



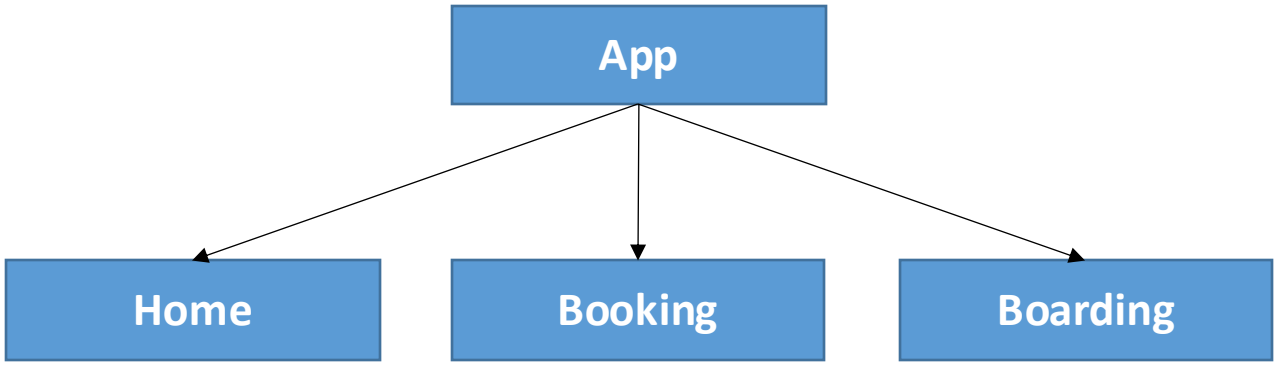
# Angular State Management with Redux and NgRx

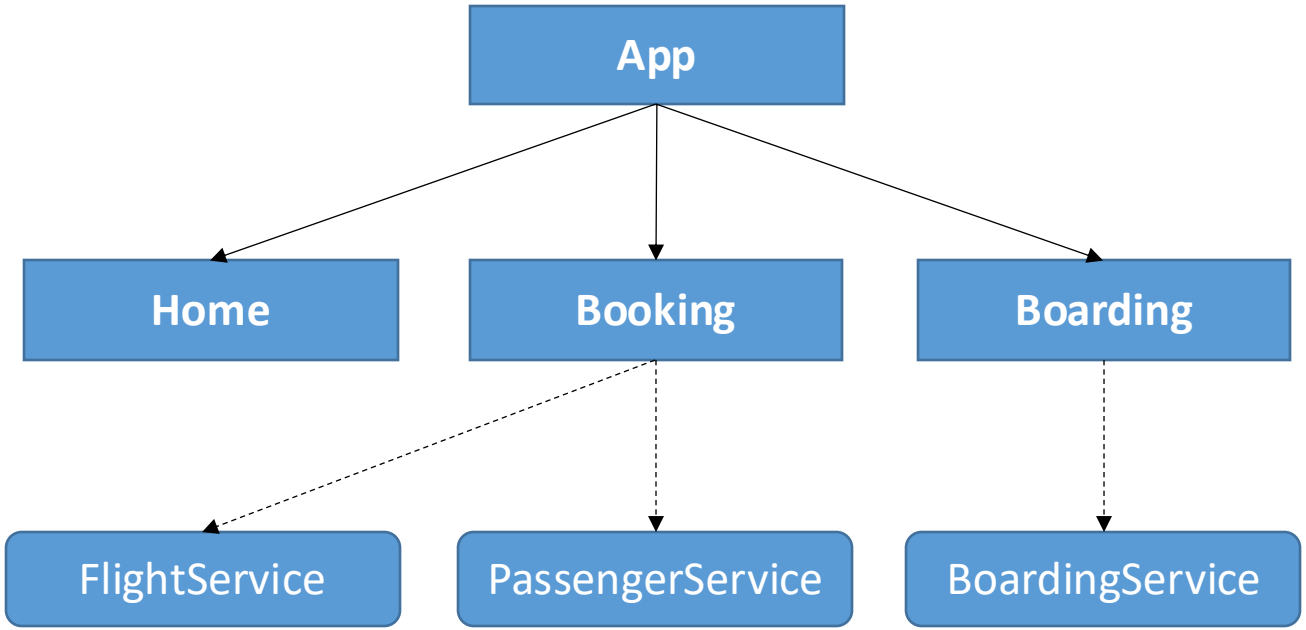
Hosted by Alexander Thalhammer
https://LXT.dev

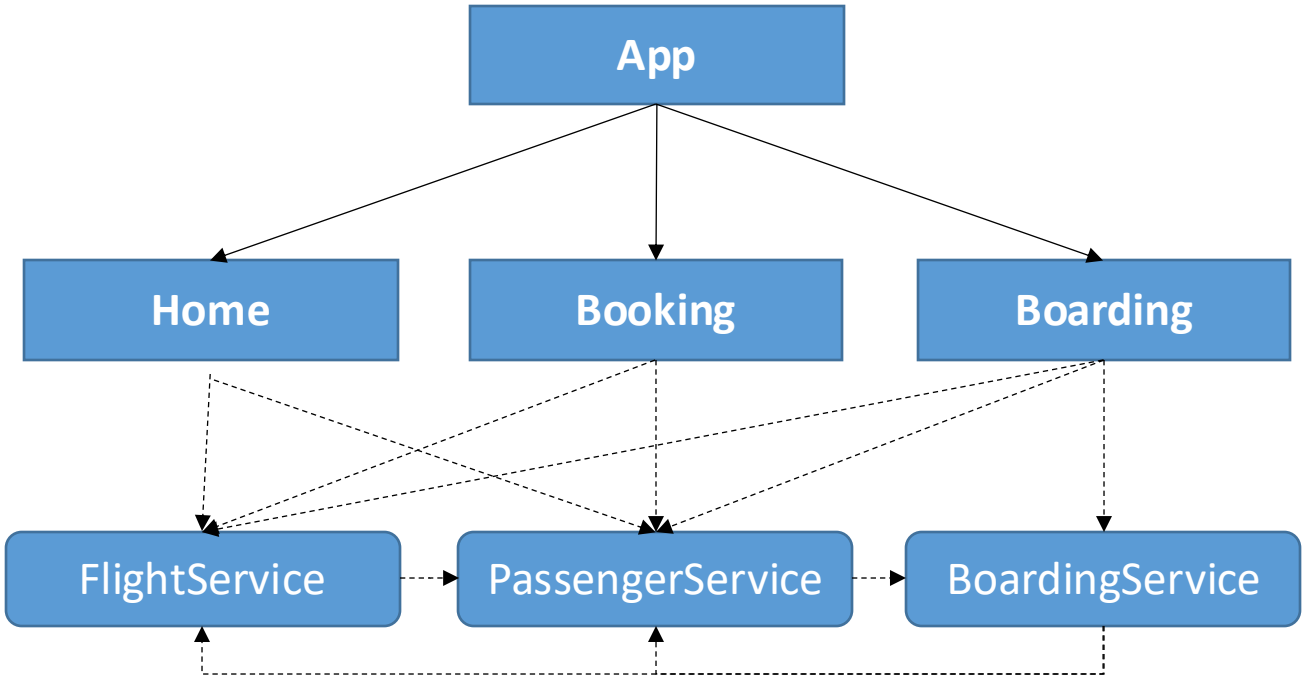
# Contents

- Motivation
  - Why State Management?
  - Why Redux pattern?
- State
- Redux
  - Actions
  - Reducer & Store
  - Selectors
  - Effects (async)

# Motivation


App
Home
Booking
Boarding


App
Home
Booking
Boarding
FlightService
PassengerService
BoardingService


App
Home
Booking
Boarding
FlightService
PassengerService
BoardingService

# Why State Management? I

- One source of truth ☺
- Good predictability
- Good performance
- Clear architecture
  - no discussions btw. Devs or Teams
- Better maintainability
- Very smooth w. Angular
  - ChangeDetectionStrategy.OnPush &
  - Signals

# Why State Management? II

- Easy to debug (with Redux DevTools)
- Easy to persist (e.g. localStorage)
- Easy to onboard new Devs
- Easy undo/redo
- Easy to test

# State Management cons

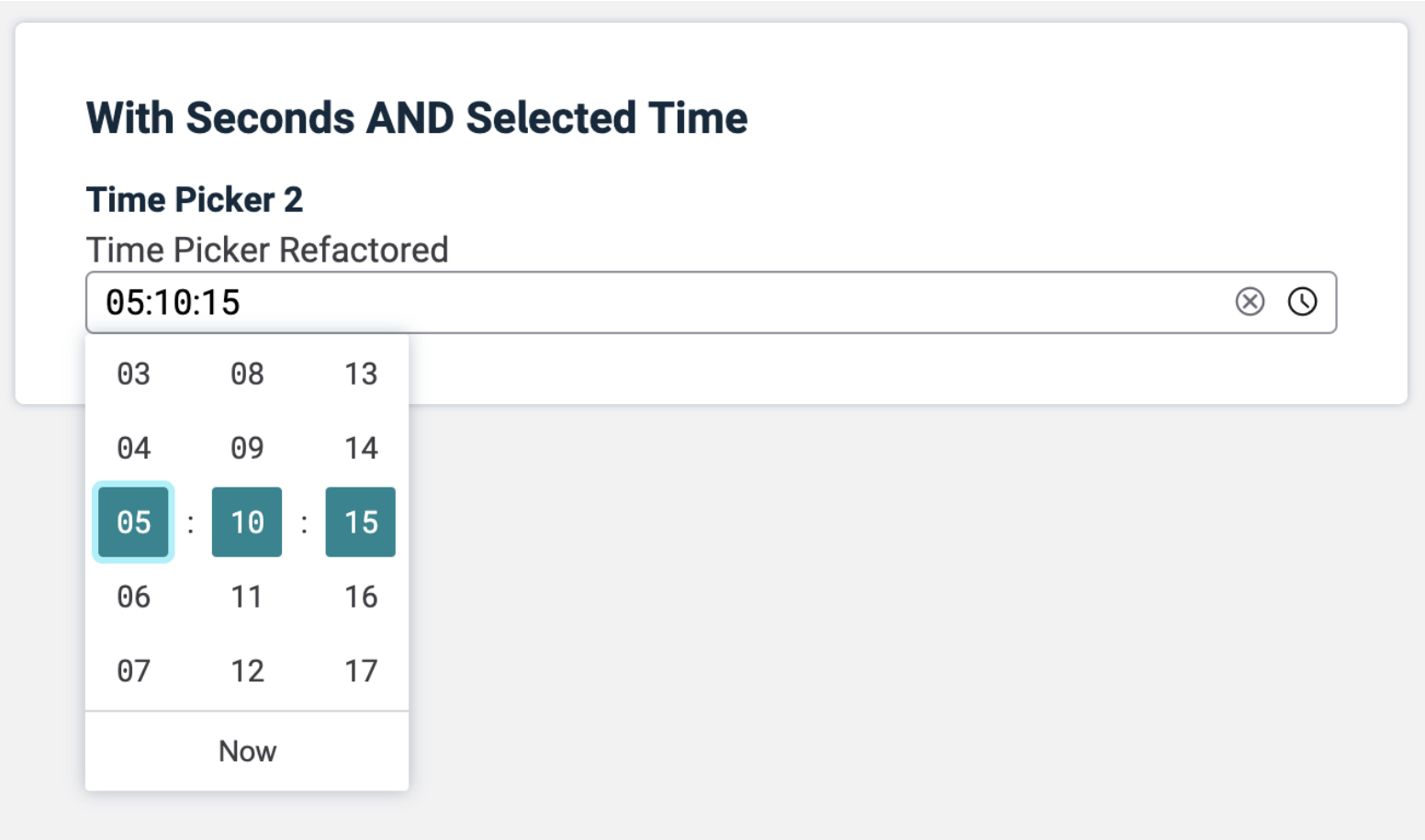
- Needs to be learned (steep curve like Angular)
- Strict architecture
- Less freedom
- Boilerplate

# When do I need State Management?

- Complex applications
  - Checkout process
  - Draft / Edit process (e.g. multiple comp. or serv. accessing the same thing)
  - Filters / pagination
  - State used by multiple routes
  - State retrieved (e.g. from API)

- Complex components
  - Container/Controller (for features)
  - real world example: TimePicker (internally in presentational component)

# Example: TimePickerComponent



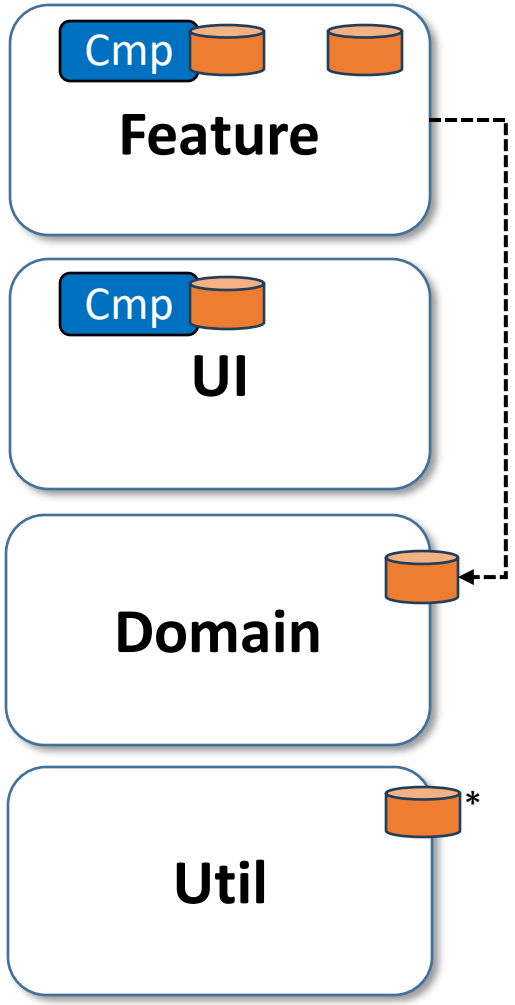

ANGULAR ARCHITECTS INSIDE KNOWLEDGE

Angular Software Tree GmbH

# Global vs feature vs local store

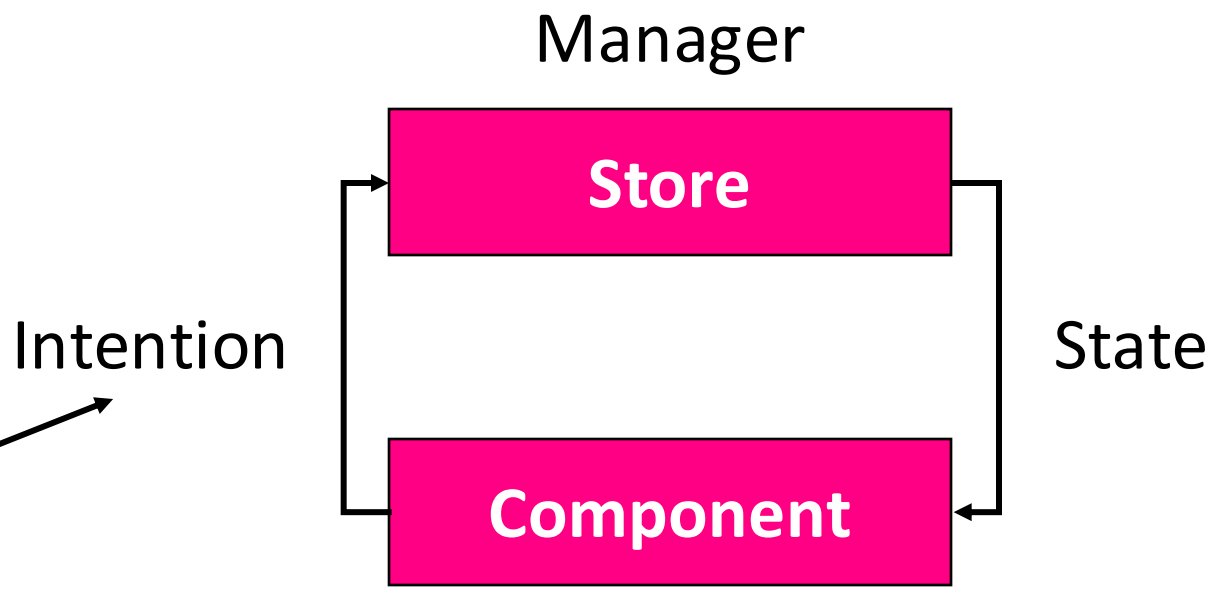
- Global store
  - classic approach
  - 1 app store, can be split into features
- Feature store
  - 1 store per feature (lib/folder)
  - Components that belong together share store
- Component store
  - 1 store for 1 component

# State Management and DDD




* seldom

# The store and the flow




- Action
- Event
- Message

- Immutables like
- Observables or
- Signals

# Which State Management solution?

BehaviorSubject(s)
in a service

ComponentStore (local)
@ngrx/component-store

NGXS (global)
@ngxs/store

ReduxStore (global)
@ngrx/store

Signal(s)
in a service

SignalStore
@ngrx/signals

ReactiveComponentState
@rx-angular/state

Lightweight ⟷ Power

# Which State Management solution?

BehaviorSubject(s)
in a service

ComponentStore (local)
@ngrx/component-store

NGXS (global)
@ngxs/store

ReduxStore (global)
@ngrx/store

Signal(s)
in a service

ReactiveComponentState
@rx-angular/state

SignalStore
@ngrx/signals

Lightweight ⟷ Power

# Redux pattern

- Redux pattern is a clear and strict way to do State Management
- Considered best practice
- Origin: React Ecosystem (by Facebook / Meta)

# Why Redux

- Global store, divided into features (vs. local)
- Separate files for
  - Actions
  - Reducer
  - Selectors
  - Effects
- Reducer (sync) vs. Effects (async)

# Why @ngrx/store

- Most used state management implemantation for Angular
- It works and it is well maintained ☺

**ng add @ngrx/store**

# Alternatives to @ngrx/store

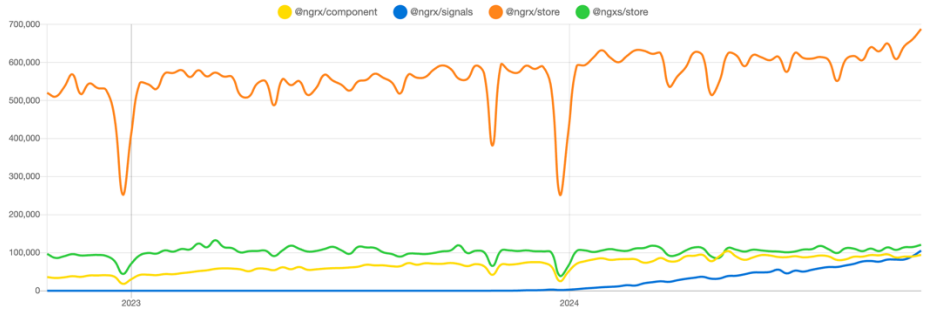
npm trends

**@ngrx/component** vs **@ngrx/signals** vs **@ngrx/store** vs **@ngxs/store**

Enter an npm package...

@ngrx/component × @ngrx/signals × @ngrx/store × @ngxs/store ×

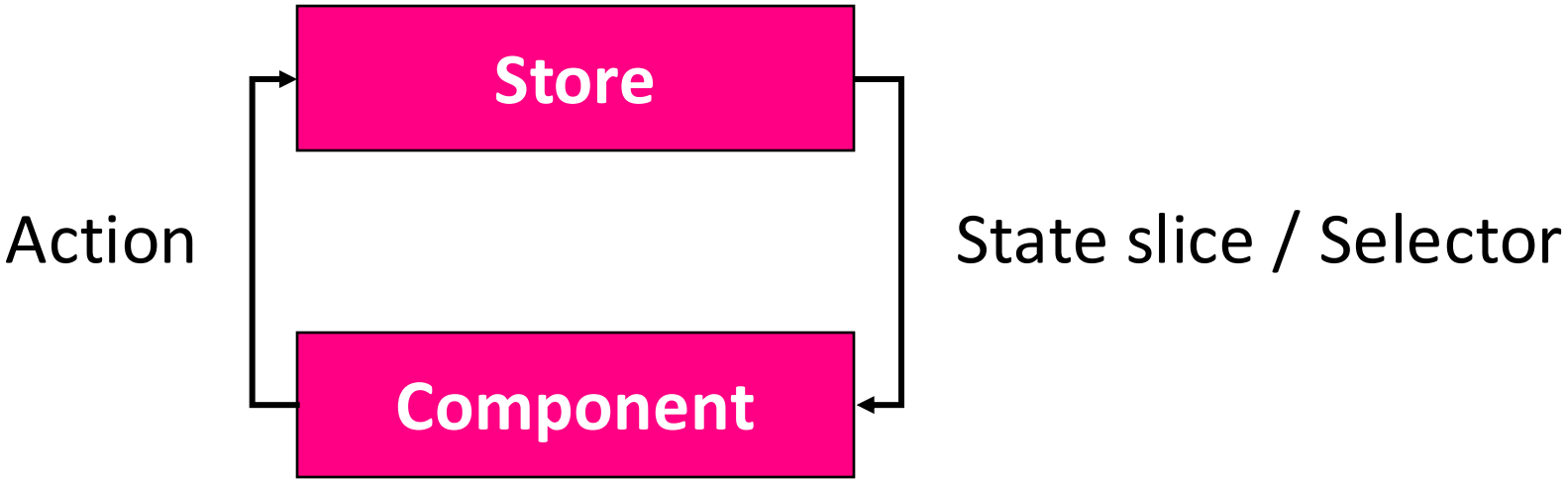
**Downloads** in past 2 Years



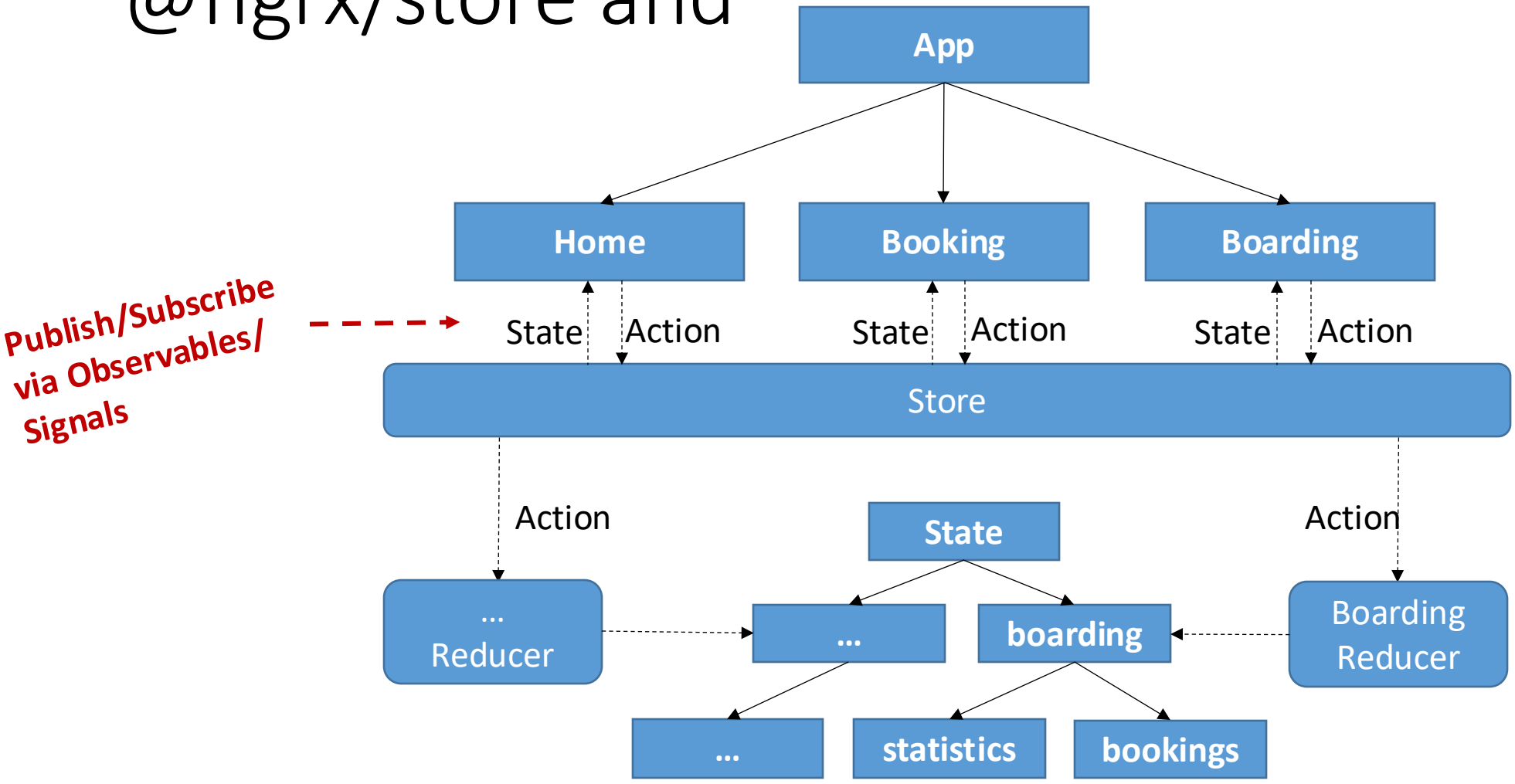

**Stats**

| | Stars | Issues | Version | Updated | Created | Size |
|---|---|---|---|---|---|---|
| @ngrx/component | 8,032 | 66 | 18.1.0 | 19 days ago | 4 years ago | install size 456 kB |
| @ngrx/signals | 8,032 | 66 | 18.1.0 | 19 days ago | a year ago | install size 744 kB |
| @ngrx/store | 8,032 | 66 | 18.1.0 | 19 days ago | 9 years ago | install size 0.961 MB |
| @ngxs/store | 3,536 | 32 | 18.1.4 | 4 days ago | 7 years ago | install size 1.05 MB |

# @ngrx/store and the flow



# @ngrx/store and




Single Immutable State Tree

ANGULAR ARCHITECTS INSIDE KNOWLEDGE

Angular Software Tree GmbH

# State

# State

```
export interface TicketsState {
  flights: Flight[];
  basket: unknown;
  tickets: unknown;
}
```

# DEMO


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

# Actions

# Actions

```
export const ticketsActions = createActionGroup({
  source: 'tickets',
  events: {
    flightsLoaded: props<{ flights: Flight[] }>(),
    updateFlight: props<{ flight: Flight }>(),
    clearFlights: emptyProps(),
  },
});
```

# DEMO

# Reducer

# Reducer

```
export const ticketsFeature = createFeature({
  name: 'tickets',
  reducer: createReducer(
    initialState,

    on(ticketsActions.flightsLoaded, (state, action) => {
      return {
        ...state,
        flights: action.flights,
      };
    }),

    on(ticketsActions.updateFlight, (state, action) => { […] })
});
```

# Reducer

```
export const ticketsFeature = createFeature({
  name: 'tickets',
  reducer: createReducer(
    initialState,

    on(ticketsActions.flightsLoaded, (state, action) => {
      return {
        ...state,
        flights: action.flights,
      };
    }),

    on(ticketsActions.updateFlight, (state, action) => { […] }
});
```

# Store

# Providing the Store (Root Level)

```
bootstrapApplication(AppComponent, {
  providers: [
    […]

    provideStore(),
    isDevMode() ? provideStoreDevtools() : [],
  ],
});
```

# Providing the Store (Feature Level)

```
export const flightBookingRoutes: Routes = [
  {
    path: 'flight-booking',
    component: FlightBookingComponent,
    providers: [
      provideState(ticketsFeature),
    ],
  },
  […]
}
```

# Using the Store

```
private readonly store = inject(Store);
```

```
this.store.dispatch(ticketsActions.flightsLoaded({ flights }));
```

```
flights$ = this.store.select(ticketsFeature.selectFlights);
```

```
flights = this.store.selectSignal(ticketsFeature.selectFlights);
```

# DEMO

# 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]()

# Selectors

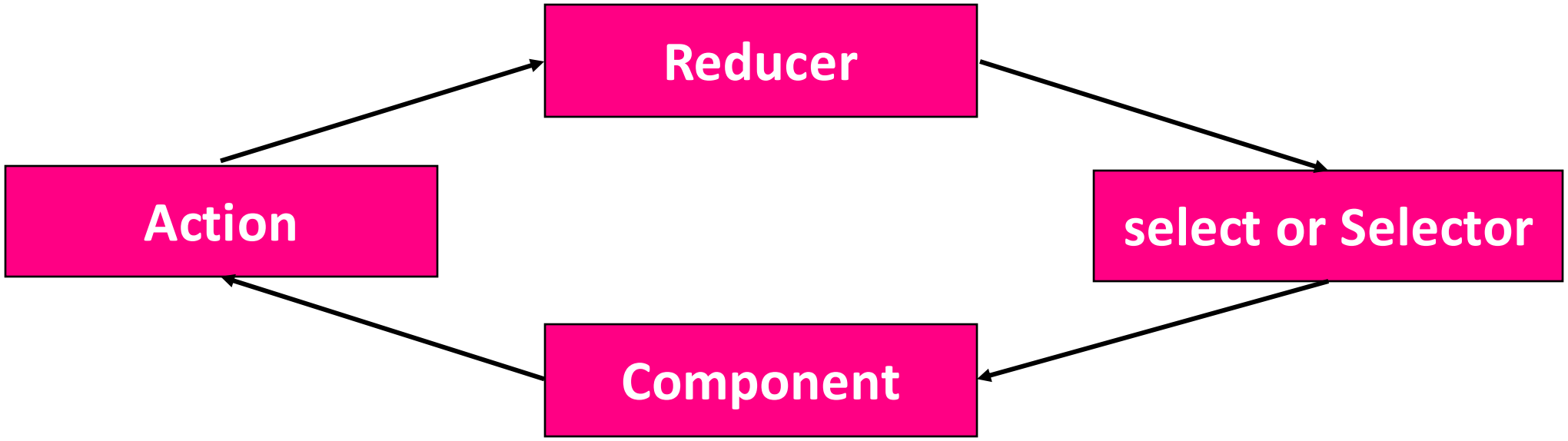
```
export const selectFilteredFlights = createSelector(
  ticketsFeature.selectFlights,
  ticketsFeature.selectHide,
  (flights, hide) => flights.filter((f) => !hide.includes(f.id))
);
```

# Using the Selector

```
private store = inject(Store);
```

```
this.store.select(selectFilteredFlights);
```

# @ngrx/store and the flow



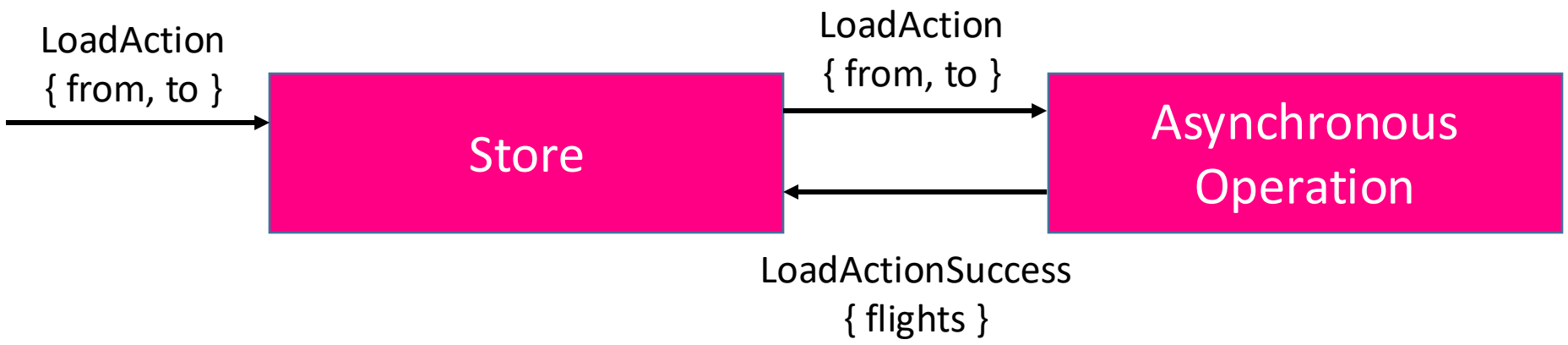
# DEMO

# Effects

# Challenge

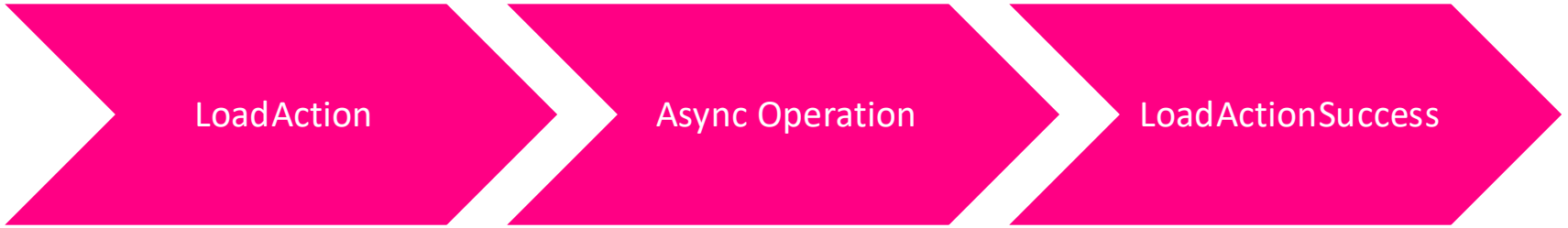
- Reducers are synchronous by definition
- What to do with asynchronous operations (side effects)?

# Solution: Effects

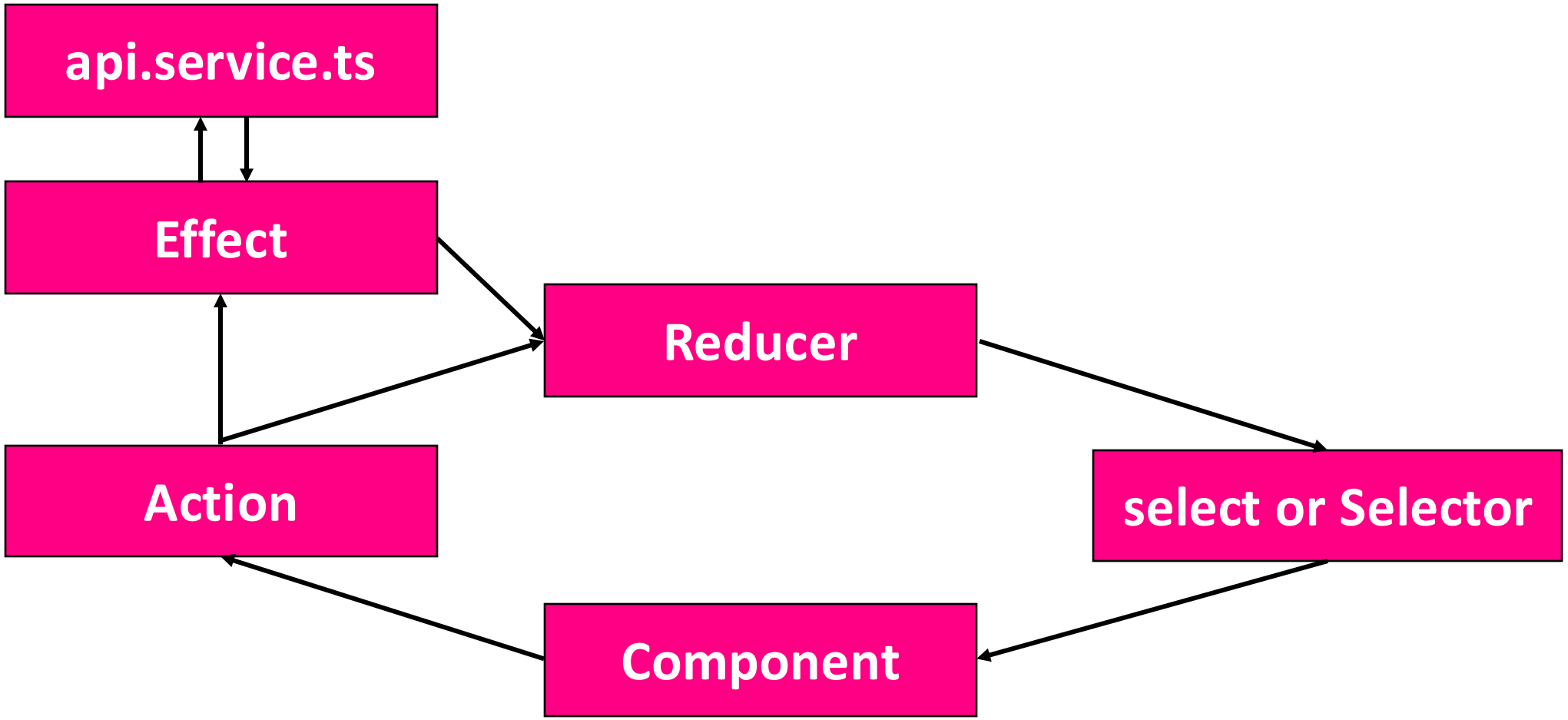


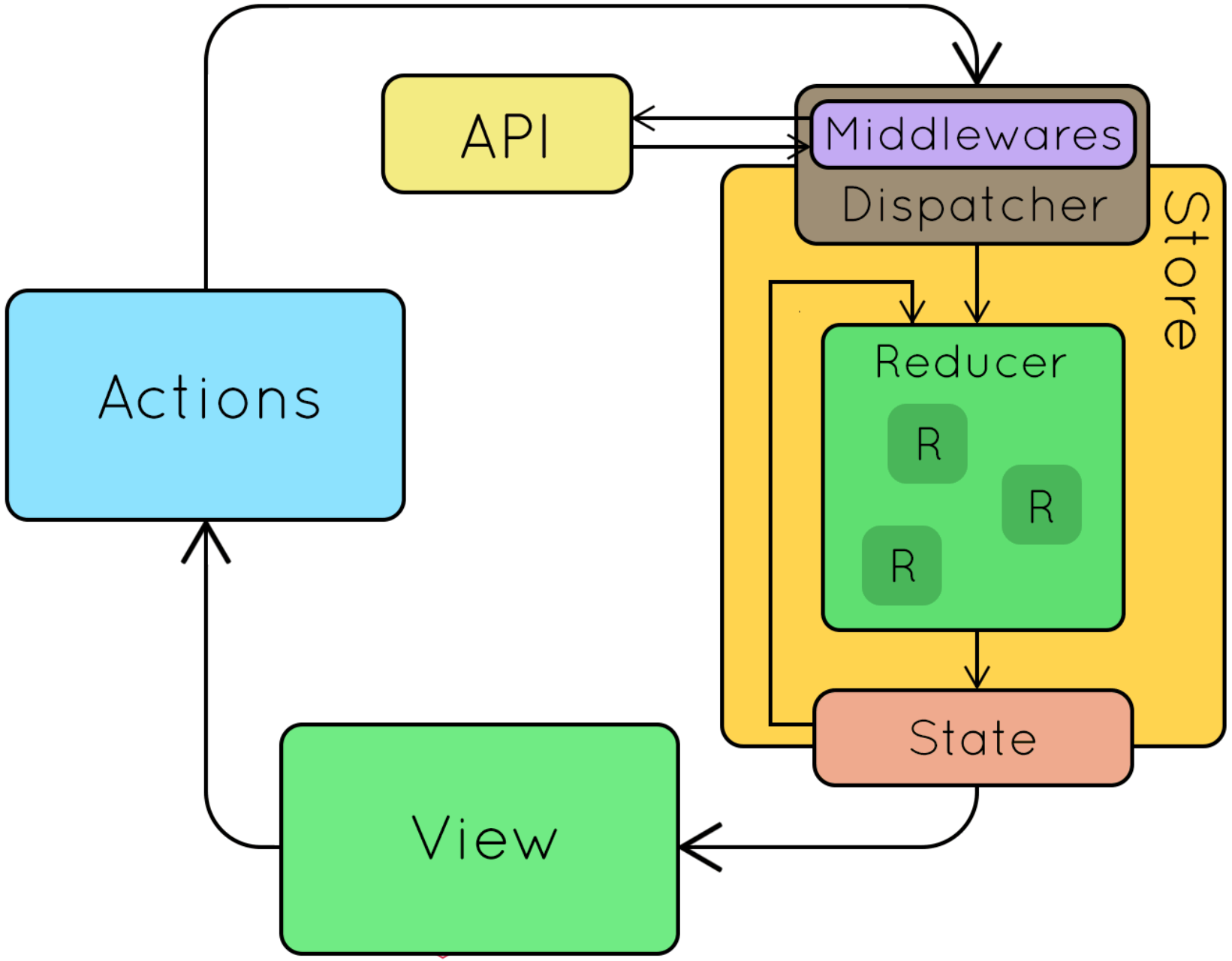

**ng add @ngrx/effects**

# Effects are Observables



# @ngrx/store and the flow




API
Middlewares
Dispatcher
Store
Reducer
R
R
R
State
Actions
View

# Defining an Effect

```
@Injectable({ providedIn: 'root' })
export class TicketsEffects {
}
```

# Defining an Effect

```
@Injectable({ providedIn: 'root' })
export class TicketsEffects {

  flightService = inject(FlightService);
  actions$ = inject(Actions);

}
```

# Defining an Effect

```
@Injectable({ providedIn: 'root' })
export class TicketsEffects {

  flightService = inject(FlightService);
  actions$ = inject(Actions);

  loadFlights = createEffect(() =>
    this.actions$
  );

}
```

# Defining an Effect

```
@Injectable({ providedIn: 'root' })
export class TicketsEffects {

  flightService = inject(FlightService);
  actions$ = inject(Actions);

  loadFlights = createEffect(() =>
    this.actions$.pipe(
      ofType(ticketsActions.loadFlights),
    )
  );

}
```

# Defining an Effect

```
@Injectable({ providedIn: 'root' })
export class TicketsEffects {

  flightService = inject(FlightService);
  actions$ = inject(Actions);

  loadFlights = createEffect(() =>
    this.actions$.pipe(
      ofType(ticketsActions.loadFlights),
      switchMap((a) => this.flightService.find(a.from, a.to)),
    )
  );

}
```

# Defining an Effect

```
@Injectable({ providedIn: 'root' })
export class TicketsEffects {

  flightService = inject(FlightService);
  actions$ = inject(Actions);

  loadFlights = createEffect(() =>
    this.actions$.pipe(
      ofType(ticketsActions.loadFlights),
      switchMap((a) => this.flightService.find(a.from, a.to)),
      map((flights) => ticketsActions.flightsLoaded({ flights }))
    )
  );

}
```

# Providing Effects (Root Level)

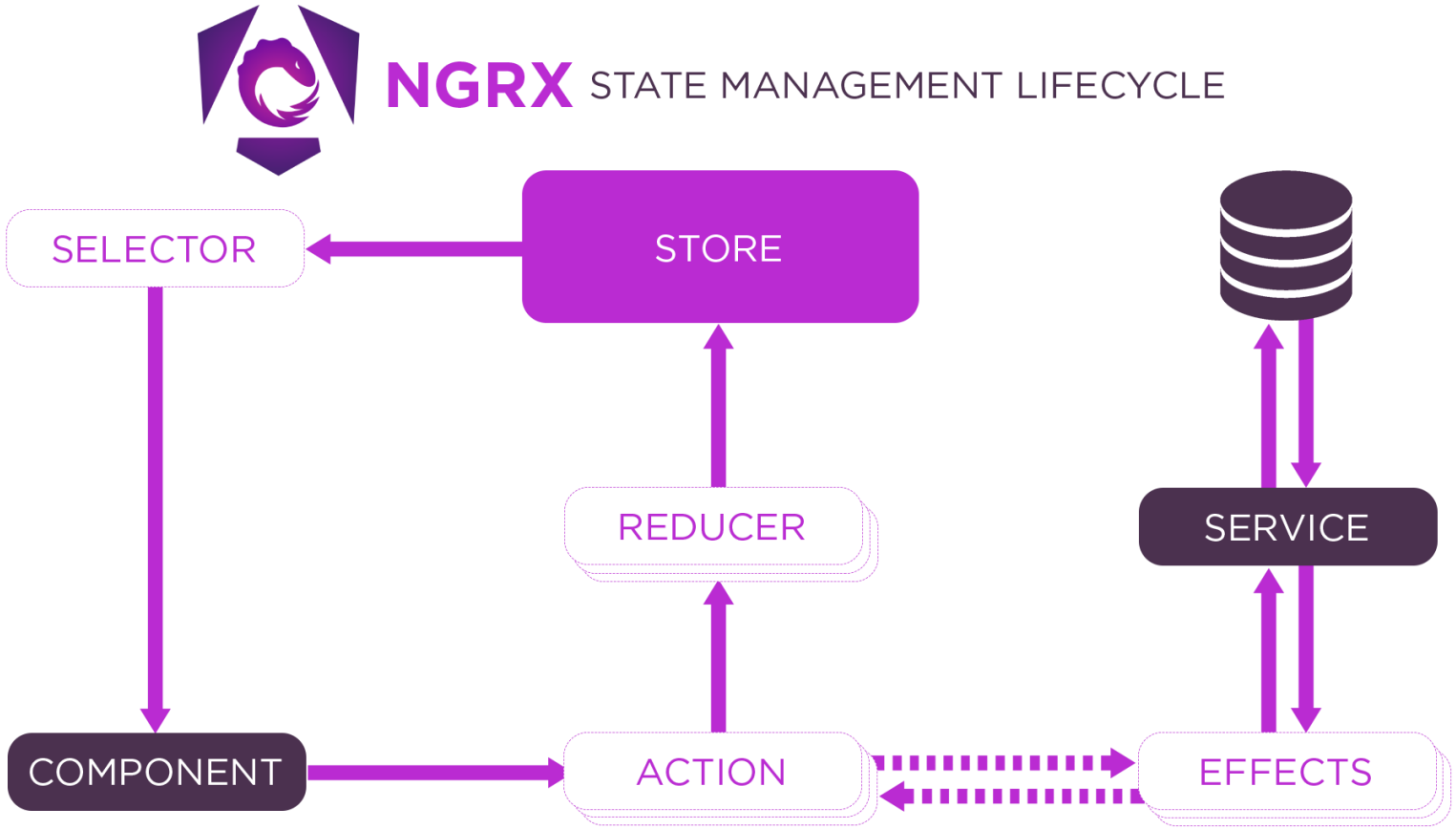
```
bootstrapApplication(AppComponent, {
  providers: [
    […]

    provideStore(),
    provideEffects()
    isDevMode() ? provideStoreDevtools() : [],
  ],
});
```

# Providing Effects (Feature Level)

```
export const flightBookingRoutes: Routes = [
  {
    path: 'flight-booking',
    component: FlightBookingComponent,
    providers: [
      provideState(ticketsFeature),
      provideEffects(ticketsEffects)
    ],
  },
  […]
}
```

# DEMO

# @ngrx redux overview




ANGULAR ARCHITECTS INSIDE KNOWLEDGE

Angular Software Tree GmbH

# Advanced topics

- store devtools
- schematics
- router store
- entities

# @ngrx/store-devtools

- very useful for debugging state management and the whole app

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

# @ngrx/schematic

- scaffolding library built on top of Angular CLI Schematics
- provides Angular CLI commands for generating files
- for new NgRx features and expanding existing ones

- **ng add @ngrx/schematics**

# @ngrx/router-store

- connects the Angular Router with the store
- on each router navigation, multiple actions are dispatched
- allows you to listen for changes in the router's state

- **ng add @ngrx/router-store**

# @ngrx/entity

- reduces boilerplate for creating reducers that manage a collection
- provides performant CRUD operations for each entity collection
- type-safe adapters for selecting entity information

- **ng add @ngrx/entity**

# That's all Folks!

ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

st Angular Software Tree GmbH