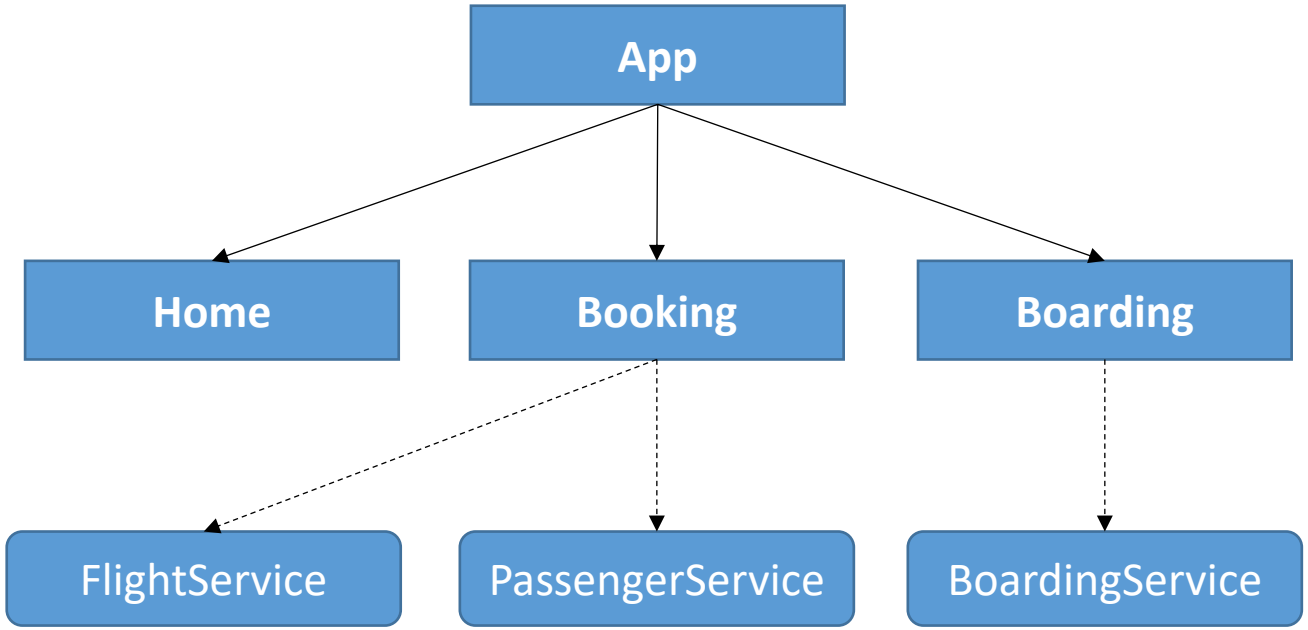
Alex Thalhammer

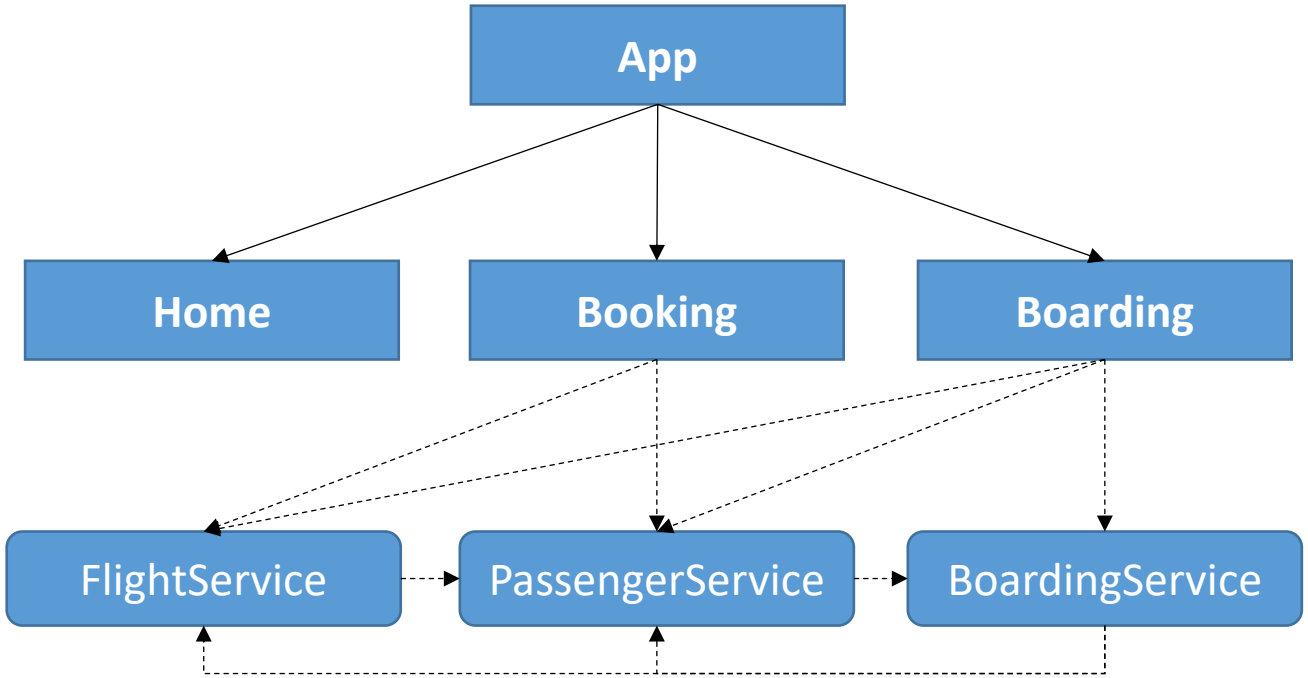
Outline

- 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

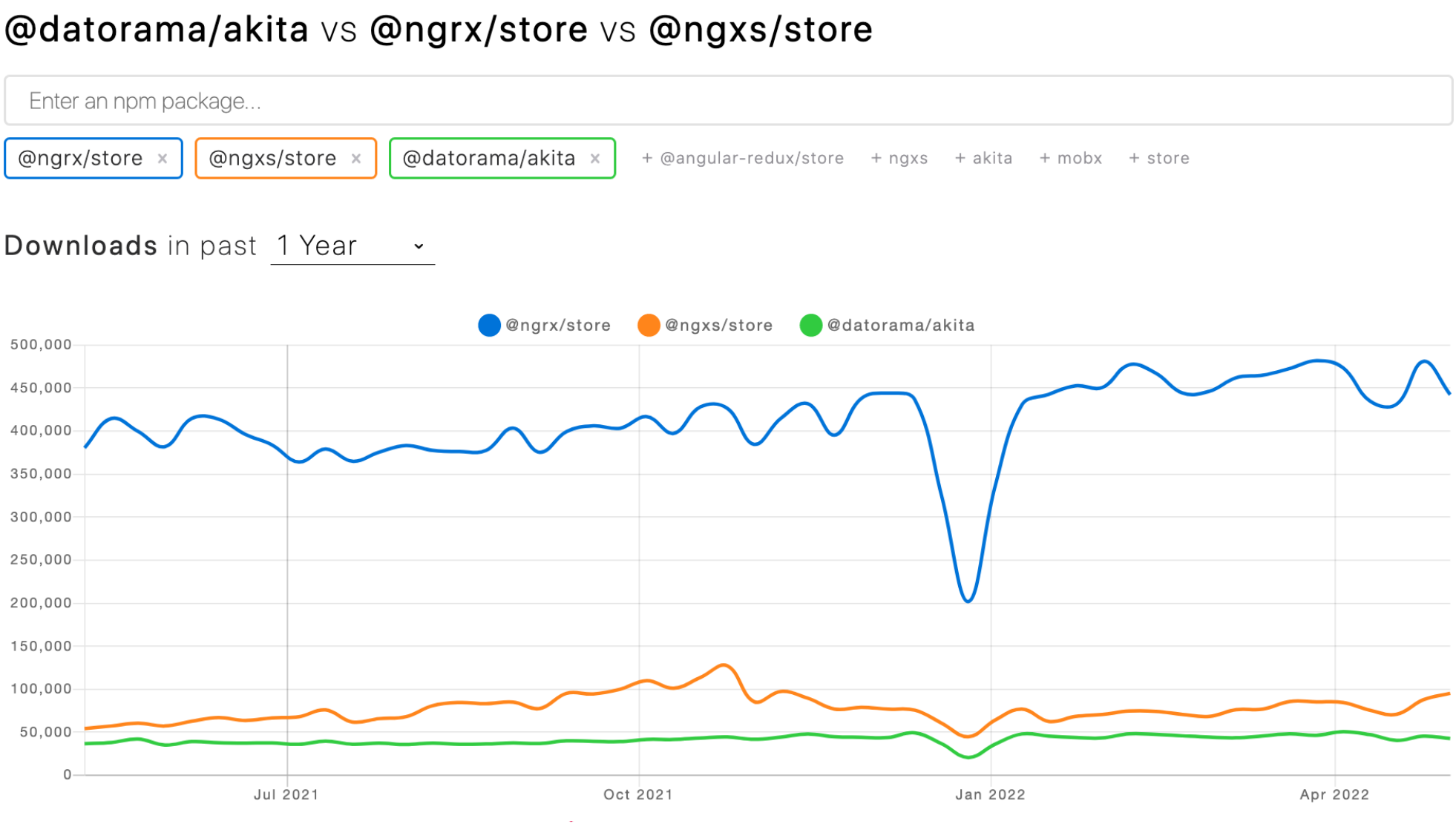
- Implementation used here: @ngrx/store
- Alternative: @ngxs/store
- Or: @dataroma/akita

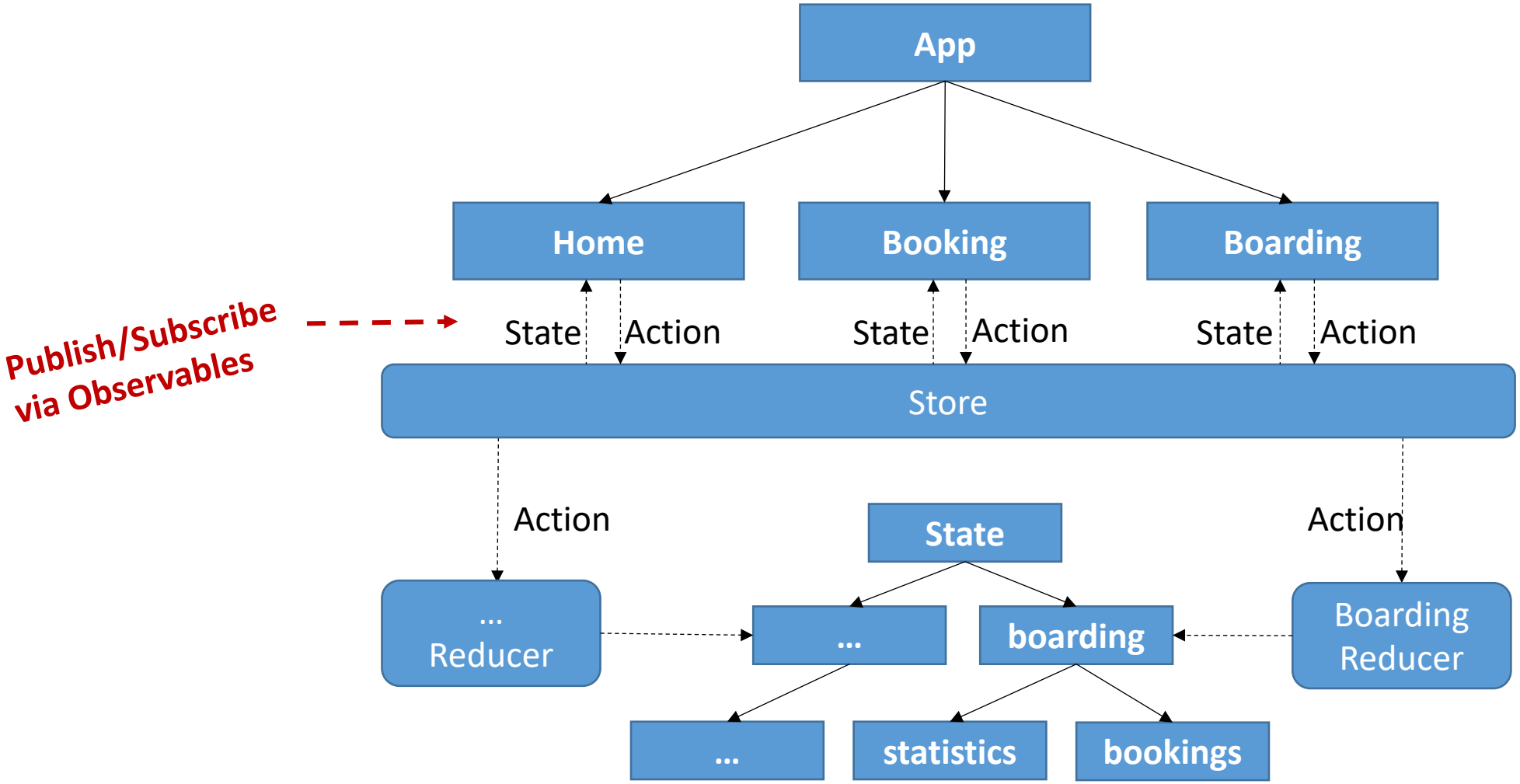
Alternatives

Alternatives

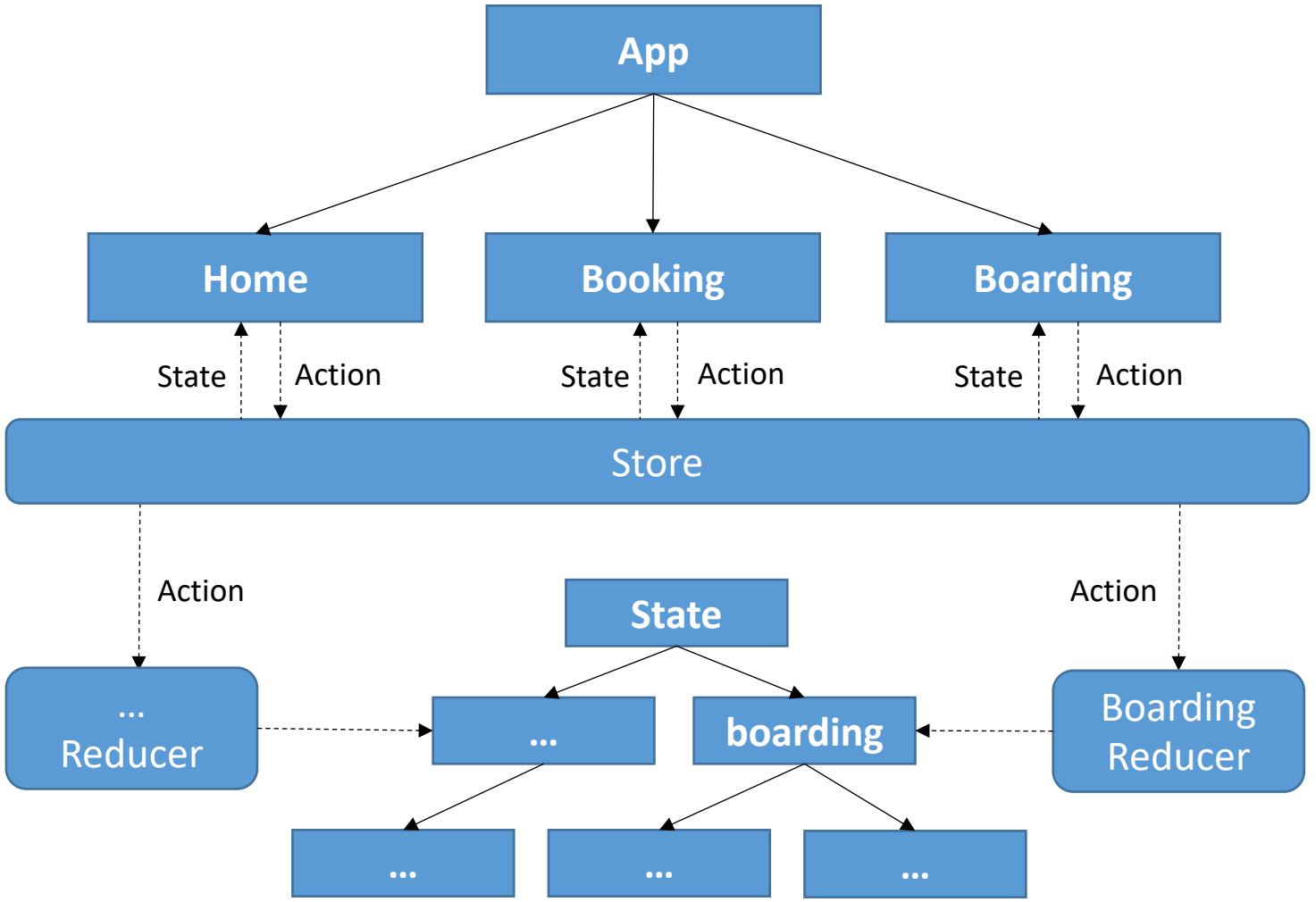
- NGXS
  - More Object-Oriented
  - Built-In Immutability
  - Reducer/Store also works with Asynchonity – no need for Effects
- Akita
  - Not based on Redux, only RxJS
  - Works very similar to @ngrx/entity
  - Can be seen as a simplified subset of ngrx
  - Very limited in terms of customisation

Alternatives



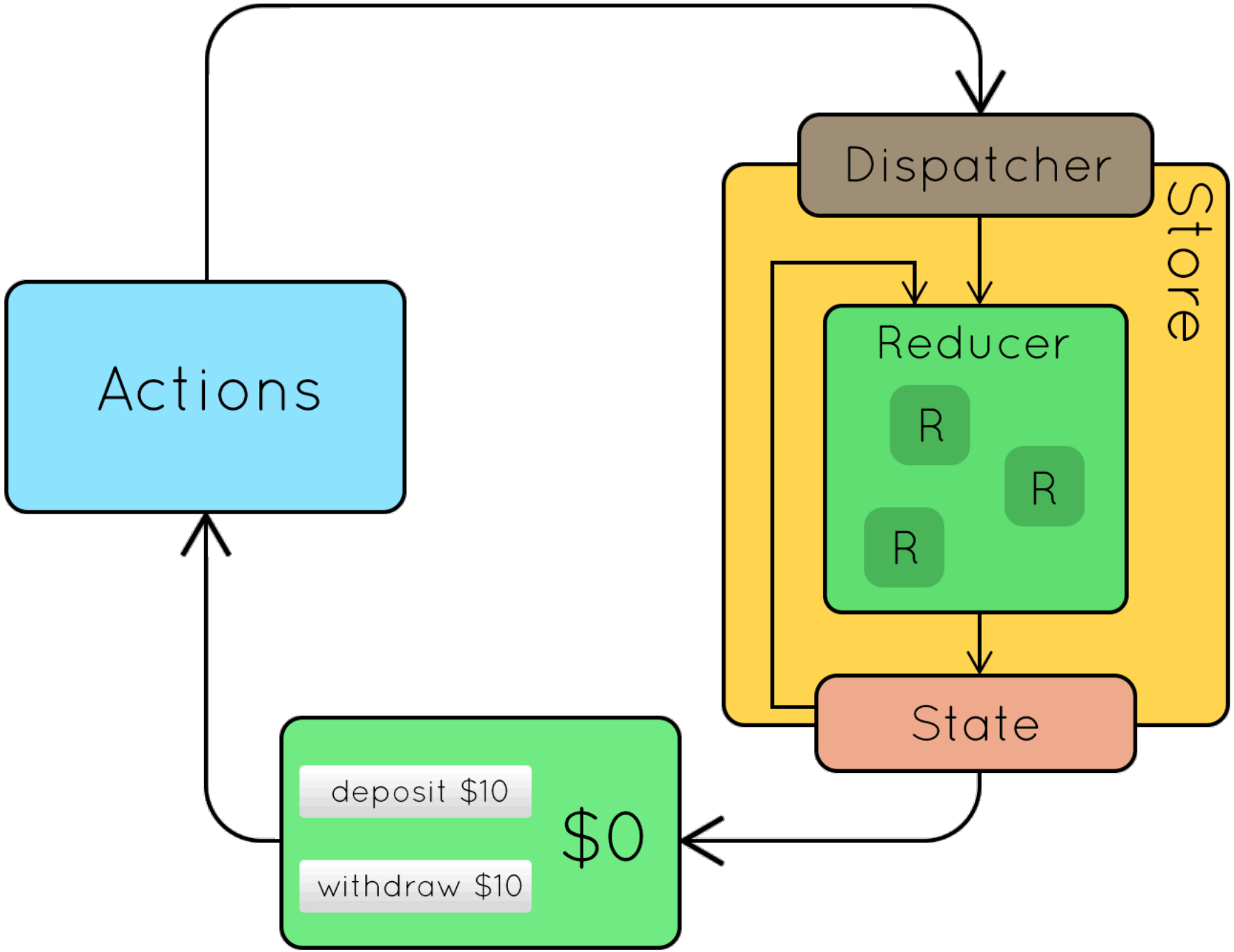


Single Immutable State Tree



Single Immutable State Tree

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


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

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
}
```


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

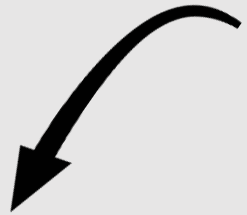
@ngrx/store-devtools

- 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

Demo & Lab

NgRx Store

ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE ARCHITECT

Selectors

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

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

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE
ARCHITECT

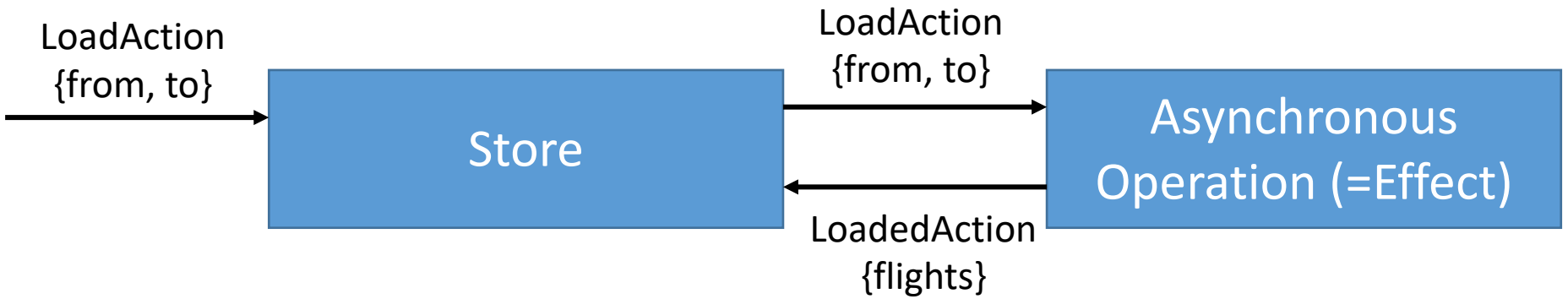
Effects

Challenge

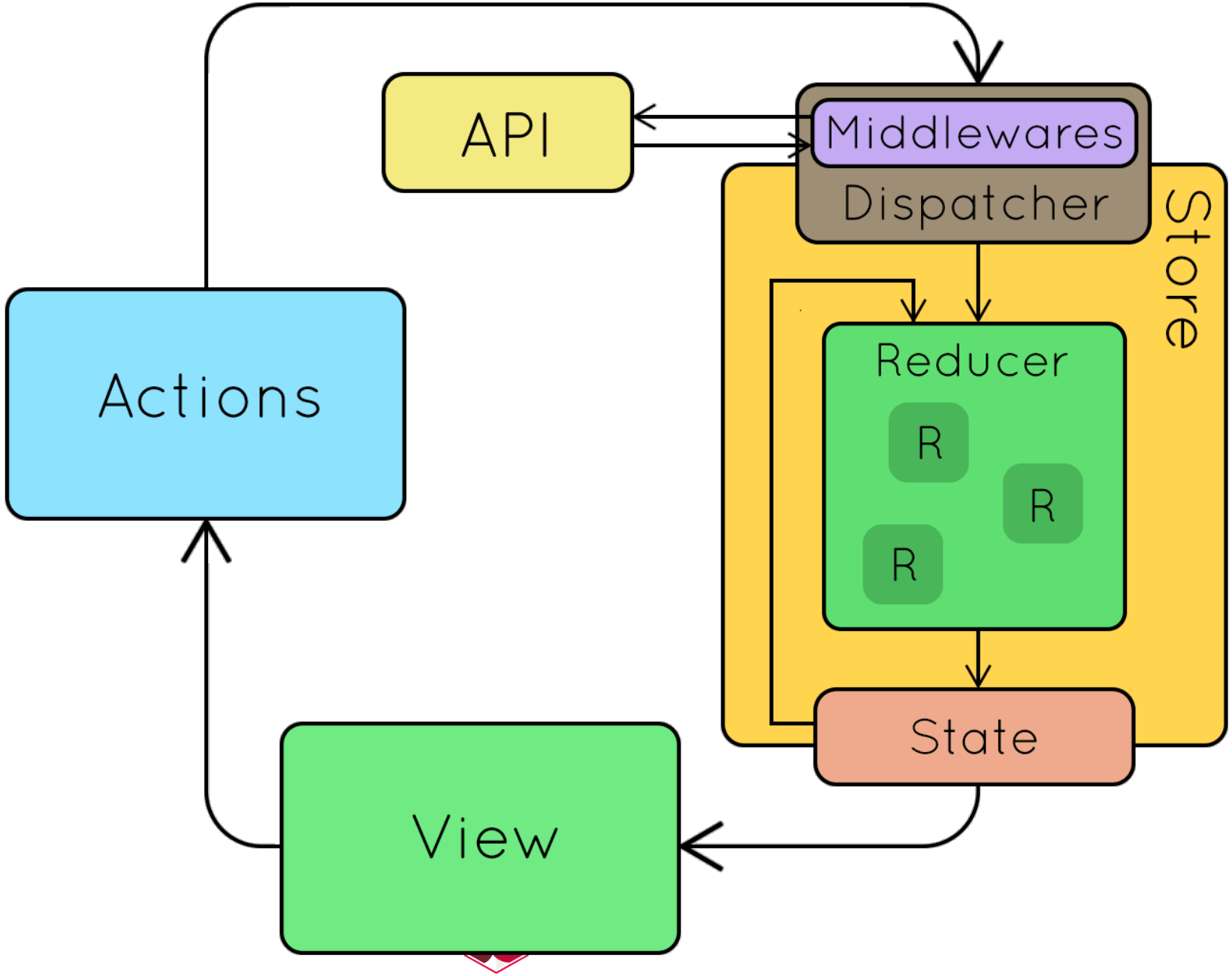
- Reducers are synchronous by definition

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

Solution: Effects



ng add @ngrx/effects


API
Middlewares
Dispatcher
Store
Reducer
R
R
R
Actions
State
View

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

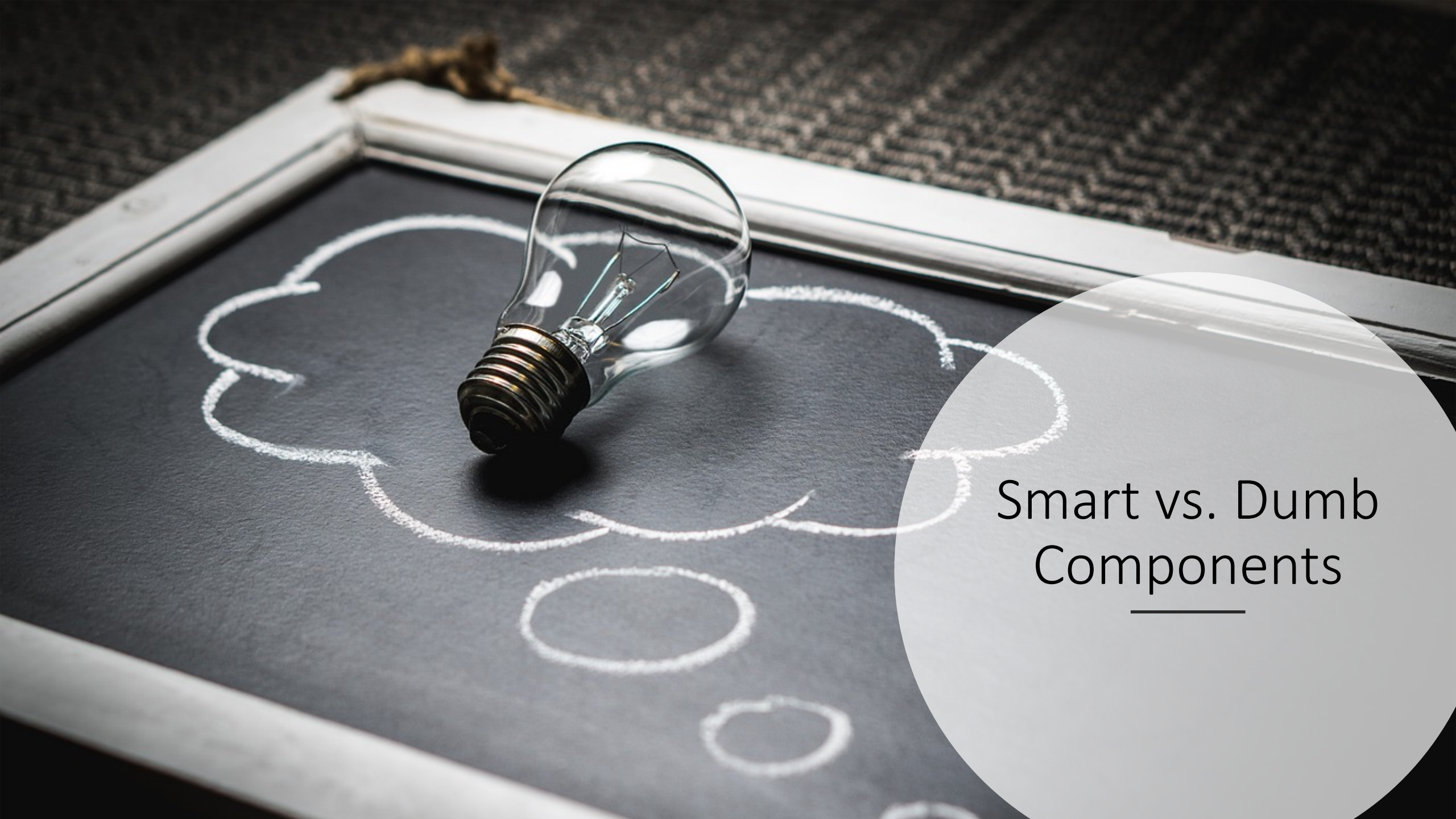
```
@NgModule({
  imports: [
    […]
    EffectsModule.forFeature([FlightBookingEffects])
  ],
  [...]
})
export class FeatureModule {
}
```

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/data/architecture-overview