RICH CLIENT STATE MANAGEMENT

WAS BEIM LETZTEN MAL GESCHAH

- Single Page Application (SPA)
- Component Architecture
- Atomic Design
- Micro Frontends
- Angular - SPA

MOTIVATION

DATENMANAGEMENT IM FRONTEND

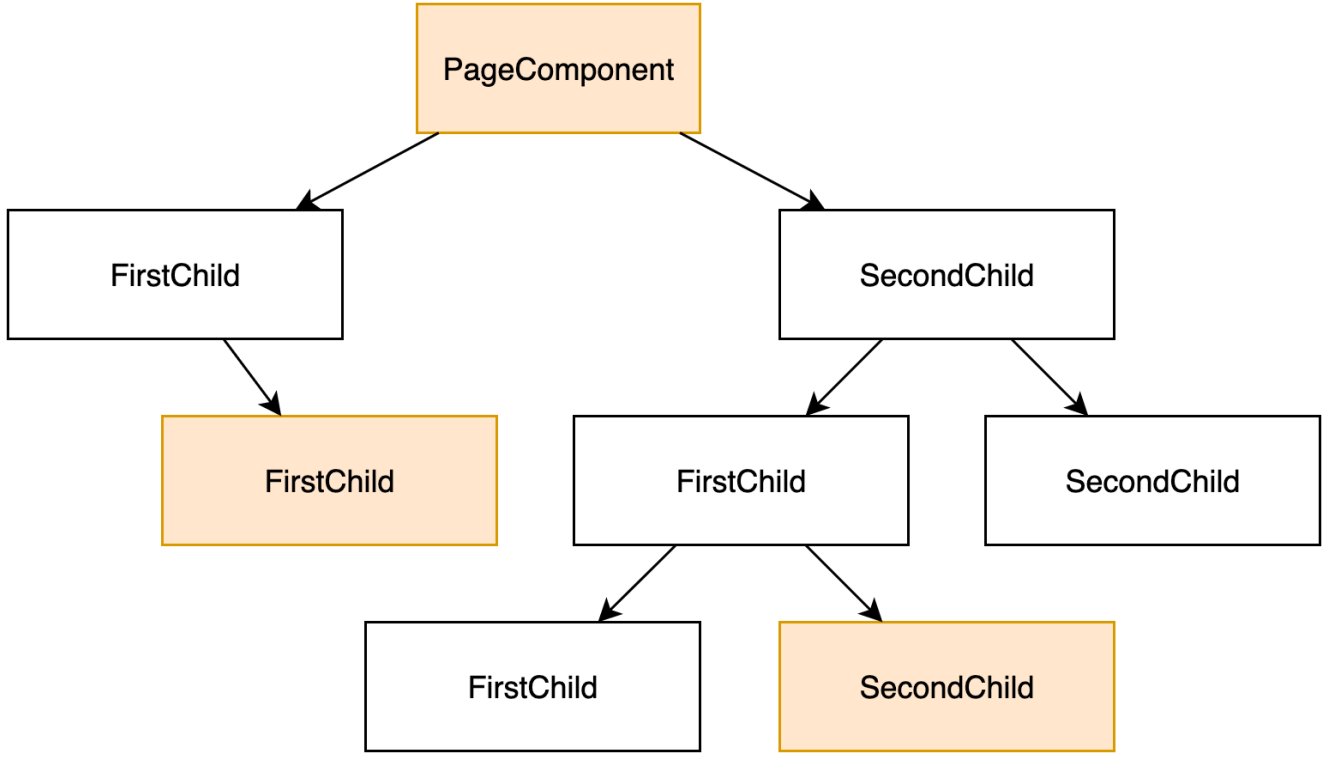
```
1 export class ShowDataComponent {
2
3     @Input()
4     someData: SomeData;
5     @Output()
6     output: EventEmitter = new EventEmitter<Output>();
7 }
```

DATENMANAGEMENT IM FRONTEND

```
1 <app-show-data
2     [someData]="{ ... }"
3     (output)="callOnOutput($event)">
4 </app-show-data>
```

TIEFE COMPONENT HIERARCHIEN

- Daten werden durch Components "hindurchgereicht"
- Boilerplate Code

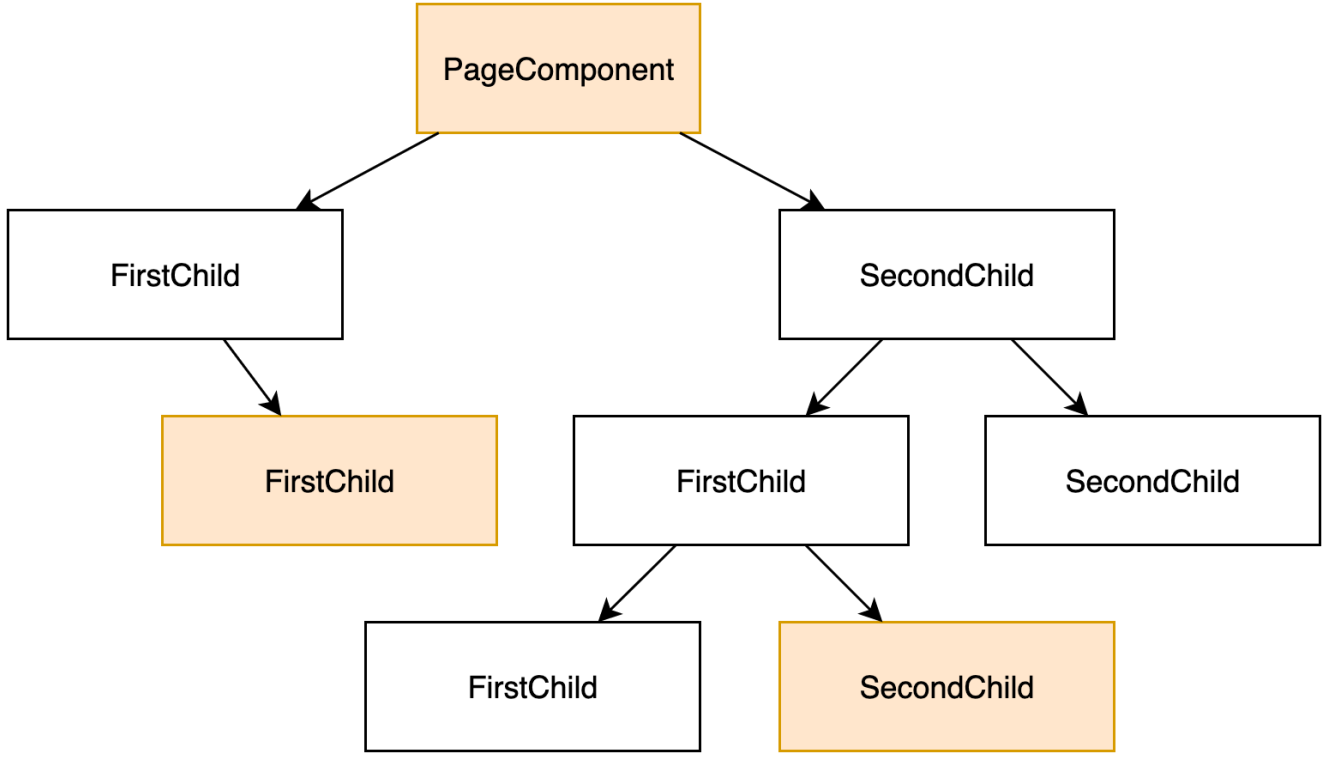


BEISPIEL COMPONENT HIERARCHY

THINK BIG

Was wenn das Frontend noch größer ist ...

und überall die Daten erneut aus dem Backend gefetched werden?



VERTEILTER DATEN

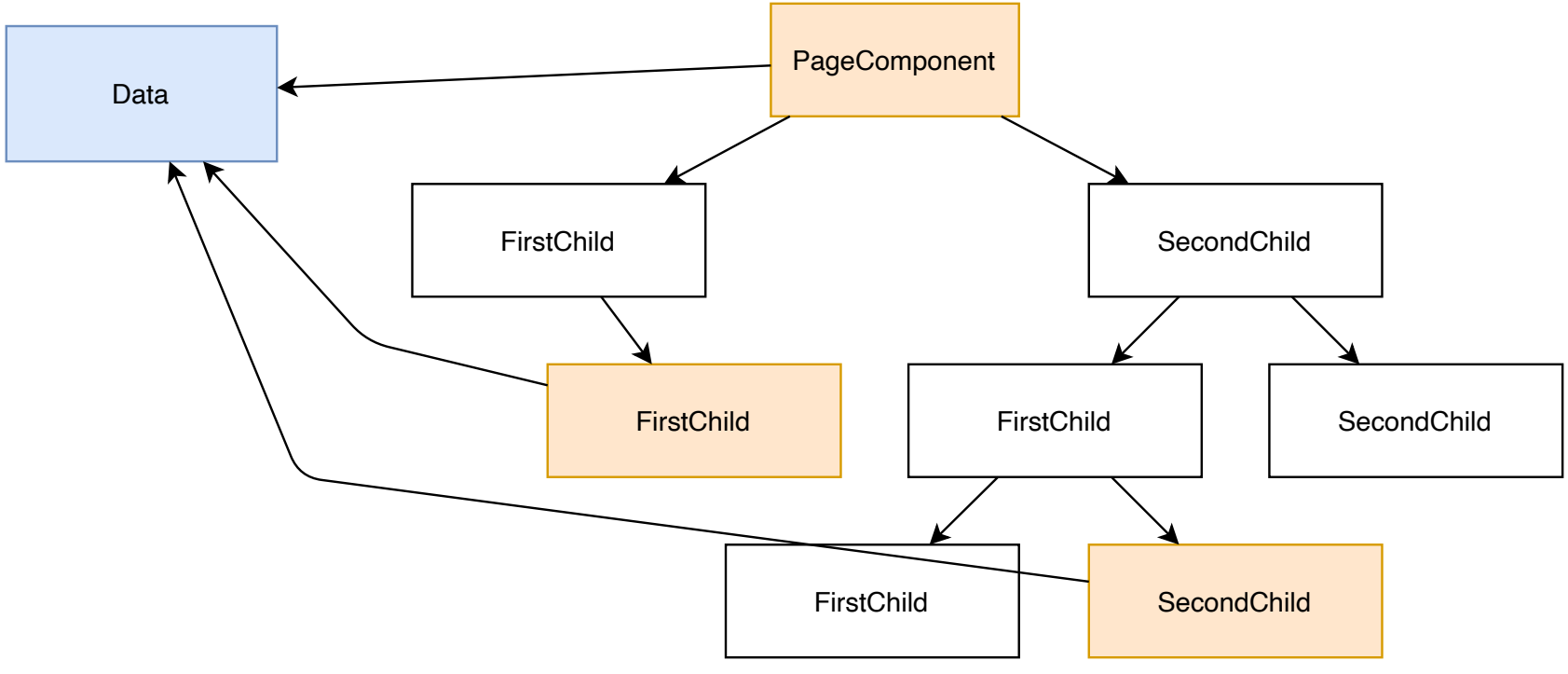
- Verteilte Daten
- unübersichtlich bei großen Anwendungen
  - fehlerhafter Daten müssen erst entdeckt werden

BEISPIEL: VERTEILTE DATEN

```
 1 public void someMethod() {
 2     var someData = new SomeData("some-title");
 3
 4     someOtherMethod(someData);
 5
 6     assert someData.getTitle().equals("some-title");
 7 }
 8
 9 private void someOtherMethod(SomeData data) {
10     data.setTitle("some-other-title");
11 }
```

STATE MANAGEMENT

LÖSUNG FÜR DAS DATENMANAGEMENT



SINGLETON ODER GLOBAL STATE

- globales Objekt
- jeder kann darauf zugreifen
- im schlimmsten Fall ein
  - direkter Zugriff
  - statischer Zugriff

MÖGLICHE NACHTEILE EINES GLOBAL STATES

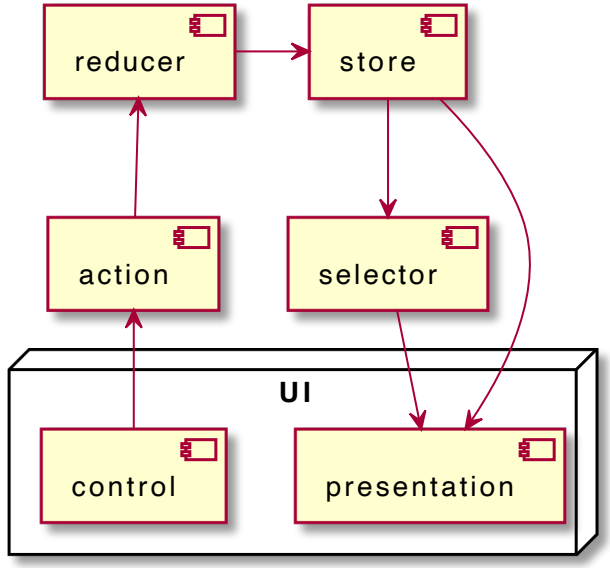
- hohe unkontrollierte Kopplung im ganzen Projekt
- direkte und statische Aufrufe sorgen für die hohe Kopplung
- Fehlersuche wird schwieriger durch:
  - Seiteneffekte
  - Verteilung im ganzen Projekt
- es wird passieren
  - Murphys Law:
  - Anything that can go wrong will go wrong.

MÖGLICHE NACHTEILE EINES GLOBAL STATES

- Unvorhersehbar
  - jeder kann den State bearbeiten
  - Seiteneffekte
- keiner weiß:
  - warum sich der State ändert
  - wo sich der State ändert

GLOBAL STATE RICHTIG

State Management mit NgRx



STORE

- enthält den aktuellen State
- State
  - ist immutable
  - wird neu erstellt bei jeder Änderung
  - wird von einem Reducer erstellt
  - kann aus mehreren Teilen bestehen

REDUCER

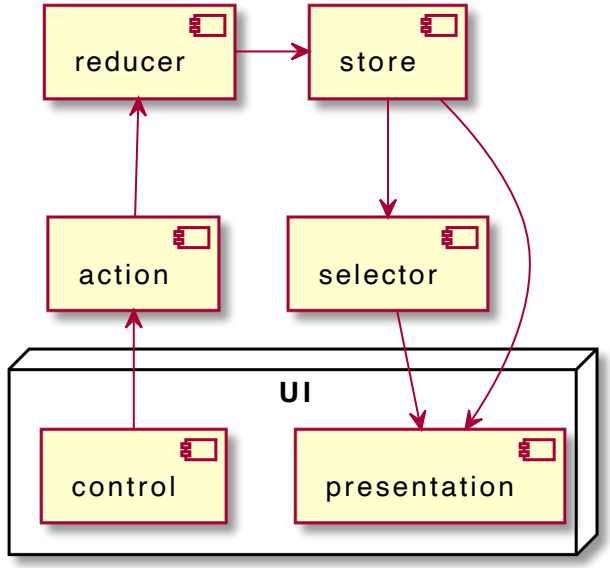
- verarbeitet Actions
- baut aus State und Action neuen State
- State darf nur ersetzt werden!
- keine Änderungen

ACTION

- eine Interaktion des Nutzers
- löst Logik im Reducer aus
- wird asynchron an einen Reducer dispatched

FUNKTIONSWEISE DES STATE MANAGEMENTS

- Nutzer löst Action aus
- Reducer verarbeitet Action und erstellt neuen State
- UI reagiert auf den State



SELECTOR

- ist nicht zwingend notwendig
- mit einem Selector kann man
  - teile des States auslesen
  - eine Projektion auf dem State auslösen
- Selectors können verschachtelt werden

VORTEILE VON STATE MANAGEMENT

- Daten bzw. State kann von überall abgerufen werden
- Fehler durch Seiteneffekte werden eingeschränkt
- Daten werden nicht mehrmals vom Backend geladen
- basiert auf Functional Reactive Programming (FRP)
  - macht die Anwendung häufig Nutzerfreundlicher
  - Anwendung soll nach einer Nutzereingabe nicht "blockieren"
- zentraler State hilf beim Entwickeln/Debuggen

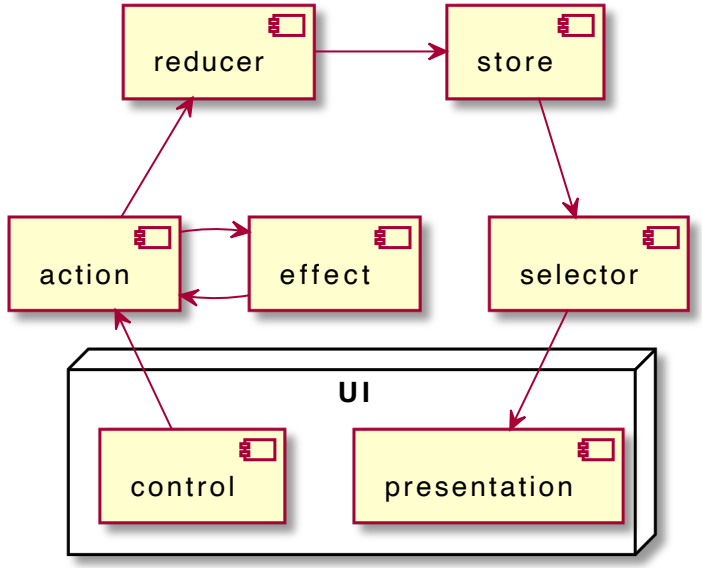
DATEN VOM BACKEND LADEN?

STATE MANAGEMENT GUT UND SCHÖN ...

- Wie bekomme ich Daten vom Backend?
- Reducer arbeitet nur synchron
- Daten über eine Action übergeben?
- Backend Call aus der Component?

NGRX EFFECTS

- Actions können von Effects abgefangen werden
- Effects können asynchrone Tasks ausführen
- zum Beispiel Backendcalls



EFFECTS

- Effects gibt es in verschiedenen Frameworks/Libraries
- heißen nur manchmal anders
- in Redux heißt es "Thunk Middleware"
- NGXS unterstützt generell asynchrone Operationen

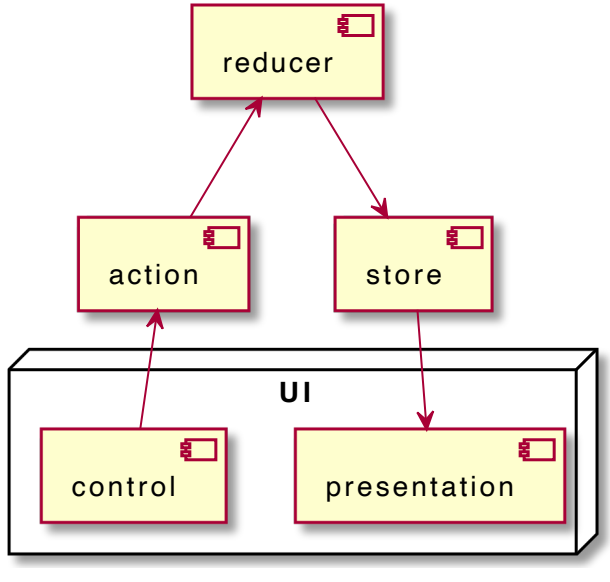
ABSEITS VON NGRX

STATE MANAGEMENT ÜBERALL

- unzählige State Management Frameworks
- jedes Frontend Framework bringt ein eigenes mit
- unter der Haube funktionieren alle gleich

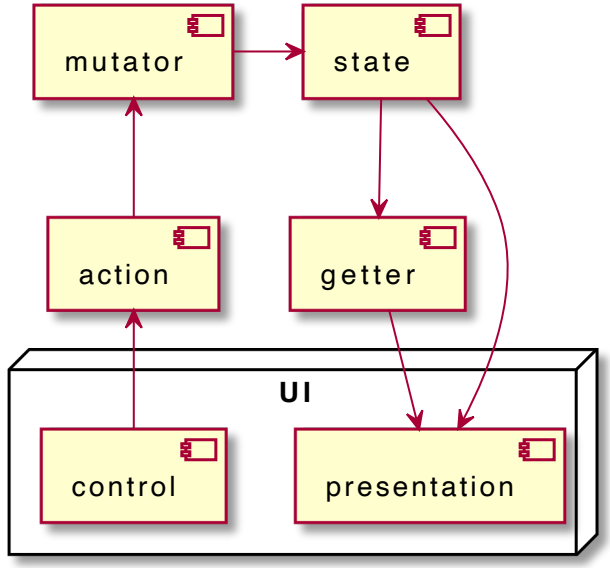
REDUX

- State Management für React
- Release 2015
- Redux verzichtet auf einen Selector
- wenig "magic"
- kann sogar ohne "magic" implementiert werden



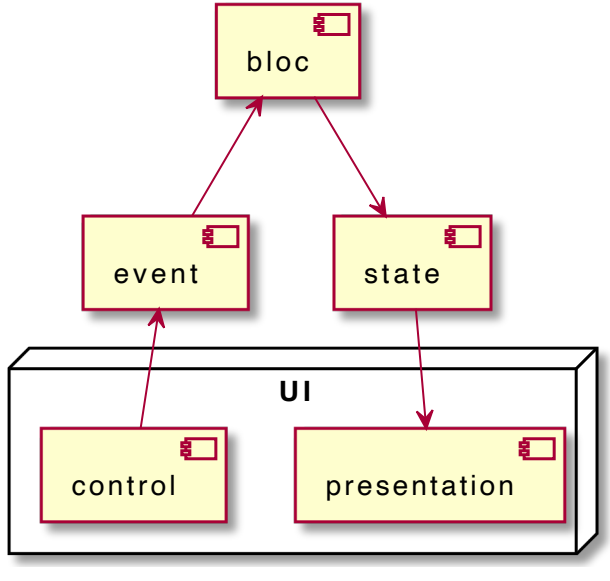
VUEX

- State Management für Vue
- Release 2020
- erinnert an NgRx
- leichtgewichtige Implementierung



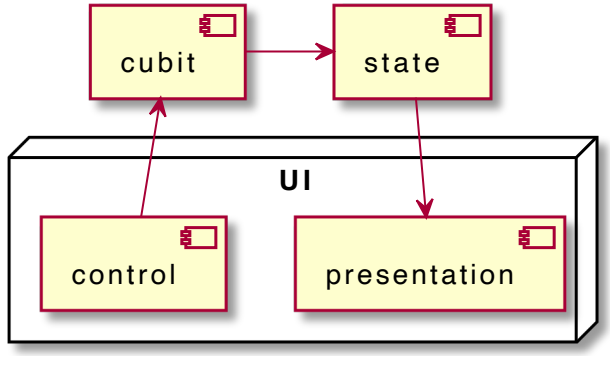
BLOC

- State Management für Dart/Flutter
- Release 2018
- ohne "Selector"
- kommt auch ohne Events aus
  - nennt man dann Cubit



BLOC (CUBIT)

- Control ruft Methoden des Cubits auf
- leichtgewichtiger Cubit



WAS MÖCHTE ICH DAMIT SAGEN?

- State Management ist ein Pattern das überall vorkommt
- im Kern funktionieren alle gleich
- Wenn ihr das Konzept verstanden habt, könnt ihr schnell neue Frameworks lernen

NGRX

MODEL

```
1 export interface Todo {
2     id?: number;
3     title: string;
4     done: boolean;
5 }
```

ACTIONS

- werden über die Funktion *"createAction"* erstellt
- brauchen einen Identifier
  - besteht aus einer Source: *"[Some Component]"*
  - und einem Event: *"Add Some Data"*
- können Properties enthalten
  - in diesem Fall *"SomeData"*

```
export const addSomeData = createAction(
    '[Some Component] Add Some Data',
    props<{ someData: SomeData }>(),
);
```

REDUCER

- braucht einen initialen State
  - dieser ist Readonly!
- wird über *"createReducer"* erzeugt
- braucht Handler für jede Action
  - dieser erstellt den neuen State

```
export const initialState: ReadonlyMap<number, SomeData>
    = new Map([]);

export const someDataReducer = createReducer(
    initialState,
    on(addSomeData, addSomeDataReducer),
);

function addSomeDataReducer(
    state: ReadonlyMap<number, SomeData>,
    { someData }: { someData: SomeData }
 ): Map<number, SomeData> {
    // do something and return new state
}
```

REDUCER REGISTRIEREN

- Reducer müssen im App Module registriert werden
- es können mehrere Reducer registriert werden

```
 1 import { someDataReducer } from './state/todo.reducer';
 2
 3 @NgModule({
 4     imports: [
 5         ...
 6         StoreModule.forRoot({ someData: someDataReducer })
 7         ...
 8     ],
 9 })
10 export class AppModule {
11 }
```

STATE

- enthält den globalen Status der Anwendung
- ist Readonly (Immutable)
- kann in mehrere Stücke unterteilt sein
- einfaches Interface

```
1 export interface AppState {
2     someData: ReadonlyMap<number, SomeData>;
3 }
```

SELECTORS

- Hilfsmittel um einen Teil des States auszulesen
- werden mit *"createSelector"* erzeugt
  - besteht aus Selector
  - und Projector

```
export const selectSomeData = createSelector(
    (state: AppState) => state.someData,
    (someData: Map<number, SomeData>)
            => Array.from(someData.values()),
);
```

DISPATCHEN EINER ACTION

- Store wird über den Constructor injected
- Action wird einfach auf dem Store dispatched

```
export class SomeDataComponent {

    someData: SomeData;

    constructor(private readonly store: Store) {}

    onSubmit() {
        this.store.dispatch(
            addSomeData({ someData: this.someData })
        );
    }
}
```

SELECTION DES STATES

- aufrufen der Funktion *"select()"* auf dem Store
- *"select()"* liefert ein Observable
  - vergleichbar zu Java Streams
- auf das Observable wird mit *"async"* subscribed

```
export class ListViewComponent {
    someData$ = this.store.select(selectSomeData);

    constructor(private readonly store: Store) {
    }
}
```

```
<div *ngIf="someData$ | async">
    <h2 class="mb-3">SomeData:</h2>
    <div *ngFor="let someData of someData$ | async">
        <app-list-view-item [item]="someData">
        </app-list-view-item>
    </div>
</div>
```

STATE OBSERVING

- auf den State kann auch direkt observed werden
- anstatt einer async Pipe
- bei neuem State wird someData überschrieben

```
export class ListViewComponent implements OnInit {
    someData: SomeData[];

    constructor(private readonly store: Store) {}

    ngOnInit() {
        this.store.select(selectSomeData).subscribe({
            someData => this.someData = someData
        });
    }
}
```

EFFECTS

- sind *@Injectable* classes
- auf *actions$* wird subscribed
- Service wird für Backendaufrufe injected

```
1 @Injectable()
2 export class SomeDataEffects {
3     // some effects here
4
5     constructor(
6         private readonly actions$: Actions,
7         private readonly someDataService: SomeDataService
8     ) {}
9 }
```

EFFECTS

- auf den *actions$* wird subscribed
- anschließend wird der Backendaufruf ausgeführt
- neue Action muss weitergegeben werden
  - diese Action landet dann beim Reducer
- Error Fall sollte beachtet werden

```
loadSomeData$ = createEffect(() => this.actions$.pipe(
    ofType(Action.LoadSomeData),
    mergeMap(() => this.someDataService.getAll()
        .then(
            someData => loadSomeDataSuccess({ someData }),
            _ => loadSomeDataError({ message: 'error' })
        ),
    ),
));
```

EFFECT REGISTRIEREN

- wird im AppModule registriert

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

PRAXIS

PRAXIS: STATE MANAGEMENT

- Umbau der bestehenden Anwendung mit NgRx
- falls jemand nicht fertig wurde
  - Branch: solution
- für die schnellen:
  - integration des Mock Services mit Effects

PRAXIS: LÖSUNGEN

- Branch: state_management_solution
- Branch: state_management_effects_solution

Was wir alles für diese Section brauchen: Simples Beispiel bauen. Typische Frontendarchitektur (statehandling): StateManagement mit Redux Konkrete Bibliothek NgRx Actions Reducer State Unsere bestehende Anwendung um ein StateManagement erweitern. Vergleich zu anderen Frameworks React, Vue, Flutter Component System StateManagement Redux, Bloc, etc. https://ordina-jworks.github.io/angular/2018/10/08/angular-state-management-comparison.html