**1. Introduction**

6. Installing NgRx and the NgRx DevTools

**NgRx Store**-ის დასაყენებლად ვუშვებთ ბრძანებას: ng add @ngrx/store

მისი **დევთულის**დასაყენებლად ვუშვებ ბრძანებას: ng add @ngrx/store-devtools

ასევე უნდა დავაყენო ქრომში **Redux DevTools** ექსთენშენი.

ბოლოს appModule-ში სავარაუდოდ ესენი დაემატებიან:

1. StoreModule.forRoot(reducers, { metaReducers }),
2. StoreDevtoolsModule.instrument({ maxAge: 25, logOnly: environment.production })

**2. NgRx Key Concepts - Actions and Reducers**

7. Configuring an NgRx Feature Module using NgRx Schematics

ng generate store auth/Auth --module auth.module.ts - მსგავსი ბრძანებით რედიუსერს ვაინსტალირებთ კონკრეტულ მოდულში.

საწყისში სასურველია { metaReducers: fromAuth.metaReducers } წავშალოთ **StoreModule.forFeature**-დან და ასევე რედიუსერის ფაილიდან environment-თან დაკავშირებული კოდი.

**2. NgRx Key Concepts - Actions and Reducers**

8. The Store Service API - Implementing the Login Screen

**NgRx**-ის სთორის კომპონენტში გამოსაყენებლად საჭიროა იგი ჯერ დავაინჯექტოთ კონსტრუქტორში და ჯენერიკ ტიპად გავუწეროთ კონკრეტული რედიუსერის ტიპი, მაგალითად **AppState**:

**reducer:**

1. export interface AppState {}
2. export const reducers: ActionReducerMap<AppState> = {};

**component:**

1. private store: Store<AppState>

**2. NgRx Key Concepts - Actions and Reducers**

9. Defining NgRx Actions using Action Creators

**NgRx**-ს აქვს ასევე ექშენები, რომლებიც დაკავშრების გარეშე, სთორზე ზეგავლენას ვერ ახდენს, თუმცა მაინც ენჯის ექშენს წარმაოდგენს და მისი დანახვა დევთულსიდან შესაძლებელია.

სთორს აქვს მეთდი **dispatch()**, რომელიც არგუმენტად იღებს ამ კონკრეტულ ექშენს.

ექშენებისთვის ცალკე ფაილი იქმნება და იქ იწერება საჭირო ლოგიკა, სადაც გამოიყენება **createAction()** მეთოდი, რომელიც სთორიდან იმპორტდება, იგი იღებს ორ არგუმენტს, პირველია ტიპი (სტრინგად), ხოლო მეორე არის **props()**, მეთდი, რომლის ჯენერიკ ტიპიც იქნება ის დეითა, რასაც ვაწვდით, შემდეგ კი ეს ექშენი **dispatch()-**ში გამოიძახება:

**actions.ts:**

1. export const login = createAction(
2. "[Login Page] User Login",
3. props<{user: User}>()
4. );

**component.ts:**

1. this.store.dispatch(login({user}))

**2. NgRx Key Concepts - Actions and Reducers**

10. Grouping Actions Together with Action Types

ექშენი, რომელიც არანაირ ინფორმაციას არ გზავსი, **props()** მეთოდს არ საჭიროებს:

1. export const logout = createAction(
2. "[Top Menu] Logout"
3. )

ასევე უკეთესი იქსპერიენსისთვის, კარგი იქნებოდა, შეგვექმნა ფაილი, სადაც ყველა ექშენს დავაიმპორტებდით და მერე დავაექსპორტებდით ისევ:

1. import \* as AuthActions from './auth.actions';
2. export {AuthActions};

**2. NgRx Key Concepts - Actions and Reducers**

11. NgRx Reducers - Step-by-Step Implementation

**Reducer**-ის დანიშნულებაა ექშენების მიხედვით მოახდინოს სთორში შენახულ დეითაზე ზემოქმედება, მაგალითად ინიშიალ დეითაზე. რედიუსერში სასურველია გვქონდეს ინტერფეისი, რომელიც ინიშიალ სტეიტის ობიექტის ტიპი იქნება და ასევე უნდა გვქონდეს მეთდი**.createReducer()**, რომელიც პირველ არგუმენტად იღებს ინიშიალ სტეიტის ობიექტს და მეორედ **@ngrx/store-**ის მეთოდი **on()**, რომლის პირველი არგუმენტია კონკრეტული ექშენი, მეორე კი ფუნქცია ორი არგუმენტით: **state**და **action**, რომელიც დააბრუნებს ობიექტს, რომელიც ინტერფეისს ემსგავსება, მნიშვნელობაში კი ექნება ექშენის მიერ შექმნილი რაიმე:

1. export interface AuthState {
2. user: User
3. }
4. export const initialAuthState:AuthState = {
5. user: undefined
6. }
7. export const authReducer = createReducer(
8. initialAuthState,
9. on(AuthActions.login, (state, action) => {
10. return {
11. user: action.user
12. }
13. }))

მოდულის ფაილში კი:

1. import \* as fromAuth from './reducers'
2. StoreModule.forFeature('auth', fromAuth.authReducer)

**3. NgRx Key Concepts - Selectors and Effects**

14. NgRx Selectors - An In-Depth Explanation

საკუთარი სელექტორის შესაქმნელად გამოიყენება ფუნქცია **.createSelector(),** იგი მრავალ არგუმენტს იღებს, მაპირების მოთხოვნის შესაბამისად, თუმცა 2 არის მინიმალური. პირველ არგუმენტად ჩავაწვდით იმ სტეიტს, რომელიც უნდა შეამოწმოს და მეორეში უკვე ბულიანის დაბრუნების ფუნქციას პასუხის შესაბამისად:

1. export const isLoggedIn = createSelector(
2. state => state['auth'],
3. (auth) => !!auth.user
4. );

**!!** - აქ ნიშნავს საწინააღმდეგოს საწინააღმდეგოს.

სხვა ფუნქციის შექმნა ასევე შეგვიძლია ამ ფუნქციის გამოყენებით:

1. export const isLoggedOut = createSelector(
2. isLoggedIn,
3. loggedIn => !loggedIn
4. )

მას კი კომპონენტში შემდეგნაირად გამოვიყენებთ **selector()**-თან:

1. this.isLoggedIn$ = this.store.pipe(
2. select(isLoggedIn)
3. )

**3. NgRx Key Concepts - Selectors and Effects**

15. NgRx Feature Selectors - a Simple Explanation

იმისათვის, რომ სტეიტსეიფ სელექტინგი გვქონდეს, იგივე .ts ფაილში, სადაც სელექტორებს ვქმნით, უნდა შევქმნათ **createFeatureSelector<AuthState>("auth")**მეთოდი, რომლის ჯენერიკ ტიპიც იქნება სტეიტის ინტერფეისი და არგუმენტად მიიღებს კონკრეტული რედიუსერის სახელს. ხოლო ის ცვლადი, სადაც ამ ფუნქციას ვინახავთ, ჩაჯდება პირველი სელექტორის შემქმნელი ფუნქციის პირველ არგუმენტად.

1. export const selectAuthState = createFeatureSelector<AuthState>("auth")
3. export const isLoggedIn = createSelector(
4. selectAuthState,
5. auth => !!auth.user
6. );

**3. NgRx Key Concepts - Selectors and Effects**

16. Implementing User Logout (Practice Lesson )

მაგ. **logout**-ის ფუნქციის რედიუსერში ჰენდლინგისთვის, რამდენადაც ჩვენი **createReducer(), on()**მეთოდით ჰენდლავს მათ და ასევე, პირველ არგუმენტად აქვს ინიშიალ სტეიტი, უბრალოდ ინიშიალ სტეიტს ისევ ანდიფაინდად გადააქცევს:

1. on(AuthActions.logout, (state, action) => {
2. return {
3. user: undefined
4. }
5. })

**3. NgRx Key Concepts - Selectors and Effects**

18. Introduction to NgRx Effects - What is a Side Effect?

სერვერთან ან ლოქალ სტორეჯთან ურთიერთობისას სარუველია გამოვიყენოთ **NgRx**-ის **Effects**-ი. თუ დაინსტალირებული არ მაქვს:

npm install @ngrx/effects --save

შემდეგ კი **appModule**-ში:

EffectsModule.forRoot([აქ წავა ეფექტების ფაილები])

უბრალოდ ყველგან უნდა გავითვალისწინო, რომ თუ ცენტრალურ მოდუში შემომაქვს, არის **forRoot()**, ხოლო თუ სხვა ფიჩურულ მოდულში -**forFeature()**.

**3. NgRx Key Concepts - Selectors and Effects**

19. Understanding NgRx Effects - A Simple Example

ყველაზე ტრივიალური გზა საიდ ეფექტის ჰენდლინგისთვის **Effect**-ში არის შემდეგნაირი:

ვქმნით ვალკე**.ts**ფაილს, შიგნით დეკლარირდება კლასი, რომელიც შემდეგ შესაბამის მოდულს ან ცენტრალიურს **forFeatures([])**-ში ან **forRoot([])**-ში ჩაეწოდება და თავად ამ კლასის კონსტრუქტორში ინჯექტდება **ngrx/effect**-ის **Actions**, რომელიც თავად ობზერვებლს წარმოადგენს და შეიძლება მისი დაუბსქრაიბება და შემდეგ მიღებული შედეგის ტიპზე დაკვირვება და შესაბამისი მოქმედების განხორციელება:

1. export class AuthEffects {
2. constructor(private actions$: Actions){
3. actions$.subscribe(action => {
4. if(action.type == '[Login Page] User Login'){
5. localStorage.setItem('user', JSON.stringify(action["user"]));
6. }
7. })
8. }
9. }

**3. NgRx Key Concepts - Selectors and Effects**

20. NgRx Effects - Step-by-Step Implementation

**NgRx**გვთავაზობს **filter**-ის ალტერნატივას, რომელიც კონკრეტულ ტიპს დაადგენს:**ofType(),** რომელიც არგუმენტად იღებს კონკრეტულ ექშენს და მის ტიპს კითხულობს ავტომატურად.

ასევე, სასურველია, აღნიშნული ეფექტი შევქმნათ **createEffect()**-ის საშუალებით, არგუმენტად იღებს ფუნქციას და აბრუნებს იგივეს, ანუ ამ შემთხვევაშ ექშენსის ობზერვებლს.

ასევე, მნიშვნელოვანია, რომ თუ კონკრეტულ ეფექტს რაიმეს არ აგზავნის სერვერზე, მეორე არგუმენტად ჰქონდეს ობიექტი: {dispatch: false}. მაგალითი:

1. export class AuthEffects {
3. login$ = createEffect(
4. () =>
5. this.actions$.pipe(
6. ofType(AuthActions.login),
7. tap((action) => {
8. localStorage.setItem("user", JSON.stringify(action.user));
9. })
10. ),
11. { dispatch: false }
12. );
14. constructor(private actions$: Actions) {}
15. }

**3. NgRx Key Concepts - Selectors and Effects**

21. Implementing the Logout Effect (Practice Lesson)

**createEffect()** - ის გამოყენების მთავარი მიზეზი არის ის, რომ თუ რამე მოხდება ეფექტთან, ანუ თუ დაიბაგა, **ngrx**თავიდან გაგზავნის მას.

**dispatch: false** - რომ არ დავაყენოთ სადაც საჭიროა, ინფინიტ ლუპი შეიქმნება.

**3. NgRx Key Concepts - Selectors and Effects**

21. Implementing the Logout Effect (Practice Lesson)

**Keep Login State:**

1. const userProfile = localStorage.getItem('user');
3. if(userProfile) {
4. this.store.dispatch(login({user: JSON.parse(userProfile)}))
5. }

**4. NgRx Development Tools In Depth**

22. Setting up NgRx Router Store and the Time-Travelling Debugger

როუტერის სთორ მოდულის გასააქტიურებლად ჯერ **appModule**-ის დონეზე უნდა დავამატოთ შემდეგი კონფიგურაცია:

1. StoreRouterConnectingModule.forRoot({
2. stateKey: 'router',
3. routerState: RouterState.Minimal
4. })

ხოლო თავად ძირითადი რედიუსერის**.ts** ფაილში უნდა იყოს შემდეგი დეკლარირება:

1. export const reducers: ActionReducerMap<State> = {
2. router: routerReducer
3. };

**4. NgRx Development Tools In Depth**

23. NgRx Runtime Checks - How do they work?

იმისათვის, რომ შემთხვევით აპის სტეიტი პირდაპირ შემთხვევით არ შევცვალოთ, შეგვიძლია **appModule**-ში **root**-ის დონეზე ჩავამატოთ მეორე ფროფერთი **runtimeChecks**, რომელიც თავადაც ობიექტს წარმაოდგენს და რამდენიმე ტიპის ფროფერთის იღებს, ჩვენ აქედან ამ კონკრეტული მიზნისთვის გვჭირდება strictStateImmutability: true, რომელიც აკრძალავს პირდაპირ წვდომას სტეიტზე.

1. StoreModule.forRoot(reducers, {
2. metaReducers,
3. runtimeChecks: {
4. strictStateImmutability: true,
5. },
6. })

**4. NgRx Development Tools In Depth**

23. NgRx Runtime Checks - How do they work?

იმისათვის, რომ შემთხვევით აპის სტეიტი პირდაპირ შემთხვევით არ შევცვალოთ, შეგვიძლია **appModule**-ში **root**-ის დონეზე ჩავამატოთ მეორე ფროფერთი **runtimeChecks**, რომელიც თავადაც ობიექტს წარმაოდგენს და რამდენიმე ტიპის ფროფერთის იღებს, ჩვენ აქედან ამ კონკრეტული მიზნისთვის გვჭირდება strictStateImmutability: true, რომელიც აკრძალავს პირდაპირ წვდომას სტეიტზე.

1. StoreModule.forRoot(reducers, {
2. metaReducers,
3. runtimeChecks: {
4. strictStateImmutability: true,
5. strictActionImmutability: true,
6. strictActionSerializability:true,
7. strictStateSerializability: true
8. },
9. })

**რაც შეეხება სხვებს:**

strictActionImmutability: true - ეს იგივეს აკეთებს, ოღონდ ექშენებზე;

strictActionImmutability: true - იგი აკონტროლებს, მაგალითად, რომ ექშენები არ ინახავენ ისეთ რამეს, როგორიცაა დეითა;

strictStateSerializability: true - იგივე ვრცელდება აქაც, ოღონდ სტეიტზე

**4. NgRx Development Tools In Depth**

24. NgRx Metareducers - Step-by-Step Implementation

**metaReducers** ჩვეულებრივი რედიუსერისგან განსხვავებით უფრო ადრე გამოიძახება, მაგალითად რაიმე ექშენი როცა გაეშვება, ჯერ მეტა რედიუსერი გაეშვება და შემდეგ ჩვეულებრივი რედიუსერი.

ისინი აქამდე გმაოიყენებოდნენ იგივე მოქმედებისთვის, ანუ აპლიკაციის გაყინვისთვის და კონკრეტული ქონდიშენების შემდეგ გაშვებისთვის, რაც წინაში იყო აღწერილი (**runtimeChecks**).

შეგვიძლია ჩვენი მეტა რედიუსერი შევქმნათ, მაგალითად ძირითად რედიუსერების **index.ts**-ში:

იგი წარმოადგენს უბრალო ფუნქციას, რომელიც არგუმენტად იღებს **reducer**-ს, ანუ ყველა რედიუსერს, რომელიც მაგის გაშვების მერე გაეშვება და იგი არის **ActionReducer<any>** ტიპის (any ჩვენს შემთხვევაში, რამდენადაც მხოლოდ ვტესტავთ) და თავად ფუნქციაც იგივე ტიპის ინფორმაციას აბრუნებს;

*გაგრძელება შემდეგში...*

**4. NgRx Development Tools In Depth**

24. NgRx Metareducers - Step-by-Step Implementation

აღნიშნული ფუნქცია დააბრუნებს ასევე რედიუსერის ფუნქციას, რომელსაც არგუმენტებად ექნება **state**და **action**და თავად ეს ფუნქცია დააბრუნებს ძირითად **reducer**-ს იგივე არგუმენტებით:

ბოლოს კი აღნიშნული მეტა რედიუსერების მასივში უნდა დავამატოთ (ფროდაქშენის მასივში)

1. export function logger(reducer:ActionReducer<any>): ActionReducer<any> {
2. return (state, action) => {
3. return reducer(state, action)
4. }
5. }
6. export const metaReducers: MetaReducer<State>[] = !environment.production ? [logger] : [];