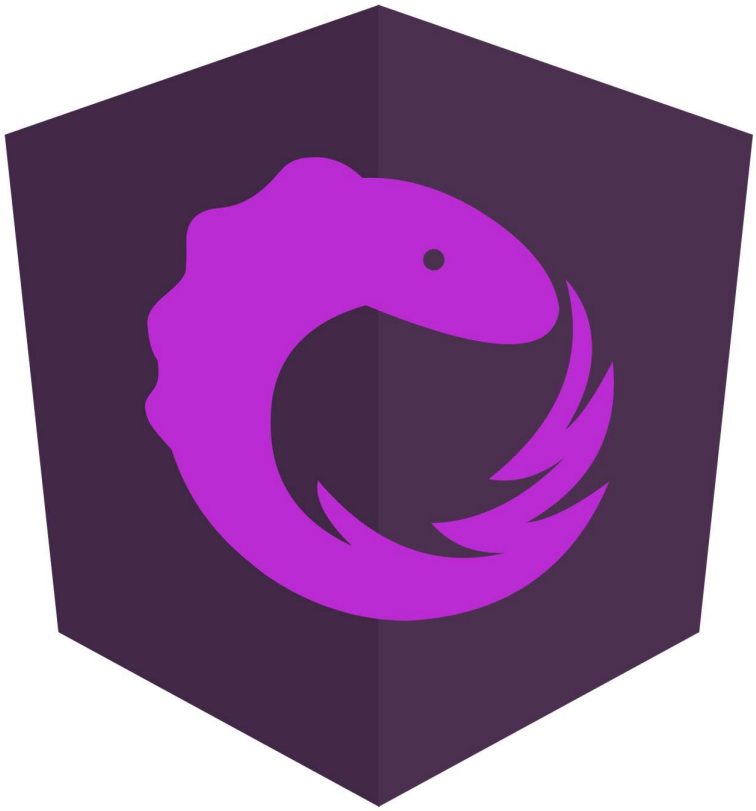
NgRx – Redux for Angular



En Synopsis af

Jakob Hammer

4. semester

Vejleder: Henrik Kryger Høltzer

Tegn med mellemrum: 23975

Indholdsfortegnelse

Opgave Introduktion 3
Problemstilling 3
Planlægning 3
Metode 3
Historien bag NgRx 4
Intro til NgRx 5
NgRx arkitektur 6
CLI ngrx/store 7
Action 8
Reducers 9
Store 10
Selectors 11
Effects 12
View 13
NgRx-Lazy Loading 14
Skal Jeg bruge NgRx 15
SHARI 15
Debugging i NgRx 16
Redux Devtool 16
Konklusion 18
Refleksion 19
Litteraturliste: 20
Skriftlige Kilder: 20
Dokumentation: 20
Video Kilder: 20
Url til mit GitHub projekt om NgRx 20

Opgave Introduktion

I denne opgave vil jeg forsøge at bruge og fremvise NgRx. NgRx, et state management library til Angular, bliver brugt i større apps stil at holde styr på appens state. Umiddelbart har Angular allerede Service klasserne, der kan dependency injectes, til at holde styr på state, men NgRx tager skridtet videre og opbevarer al staten samlet i et objekt.

Jeg vil samtidig forsøge at lave en app, der bruger Angular services i et component, og NgRx i et andet, så vi kan sammenligne tilgangen, og hvor stor en fordel NgRx er.

Problemstilling

Hvad er NgRx?

Hvordan kan NgRx gøre en Angular app bedre?

Hvad er forskellen mellem NgRx og normale Angular-services?

Planlægning

Uge	Aktivitet	
17	Research af emne	
18	Synopsis-skrivning og programmering af projekt	
19	Synopsis-rettelser	
20	Synopsis-rettelser	
21	Synopsis-rettelser	Aflevering d. 28/05/2021

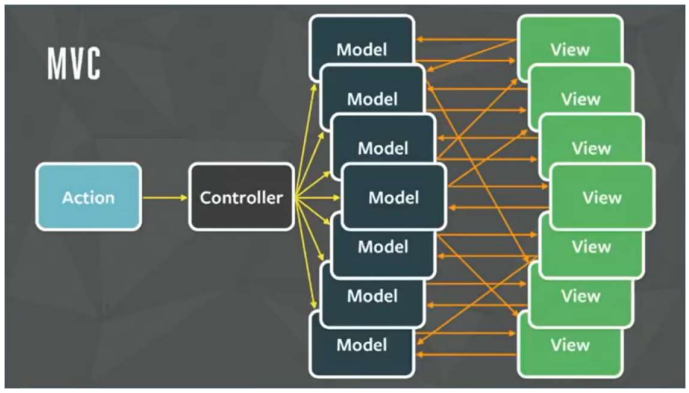
Metode

Jeg vil arbejde med min synopsis på følgende måder:
Kursus på Pluralsight + tilknyttede opgaver.
Herefter skriver jeg min synopsis samtidig med at jeg laver min app.
Jeg synes at denne måde er optimal, da før jeg skriver et emne har friske erindringer om hvordan arkitekturen hænger sammen, og hvordan koden skal skrives, samtidig tror jeg det hjælper på min forståelse at jeg nedskriver de tanker og det jeg opnår når jeg laver appen. Det tvinger mig til at undre mig over hvorfor man gør det på en bestemt måde.

Historien bag NgRx

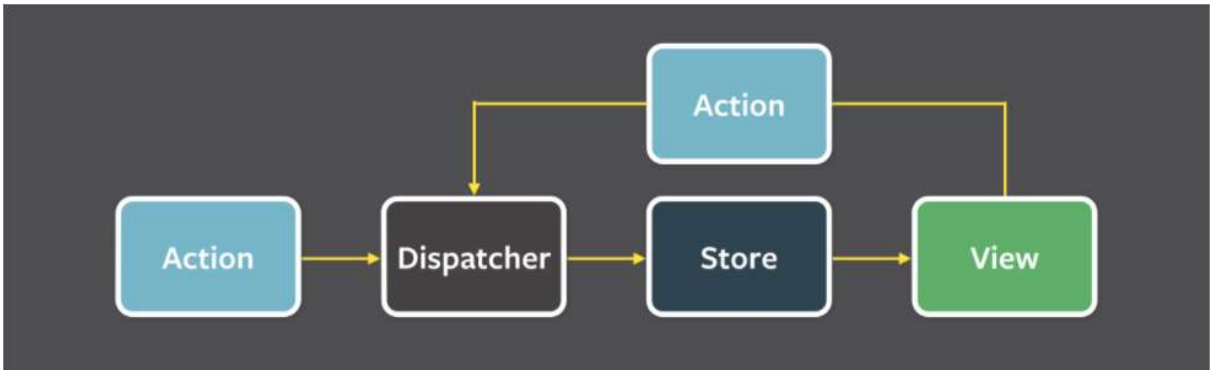
NgRx er et Angular library inspireret af Redux til React, Redux er inspireret af Flux, som er et state management library udviklet af Facebook, da de havde problemer med en "unseen-message-counter"[1].

Da Facebooks udviklere udviklede Messenger appen brugte de et lille tal til at fortælle brugere om hvor mange ikke-sete beskeder de havde i deres indbakke. Da Facebook derefter videreudviklede Messenger konceptet blev de mødt af en bug.
Der ville være et lille tal ved beskederne, men når man trykkede på knappen, var der intet nyt at vise. Dette viste sig at være en bug i deres state. På dette tidspunkt brugte Facebook arkitekturen MVC, problemet ved dette var at det var ekstremt svært at få et overblik over hvilke påvirkninger de forskellige komponenter havde på hinanden.



Figur 1 https://youtu.be/nYkdrAPrdcw?t=644

Det er tydeligt at se på billedet (figur 1). at det er meget svært at overskue hvor modellerne og views interagere. Dette kan hurtigt forårsage infinite loops og uventet opførsel.



Figur 2 https://facebook.github.io/flux/docs/in-depth-overview

Når Facebook fik fixet buggen, kom den hurtigt igen ved andre ændringer.

For at løse dette permanent udviklede Facebook Flux.

[1] Video fra Facebook konference "Hacker Way" om Flux: https://youtu.be/nYkdrAPrdcw?t=619

I flux-modellen starter alle ændringer i en ”store” med en action, denne action bliver sendt videre til en dispatcher, som sørger for at der ikke kommer en action ind til storen før den er klar. Storen er hvor data er gemt. Man kan forestille sig storen som en slags model som i mvc, storen i Flux indeholder dog mere data og har mange gange al staten af en del af applikationen. Views’ne er i stand til at opdatere sig selv når storen er opdateret og derved har emittet et event.

Intro til NgRx

NgRx er et state management library, inspireret af React Redux[2]. NgRx introducerer en store i Angular, denne store indeholder det meste af ens state. NgRx følger princippet i Angular om at bruge TypeScript og har derfor Typesikkerhed. Det gør at TypeScript kompilerer ens kode og giver fejl og advarsler, samtidig kan man gøre brug af IntelliSense, dette gør at man bl.a. ikke skal huske hvad ens Actions hedder og at man derfor bliver mere produktiv.

NgRx gør det også nemmere at teste sin app, da al staten ligger i storen og er abstraktet væk fra component’erne.

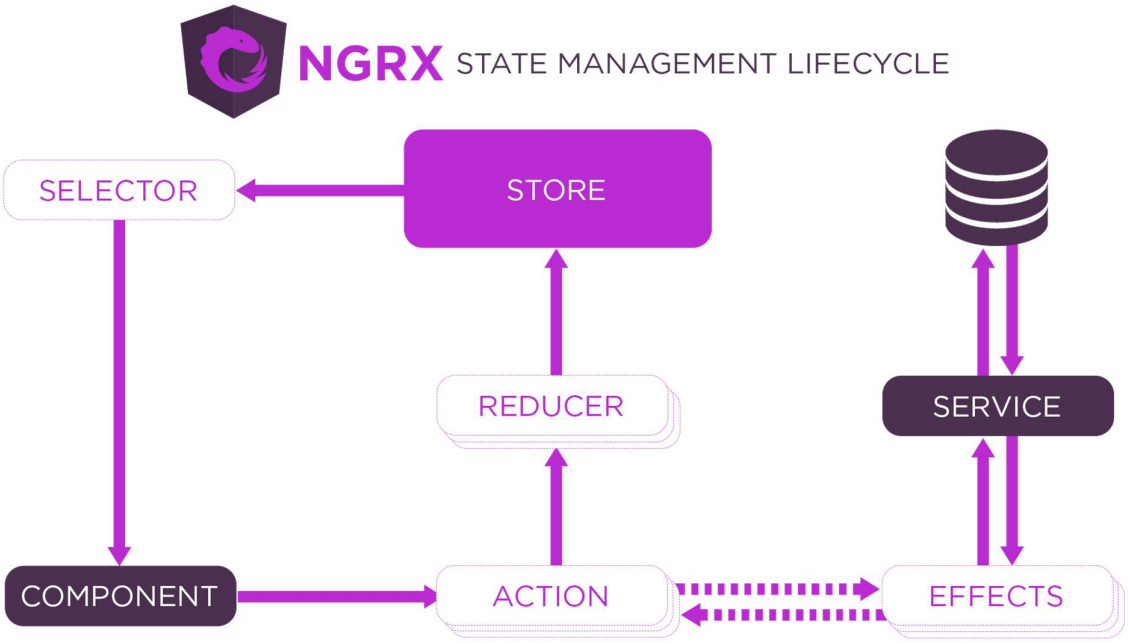
Hvor en af de helt store forskelle fra React Redux selvfølgelig er TypeScript, er en anden at NgRx er bygget ved hjælp af RxJs, hvilket betyder at state ændringer foregår reaktivt ved hjælp af Observables, Observers og subjects. Det betyder bl.a. at man tilgår staten i component’erne ved hjælp af subscription eller async pipe.

[2] https://ngrx.io/guide/store

NgRx arkitektur

Arkitekturen bag NgRx gør det muligt for libray'et at udføre det store ansvar det er at holde styr på staten i en hel applikation og samtidig gøre det uden at have for mange bivirkninger. Som man kan se nedenfor, er hele NgRx delt op i en slags cirkel, vi starter med kernedelen af NgRx, så vi tager effect til sidst.

En af de større ulemper ved NgRx er, at man ikke kan forstå en enkel del før man forstår dem alle. De er alle bundet sammen, og derfor kan det godt tage tid før man har forstået hvordan de forskellige komponenter interagerer med hinanden.



Figur 3 Diagram over arkitekturen i NgRx[3]

Storen: Oversat til lagret. Stedet hvor al staten er opbevaret.

Selector: De handlers der tager en del af staten fra storen og serverer det til view'et, her kan man med fordel tænke på at storen er en Database og selector'en er en slags SQL-SELECT statement.

Component: Component'et er view'et, og i NgRx forsøger vi at holde view'et så pure som muligt, bl.a. ved at undgå at view'et indeholder state.

[3] https://ngrx.io/guide/store

Action: En action er både inputs og outputs, de er en slags forbindelse mellem reducer'en og storen. Man kan forestille sig at de er en slags HTTP-Requests, som bliver sendt bl.a. fra view'et og reducer'en.
Reducer: Dette er de funktioner der skriver til storen, de gør noget når de modtager actionen. Man kan med fordel tænke på dem som en primitiv API, der når de modtager en HTTP-Request gør et eller andet bestemt. F.eks. Poster til databasen (ændrer staten i storen).

Nu da vi har det basale overstået, kan vi med fordel gennemgå **Effects**. Effects er et sted hvor man flytter alle de Actions, der har side-effects, f.eks. API-kald, hen. Derved kan man kalde en API asynkront og effekten svarer så tilbage med en Action hvis den er en succes og en anden hvis den får en fejl.

CLI ngrx/store

For at bruge NgRx bør man gøre brug af Angulars CLI, den kan hjælpe en med at hente og importere NgRx ind i ens Angular app, CLI-kommandoen ser således ud:

```
ng add @ngrx/store@latest
```

Angular opdaterer dependencies i package.json med NgRx.
Og importerer NgRx ind i dit root-modul, herefter adderer den boilerplate kode til importslisten.

NgRx har deres egen overbygning på Angulars CLI, nemlig ngrx/schematics. Schematics tilføjer en masse nye kommandoer til CLI'en, som gør oprettelse af Stores, Actions, osv. meget nemmere.

Dette gør også at den ellers "store" mængde af boilerplate kode man skal skrive, bliver mindsket.

Samtidig er det en god idé at installere NgRx schematics, da vi skal bruge det senere. NgRx schematics er NgRx's udvidelse til Angular CLI.

```
ng add @ngrx/schematics@latest
```

Action

Hvad er en Action?

En Action er et objekt, de bliver ”emittet” ved f.eks. brugerinteraktioner eller API-kald. En action i NgRx kan laves typesikkert, ved hjælp af det simple Action-interface i NgRx, der indeholder en property af typen string[4].

En action bruges til at kommunikerer mellem: reducer, effect og component. Det fungerer ved at en bruger trykker på en knap i UI’en. Component’et dispatcher herefter en action og reducer’en fanger den.

```
3   export const loadEvents = createAction('[Event Page] Load');
```

Figur 4 event-page.action.ts[5]

Her kan man se hvordan man opretter en action; createAction er en funktion, der opretter et objekt af typen Action med propertien ”type” sat til metodens argument. Det er good-practice at man som det første i strengen giver en definition på, hvor actionen hører til.
I dette eksempel vil Event Page betyde at den hører til de kommandoer, som view’et bruger.

```
4   export const loadEventsSuccess = createAction(
5     '[Events API] Load Success',
6     props<{ events: EventElement[] }>()
7   );
```

Figur 5 event-api.actions.ts[6]

Action’en her har en props metode, her definerer man hvilken payload actionen har med, i dette tilfælde en liste af EventElements.

[4] https://github.com/ngrx/platform/blob/master/modules/store/src/models.ts linje 1-3
[5] Kan findes på GitHub
[6] Kan findes på GitHub

Reducers

Hvad er en Reducer?

En reducer er en pure-function, det betyder at den ikke har nogle side-effects og at den er idempotent. Side-effects er f.eks. API-kald. API-kald håndteres af effects, som vi kommer til senere.

```
15  export const eventReducer = createReducer(
16    initialState,
17    on(eventApiActions.loadEventsSuccess, (state, action) => {
18      return {
19        //spreading the state to allow nonchanged state to be copied
20        ...state,
21        //overwriting the events array with the new events
22        events: action.events,
23      };
24    })
25  );
```

Figur 6 event.reducer.ts[7]

Her er et screenshot af en reducer, den tager en InitialState, som er et objekt af vores start state. Herefter tager reducer'en en eller flere on-funktion. On-funktionen tager en action og sammenligner den med de actions den har i dens on-funktion, hvis der er en der passer, kører den body'en af on-funktionen. Herefter har jeg givet on-funktionen en callback funktion. Callback funktionen returnerer et objekt-literal, der bruger JavaScript spread-operator til at sprede state objektet ud og kopiere det. Herefter overskrives events, med den nye events-state, som kommer i form af parameter-objektet "action". Denne parameter kommer fra metoden props, som vi gennemgik i Action-afsnittet. Hele state objektet bliver så returneret til storen, der så opbevarer det indtil en ændring i staten igen sker.

Som man kan se, har vores action typesikkerhed, da den er et objekt, men i princippet kunne man godt give on-funktionen en string. Dette vil dog betyde at man ude i at component'et skulle matche den string eksakt, og det er især her hvor man vil opleve at man kan komme til at stave et eller andet forkert og derfor virker ens actions ikke. Der kommer heller ingen advarsler, da TypeScript ikke kontrollerer at tekststrengene matcher. Derfor er den typestærke fremgangsmåde en lille smule mere kode upfront, men til gengæld vil det være nemmere at implementere nye features og vedligeholde den koder der er skrevet.

[7] Kan findes på GitHub

Store

Hvad er en Store?

Storen i NgRx er et modul hvor alt data'en gemmes og kan hentes fra. Store er global, hvilket betyder at man kan tilgå alt i storen i alle component'erne ved hjælp af selectors. Når brugerne interagerer med en NgRx app, dispatcher de actions til reduceren, storen opdateres og sender den opdaterede state tilbage til component'erne.

Helt konkret er en store en samling Observables[8], subjects og nogle andre RxJs objekter. Storen er abstraktet væk så man behøver ikke bekymre sig om den i NgRx. Andet end at den skal defineres i root-modulets importliste.

Hvad der dog er vigtigt at vide, er at staten er immutable, hvilket betyder at man ikke kan lave ændre i statens properties. For at tilføje ændringer til storen, skal man derfor overskrive hele staten i storen. Fordelen ved dette er at det er nemmere at finde fejl, især når man samtidig bruger NgRx Devtools, i ens browser. Det betyder også at Angulars ChangeDetection system altid bliver kørt, så man ikke risikerer at en ændring ikke bliver opdaget af Angular og derfor ikke renderer.

[8] https://github.com/ngrx/platform/blob/master/modules/store/src/store.ts linje: 11 - 17

Selectors

Hvad er en Selector?

Når nu storen er en Observable indeholdende vores applikations state, kunne man ikke bare subscribe til storen direkte?

```
29        this.store.subscribe((state: State) => (this.events = state.events.events));
```

Figur 7 event-list-ng.component.ts

Dette kan sagtens lade sig gøre og virker også helt fint! Der er dog ét problem, hver gang en del af staten ændrer sig i storen, vil storen pushe en ændring, som så vil få view'et til at opdaterer sig selv, også selvom den del af staten man skal bruge, ikke er blevet ændret. Når man har meget state i sin store, så vil der hurtigt være mange elementer i views'ne der bliver opdateret. Dette er ikke optimalt, da appens hastighed vil blive sænket.

Ind kommer Selectors: En selector er en funktion. Man kan forestille sig at selector'en går ind i storen og tager den del af storen man skal bruge ned af hylden og tager den med ud der hvor den skal bruges.

```
16  const getEventsFeatureState = createFeatureSelector<EventState>('events');
17
18  export const getEvents = createSelector(
19    getEventsFeatureState,
20    (state) => state.events
21  );
```

Figur 8 state/index.ts linje[9] En Feature selector, der henter hele feature state og en normal selector, der henter en del af feature state

En selector er "intelligent", den tager kun staten ned af hylden, hvis der er sket ændringer. Så man slipper for at skulle gentegne siden hver gang der f.eks. bliver trykket på en knap.
Samtidig husker en selector den sidst hentede state. Den cacher simpelthen returværdien, så man kan hente værdien hurtigere næste gang man bruger selector'en.

En selector er kort fortalt en mapping funktion, der cacher tidligere resultater.

[9] Kan findes på GitHub

```
25      this.events$ = this.store.select(getEvents);
```

Figur 9 event-list-ng.component.ts[10]

Her er selector'en brugt, ved at bruge Observable i stedet for den konkrete værdi, har man mulighed for at lade Angular styre subscription og unsubscription med Async pipe'en.

Effects

Hvad er en Effect?

Da et api-kald er en side-effect, bryder de NgRx's regler om en pure-function reducer, derfor er man nødt til at lave de kald et andet sted, og her gør man brug af effects. Effects er funktioner hvis formål f.eks. kunne være at hente data fra en api, web-socket eller andet asynkront.

Da Effects ikke er en del af standardpakken, er vi nødt til at tilføje dem med Angular CLI således:

```
ng add @ngrx/effects@latest
```

```
43      createEvent$ = createEffect(() =>
44        this.action$.pipe(
45          ofType(eventPageActions.createEvent),
46          concatMap((action) =>
47            this.eventService.postEvent(action.event).pipe(
48              map((event) => eventApiActions.createEventSuccess({ event })),
49              catchError((error) =>
50                of(eventApiActions.createEventFailure({ error }))
51              )
```

Figur 10 event.effect.ts[11]

Effect'en lytter til en action (fra Action$-observable) f.eks. "createEvents". Det gør den ved hjælp af ofType metoden. Når den opfanger en pågældende action, tager funktionen action stream observablen og den nye observable og flader dem ud (concatMap), da man ikke vil arbejde med to observables ad gangen.

Herefter kaldes eventServices, som er en service der har en http post-request metode kaldet postEvent(). Hvis postEvent responderer med OK, bliver createEventSuccess actionen mappet ind i observablen med en prop af resultatet, i dette tilfælde det event vi har oprettet.
Hvis der opstår en fejl, bliver der returneret en ny observable af createEventFailure, hvor error'en sendes med. Denne error opstår i eventServices og forplanter sig videre op så view'et kan tage stilling til fejlen.

[10] Kan findes på GitHub
[11] Kan findes på GitHub

Når observablen med enten createEventsSuccess eller createEventsFailure bliver returneret kommer den en tur igennem reducer'en der så tager sig af at opdatere storen med den nye state.

View

View'et er en betegnelse på hele component'et. I NgRx vil man gerne have stateless-components, altså component'er der ikke har ansvaret for at opbevare state. Det gør man ved at holde al staten i storen, og hente den til component'et når man har brug for det ved hjælp af selectors.

Component'et dispatcher til reducer'en, som så sørger for at ændre staten i storen. Disse ændringer rapporteres tilbage til component'et ved hjælp af en selector, når den nye state kommer til view'et opdaterer view'et sig selv

ChangeDetectionStrategy.OnPush

Det er også i view'et man kan bruge en anden changeDetection. Her kan man bruge ChangeDetectionStrategy.OnPush til at få Angular til kun at opdatere de steder hvor referencen på en del af staten er ændret. Dette gør at man sparer beregnings-resurser i forhold til ChangeDetectionStrategy.Default, der kontrollerer alt, også properties på state-objekter. Default bliver f.eks. kørt hver gang der bliver kaldt et event eller der kommer et http-response. Så man kan forestille sig at ChangeDetection bliver kørt tit, derfor er det en god idé at optimerer her.

```
11  @Component({
12    selector: 'app-event-ng',
13    templateUrl: './event-ng.component.html',
14    styleUrls: ['./event-ng.component.css'],
15    changeDetection: ChangeDetectionStrategy.OnPush,
16  })
```

Figur 11 event-ng.component.ts[12] Her ses hvordan man ændrer changeDetection til OnPush.

OnPush er hurtigere end default, men vi skal også følge nogle simple regler. Ved OnPush tjekker Angular kun referencen på state-objekter, derfor er det nødvendigt at have immutable state, hvilket spiller godt med NgRx. NgRx bruger, som sagt Observables, og her kan async pipe igen hjælpe os. Når en Observable emitter en ny værdi planlægger async pipe at kører ChangeDetection på værdien. Derfor kan man med fordel bruge OnPush i NgRx.

[12] Kan findes på GitHub

NgRx-Lazy Loading

En af fordelene ved Angular er, at man nemt kan lazyloade moduler. NgRx følger dette princip ved at man både kan lazyloade, reducer, effect og store.

```
23        StoreModule.forFeature('events', eventReducer),
24        EffectsModule.forFeature([EventEffects]),
```

Figur 12 event-ng.module.ts[13]

Forfeature er NgRx's måde at fortælle at denne reducer, eventReducer, hører til dette feature-modul events.

Staten er en anden udfordring, hvordan kan man opbevare al sin state i ét objekt, hvis staten er lazyloaded? Ved at lave inheritance, featurestaten nedarver fra root staten.

Root-state:

```
1   //empty state for extending in the feature, makes sure we can still do
    lazyloading :)
2
3   export interface State {}
```

Figur 13 src/state/state.ts[14]

Feature-State:

```
6   // Extends the app state to include the event feature.
7   // This is required because events are lazy loaded.
8   // So the reference to eventstate cannot be added to our root state directly.
9
10  //State :
11  export interface State extends appState.State {
12    events: EventState;
13  }
```

Figur 14 eventComponentsNgRx/state/index.ts[15]

Dette betyder at man igennem hele applikationens levetid, i runtime, kan loade nye properties ind i sin state igennem lazyloading, det betyder også at man ikke behøver at hente al sin state når applikationen starter op, hvilket gør vores initial load hurtigere. Samtidig skal man dog huske på at man skal loade resten af applikationen, men dette vil foregå i små biddet spredt ud over applikationens levetid. Lazyloading er

[13] Kan findes på GitHub
[14] Kan findes på GitHub
[15] Kan findes på GitHub

forholdsvis nemt at sætte op når man bruger Angular, og der kan man mærke at NgRx-holdet har gjort deres for ikke at forstyrre denne feature for meget.

Skal Jeg bruge NgRx

Et af de helt store spørgsmål er om man skal bruge NgRx i sin app. Det er vigtigt at huske på at der er fordele og ulemper ved at bruge NgRx.

SHARI

SHARI et godt huskeord, hvis formål er at finde ud af om man i virkeligheden har brug for NgRx eller om man kan nøjes med Angular's build-in services[16]:

Shared: *Ens State bliver brugt af mange komponenter og services.* Her kan man blandt andet som eksempel tænke på Autentifikation, da mange komponenter skal vide om en bruger er tilladt at gå ind på f.eks. bestemte routes.

Hydrated: *Ens state bliver "rehydrated" fra udefrakommende hukommelse.* Hvis man har state der kommer udefra, f.eks. lokalt på computeren. For det meste tænkes der her på bruger præferencer/indstillinger.

Available: *Hvis man skal kunne gå tilbage i routes og stadig have informationen som brugeren har tastet ind før.* Dette kunne f.eks. være når en bruger udfylder en form og går tilbage i formen for at ændre noget, så skal den information personen tidligere har udfyldt stadig være tilgængelig. Det kan også være simple ting som, hvilke filtre man vil filtrere en list med.

Retrieved: *Hvis ens State bliver hentet med en side-effekt (se f.eks. http)*. Dette betyder at man ved at bruge NgRx har en mulighed for at cache http-responses. Det kan betyde meget for performance ved store applikationer, hvor der skal hentes meget data.

Impacted: *State der bliver ændret af actions fra andre "kilder".* Når du filtrerer en liste baseret på noget state og der så kommer nye items i din liste, bliver du nødt til at køre filtreringen igen med de nye elementer for at få dem vist. Hvis man bruger NgRx kan man lade Selectoren stå for at importerer den filtrede state og man gør derved sin arkitektur mindre kompleks.

Man bør altid vurdere om det er nødvendigt at have den ekstra kode og kompleksitet som NgRx introducerer til ens applikation, da det ikke altid er nødvendigt og NgRx er ikke et mirakel-middel.

[16] https://youtu.be/omnwu_etHTY?t=797

Debugging i NgRx

Debugging kan være svært i NgRx, bl.a. fordi view'et taler med actions, som taler med reducer'en, som taler med storen, som taler med selector'en, som taler med view'et. Alt sammen pakket ind i observables. Hertil kommer Redux Devtools til undsætning.

Redux Devtool

For at få Redux Devtools skal man først downloade dem til ens browser. Det kan man bl.a. gøre på Chromes extensions store.[17]

Redux Devtools er ligesom effects, en ekstra pakke man kan downloade og installere ved hjælp af CLI'en. Det er vigtigt at man husker at man både skal downloade Devtools til både ens browser og til ens projekt.

```
ng add @ngrx/store-devtools@latest
```

```
27    StoreDevtoolsModule.instrument({
28      maxAge: 25,
29      logOnly: environment.production,
30    }),
```

Figur 15 app.module.ts[18]

Inde i importslisten i root component'et, importere man Devtools, herefter kan man give nogle indstillinger til modulet.

Her er givet maxAge: 25, hvilket er en max grænse på hvor mange actions der kan være i vores history tree.

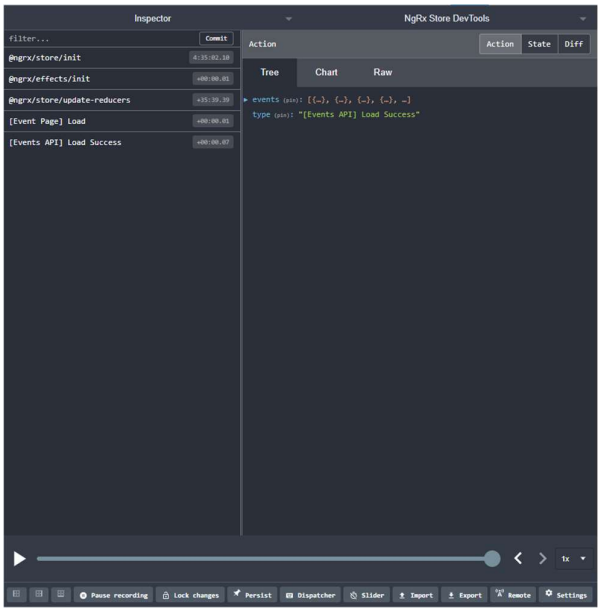
logOnly: environment.production, hvilket faktisk er en reference til en false boolean når vi ikke er i prod-mode. Når vi builder vores Angular app vil filen der indeholder den property bliver byttet ud med en der refererer en true værdi, hvilket betyder at der ikke vil blive logget, dette er så Devtools ikke logger på en side i produktion.

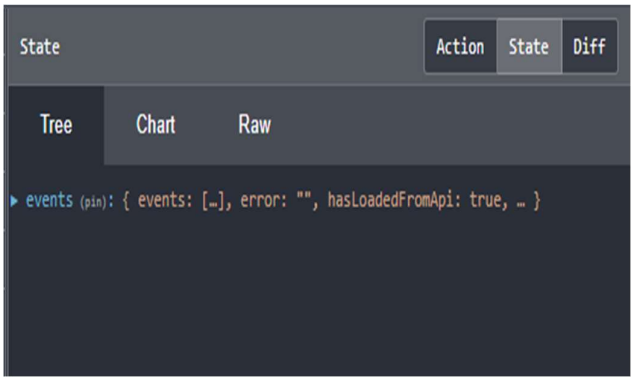
Når man endelig har installeret Devtools og ens Angular app kører og åbent i browseren. Kan man gå ind i Chromes developer tools (F12) og her burde der nu komme en tab frem, der hedder Redux. Trykker man på den kommer man ind i selve redux Devtools.

I redux Devtools er der Action, State Diff. Action er de actions der bliver dispatchet. State er applikationens state, og diff er forskellen fra før/efter.

[17] https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=da
[18] Kan findes på GitHub

Trykker man nu på ”NgRx Events” i webappens navbar, kommer dette frem:





Disse skærmbilleder viser actions der er blevet dispatchet, hvordan staten ser ud og ændringer i Staten. Der er også mulighed for charts og raw view, som viser de forskellige kategorier på andre måder.

Dette er en rigtig god måde at finde bugs i ens app f.eks. hvis staten ikke bliver ændret i appen, kan man gå ind og se om den korrekte action er blevet dispatchet. Dette afmystificerer også hvad der sker inde i NgRx en del.

Konklusion

Hvad er NgRx?
Hvordan kan NgRx gøre en Angular app bedre?
Hvad er forskellen mellem NgRx og normale Angular-services?

NgRx er et State-management library, som abstrakter staten væk fra component'erne og over i en store. Staten er Immutable og der er et uni-directional-data-flow, hvilket kan gøre det nemmere at få et overblik over især meget store webapps. NgRx er også en løsning til at få "renere" component'er, som ikke har ansvaret for at administrere staten, men kun være forbruger af den.

NgRx er et godt library, men det er bestemt reserveret for, hvis ikke store, så applikationer, der har enorme mængder forbindelser mellem component'er. Når man bruger NgRx er der store fordele ved at ens state er abstraktet væk fra component'erne, men især i mindre applikationer kan den boilerplate kode der kommer til at være i stedet for, være så meget at det kan blive uoverskueligt.
Man skal dog altid huske på SHARI, det er et godt huskeord for om det overhovedet er nødvendigt med NgRx.

NgRx fremmer S'et i SOLID. Da component'erne kun kommer til at have ansvar for at have interaktionen mellem brugeren og UI'en og at vise noget data, logikdelen bliver holdt et andet sted, f.eks. i reducer'en, effect'en, services, eller andet.

I en Angular applikation uden NgRx ville man gøre brug af services til at holde bestemt state, dette kan dog hurtigt fører til at man har rigtig mange serviceklasser, hvis man har et stort projekt. Man kan hurtigt forestille sig at hvis man skal gemme på en masse state i service-klasserne, ville man hurtigt bliver overrumplet af mængden af serviceklasser, og netop dette kommer til at gøre det uoverskueligt, i sådanne tilfælde kan man måske begynde at overveje NgRx.

Refleksion

NgRx kan godt virke uoverskueligt, det er det også i starten. Der er meget mystik omkring netop dette library. Jeg synes bestemt at have lært en del, både om RxJs og NgRx. Jeg skulle nok have valgt at starte med at begrænse mig selv mere. Der er dog ting jeg har følt var out-of-scope bl.a. entity, component-store og router-store. Der er også et andet framework jeg har læst om ved navn NgXs, hvilket også er et state-management library, hvor de har videreudviklet NgRx, til at have mindre boilerplate kode, og et mindre krav til brugerens kendskab til RxJs.[19]

NgRx er et solidt library, dog med høje krav til tiden man vil bruge på det, da det kræver lang tid at få en ordentlig forståelse for hvordan NgRx fungerer, og jeg har nok ikke engang skrabet i overfladen.

Jeg synes også det har været svært at finde en god vejledning i hvordan NgRx fungerer inde bagved, og har ofte måtte ty til at læse kildekoden, som ligger på GitHub, hvilket nogle gange har haft den modsatte effekt og gjort det synes mere komplekst.

Jeg vil dog påpege at jeg til at starte med var meget negativ overfor NgRx og synes det var lidt omsonst at man håndterede state på denne måde, da man sagtens kan gøre det med Angulars serviceklasser og dependency Injections. Men efterhånden som jeg har arbejdet med NgRx må jeg nok indrømme at jeg er begyndt godt at kunne lide idéen om at have en samlet state.

Min metode synes jeg har fungeret godt, og jeg synes jeg har opnået det, der var målet med metoden, at have større forståelse for emnet før jeg skriver det. Det har dog affødt at jeg har været nødt til at gennemgå hvert emne og omskrive eller rette op til flere gange.

[19] NgXs dokumentation https://www.ngxs.io/

Litteraturliste:

Skriftlige Kilder:

En artikel omkring Angular ChangeDetection:

https://netbasal.com/a-comprehensive-guide-to-angular-onpush-change-detection-strategy-5bac493074a4

Reddit Angular subreddit, første kommentar beskriver personens skifte fra NgRx til NgXs og hvorfor:

https://www.reddit.com/r/Angular2/comments/8x71fk/ngrx_or_ngxs/

Dokumentation:

Flux dokumentation: https://facebook.github.io/flux/docs/in-depth-overview

NgRx's officielle dokumentation: https://ngrx.io/docs

NgRx source code, GitHub:

https://github.com/ngrx/platform/tree/a9e7fb177c4750ac5c8a39ea3898454669cde847/modules/store

Video Kilder:

Video fra Facebook konference "Hacker Way" om Flux: https://youtu.be/nYkdrAPrdcw?t=619

Pluralsight video af Deborah Kurata og Duncan Hunter om NgRx:

https://app.pluralsight.com/library/courses/angular-ngrx-getting-started/table-of-contents

Video fra en af developerne på Core Team på NgRx " You might not need NgRx", af Mike Ryan:

https://www.youtube.com/watch?v=omnwu_etHTY

Url til mit GitHub projekt om NgRx

Først skal man køre:

npm install

I én terminal kørers *npm run api*

I én anden terminal køres *ng serve -o*

https://github.com/Jakob-Droid/AngularNgrx-Exam