

# **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

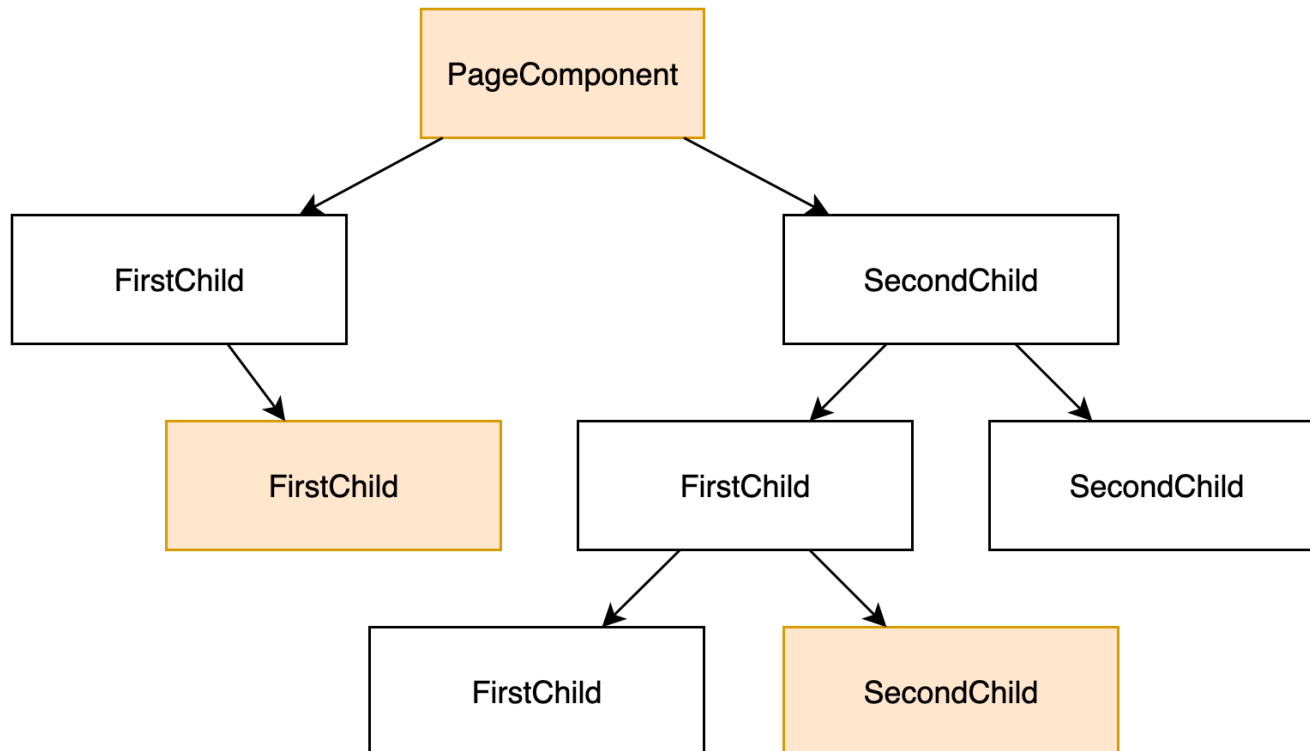
```
1 export class ShowDataComponent {  
2  
3     @Input()  
4     someData: SomeData;  
5     @Output()  
6     output: EventEmitter = new EventEmitter<Output>();  
7 }
```

# 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

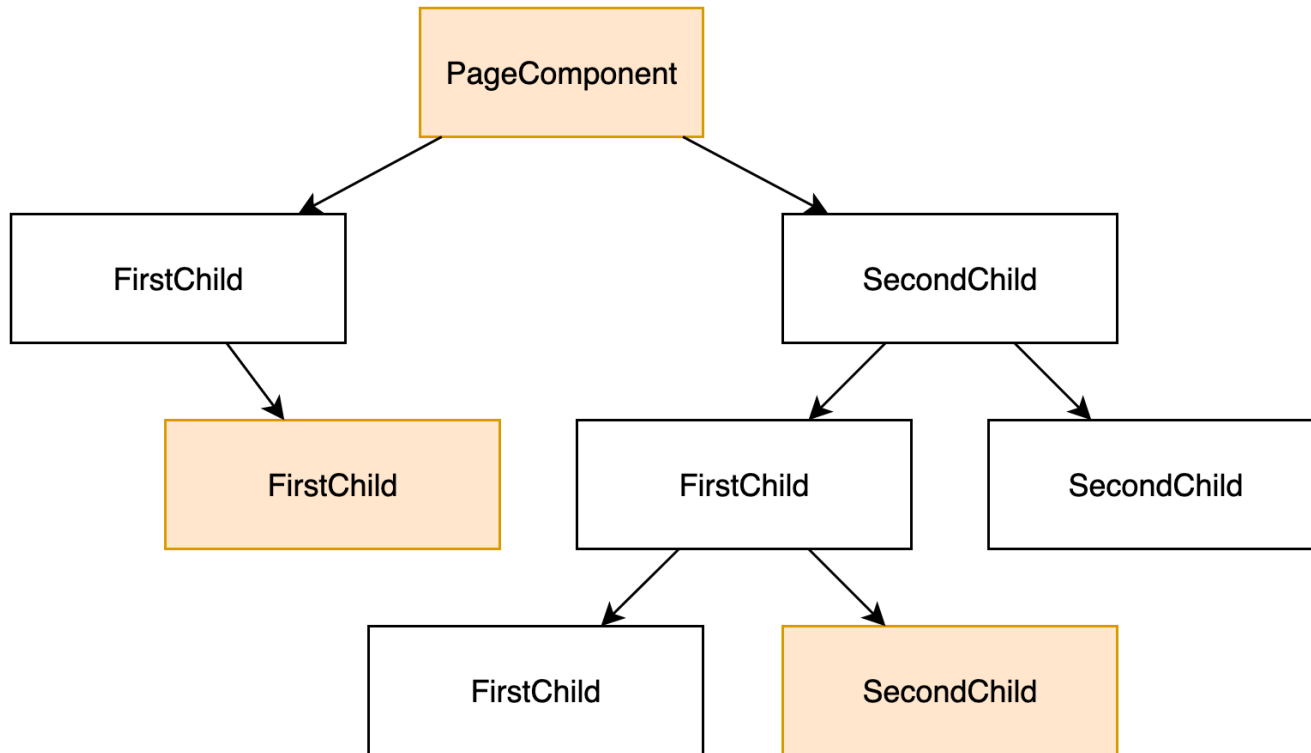


# BEISPIEL COMPONENT HIERARCHY

# THINK BIG

## Was wenn das Frontend noch größer ist ...

und überall die Daten erneut aus dem Backend gefetched werden?





# ROUTING

- Wie werden Daten über mehrere Screens geteilt?
- Router Outlet nimmt keine Parameter entgegen
  - Wie auch, es können unterschiedliche Screens eingebettet werden
  - Die Screens könnten alle unterschiedliche Parameter haben
- Services könnten die Daten cachen
  - dies geht schon in die Richtung State Management

```
1 <router-outlet></router-outlet>
```

# VERTEILTE DATEN

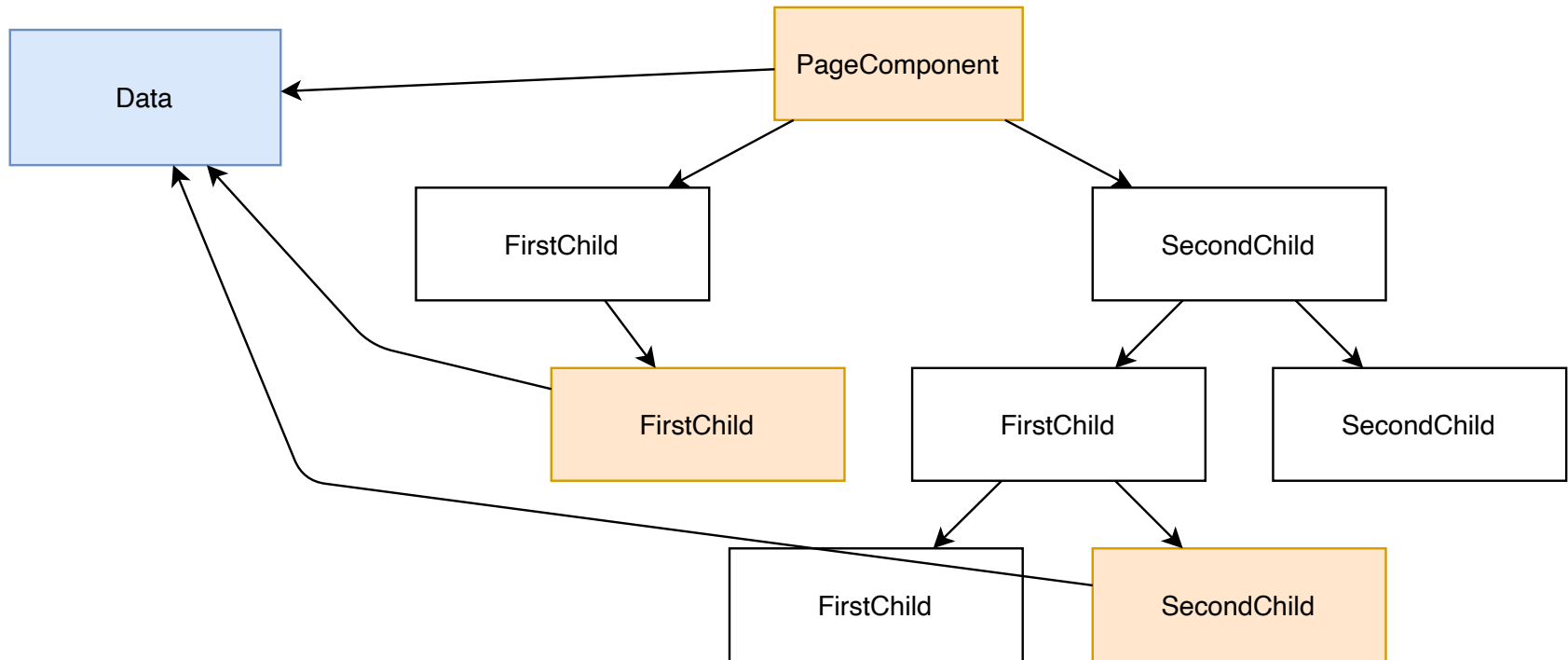
- Verteilte Daten
- unübersichtlich bei großen Anwendungen
  - fehlerhafte Daten müssen erst entdeckt werden

## 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.

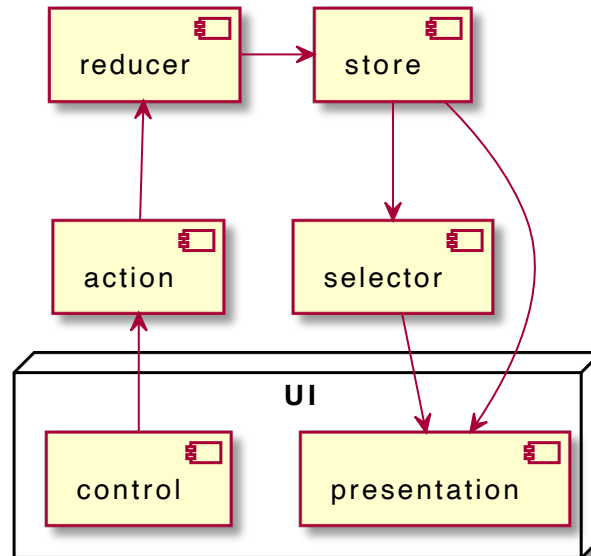
# MÖGLICHE NACHTEILE EINES GLOBAL STATES

- 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

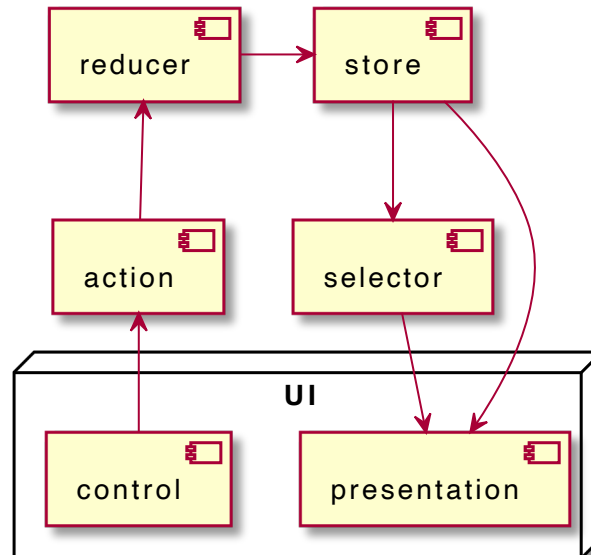
- verarbeitet Actions
- baut aus State und Action neuen State
- State darf nur ersetzt werden!
- keine Änderungen

# 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?**

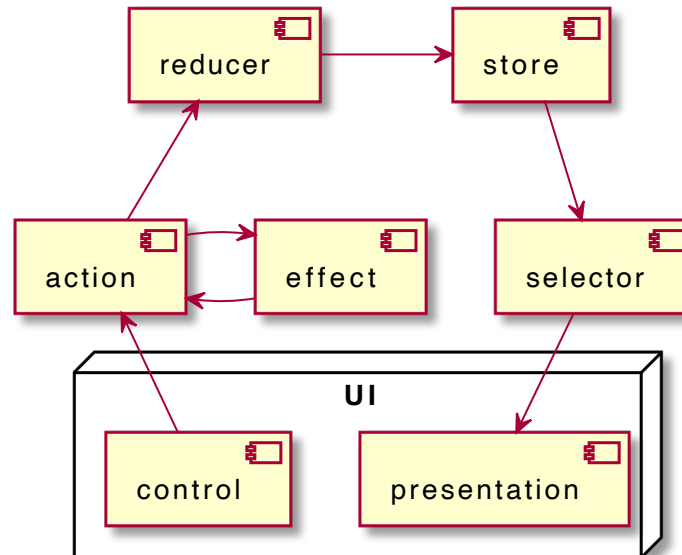


# STATE MANAGEMENT GUT UND SCHÖN ...

- 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

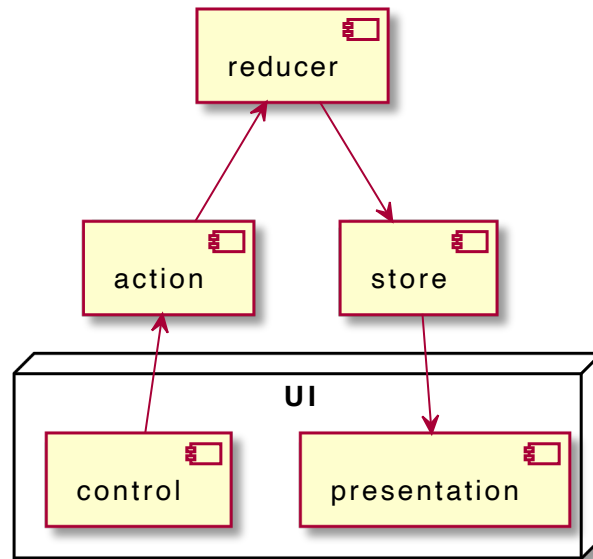
# ABSEITS VON NGRX

# STATE MANAGEMENT ÜBERALL

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

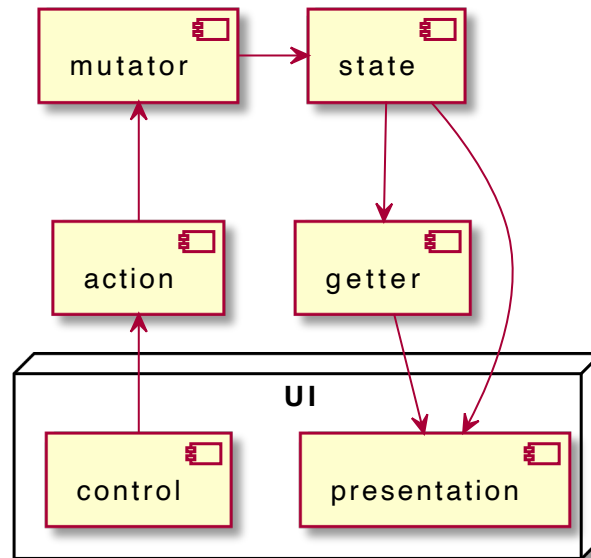
# REDUX

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



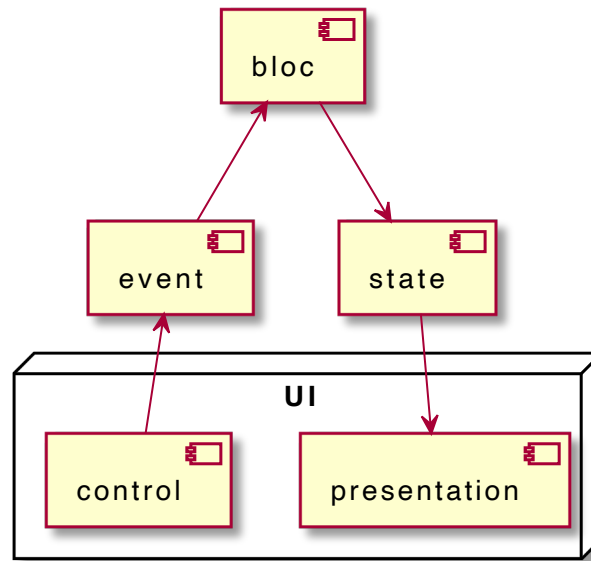
# VUEX

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



# BLOC

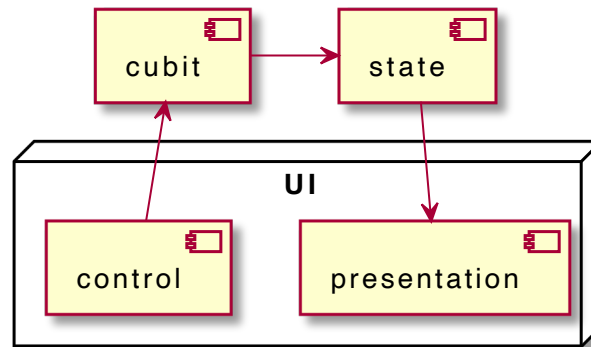
- State Management für Dart/Flutter
- Release 2018
- ohne "Selector"
- kommt auch ohne Events aus
  - nennt man dann Cubit





# 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"*

```
1 export const addSomeData = createAction(  
2   '[Some Component] Add Some Data',  
3   props<{ someData: SomeData }>(),  
4 );
```

# REDUCER

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

```
1 export const initialState: ReadonlyMap<number, SomeData>
2   = new Map([]);
3
4 export const someDataReducer = createReducer(
5   initialState,
6   on(addSomeData, addSomeDataReducer),
7 );
8
9 function addSomeDataReducer(
10   state: ReadonlyMap<number, SomeData>,
11   { someData }: { someData: SomeData }
12 ): Map<number, SomeData> {
13   // do something and return new state
14 }
```

# 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

```
1 export const selectSomeData = createSelector(  
2   (state: AppState) => state.someData,  
3   (someData: Map<number, SomeData>)  
4     => Array.from(someData.values()),  
5 );
```

# DISPATCHEN EINER ACTION

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

```
1 export class SomeDataComponent {  
2  
3   someData: SomeData;  
4  
5   constructor(private readonly store: Store) {}  
6  
7   onSubmit() {  
8     this.store.dispatch(  
9       addSomeData({ someData: this.someData })  
10    );  
11  }  
12 }
```

# 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

```
1 export class ListViewComponent {
2     someData$ = this.store.select(selectSomeData);
3
4     constructor(private readonly store: Store) {
5     }
6 }
```

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

# STATE OBSERVING

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

```
1 export class ListViewComponent implements OnInit {  
2     someData: SomeData[];  
3  
4     constructor(private readonly store: Store) {}  
5  
6     ngOnInit() {  
7         this.store.select(selectSomeData).subscribe({  
8             someData => this.someData = someData  
9         });  
10    }  
11 }
```

# 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

```
1 loadSomeData$ = createEffect(() => this.actions$.pipe(  
2   ofType(Action.LoadSomeData),  
3   mergeMap(() => this.someDataService.getAll()  
4     .then(  
5       someData => loadSomeDataSuccess({ someData }),  
6       _ => loadSomeDataError({ message: 'error' })  
7     ),  
8   ),  
9 ));
```

# EFFECT REGISTRIEREN

- wird im AppModule registriert

```
1 @NgModule({
2     imports: [
3         ...
4         EffectsModule.forRoot([SomeDataEffects]),
5         ...
6     ],
7 })
8 export class AppModule {
9 }
```

# 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