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 }
```

Speaker notes

- wir erinnern uns an die letzte Vorlesung
- Daten werden über Input und Output Parameter in Components hinein und heraus gegeben

DATENMANAGEMENT IM FRONTEND

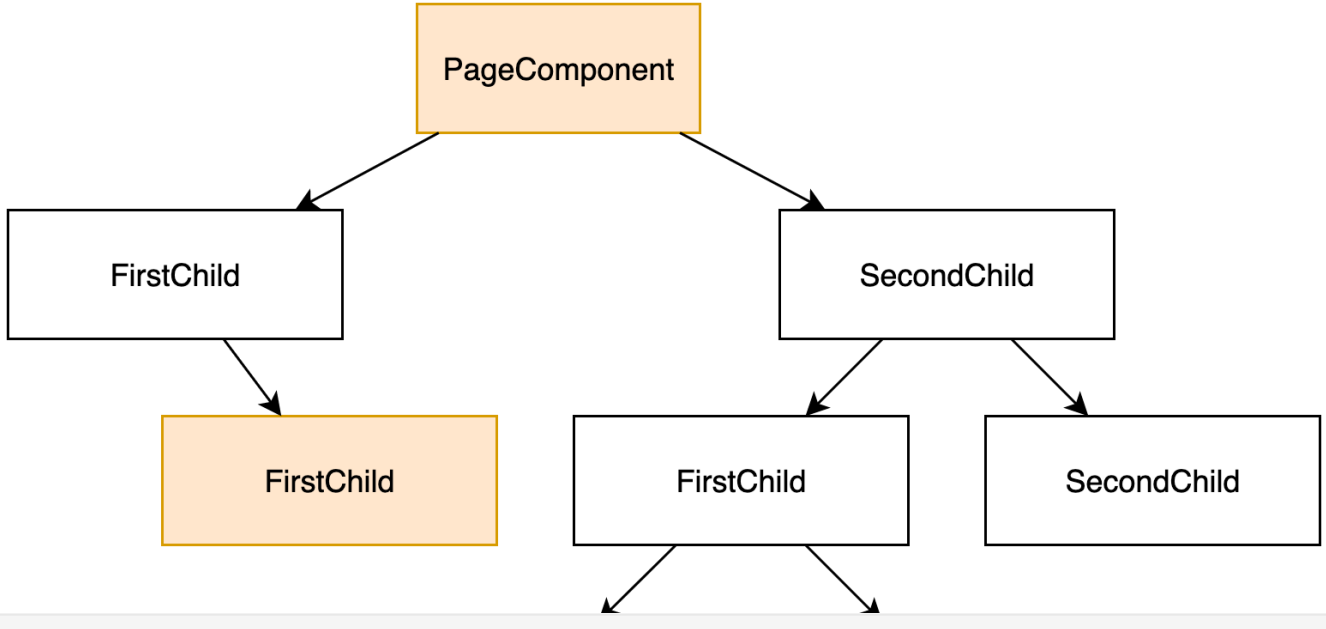
```
<app-show-data
    [someData]="{ ... }"
    (output)="callOnOutput($event)">
</app-show-data>
```

Speaker notes

- Daten können von der Parent Component an eine Child Component weitergegeben werden
- Genauso umgekehrt von einer Child Component an die Parent Component

TIEFE COMPONENT HIERARCHIEN

- Daten werden durch Components "hindurchgereicht"
- Boilerplate Code



Speaker notes

- Daten werden in beide Richtungen "durchgereicht"
- als Input durch die Component nach untern
- und als Output wieder hoch
- dies erzeugt eine Menge Code der irgendwie falsch erscheint

BEISPIEL COMPONENT HIERARCHY

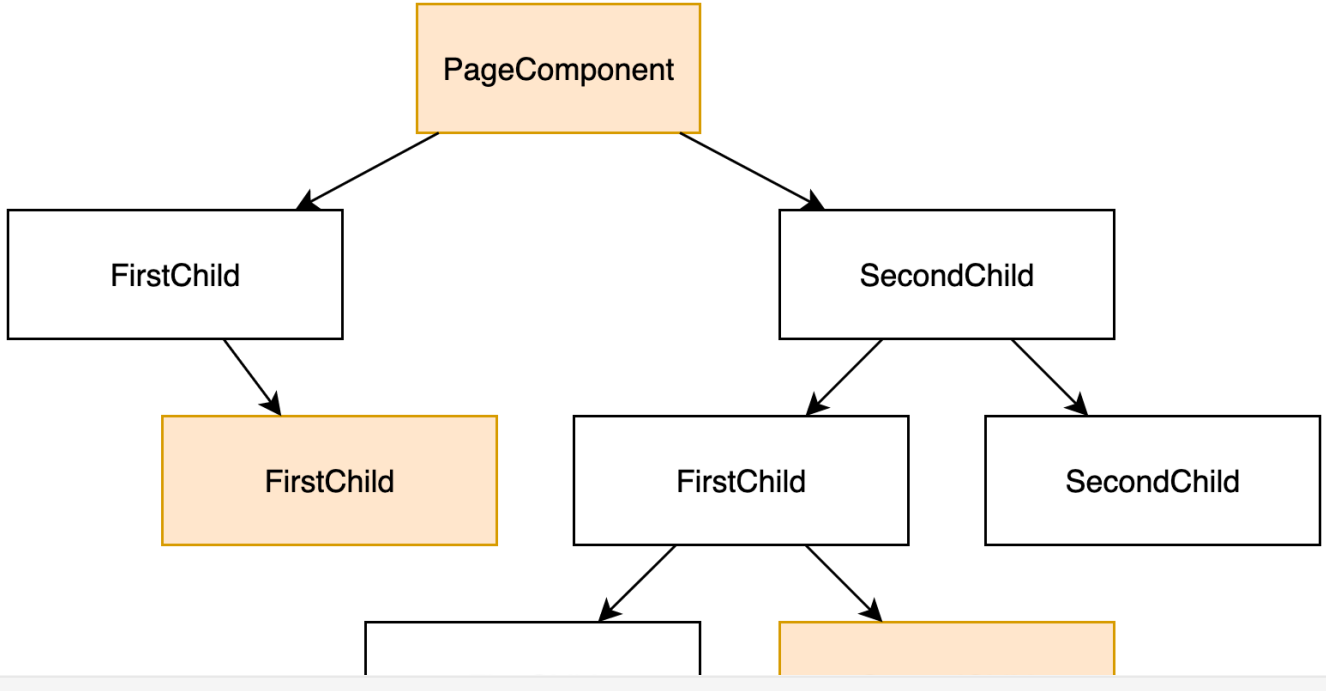
Speaker notes

- hier ziehe ich ein Codebeispiel hinzu
- es geht dabei darum den Boilerplate Code an einem Beispiel zu zeigen

THINK BIG

Was wenn das Frontend noch größer ist ...

und überall die Daten erneut aus dem Backend gefetched werden?



Speaker notes

- Daten werden mehrmals geladen
- unnötige Datenübertragung

VERTEILTER DATEN

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

Speaker notes

- Daten sind meist über die Anwendung verteilt
- Besonders bei größeren Anwendungen wird es unübersichtlich
  - Man kann sich manche Fehler nicht erklären, weil Daten überall übers Frontend verteilt sind
  - Wechselwirkungen und Seiteneffekte treten dann gerne auf

BEISPIEL: VERTEILTE DATEN

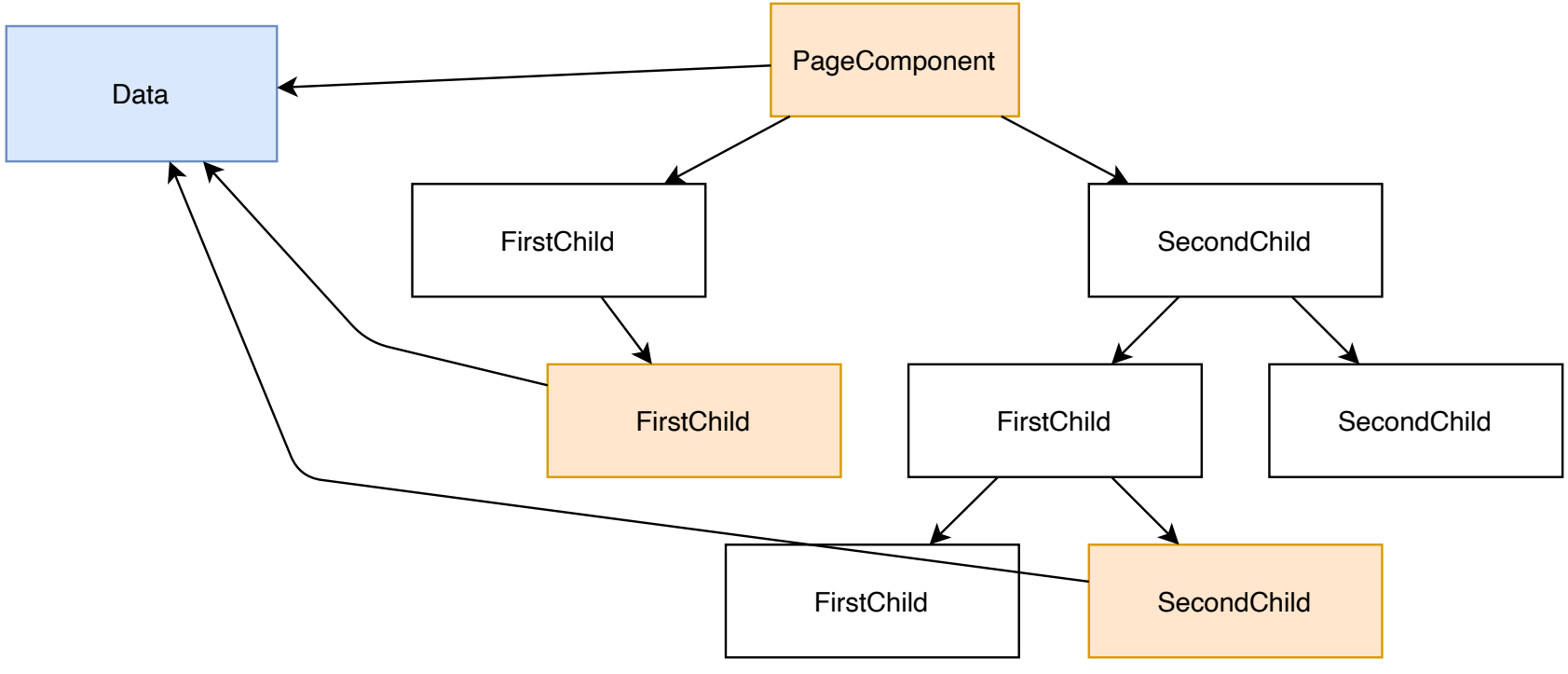
```
 1 public void someMethod() {
 2     var someData = new SomeData("some-title");
 3
 4     someOtherMethod(someData);
 5
 6     assert someData.getTitle().equals("some-title");
 7 }
 8
 9 private void someOtherMethod(SomeData data) {
10     data.setTitle("some-other-title");
11 }
```

Speaker notes

- mit Methoden in Java verhält es sich ähnlich wie mit Components in Component Architekturen
- Seiteneffekte in Components / Methoden führen oft zu ungewolltem Verhalten
- es wird schwer nachzuvollziehen, warum und was grade passiert
- auch in Components können Seiteneffekte die Daten verändern und damit kann es zu merkwürdigen Effekten führen

STATE MANAGEMENT

LÖSUNG FÜR DAS DATENMANAGEMENT



Speaker notes

- an welches Pattern erinnert euch dieser Aufbau?

SINGLETON ODER GLOBAL STATE

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

Speaker notes

- erinnert an ein Singleton, das oft als Antipattern beschimpft wird
- ein Global State bringt viele Nachteile mit, wenn er schlecht gebaut ist

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.

Speaker notes

- theoretisch müssen die Aufrufe nicht unkontrolliert im ganzen Projekt statt finden
- besonders in Entwicklungsteams mit mehreren Entwicklern (junior wie senior), wird es dazu kommen

MÖGLICHE NACHTEILE EINES GLOBAL STATES

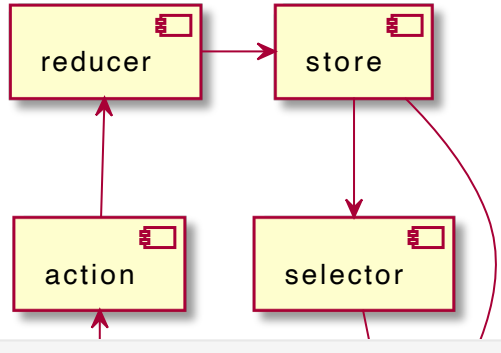
- Unvorhersehbar
  - jeder kann den State bearbeiten
  - Seiteneffekte
- keiner weiß:
  - warum sich der State ändert

Speaker notes

- Projekte mit dieser Art Global State werden schnell:
  - unübersichtlich
  - schlecht wartbar
  - schlecht erweiterbar
  - Seiteneffekte wollen wir ja eigentlich mit State Management vermeiden
- trotzdem wollen wir einen Global state verwenden
- schließlich wollen wir uns den Code sparen, um Daten durch die Components zu reichen

GLOBAL STATE RICHTIG

State Management mit NgRx



Speaker notes

- um ein besseres State Management zu erhalten müssen erstmal die Abhängigkeiten aufgetrennt werden
- es gibt nicht nur noch den einen Global State, auf den jeder zugreifne kann
- das State Management wird in mehrere Komponenten aufgesplittet
- in den meisten Fällen gibt es Actions, Reducer und einen State
- diese Grafik zeigt State Management mit NgRx
  - hier kommen zusätzlich Selectors hinzu
  - diese sind nicht zwingend notwendig
- im Folgenden gehen wir genauer auf die einzelnen Komponenten ein

STORE

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

Speaker notes

- State bildet den ganzen Anwendungsstate ab
- oftmals lohnt es sich den State in logische Teile zu unterteilen

REDUCER

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

Speaker notes

- er reduziert die Action und den State auf einen neuen State

ACTION

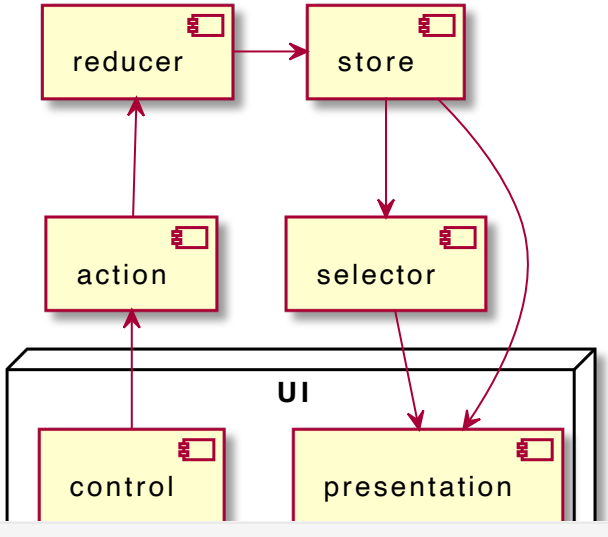
- eine Interaktion des Nutzers
- löst Logik im Reducer aus
- wird asynchron an einen Reducer dispatched

Speaker notes

- Actions können natürlich auch Daten an den Reducer leiten
- z.B. wenn der Nutzer etwas eingegeben hat

FUNKTIONSWEISE DES STATE MANAGEMENTS

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



Speaker notes

- Änderungen werden nur über Actions ausgelöst
- der State ist immutable
- Reducer enthalten Logik zum verändern des States

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

Speaker notes

- kein unnötiger Code mehr zur Weitergabe von Daten
- Seiteneffekte werden vermieden. Action führt zu neuem State. State zu einem UI rerender.
- in großen Projekten werden Daten in den verschiedensten Components geladen. Somit kann es schnell passieren, dass Daten doppelt geladen werden
- FRP wird häufig eingesetzt um Anwendungen Nutzerfreundlicher und reaktiver zu gestalten.
- State Management arbeitet quasi mit FRP
- mit einem zentralen State hat man zu jeder Zeit einen Überblick über die Anwendung und kann Fehler schneller finden.

DATEN VOM BACKEND LADEN?

STATE MANAGEMENT GUT UND SCHÖN ...

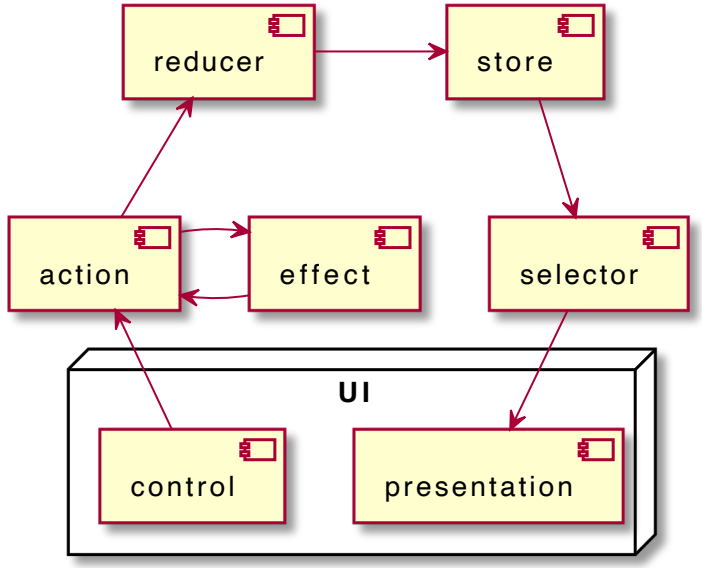
- Wie bekomme ich Daten vom Backend?
- Reducer arbeitet nur synchron
- Daten über eine Action übergeben?
- Backend Call aus der Component?

Speaker notes

- Daten sollten natürlich nicht in einer Component geladen werden
- Components sollten nur fürs Rendering da sein
- sie dürfen gerne auch Actions des Nutzers an den Reducer weiterleiten
- mehr aber bitte nicht!

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

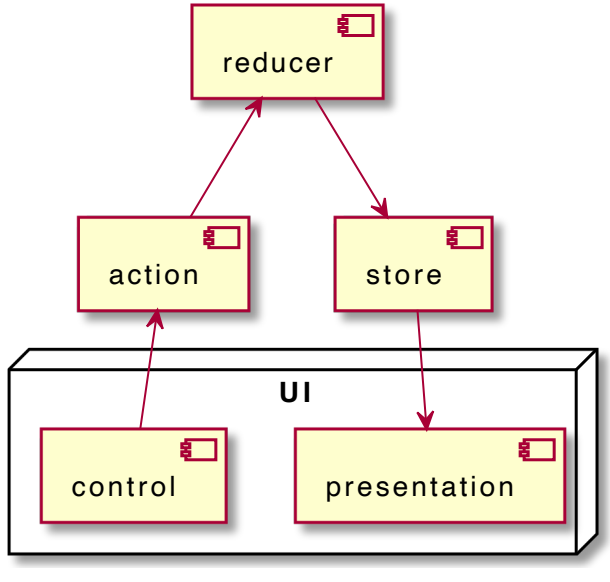
- unzählige State Management Frameworks
- jedes Frontend Framework bringt ein eigenes mit
- unter der Haube funktionieren alle gleich

Speaker notes

- jedes "Frontend" Framework ist bewusst gewählt. Hier geht es nicht nur um Web, sondern auch um App.
- Wie anfangs erwähnt sind Apps sehr ähnlich zu SPA Anwendungen
- wir schauen uns jetzt ein paar Beispiele an

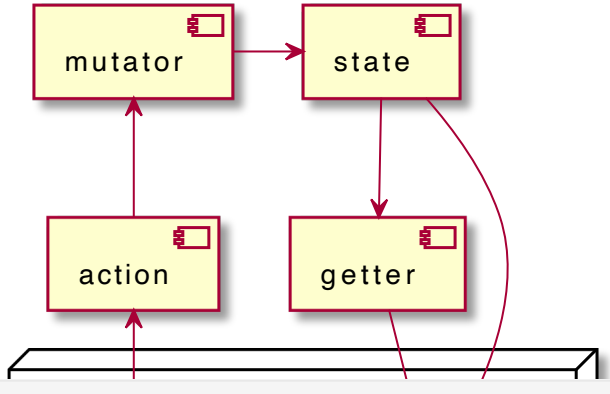
REDUX

- 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

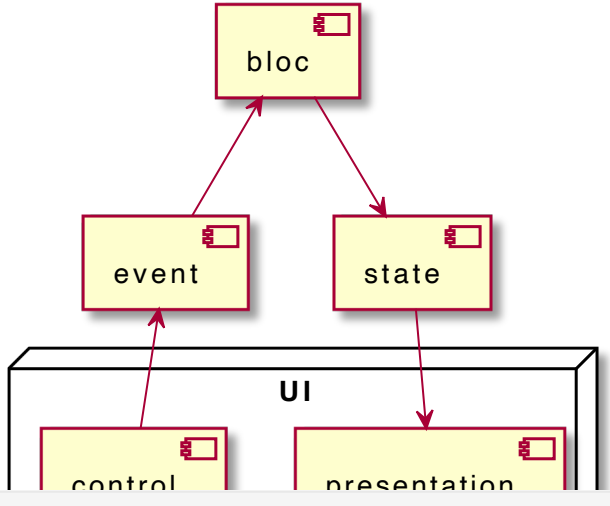


Speaker notes

- wie NgRx nur mit anderen Namen
- es sieht nach mehr "magie" auf den ersten Blick aus
- ich habe allerdings nur mal das Tutorial durchgeklickt
- passt allerdings mit der Aussage eines Kollegen von mir überein, der meinte das Vue allgemein sehr anfängerfreundlich sei

BLOC

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

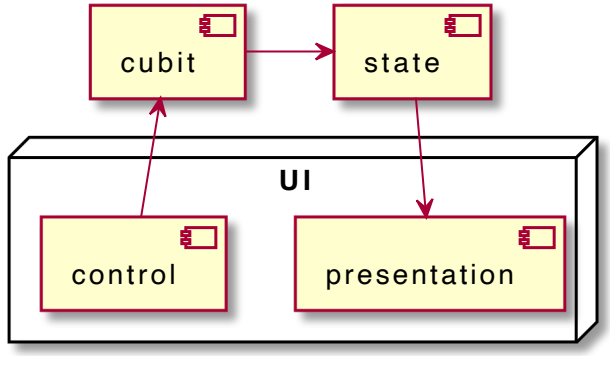


Speaker notes

- Bestandteile wurden wieder nur umbenannt
- im Kern steckt dasselbe Pattern dahinter

BLOC (CUBIT)

- Control ruft Methoden des Cubits auf
- leichtgewichtiger Cubit



Speaker notes

- Bloc's und Events sind in vielen Fälle zu überdimensioniert
- deshalb hat man sich entschieden Cubits zu nutzen

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 }
```

Speaker notes

- habt ihr ja bereits in eurer Anwendung und kennt ihr aus der ersten Praxisübung

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',
```

Speaker notes

- Identifier
  - Source soll zeigen wo diese Action geworfen wird
  - Event beschreibt was diese Action bewirkt
- Properties werden über die Funktion props<>() hinzugefügt
  - werden normalerweise in ein Objekt gewrapped um künstliche "named" Parameter zu haben

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([]);
```

Speaker notes

- Initialer State kann auch Daten enthalten
- State muss Readonly (Immutable) sein, weil der State von keinem einfach so verändert werden darf
  - Erinnerung an die Regeln des State Managements
- Reducer wird über die Methode "createReducer" erzeugt
  - bekommt den initialen State
  - außerdem werden hier die Handler für die Actions definiert
- Handler
  - für jede Action die dieser Reducer bearbeiten soll
  - es macht keinen Sinn eine Action von mehreren Reducern bearbeiten zu lassen
  - ich habe den Handler als Funktion ausgelagert
  - dies muss nicht sein, man kann es auch als anonyme Funktion schreiben
  - Wichtig: der Handler verändert den State nicht, sondern erzeugt einen neuen!

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()),
);
```

Speaker notes

- mit der Selector Funktion wird aus dem AppState ein Teilstate ausgelesen
- mit dem Projector kann man diesen Teilstate umwandeln (projezieren)
- in diesem Fall wird aus dem State die Map von SomeData ausgelesen und ein Array aus dem Values gemacht

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 }
```

Speaker notes

- ein Effect fängt Actions ab und führt seinen Code aus, z.B. Backendcalls
- Actions werden wir herkömmliche Injectables in die class injected

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(
```

Speaker notes

- mit *createEffect()* wird ein Effect erzeugt
- anschließend wird auf den Actions subscribed (piped)
  - ofType filtert die richtigen Actions -> es können auch mehrere sein
  - mergeMap sorgt dann dafür, dass asynchroner Code ausgeführt werden kann
  - in der mergeMap muss eine Action zurückgegeben werden
  - dies kann die eingehende Action, aber auch eine neue sein
  - Fehlerhandling sollte beachtet werden, falls ein Fehler auftreten kann

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