Anleitung - NgRx

Freitag, 1. September 2023 09:55

Ableitung Beispiel: Zähler/Counter

Version 1

stateService.ts

```
  private state: State = {
    counter: 0,
    text: 'foo'
  }
  state$ = new BehaviorSubject<State>(this.state);
-------------------------------------------------------->
  incrementCounter() {
    this.state.counter++
    this.state$.next(this.state);
  }
-------------------------------------------------------->
```

stateComp.ts

```
  counter$ = this.service.state$.pipe(
    map(state=>state.counter)
  );
```

stateComp.hmtl

```
  {{ counter$ | async }}
  <button (click)="service.incrementCounter()">
```

Version 2 (Pseudo Immutable)

```
  incrementCounter() {
    this.state = {
      ...this.state,
      counter: this.state.counter +1
    };
    this.state$.next(this.state);
  }
```

Version 3 (Nachrichten)

werden auch Actions genannt

stateComp.ts

```
  increment() {
    this.service.dispatch('INCREMENT');
  }
```

Nachrichten sind INCREMENT, DECREMENT, RESET
service.incrementCounter wird nicht mehr benutzt

Version 4 (Funktion auslagern)

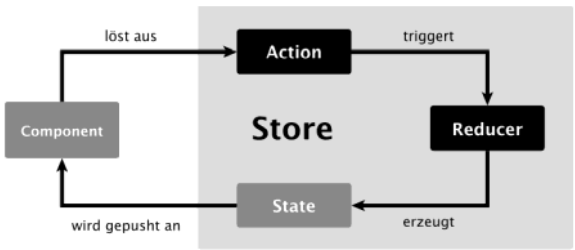
Reducers verarbeiten Nachrichten

stateFunc.ts

```
export function calculateState(state: State, message: string): State {
  switch(message) {
    case 'INCREMENT':{
      return {
        ...state,
        counter:state.counter +1
      }
    } ...
```

stateService.ts

```
  dispatch(message: string){
    this.state = calculateState(this.state, message);
    this.state$.next(this.state);
  }
```



Version 5 (Nachrichten im Stream)

stateService.ts

```
  private message$ = new Subject<string>();
  private state: State = {
    counter: 0,
    text: 'foo'
  }
```

Ein Subject wird hinzugefügt, welches von allen Komponenten die Nachrichten aufsammelt.

```
  readonly state$ = this.message$.pipe(
    startWith('INIT'),
    scan(calculateState, this.state),
    shareReplay(1)
  );
```

Das ehemalige BehaviorSubject nimmt alle Nachrichten auf und jagt sie durch die Funktion calculateState (hier als Callback des scan Operators)

```
  dispatch(message: string){
    this.message$.next(message);
  }
```

dispatch reicht die Nachrichten nur noch weiter

NgRx Installieren

```
ng add @ngrx/store --defaults
ng add @ngrx/store-devtools
ng add @ngrx/effects
ng add @ngrx/schematics
ng add @ngrx/entity
```

Struktur von NgNx

Jedes Feature erhält einen eigenen Satz an Actions, Reducers und Effects. Sie sind nur für dessen State verantwortlich. Alle Dateien dazu sollten in einem Unterordner gesammelt werden.

Eine Convention beschreibt Actions in drei Teile: name(), nameSuccess() und nameFailure() Das ergibt eine ActionGroup

```
export const BookActions = createActionGroup({
  source: 'Book',
  events: {
    'Load Books': emptyProps(),
    'Load Books Success': props<{ books: Book[] }>(),
```

Begriffe genauer erklärt

Store: Das zentrale Datenobjekt in dem der gesamte State der Anwendung gespeichert wird. Es ist ein Observable.

Actions: Aktionen sind Informationspakete, die Daten von der Anwendung zum Store senden. Jede Action hat einen Typ und, wenn nötig, zusätzliche Daten. Man kann sich Actions wie Befehle oder Intentionen vorstellen, die etwas in deinem Store ändern sollen. Zum Beispiel könnte eine Action den Typ "ADD_ITEM" haben und als zusätzliche Daten das hinzuzufügende Element enthalten.

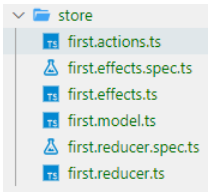
Reducers: Reducers sind Funktionen, die den aktuellen Zustand und eine Action entgegennehmen und einen neuen Zustand zurückgeben. Sie sind dafür verantwortlich, den State basierend auf der gegebenen Action zu aktualisieren. Es ist wichtig zu betonen, dass Reducers "rein" sind, d.h. sie ändern den aktuellen Zustand nicht direkt, sondern geben einen neuen Zustand zurück. Wenn zum Beispiel eine Action vom Typ "ADD_ITEM" ankommt, gibt der Reducer einen neuen State zurück, in dem dieses Element hinzugefügt wurde.

Effects: Effects sind mit NgRx eingeführt und sie handhaben

```
export const BookActions = createActionGroup({
  source: 'Book',
  events: {
    'Load Books': emptyProps(),
    'Load Books Success': props<{ books: Book[] }>(),
    'Load Books Failure': props<{ error: HttpErrorResponse }>(),
  }
});
```

Erstellen eines Store Ordners
NgNx erstellt einen Ordner mit allen benötigten Files.
Dafür wird dieser Befehl verwendet:

```
ng g feature store/book --api --defaults
```



Store beschreibt den Namen des neuen Ordners. Ich finde store ist ein guter Name, es sagt klar was drin ist. Book ist der Präfix der neuen Dateien. Auf diesem Bild ist es first, weil es mein erster Versuch war
(Ich vermisse die Datei für Selectoren …)

"ADD_ITEM" ankommt, gibt der Reducer einen neuen State zurück, in dem dieses Element hinzugefügt wurde.

Effects: Effects sind mit NgRx eingeführt und sie handhaben "nebenläufige" oder "asynchrone" Zum Beispiel Daten von einem Server abrufen. Das wäre eine asynchrone Aufgabe. Anstatt dies direkt in dem Reducer zu tun, würdest man einen Effect verwenden. Der Effect würde die Daten abrufen und dann eine neue Action auslösen, um den State mit den abgerufenen Daten zu aktualisieren. Es wird benutzt um mit Services zu kommunizieren.

Selectors: Das sind Funktionen die verwendet werden um nur Teile aus dem State abzurufen. Es sind die Hauptkomponenten um Daten aus dem Store in die Komponente oder den Service zu bringen.

Entity: Sie stehen für Objekte in einer Objektsammlung. NgRx bietet eingebaute Methoden für CRUD-Operationen. Es bietet einen effizenten Weg um Objekte abzufragen und zu verändern, was die Performance bei großen Datensätzen verbessert und den Code einfacher macht.

https://angular.schule/blog/2018-06-5-useful-effects-without-actions

---

Ein State mit NgRx für Standalone Komponenten

0. app.config (Bekantmachung)

```
providers: [  ...
              provideStore(),
              provideState(myFeatureKey,myReducer),
              provideStoreDevtools({ maxAge: 25, logOnly: !isDevMode() }),
              provideEffects(MyEffects)
           ]
```

---

1. name.reducer.ts (State einrichten)
Hier befindet sich ein Interface vom State und ein initialState als Konstante. Dieser beschreibt den Zustand der Anwendung beim Start. Im State werden alle Daten und Zustände der Anwendung gespeichert.

In dieser Datei wird auch ein FeatureKey angelegt. Eine Konstante mit einem String. Dieser wird später mit den Selektoren verbunden. Er wird auch bei den Providern angegeben, zusammen mit dem Reducern.

```
export const myFeatureKey = 'myFeature';

export interface FeatureState {
  data: string;
}
export const initialState: FeatureState = {
  data: 'nothing',
};
```

---

2. name.actions.ts (Nachrichten beschreiben)
Bevor die Funktionen geschrieben werden, sollten erst die Actions erstellt werden. Wichtig ist das sie den Namen des Events bekommen und nicht den ihrer technischen Aufgabe. Ein Ation hat diesen Aufbau:

```
{
  type: 'Eindeutiger Name',
  payload?: any;
}
```

Ein solches Objekt wird nicht manuell erzeugt, um Fehler zu vermeiden macht das eine Funktion, die createAction()

```
export const myActions = createAction(
  '[Etwas] ist erfolgreich gewesen',
  props<{data: string}>()
);
```

Für eine HTML-Schnittstelle braucht man normalerweise drei zusammengehörige Actions, Load, Success und Failure.

---

3. store.dispatch() (Nachrichten an den Store schicken)
Die Komponenten senden ihre Nachrichten, die Actions in den Store mit der Funktion dispatch() Der Store kann als Service in die Komponente injiziert werden.

```
constructor(private store: Store<FeatureState>){}

doAny(data: string){
  this.store.dispatch(myActions({data}))
}
```

---

4. name.reducers.ts (State aktualisieren)
Eine Reducher-Funktion bekommt zwei Übergabewete, den aktuellen State und die eintreffende Action.

```
export reducer(state: State, action: Action): State{};
```

Statt mit switch/case die Nachrichten zuzuordnen wird in NgRx eine weitere Funktion zur verfügung gestellt, das createReducer() Jede Action bekommt einen eigenen Reducer. Der Reducer hat statt einem Switch die Funktion on() für jede Action. Das erste Argument ist immer die Action und das zweite Argument ist die Funktion welche ausgeführt werden soll.

```
export const myReducer = createReducer(
  initialState,
  on(myActions, (state, action)=>({...state, data: action.data})),
);
```

Für alle unbekanten Action liefert der Reducer den aktuellen State unverändert zurück.

Reducer sind "Pure Funktionen"
- **deterministisch** (gleiche Eingabe, gleiche Ausgabe)
- **keine äußeren Zustäne** (kein Kontakt nach außen)
- **keine Seiteneffekte** (kein Einfluss auf die Außenwelt)

---

5. name.selectors.ts (Daten aus dem State selektieren)

Um nicht immer den gesamten State (Ein Observable) zu senden gibt es die Selectoren. NgRx bring auch hier Funktionen mit die das Händling vereinfachen. Jeder Selector speichert seine zuletzt verarbeiteten Eingabewerte (Memoization) Auch Selectoren müssen Pure Funktionen sein. Mit der ersten Konstante wird der FeatureKey mit dem State verbunden. Jeder Selector hat als ersten Übergabewert diese Konstante. Der zweite Überabewert enthält den ausgewählten State-Wert

```
export const selectMyFeatureState = createFeatureSelector<FeatureState>(myFeatureKey);

export const selectMyStateData = createSelector(
  selectMyFeatureState,
  (state: FeatureState) => state.data
);
```

Die Daten können vorher auch gefiltert werden, oder mit mehreren States verbunden werden, indem mehrere FeaturerKeys dem Selektor übergeben werden. Die Funktionen können mit einer Datenbankabfrage verglichen werden.

---

6. name.component.ts (Daten ins Template bringen)

Die Daten im Template werden mit der Async-Pipe abonniert. Im Constructor wird über den Selector auf die benötigten Dateien zugegriffen und einer Membervariable zugewiesen

```
data$: Observable<string> = of('');

constructor(public service: MyStateService, private store: Store<FeatureState>){
  this.data$ = this.store.select(selectMyStateData);
}
```

---

7. name.actions.ts (ActionGroup erstellen)

Um nun ein Effect zu erstellen, der seine Infos von einer API bekommt, ist es sinnvoll eine actionGroup zu erstellen. Das ist eine Sammlung zusammengehöriger actions. Ihr typischer Aufbau sieht so aus:

```
export const MyEffects = createActionGroup({
  source: 'Random Value API',
  events:{
    'Load':emptyProps(),
    'Load Success':props<{ digit: number }>(),
    'Load Failure':props<{ error: any }>(),
  }
});
```

source ist der Name der Gruppe und soll beschreiben worum es geht
events ist ein Objekt mit der Auflistung aller actions
emptyProbs wird verwendet wenn die action keine zusätzlichen Daten annimmt
props beschreibt die Datenstruktur der actions

Eingesetzt werden die Actions dann so

```
MyEffects.load
MyEffects.loadSuccess({digit: 12})
MyEffects.loadFailure({error:'error'})
```

---

8. name.reducer.ts (Reducer mit ActionsGroup)

Die Reducher-Funktion wurde um die neuen actions erweitert. Die loadFailure action wurde so beschrieben das sie einen error sendet. In diesem Beispel wird mit dem Wert nichts gemacht. Der State hat noch einen weiteren Wert bekommen, das loading.
(Da ich bei Schritt 7 bei 'Load' keine Daten übermittle wird sich loading hier nicht verändern)

```
export const myReducer = createReducer(
  initialState,
  on(myActions, (state, action) => ({ ...state, data: action.data })),

  on(MyEffects.load, (state) => ({
           ...state,
           loading: true,
         })),
  on(MyEffects.loadSuccess, (state, { digit }) => ({
           ...state, digit,
           loading: false,
         })),
  on(MyEffects.loadFailure, (state) => ({
           ...state,
```

```
          loading: false,
        }))
);
```

---

9. name.effects.ts (Effect erstellen)

Effekte benutzen die Higher-order Observable, zu denen weiter unten noch genauerer Erklärungen stehen. NgRx bringt einen eigenen Operator mit, den **ofType(<action>)** Der Übergabewert ist die gewünschte Action. Mit einem weiteren Flattening-Operator werden die Observable zu einem flachen vereint. Hier wird die die Funktion im Service aufgerufen. Diese gibt ein Observable zurück. Mit map() und catchError() werden die actions getriggert für den Fall das der Wert erhalten wird, oder ein Fehler auftritt.

```
loadMyDigit$ = createEffect(() => {
  return this.actions$.pipe(
    ofType(MyEffects.load),
    mergeMap(() =>
      this.service.getRandomValue().pipe(
        map((digit) => MyEffects.loadSuccess({ digit })),
        catchError((error) => of(MyEffects.loadFailure({ error })))
      )
    )
  );
});
```

---

10. name.selectos.ts (Konsolenausgabe)

Die Selectoren für die Effekte funktionieren genau so wie bei normalen Actions. Darum nutze ich diesen Platz um zu zeigen wie Konsolenausgaben in Selectoren oder Reducern eingefügt werden können, was beim debuggen hilft

```
export const selectMyStateDigit = createSelector(
  selectMyFeatureState,
  (state: FeatureState) => {
    console.log('Selector:', state.digit);   <------------------
    return state.digit;
  }
);

on(MyEffects.loadSuccess, (state, { digit }) => {
console.log('Reducer: ', digit);             <------------------
return {
  ...state,
  digit,
  loading: false,
};
```

---

11. name.componente.ts (Effekt einbinden)

Effekte können von OnInit starten oder durch eine Funktion ausgelöst werden. Auch hier unterscheidet es sich nicht von normalen actions. Im Template werden sie mit einer Async-Pipe eingebunden.

```
digit$: Observable<number> = of(0);
loading$: Observable<boolean> = of(false);

constructor(private store: Store<FeatureState>){
  this.digit$ = this.store.select(selectMyStateDigit);
  this.loading$ = this.store.select(selectLoadingStatus);
}

ngOnInit(): void {
  this.store.dispatch(MyEffects.load());
}
```

Higher-order Observable (Flattening)

Effekte benutzen die Higher-order Observable, also ein Observable das ein Observable ausgibt. Das hier währe ein solches Observable:

```
fileObservable = urlObservable.pipe(map((url) => http.get(url)));
```

Um mit diesen Observablen arbeiten zu können muss es "flach" gemacht werden. Dafür wird ein Observable erstellt das die Werte des Inneren Observable ausgiebt. Wie es das genau macht kann mit vier Operatoren gesteuert werden.

- **concatAll()**: Abonniert jedes innere Observable nacheinander und gibt alle seine Werte aus.
- **mergeAll()**: Abonniert alle inneren Observables gleichzeitig und gibt Werte aus, sobald sie ankommen.
- **switchAll()**: Abonniert das erste innere Observable und gibt dessen Werte aus. Sobald ein neues inneres Observable ankommt, wird das alte deabonniert und das neue abonniert.
- **exhaustAll()**: Abonniert das erste innere Observable und ignoriert alle weiteren, bis das erste abgeschlossen ist.

Es gibt Operatoren die sowohl flattern als auch mappen können. Das ist sehr nützlich bei komplexen Aufgaben.

concatMap(): Führt für jeden Wert ein Mapping durch und wartet, bis das innerhalb des Mappings erzeugte Observable abgeschlossen ist, bevor es mit dem nächsten fortfährt.

mergeMap(): Führt das Mapping für jeden Wert durch und abonniert alle resultierenden Observables gleichzeitig, gibt deren Werte dann parallel aus.

switchMap(): Sobald ein neuer Wert erscheint, wird das aktuell abonnierte Observable deabonniert und ein neues Observable abonniert. Nur das aktuellste Observable wird beachtet.

exhaustMap(): Ignoriert alle neuen Werte, bis das aktuelle Observable abgeschlossen ist. Erst dann wird ein neues Observable abonniert.

Hinzufügen von Entity

1. name.entity.ts (Entity beschreiben)
In einer neu erstellten Datei wird zuerst das nötigste importiert

```
import { EntityState, EntityAdapter, createEntityAdapter } from '@ngrx/entity';
```

Dann wird die Datenstruktur des Entity beschrieben und den State. Dieser muss von EntityState<Entity> erben und die Datenstruktur mitgeben. Im State können die Zustände des Entity beschrieben werden

```
export interface User {
  id: string;
  name: string;
}

export interface UserState extends EntityState<User>{
  isAnything: boolean;
}
```

Um die Entyties leichter zu pflegen wird ein Adapter zur verfügung gestellt. Er benutzt die Eigenschaft id als Primärschlüssel. Soll eine andere Eigenschaft diese Rolle übernehmen muss die Funktion ein selectId angeben

```
export const adapterById: EntityAdapter<User> = createEntityAdapter<User>();

export const adapterByName: EntityAdapter<User> = createEntityAdapter<User>({
  selectId: user => user.name,
});
```

---

2. name.reducer.ts (In bestehendes State einbinden)
Der neue State kann in den schon vorhandenen eingehangen werden.

```
export interface FeatureState {
   data: string;
   digit: number;
   loading: boolean;
   users: UserState;            <--------------------------
}

export const initialState: FeatureState = {
  data: 'nothing',
  digit: 10,
  loading: false,
  users: adapterById.getInitialState({      <--------------------------
    isAnything: false,
  })
};
```

---

3. name.action.ts (CRUD)
Bevor die Reducer-Funktionen für die Entities erstellt werden können, müssen die Actions erstellt werden. In diesem Beispiel habe ich die CRUD Funktionen genommen und sie in einer ActionGroup beschrieben

```
export const UserActions = createActionGroup({
  source: 'User API',
  events: {
    'Add User': props<{ user: User }>(),
    'Update User': props<{ user: User }>(),
    'Delete User': props<{ id: string }>(),
    'Load Users': emptyProps(),
    'Load Users Success': props<{ users: User[] }>(),
    'Load Users Failure': props<{ error: any }>(),
  },
});
```

---

4. name.reducer.ts (Funktionen des Adapters)
Der Adapter stellt eine Reihe an Methoden zur Verfügung, mitdenen die Sammlung an Entities verwaltet werden kann. Diese werden in den Reducer-Funktionen eingesetzt. Diese fertigen Methoden sind:

```
- addOne      - removeOne     - updateOne
- addMany     - removeMany    - updateMany
- setAll      - removeAll     - upsertOne
- setOne                      - upsertMany
- setMany                     - map
```

```
on(UserActions.addUser, (state, { user }) => ({
  ...state,
  users: adapterById.addOne(user, state.users),
})),
on(UserActions.updateUser, (state, { user }) => ({
  ...state,
  users: adapterById.updateOne({ id: user.id, changes: user }, state.users),
})),
on(UserActions.deleteUser, (state, { id }) => ({
  ...state,
```

```
    ...state,
    users: adapterById.updateOne({ id: user.id, changes: user }, state.users),
  })),
  on(UserActions.deleteUser, (state, { id }) => ({
    ...state,
    users: adapterById.removeOne(id, state.users),
  })),
```

---

5. name.selectors.ts (Selectoren des Adapters)

Der EntityAdapter hat zu den Funktionen noch ein paar Selektoren dabei. Diese werden mit adapter.getSelector() erzeugt und dann zugewiesen. Es gibt zwei Herangehensweisen, die eine ist generischer und eignet sich wenn man auf den gesamten State zugreifen möchte ...

```
const { selectIds, selectEntities, selectAll, selectTotal } = adapter.getSelectors();

export const selectUserIds = selectIds;
export const selectUserEntities = selectEntities;
export const selectAllUsers = selectAll;
export const selectUserTotal = selectTotal;
```

---

6. name.selectors.ts (Selectoren für einen Bestimmten State)

... die zweite Methode ist spezifischer und bezieht sich auf einen bestimmten Teil des States. In diesem Beispiel soll auf die User-Entities zugegriffen werden. Dafür muss der UserState, der sich im übergeordneten FeatureState befindet extrahiert werden, dann können sie den Selektoren zugewiesen werden.

```
export const selectUserState = createSelector(
  selectMyFeatureState,
  (state: FeatureState) => state.users
);

export const {
  selectIds: selectUserIds,
  selectEntities: selectUserEntities,
  selectAll: selectAllUsers,
  selectTotal: userCount,
} = adapterById.getSelectors(selectUserState);
```

---

7. name.componente.ts (Wiederholung)

Jetzt kann die Liste mit den Entitäten im Template dargestellt werden. Das funktioniert genau so wie bei den anderen Statewerten.

```
users$: Observable<User[]>;
constructor(private store: Store) {}
ngOnInit(): void {
  this.users$ = this.store.select(selectAllUsers);
}
```

```
<ul>
  <li *ngFor="let user of users$ | async">
    {{ user.name }}
  </li>
</ul>
```