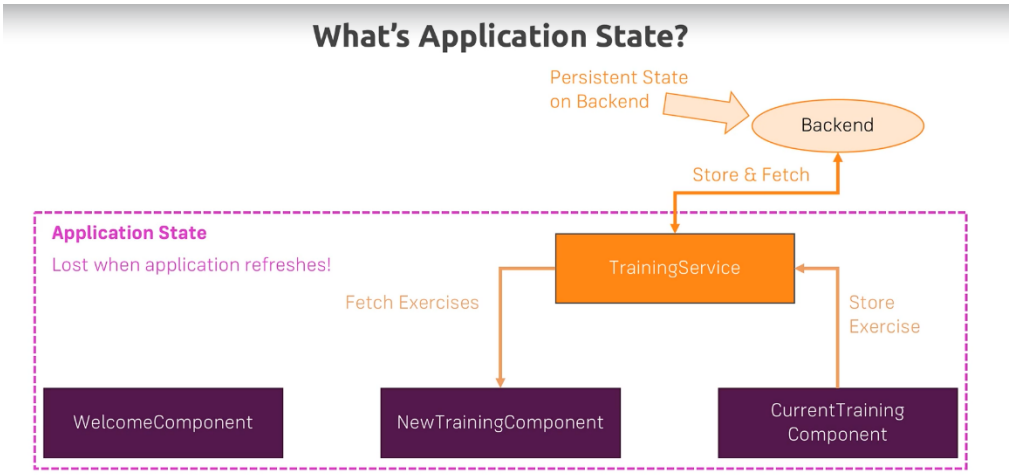
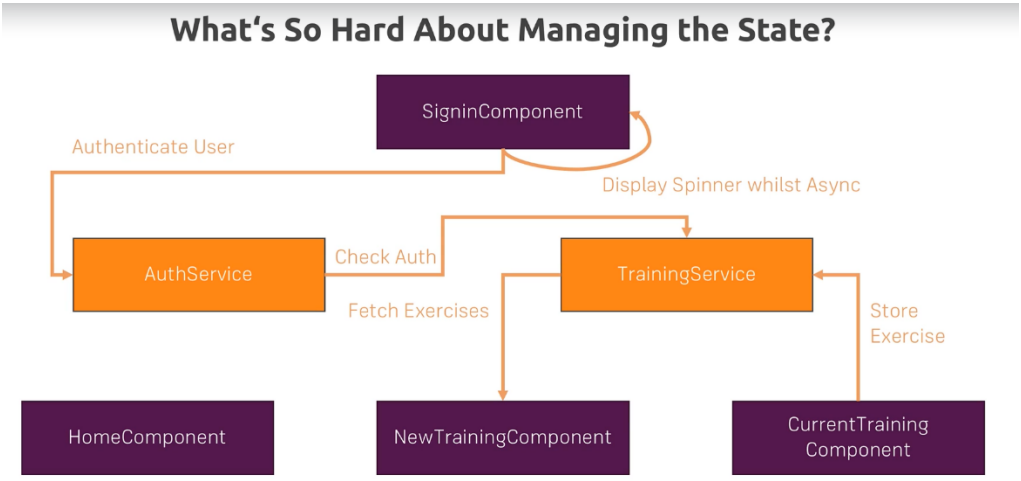
1. INTRODUCTION

NgRx angular'ın state management için redux based solution'ıdır. Şuan tam oturmasa anladığım kadarıyla, verilerin biryerden bir yere gönderilmesi, kullanıcın auth olması, verilerin store etilmesi gibi lokaldeki durumlara application state diyoruz. Yani aslında uygulamanın bulunabileceği farklı durumlar var ve bunlara application state diyoruz.

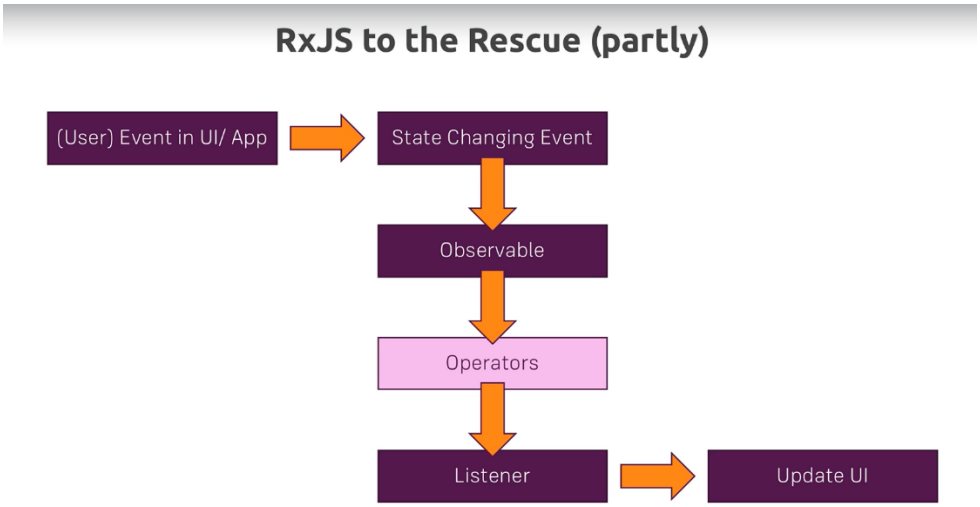


State management zor bir iş, aşağıdaki bizim uygulamamıza benzer bir örnek için bu grafik var, farklı application state'ler ve componenler arası bağlantılar görünüyor, belki küçük uygulamarda state management çok zor değil ama uygulama büyüdükçe işler karmaşıklaşmaya başlıyor.



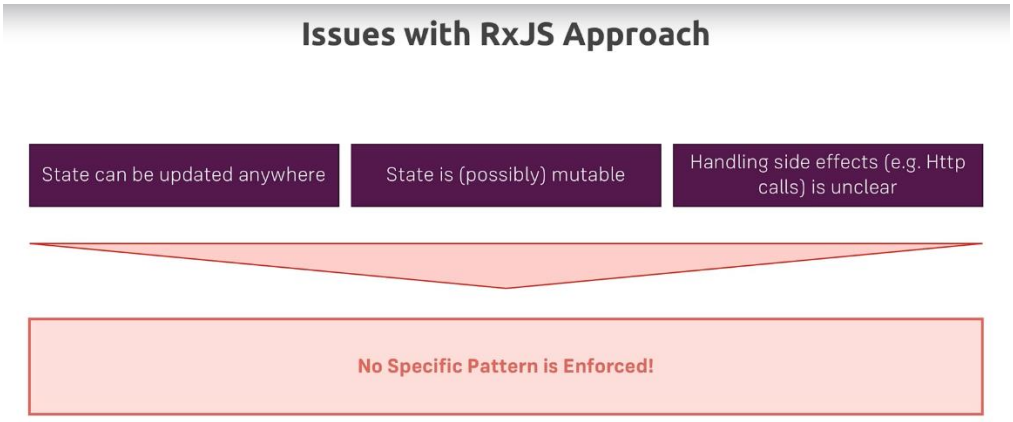
State management için kullanılabilecek önemli bir tool RxJS'tir ki bunu zaten kullandık. Burada bir event dinleniyordu ve buna bağlı olarak, state değişiyor, bunu da observables ile yani subjects ile emit edebiliyoruz hatta bu sırada values'a operations uygulayabiliyoruz ve sonuçta emit edilen value'yı subscription ile dinliyoruz ve sonuç olarak UI'ı update ediyoruz yani

uygulamanın state'ini eventler ile değiştirmiş oluyoruz. Şimdiye kadar state management'ı böyle yaptık.



State management diyince havalı oluyor ama bunu uygulamanın farklı durumları olarak düşün, örneğin kullanıcı doğru bilgi girerse, isAuth true olsun isAuth true olursa UI'ın belirli kısımlaır görünsün belirli route'lar kullanıcıya açılsın bu bir state olmuş oluyor.

Şimdiye kadar ngRx kullanarak veya doğrudan state management'ı yaptık ancak state management için kullanılan bu pattern'ın problemleri de var, örneğin state'i uygulama içinde heryerden update edebiliriz, state'ler kararsız olabilir, ayrıca bi belirsizlik var örneğin http request'i nerede göndereceğiz, angularfire'ı nerede kullanacağız, service içerisinde mi kullanalım, yoksa component içerisinde mi ortada bir belirsizlik var.

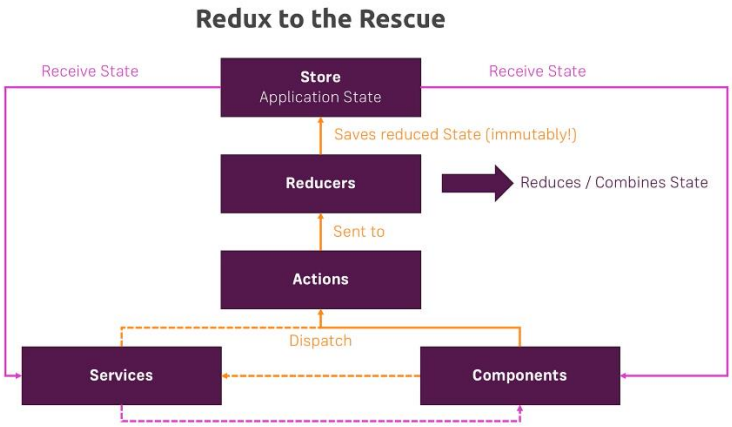


İşte Redux'ın çözeceği problem bu olacak sanırım ortaya bir strateji koyacak, ve uygulama durumlarını daha düzenli bir şekilde kontrol etmemizi sağlayacak.

2. WHAT IS REDUX?

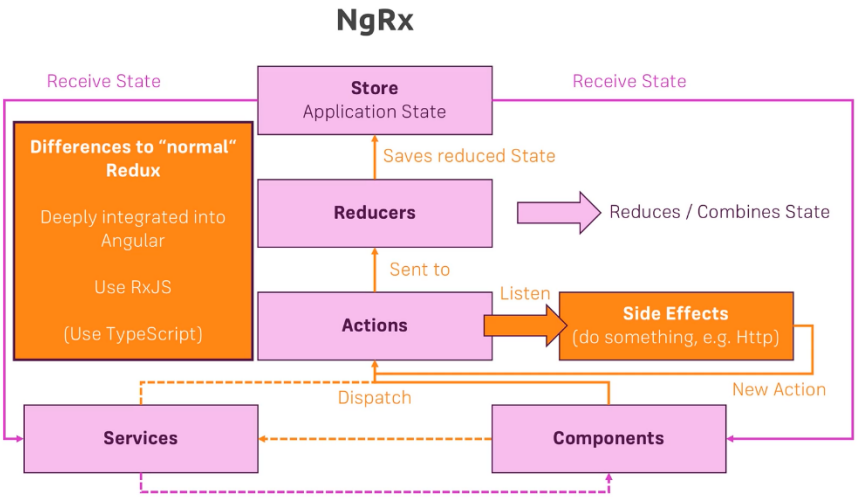
Redux tüm application states'i store edebileceğimiz central bir yapı ortaya koyar. Yani şuan bir şey mi yüklüyoruz? Exercises array of exercises'a yüklendi mi gibi state'leri store edebiliriz.

Bu state'ler central store'da store edilir ve uygulamamızın servisleri ve componentleri kendi arasında haberleşmenin yanında application state bilgisini bu central store'dan alır.



Bunun yanında state'leri set etmek için component ve servisler doğrudan store ile iletişim kurmaz bunun yerine actions dispatch ederek reducers'a ulaşır, reducers action'ı ve payload'ı alarak bunu central store'un store edebileceği bir value'ya reduce eder ve bu value state olarak kaydedilir.

Bu state management yaklaşımına Redux denir, Angular ile bu approach'ı uygulamak için NgRx pattern'ı kullanırız.



NgRx ile ekstra olarak side effects kavramından bahsedilebilir, route değiştirme request gönderme gibi işlemleri side effects ile yapabiliriz.

3. NGRX CORE CONCEPTS

Projemizde ngrx yapısını kullanabilmek için ilk olarak aşağıdaki komutu çalıştırıyoruz:

npm install --save @ngrx/store

Öncelikle app.module'ün hemen yanına **app.reducer.ts** ismiyle main reducer oluşturuyoruz, bu reducer'ı main store'um ile bağlantı kurmak için kullanacağız.

1. Reducer

Şimdi app.reducer.ts içerisine basit bir reducer yazalım:

```
export interface State {
  isLoading: boolean;
}

const initialState: State =  {
  isLoading: false
}

export function appReducer(state = initialState, action) {
  switch(action.type) {
    case 'START_LOADING':
      return {
        isLoading: true
      };
    case 'STOP_LOADING':
      return {
        isLoading: false
      };
    default:
      return state;
  }
}
```

Reducer dediğimiz temelde bir fonksiyon, input olarak current state'i ve dispatched action'ı alıyor, ve bu action'a göre yeni state'i return ediyor.

Örneğin application state sadece boolean isLoading içeren bir javascript object olsun. Sonuçta bu reducer'a current state verilecek ve yanında action verilecek ve reducer da bu action ile next state'i return edecek.

Bu reducer'ı kullanabilmek için **app.module.ts** içerisinde, aşağıdaki importları yapmanın yanında

```
import { StoreModule } from '@ngrx/store';
import { appReducer } from './app.reducer';
```

imports arrayine, aşağıdaki yapıyı ekliyorum, burada ui kendi verdiğim bi isim, başka reducer'lar da olacak ui'dan sorumlu reducer bu olacak o yüzden ui dedik.

```
StoreModule.forRoot({ui: appReducer})
```

Yukarıki yapıda ui key'i ile appReducer'a ulaşmış olacağız bu reducer sadece ui ile ilgili state'leri kontrol edecek, başka state slices için başka keyler kullanırız.

2. Dispatching Actions

Şimdi bir reducer tanımladık ve bunu app.module.ts içerisinde ui key'ini de kullanarak import ettik artık bu reducer'a ulaşıp, initial state ile action'ı vererek next state'i elde edebiliriz.

Şimdi bu loading state'i önceden nasıl değişiyordu onu hatırlayalım, auth.service içerisinde login veya registerUser methodları çalıştığı zaman, bir subject true olarak emit ediliyordu, firebase'den sonuç gelince de .then veya .catch içerisinde yakalanıyordu, yakalanınca tekrar aynı subject false olarak emit ediliyordu, bu sonuçlar da login ve signup componentlerinde yakalanıp, ona göre ekranda spinner dönüyordu.

O zaman biz de action dispatching'i aynı noktalarda yapacağız, aynen bir subject gönderiyormuşuz gibi action dispatch edeceğiz.

Şuanda tek bir store slice'ımız var, bunu app.module'e bakarak anlayabiliriz:

```
StoreModule.forRoot({ui: appReducer})
```

Bir tek ui slice'ı var ve bu da appReducer'ın state'lerini store etmek için kullanılır.

Action dispatching için, bu işlemin yapılacağı yere bir store inject etmemiz gerekiyor ki dispatch edilen actionlar sonucunda reducer'ın return ettiği yeni state store'da kaydedilebilsin, bunun için **auth.service.ts** içerisinde aşağıdaki importları yaptık:

```
import { Store } from '@ngrx/store';
import * as fromApp from '../app.reducer';
```

Ardından da constructor içerisine ui reducer'ına yani appReducer'a özel bir store slice'ı inject ettik, bu store slice'ının kaydedeceği state'in tipini de appReducer içerisinde tanımlanmış state interface'i belirleyecek.

```
private store: Store<{ui: fromApp.State}>
```

Şimdi registerUser ve login methodları içerisinde action dispatching yapalım, eski subject emit satırını comment'e aldık:

```
  registerUser(authData: AuthData){
    // this.uiService.loadingStateChanged.next(true);
    this.store.dispatch({type: 'START_LOADING'});
    this.afAuth.auth.createUserWithEmailAndPassword(
      authData.email,
      authData.password
      ).then(result => {
        //this.uiService.loadingStateChanged.next(false);
        this.store.dispatch({type: 'STOP_LOADING'});
      }).catch(error => { // Eğer register hata döndürürse burada error olarak
 yakalanır.
        //this.uiService.loadingStateChanged.next(false);
        this.store.dispatch({type: 'STOP_LOADING'});
        this.uiService.showSnackbar(error.message,'OK',8000);
      });
```

Unutma ki actions her zaman object tipindedir ve her zaman bir type'ı olmak zorundadır çünkü reducer bu type'a bakarak next state'i belirliyor.

```
  login(authData: AuthData){
    //this.uiService.loadingStateChanged.next(true); // Bir servisin subject'i
 başka bir servis içinden emit ediliyor.
    this.store.dispatch({type: 'START_LOADING'});
    this.afAuth.auth.signInWithEmailAndPassword(
      authData.email,
      authData.password
```

```
      ).then(result => {
        //this.uiService.loadingStateChanged.next(false);
        this.store.dispatch({type: 'STOP_LOADING'});
      }).catch(error => {
        //this.uiService.loadingStateChanged.next(false);
        this.store.dispatch({type: 'STOP_LOADING'});
        this.uiService.showSnackbar(error.message,'OK',8000);
      });
  }
```

Aynı işlemi login için de yaptık.

3. Subscribing Central Store Slice

Artık service içerisiden nasıl action dispatching yaptığımızı gördük, istediğimiz event gerçekleşince aynı bir subject emit eder gibi action dispatch ettik ve sonucunda store slice'ı için kullanılan reducer fonksiyonunun belirlediği kurallara göre next state ilgili store slice'ına kaydedildi.

Sıradaki adım ise store'dan ilgili state'i almak yani subscribe etmek. Bunu aynen emitted subject'i subscribe etmek gibi düşünebilirsin sadece ortada bir centralization var, subjectler rastgele havada uçuşmuyoru hepsi tek bir merkeze gidip oradan subscribe ediliyor gibi düşün.

O halde şimdi yapılacak işlem, state'in okunması gereken component'e gidip yine store'u inject etmek böylece store üzerinden ilgili slice'a subscription yapabiliriz ve aynen bir subject'e subscription yapmışız gibi işimizi görebiliriz.

Öncelikle ilgili importları login component'i içerisinde yapıyoruz:

```
import { Store } from '@ngrx/store';
import * as fromApp from '../../app.reducer';
```

Şimdi sırada component'e ilgili store'u ui slice'ı ile inject etmek, bunun için aşağıdaki satırı constructor içerisine gömüyoruz.

```
private store: Store<{ui: fromApp.State}>
```

Şimdi artık, örneğin init methodu içerisinde ilgili store'a subscribe ederek store'un tamamına ulaşabiliriz, şuanda tek bir slice'ı var o da ui slice'ı.

```
this.store.subscribe(data => console.log(data));
```

Birden fazla slice olabilirdi, bu store action dispatchings ile güncelleniyor ve güncel değerler subscription ile store'dan çekilebiliyor böylece bu state management gerçekleşimiş oluyor. Burada ui slice'ı içerisindeki state'in tipini ise appReducer yani ui'a atanmış reducer içerisindeki State interface'i ile belirledik. Başka store slice'lar daha farklı state'ler tutabilir.

Şimdi console'da log edilen yapıya bakarsak, bir ui slice'ı içerisinde bir state object ve state'in güncel durumunu görebiliriz:

```
▼ {ui: {…}}
  ▶ ui: {isLoading: true}
  ▶ __proto__: Object
```

Bu şekilde isLoading değerini login component içerisine alabiliriz, ancak burada farklı bir şey yapacağız:

Öncelikle yine login component içerisinde, aşağıdaki observable'ı tanımlayacağız, bu observable NgRx ile kontrol edileceği için convention olarak sonuna $ ekleriz:

```
isLoading$: Observable<boolean>;
```

Daha sonra da store içerisinden observable'ı direkt çekeriz ve buna asynch pipe'ı kullanarak template içerisinde subscribe ederiz, bunu daha önce yapmıştık: (Burada originalde sadece map kullanıldı, versiyona bağlı)

```
this.isLoading$ = this.store.pipe(map(state => state.ui.isLoading));
```

Ayrıca burada map fonksiyonunu observable'a uygulayarak, demin yukarıda görünün ui property'si olan object'i state value olarak alıp manipüle edebiliriz, örneğin burada isLoading$'i object value yerine isLoading parametresini yani boolean value taşıyan bir observable'a dönüştürdük. Böylece template içerisinde bu boolean value'ya göre aşağıdaki gibi işimizi görebiliriz:

```
<button *ngIf="!(isLoading$ | async)" type="submit" mat-raised-
button color="primary" [disabled]="loginForm.invalid">Submit</button>
    <mat-spinner *ngIf="isLoading$ | async"></mat-spinner>
```

4. MULTIPLE REDUCERS

Şuan elimizde tek bir global reducer var, bu reducer'ı da ui state'lerini manage etmek için kullanıyoruz.

Şimdi ise 3 reducer kullanacağız, bir tanesi yine ui için olacak biri training states için diğer ise auth states için kullanılacak. Servislerimizi de bunların her biri için ayırmıştık.

Şuanda kullanılan app.reducer.ts'i ise bu üç reducer'ı merge etmek için kullanacağız.

Şimdi ilk olarak **shared** klasörü içerisine **ui.reducer.ts** dosyası açıyoruz ve burada ui reducer'ımızı oluşturacağız, aslında bu biraz evvelki app.reducer'a çok benzer olacak sadece fonksiyon ismini uiReducer koyacağız.

Ancak bunun yanında sadece ACTION type isimlerini string olarak verip hataya yol açmamak için bazı geliştirmeler yapıldı, ui.actions.ts isminde bir file daha eklendi ve burada bu işlem halledildi, basitçe artık bu actions'ın import'u ile action isimlerini typescript otomatik olarak doldurmamıza yardım edebilecek:

Yine shared içindeki ui.actions.ts'in içi aşağıdaki gibi:

```
import { Action } from '@ngrx/store';

export const START_LOADING = '[UI] Start Loading';
export const STOP_LOADING = '[UI] Stop Loading';

export class StartLoading implements Action {
  readonly type = START_LOADING;
}

export class StopLoading implements Action {
  readonly type = STOP_LOADING;
}

export type UIActions = StartLoading | StopLoading;
```

Dediğim gibi buna çok takılma, sadece önceden START_LOADING action tipini string olarak veriyorduk bu hataya açık bir yöntemdi o yüzden bunu yaptık, bu sebeple ui reducer'ın içi de biraz değişecek:

```
import { UIActions, START_LOADING, STOP_LOADING } from './ui.actions';

export interface State {
  isLoading: boolean;
}

const initialState: State =  {
  isLoading: false
}

export function uiReducer(state = initialState, action: UIActions) {
  switch(action.type) {
    case START_LOADING:
      return {
        isLoading: true
      };
    case STOP_LOADING:
      return {
        isLoading: false
      };
    default:
      return state;
  }
}

export const getIsLoading = (state: State) => state.isLoading;
```

Sondaki yapı da bir arrow function, getIsLoading() ile ilgili state'in isLoading değerini elde edebilmemizi sağlayacak.

Şimdi ise app.reducer.ts içerisine bakalım dediğimiz gibi bu reducer artık diğer tüm reducer'ların merge edildiği bir global reducer rolünü üstelenecek:

```
import * as fromUi from './shared/ui.reducer';
import { ActionReducerMap, createFeatureSelector, createSelector } from '@ngrx
/store';

export interface State {
  ui: fromUi.State;
}

export const reducers: ActionReducerMap<State> =  {
  ui: fromUi.uiReducer
}

export const getUiState = createFeatureSelector<fromUi.State>('ui');
export const getIsLoading = createSelector(getUiState, fromUi.getIsLoading);
```

Ne olup bittiğini açıklamaya çalışalım, öncelikle biraz önce tanımlanan ui.reducer'ı fromUi ön ismi ile buraya import ediyoruz.

Ayrıca bazı diğer gerekli importları da yaptık.

İlk olarak global reducer'ın uğraşacağı state'in tipini belirtiyoruz, örneğin ui reducer'ın uğraştığı state tipi tek bir boolean attribute'lu bir object idi.

Global reducer ise her bir farklı reducer için ayrı bir key ve buna karşılık ilgili reducer'ın uğraştığı state objesi ile uğraşacak, adı üstünde bu global reducer o zaman state'i de tüm stateleri içermeli.

Şimdi sırada tüm reducers'ı gruplamak var, bunun için yukarıdaki ActionRecıerMap yapısını kullanıyoruz.

getUiState funksiyonu'nu çağırarak sadece ui state'ini ulaşabilmemiz mümkün olacak, aslında bunları yapmadan da global state üzerinden state.ui falan şeklinde ilgili ui state'ine ulaşabiliriz.

Son olarak bu kolaylığı bir adım ileriye götürüp sadece ui state'ine değil o state'in içerisindeki bir key'e de ulaşabiliyoruz.

Bu noktada olaylar biraz karmaşık görünebilir ama aslında değil, olay özünde aynen subject emit etmek ve subsribe etmek gibi, çok handy olsun diye birsürü ekstra method vesaire tanımlanıyor, ancak burada olay farklı reducer'lar farklı state'ler ile ilgilenecek, bu reducer'lar da tek bir reduce rolan app reducer'da birleşecek ve bunun üzerinden ulaşılabilecek.

5. ACTION DISPATCHING AGAIN

Reducer yapısı biraz değiştiği için şimdi dispatching işlemlerine tekrar bakalım.

Öncelikle app.module içerisinde reducer'ı import edişimiz biraz değişecek, bu kez tüm reducer'ları app.reducer.ts üzerinden import edeceğiz:

```
import { StoreModule } from '@ngrx/store';
//import { appReducer } from './app.reducer';
import { reducers } from './app.reducer';
```

yine app.module içerisinde imports arrayinde ise yine reducers'ı import edeceğiz:

```
    //StoreModule.forRoot({ui: appReducer})
    StoreModule.forRoot(reducers)
```

Bu reducers'ı hatırlarsak app.reducer.ts içerisindeki aşağıdaki kısıma karşılık geliyor burada tüm reducer'lar birleşecek.

```
export const reducers: ActionReducerMap<State> =  {
  ui: fromUi.uiReducer
}
```

Şimdi ise login ve register ekranındaki spinner'ı sağlayabilemek için action dispatching ile global state'leri nasıl değiştireceğimize bakacağız, bunu subject emit etmeye benzetiyorduk. Öncelikle action dispatch edilecek yerde Store injection'ı gerçekleştirelim:

```
import { Store } from '@ngrx/store';
import * as fromRoot from '../app.reducer';
import * as UI from '../shared/ui.actions';
```

```
  constructor(
      private store: Store<fromRoot.State>
  ) {}
```

Bu kez app.reducer'ın kütüphanesini fromRoot ismiyle import ediyoruz, ui.actions'ı da biraz sonra göreceğimiz üzere dispatching'i kolaylaştırsın diye import ediyoruz.

Şimdi örneğin login için dispatching'e bakalım, eski yapılar da comment'li duruyor:

```
  login(authData: AuthData){
    //this.uiService.loadingStateChanged.next(true); // Bir servisin subject'i
 başka bir servis içinden emit ediliyor.
    //this.store.dispatch({type: 'START_LOADING'});
    this.store.dispatch(new UI.StartLoading());
    this.afAuth.auth.signInWithEmailAndPassword(
      authData.email,
      authData.password
      ).then(result => {
        //this.uiService.loadingStateChanged.next(false);
        //this.store.dispatch({type: 'STOP_LOADING'});
        this.store.dispatch(new UI.StopLoading());
      }).catch(error => {
        //this.uiService.loadingStateChanged.next(false);
        //this.store.dispatch({type: 'STOP_LOADING'});
        this.store.dispatch(new UI.StopLoading());
        this.uiService.showSnackbar(error.message,'OK',8000);
      });
  }
```

Login olunca global store'a bir global state gönderiyoruz, bu global state'in içeriği app.reducer içerisinde tanımlı.

Dispatch'in için bir UI action objesi input olarak verilir önceden type'ı string olan bir action objesi veriliyordu, burada verilen objenin otomatik olarak tipi atanıyor zaten. App reducer'ımız ilgili action objesini ilgili ui reducer'ına yönlendiriyor, ilgili ui reducer bu action objesinin tipine göre state'i değiştiriyor, bu state app reducer state'inin içerisinde global store'a yazılıyor. Artık istersek global store üzerinden ilgili state'e ulaşabiliriz.

Bunun için hemen login component'ine gidelim, yine app reducer library'sini fromRoot olarak import ediyoruz:

```
import { Store } from '@ngrx/store';
import * as fromRoot from '../../app.reducer';
```

Ardından global store'u app reducer'ın içindeki State yapısını baz alarak buraya inject ediyoruz:

```
constructor(private authService: AuthService, private uiService: UiService, pr
ivate store: Store<fromRoot.State>) {}
```

Şimdi artık bu store'a subscription işlemlerini gerçekleştirebiliriz, ancak geçen seferden hatırlarsan, biz store'u observable olarak elde etmiştik, sadece value'su üzerinde bazı operasyonlar yapıp, subscription'ı asynch pipe'ı ile template üzerinde yapmıştık, yine aynı işlemi yapacağız.

Ancak bu kez, app reducer içine tanımlanan selecter functions yapısını kullanarak herhangi bir pipe-map yapısı kullanmadan doğrudan istediğimiz value'nun observable'ını elde edebileceğiz, select store'un kendi methodu, ancak getIsLoading'i biz tanımladık bu sayede global state'in içinden ui state'inin tek bir attribute'unu observable value olacak şekilde spesifik olarak çekebiliyoruz:

```
  ngOnInit() {
    //this.store.subscribe(data => console.log(data));
    //this.isLoading$ = this.store.pipe(map(state => state.ui.isLoading));
    this.isLoading$ = this.store.select(fromRoot.getIsLoading);
    //this.loadingSubs = this.uiService.loadingStateChanged.subscribe(isLoadin
g => {this.isLoading = isLoading;});
    this.loginForm = new FormGroup({
      email: new FormControl('', {
        validators: [Validators.required, Validators.email]
      }),
      password: new FormControl('', { validators: [Validators.required] })
    });
  }
```

Benim bu noktada kafama takılan tek şey, dispatching sırasında app.reducer'ın ilgili action'ı hangi alt reducer'a yönlendireceğini nasıl anladığı?

Bunu anlarsam açıklayacağım.

Sanıyorum, dispatch içine verilen action'ın tipine göre anlıyor, zaten action'ı object olarak veriyoruz, her reducer için de ayrı action classes kullanıyoruz, bu yüzden buradan anlaşılıyor olabilir.

6. DEVAM

Kursun devamını sadece izleyeceğim, önemli conceptual bir şey olursa onları not düşeceğim.

Yukarıdaki kısımda auth service içerisinde fromRoot ismiyle app.reducer libaray'sini import ettik, store'u import ettik ve store'u kullanarak action dispatching ile bir nevi global store'a subject emit ettik. Yani app reducer üzerinden ui reducer'ına ulaşarak, isLoading state'ini global store'da değiştirdik.

Bu state değişimini daha sonra Login component'inde observable olarak store üzerinden elde ettik, subscription'ını da login template içerisinde yaptık. Bu noktada select fonksiyonunu kullanarak, global state içerisinden sadece bize gerekli olan isLoading boolean'ını almayı başardık.

Sonuçta ui reducer'ını kullanarak yaptığımız işlem, login isteği firebase'e gönderildiğinde isLoading state'ini true yapmak ve bu süreçte ise kullanıcıya bir spinner göstermek, isLoading sonlandığında yani firebase cevap verdiğinde ise bu state'i false'a geri çevirmek böylece spinner'ı da ekrandan yok etmek şeklindeydi.

Biz aynı işlemi öncelikle signup component'inde de yapacağız bu componen de auth.service içerisindeki registerUser methodunun gönderdiği state ile tetiklenecek, state aynı state bu arada, sadece başka noktalardan değiştiriliyor.

Bunun dışında aynı mantığı training.service ve new-training component arasında da uygulayabiliriz, burada da fetchAvailableExercises methodu ile servis üzerinden firebase'e ulaşıyorduk, bu ulaşma başlayınca bir action dispatch ile state'i değiştirebilir, ve daha sonra da subscription'ın başarılı veya başarısız olduğu noktalarda - ki bu noktalarda önceden false olarak subject emit ediyorduk – yeni bir action dispatching ile global state değiştirilebilir. Ardından da spinner'ın gösterileceği noktada ki bu new-training.component.ts oluyor, global state dinlenerek istenilen işlemler state durumuna göre yapılabilir.

Mantık artık bu, pattern'ı öğrendik aynı mantıkla bağımsız state'ler için bağımsız reducer'lar tanımlayabiliriz, örneğin auth veya training state'leri için

ayrı reducer’lar tanımlarız ve benzer şekilde işlemlerimizi global reducer’ı kullanarak yapabiliriz. Bu kısımları sadece izleyeceğim not almayacağım.