**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] : [];

**5. NgRx Entity In Depth**

29. Understanding the NgRx Entity Format

რეკომენდირებულია, პირველ რიგში გავაკეთოთ ექშენები, შესაბამისად ეფექტებიც და ბოლოს რედიუსერი, რომელიც ამათ კორესპონდს.

**5. NgRx Entity In Depth**

29. Understanding the NgRx Entity Format

ენთითის შექმნა რეკომენდირებულია, ვიდრე სტანდარტული ტიპის, როდესაც დეითას სტეიტის ინტერფეისს ვქმნით რედიუსერში. იგი შემდეგნაირად გამოიყურება:

1. export interface CoursesState {
2. entities: {[key: number]:Course}
3. }

**5. NgRx Entity In Depth**

29. Understanding the NgRx Entity Format

ენთითის შექმნა რეკომენდირებულია, ვიდრე სტანდარტული ტიპის, როდესაც დეითას სტეიტის ინტერფეისს ვქმნით რედიუსერში. **entity**იგივე ბიბლიოთეკაა, ხოლოდ **ids**კი - განსაზღვრავს ბუნებრივ წესრიგს ჩვენი ენთითს. იგი შემდეგნაირად გამოიყურება:

1. export interface CoursesState {
2. entities: {[key: number]:Course},
3. ids: number[]
4. }

თუმცა აღნიშნულ შემთხვევაში, ბევრი ენთითის შექმნა მოგვიწევდა დიდი აპლიკაციის შემთხვევაში. ამის მოსაგვარებლად, **ngrx** გვთავაზობს ჩაშენებულ **EntityState**-ის მეთოდს:

1. import { EntityState } from "@ngrx/entity";
2. import { Course } from "../model/course";
4. export interface CoursesState extends EntityState<Course> {}

*გაგრძელდება...*

**5. NgRx Entity In Depth**

30. Implementing Reducers Using the NgRx Entity Adapter

იმისათვის, რომ უფრო გავამარტივოთ ჩვენს ენთითიში დეითას შენაცვა, ვიყენებთ **NgRx**-ის მეთდს **createEntityAdapter<Course>()**, რომელსაც ასევე აქვს ინიშიალ სტეიტი, რომლითაც ვისარგებლებთ. შემდეგ სტანდარტულად **createReducer()**-ით შევქმნით რედიუსერს და მოდულშიც დავაინჯექტებთ.

**reducer.ts:**

1. export interface CoursesState extends EntityState<Course> {}
3. export const adapter = createEntityAdapter<Course>();
5. export const initialCoursesState = adapter.getInitialState();
7. export const coursesReducer = createReducer(
8. initialCoursesState,
9. on(CourseActions.allCoursesLoaded, (state, action) => adapter.addAll(action.courses, state))
10. )

**module:**

1. imports: [
2. EffectsModule.forFeature([CoursesEffects]),
3. StoreModule.forFeature("courses", coursesReducer)
4. ],

**5. NgRx Entity In Depth**

31. NgRx Entity Selectors - Refactoring the Home Component

შემდეგ ვქმნით სელექტორებს, რომლებითაც კომპონენტებში გამოვიძახებთ სასურველ ინფორმაციას.

რამდენადაც ადაპტერი გვაძლებს საშალებას სტეიტში არსებულ ინფორმაციაზე მანიპულირება მოვახდინოთ, მისი **getSelectors()** მეთოდისთ ვისარგებლებთ და იქიდან წავიღებთ **selectAll**-ს.

**reducer.ts:**

1. export const {selectAll} = adapter.getSelectors();

**selectors.ts:**

1. ...
3. import \* as fromCourses from './reducers/course.reducers'
5. export const selectCoursesState = createFeatureSelector<CoursesState>("courses");
7. export const selectAllCourses = createSelector(
8. selectCoursesState,
9. fromCourses.selectAll
10. )
11. export const selectBeginnerCourses = createSelector(
12. selectAllCourses,
13. courses => courses.filter(course => course.category == "BEGINNER")
14. )
15. ...

**component.ts:**

1. reload() {
2. this.beginnerCourses$ = this.store.pipe(select(selectBeginnerCourses))
3. ...
4. }

**5. NgRx Entity In Depth**

32. Entity Adapter Configuration - Understanding sortComparer and selectId

**createEntityAdapter<>()**-ს შეგვიძლია ობიექტი ჩავაწოდოთ არგუმენტად, რომელსაც აქვს ორი ფროფერთი: **sortComparer**და **selectId**, აქედან **sortComparer**იღებს სორტირების ფუნქციას ხოლო **selectId**იმ შემთხვევაში გამოიყენება, თუ მაგალითად ჩვენს ობიექტში ინტერფეისში არ არსებობს **id**ფროფერთი, რამდენადაც ჩვენთან არის, აღნიშნული არ გვჭირდება, თუმცა იგი შემდეგნაირი იქნებოდა:

1. export const adapter = createEntityAdapter<Course>({
2. sortComparer: compareCourses,
3. selectId: course => course.courseId
4. });

**5. NgRx Entity In Depth**

33. NgRx Data Fetching Solution - How to Load Data Only If Needed

თუ რაიმეს ინიციალიზება დამჭრდებოდა, მაგალითად იმის შემოწმება, დეითა მოტანილია თუ არა, თავად**getInitialState()**-ისთვისაც შემეძლო ობიექტის ჩაწოდება, რომელიც **EntityState**-ის ექსთენდის ფროფერთის უპასუხებდა და კონკრეტულ ფროფერთის მნიშვნელობას მიანიჭებდა:

1. export interface CoursesState extends EntityState<Course> {
2. allCoursesLoaded: boolean
3. }
4. export const initialCoursesState = adapter.getInitialState({
5. allCoursesLoaded:false
6. });

**6. NgRx Data In Depth**

37. Why NgRx Data? New Section Introduction

წინაზე ყველაფერი, რაც არ მიწერია კონსპექტში, კოდშია მითითებული, ახლა იგივე კოდს გადავაკეთებთ **NgRx**-ის **Data**-ს გამოყენებით, რამდენადაც წინა ენთითის მანუალურად გამოყენებაზე იყო დაფუძნებული, რაც დიდი აპლიკაციის შემთხვევაში საქმეს გაართულებს.

**6. NgRx Data In Depth**

38. Setting Up NgRx Data in a Lazy Loaded Module

ენთითის დეითას მოდული ჯერ შემოგვაქვს **appModule**-ში შემდეგნაირად:

EntityDataModule.forRoot({})

ხოლო სხვა მოდულში კი, ჯერ შევქმნით ცვლადს, რომელიც იქნება **EntityMetadataMap**-ის ტიპის და იქნება ობიექტი, რომელიც შეინახავს სხვადასხვა სთორების ენთითიებს, რომლებიც ასევე თავადაც ობიექტები არიან, მაგალითად lessons და courses:

1. const entityMetadata: EntityMetadataMap = {
2. Course: {
3. sortComparer: compareCourses,
4. entityDispatcherOptions: {
5. optimisticUpdate: true
6. }
7. },
8. };

შემდეგ კი ამ მოდულში (რომელიც ასევე წარმაოდგენს ლეიზი ლოადედ მოდულს) უნდა დავაინჯექტოთ **EntityDefinitionService**, რომელშიც უნდა დავარეგისტრიროთ ჩვენი ენთითი შემდეგნაირად:

1. constructor(private eds: EntityDefinitionService,) {
2. eds.registerMetadataMap(entityMetadata);
3. }

**6. NgRx Data In Depth**

38. Setting Up NgRx Data in a Lazy Loaded Module

მოდულებში დეკლარაციების მერე საჭროა შევქმნათ სერვისი, რომელიც მაგალითად მოემსახურება **Course**ენთითის. იგი იქნება სტანდარტული სერვისის მსგავსი, უბრალოდ მისი კლასი **extends**გაუკეთებს **ENtityCollectionServiceBase**-ს, რომელიც ენთითის ჯენერიკ ტიპი იქნება. მას ექნება კონსტრუქტორი, რომელიც დაინჯექტდება **EntityCollectionServiceElementsFactory**-ით და კონკსტრუქტორის ბლოკში ჩაჯდება **super()**, რომელიც პირველ არგუმენტად ენთითს ტიპს მიიღებს სტრინგად, ხოლო მეორე არგუმენტად თავად ფაქტორის (ზემოთ რაცაა ნახსენები):

ამ სერვისს გამოვიყენებთ დანარჩენ ყველაფერში:

1. @Injectable()
2. export class CourseEntityService extends EntityCollectionServiceBase<Course> {
4. constructor( serviceElementsFactory: EntityCollectionServiceElementsFactory) {
5. super('Course', serviceElementsFactory);
6. }
8. }

აღნიშნული სერვისი ასევე უნდა დაემატოს კონკრეტული მოდულის პროვაიდერებშიც.

**6. NgRx Data In Depth**

39. How Does NgRx Data Work? Transparent Fetching Data In Action

ბექიდან დეითას მოტანის კონტროლისთვის აღნიშნული სერვისი ისარგებლებს ცალკე ფაილით, რომელიც **Resolver**-ს წამოარდგენს, რომელიც იტვირთება მანამ, სანამ კონკრეტული როუტი გააქტიურდება.

რესოლვერის სტანდარტული დეკლარირების ტიპი ქვემოთა მოცემული, იგი ასევე ჩვეულებრივ სერვისს წარმოადგენს, ამიტომაც კონკრეტული მოდულის პროვაიდერებში უნდა დადეკლარირდეს და ასევე, რამდენადაც რეზოლვერია, კონკრეტულ როუტზეც დაჯდება და რაარესოლვებს.

**resolver.ts:**

1. @Injectable()
2. export class CoursesResolver implements Resolve<boolean> {
3. resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> {}

**module.ts:**

1. export const coursesRoutes: Routes = [
2. {
3. path: '',
4. component: HomeComponent,
5. resolve: {
6. courses: CoursesResolver
7. }
8. },
9. ]
11. @NgModule({
12. providers: [
13. CoursesResolver,
14. ]
15. })

**6. NgRx Data In Depth**

39. How Does NgRx Data Work? Transparent Fetching Data In Action

შემდეგ ჩვენს რესოლვერს ვაინჯექტებთ ახალხანს შერქმნილი ენთითი სერვისით, მას კი ცალკე მოყვება ყველა საჭრო მათოდი, მათ შორისაა **http**რექვესტების გასაკეთებელი მეთოდებიც. მას აქვს მეთდი **.getAll()**მაგალითად,  რომელიც მიიღებს დეითას ბექიდან და მიღება თუ წარმატებით დასრულდა, სთორშიც შეინახავს.

დეითას ბულიანად გადასაქცევად ვიყენებთ **!!** პრეფიქსს: loaded => !!loaded

**NgRx Store** ასევე სერვერთან კომუნიკაციის მეთდერბის გამოყენებისას, იყენებს ჩაშენებულ კონვენციას, რისი მიხედვითაც, კონკრეტულ მისამართს უმატებს **/api** -ის ლინკში და ასევე ენთითის სახელს, ოღონდ მრავლობითი სუფიქსის დამატებით, ჩვენს შემთხვევაში ასე გამოიყურება:

**localhost:3000/api/courses**

თუმცა აღნიშნული კონვენციის შეცვლა შეიძლება.

**6. NgRx Data In Depth**

40. NgRx Custom Data Service - Fetching Data From the Backend

იმისათვის, რომ კოსტუმაცია შევძლოთ დეითასი, საჭიროა შევქმნათ ცალკე დეითა სერვისი, რომელიც მოემსახურება კურსების ენთითის სერვისს.

იგი extends გაუკეთებს **DefaultDataService**-ს, რომელიც ენთითის დეითას ტიპის იქნება და მის კონსტრუქტორში დაინჯექტდება **HttpClient**(ამ შემთხვევაში) და **HttpUrlGenerator**-ს და მისსავე ბლოკში **super()-**ში მიიღებს ჯერ ენთითის სახელს, შემდეგ **http**-ს და ბოლოს httpUrlGenerator-ს.

რამდენადაც აღნიშნულიც ჩვეულებრივი სერვისია, კონკრეტული მოდულის პროვაიდერებში წავა.

ასევე, აღნიშნული უნდა დაინჯექტდეს კონკრეტული მოდულის კონსტრუქტორშიც. ჯერ დაინჯექტდება **EntityDataService**, ხოლო მერე ჩვენი სერვისი და ამ დეითა სერვისში დარეგისტრირდება ჩვენი სერვისი (პირველი არგუმენტი ისევ ენთითის სახელი იქნება, მეორე კი თავად სერვისი).

**data.service.ts:**

1. @Injectable()
2. export class CoursesDataService extends DefaultDataService<Course> {
3. constructor(http:HttpClient, httpUrlGenerator: HttpUrlGenerator) {
4. super('Course', http, httpUrlGenerator);
5. }
6. }

**6. NgRx Data In Depth**

40. NgRx Custom Data Service - Fetching Data From the Backend

**course.module.ts:**

1. export class CoursesModule {
2. constructor(
3. private eds: EntityDefinitionService,
4. private entityDataService: EntityDataService,
5. private coursesDataService: CoursesDataService) {
7. eds.registerMetadataMap(entityMetadata);
8. entityDataService.registerService('Course', coursesDataService);
10. }
11. }

**6. NgRx Data In Depth**

40. NgRx Custom Data Service - Fetching Data From the Backend

მას შემდეგ, რაც დეითა სერვისს შევქმნით, დეფაულტ ბიჰეივორის ნაცვლად, რასაც ჩვენ მივუთითებთ, იმას გააკეთებს.

ამისთვის საჭიროა გადავაწეროთ დეფაულტს კონკრეტული მეთდების გამოძახებით. ჩვენს შემთხვევაში გვსურს, რომ **.getAll()**-ს გადავაწეროთ. იგი დააბრუნებს ობზერვებლს და შიგნით ჩაჯდება **http**ქოლი და ხელით მივუთითებთ სადაც გაგზავნოს რექვესტი და როგორ გაფილტროს დეითა.

**data.service.ts:**

1. @Injectable()
2. export class CoursesDataService extends DefaultDataService<Course> {
4. constructor(http:HttpClient, httpUrlGenerator: HttpUrlGenerator) {
5. super('Course', http, httpUrlGenerator);
6. }
8. getAll(): Observable<Course[]> {
9. return this.http.get('/api/courses')
10. .pipe(
11. map(res => res["payload"])
12. );
13. }
15. }

**6. NgRx Data In Depth**

40. NgRx Custom Data Service - Fetching Data From the Backend

თუ კონკრეტულ ქონვენშენს მიყვებოდა ჩვენი ბექი, რაც **NgRx Store**-შია, **store** სერვისის ცალკე შექმნა საჭირო აღარ იქნებოდა.

**6. NgRx Data In Depth**

41. Controlling Data Loading with the NgRx Data loaded flag

მას შემდეგ, რაც დეითა შეიქმნება ენთით სერვისის მიერ, მას ასევე ექნება თავისივე სტეიტების მენეჯმენტი, მაგალითად იტვირთება, თუ ჩატვირთულია დეითა: **loaded (loaded$), loading (loading$)**,  რომლებსაც გამოვიყენებთ დაკვირვებისთვის და შესაბამისი მოქმედებისთვის

**courses.resolver.ts:**

1. export class CoursesResolver implements Resolve<boolean> {
3. constructor(private coursesService: CourseEntityService) {}
5. resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> {
6. return this.coursesService.loaded$
7. .pipe(
8. tap(loaded => {
9. if (!loaded) {
10. this.coursesService.getAll();
11. }
12. }),
13. filter(loaded => !!loaded),
14. first() //პირველივე მნიშვნელობის მიღებისას დააკომპლიტებს
15. );
17. }
19. }

**6. NgRx Data In Depth**

42. Querying Store Data with NgRx Data and the entities$ Observable

მას შემდეგ, რაც გაიმართება ენთითი სერვისი, აპლიკაციის კომპონენტებში **EntityService**-ის გამოძახებით გვექნება წვდება მთელის დეითაზე ობზერვებლის სახით (**entities$.**):

1. constructor(private coursesService: CourseEntityService) {}
3. this.beginnerCourses$ = this.coursesService.entities$
4. .pipe(
5. map(courses => courses.filter(course => course.category == 'BEGINNER'))
6. );

ასევე, სორტირებაც შეგვიძლია ჩავრთოთ კონკრეტულ მოდულში დეკლარირებული ენთთიდან (როგორც ენთითს შემთხვევაში ვიქცეოდით) **sortComparer**ფროფერთის გამოყენებით, რომელიც სორტირების ფუნქციას იღებს მნიშვნელობად:

1. const entityMetadata: EntityMetadataMap = {
2. Course: {
3. sortComparer: compareCourses,
4. },
5. };

**6. NgRx Data In Depth**

43. NgRx Data CRUD - Why use Optimistic Updates?

პესიმისტური მიდგომა ნიშნავს, როცა **UI**მას შემდეგ აფდეითდება, რაც ბექში ოპერაცია დასრულდება, ხოლო ოპტიმისტურია - როცა **UI**ბექში ოპერაციის დასრულებამდე აფდეითდება.

ამისთვის შეგვიძლია გამოვიყენოთ ენთითის სერვისის მეთდი **.update()**, რომელიც არა მხოლოდ დააფდეითებს სთორს, ასევე პესიმისტურად სინქრონში მის შემდგომ ბრძანებასაც წაიკითხავს, რომელიც დახურავს კონკრეტულ დიალოგის ფანჯარას და ასევე ბექშიც გაუშვებს **PUT**რექვესტს. არგუმენტად კი იღებს მხოლოდ იმ დეითას, რომელსაც ააფდეითებს:

1. this.coursesService.update(course);
2. this.dialogRef.close();

იგი კონვენციით ავტომატურად მიხვდება სად შეინახოს დეითა, ანუ რომელ ენდპოინტში.

თუმცა მაინც არ დაააფდეითებს **UI**-ს, რადგან ბექში რექვესტი მიდის. რამდენადაც დეფაულტზე **.update()**-ს აქვს პესიმისტური ეპროაჩი, აღნიშნული კონკრეტულ მოდულში დეკლარირებული ენთითიდან შეგვიძლია შევცვალოთ.

*გრძელდება...*

**6. NgRx Data In Depth**

43. NgRx Data CRUD - Why use Optimistic Updates?

**sortComparer**-ის ქვევით ჩავამატებთ ახალ ფროფერთის - **entityDispatcherOptions**, რომელიც არის ობიექტი და ჩვენს შემთხვევაში გვჭირდება, რომ **optimisticUpdate true**იყოს, შესაბამისად:

1. const entityMetadata: EntityMetadataMap = {
2. Course: {
3. sortComparer: compareCourses,
4. entityDispatcherOptions: {
5. optimisticUpdate: true
6. }
7. },
8. };

**6. NgRx Data In Depth**

44. NgRx Data CRUD - Why Pessimistic Data Creation?

იმისათვის, რომ გამოვიწვიოთ **POST**რექვესტი, ანუ შევქმნათ ახალი დეითა, **EntityService**გვაძლევს იმის საშალებს**.add()** მეთოდის გამოყენებით, რომელიც არა მხოლოდ ქეშში ინახავს დეითას, არამედ ბექის რექვესტსაც აკეთებს.

არგუმენტად მიიღებს დეითას, რაც უნდა შეინახოს.

იგი ობზერვებლს დააბრუნებს, რომლის სუბსქრაიბიც შეიძლება და დეითას მოსმენა:

1. this.coursesService.add(course).subscribe( newCourse => {
2. console.log('New Course', newCourse);
3. this.dialogRef.close();
4. });

დეფაულტზე **add()** პესიმისტურია და იგი იდეალურია ოფლაინის დროსაც, რამდენადაც **UI**არ განახლდება, თუ არ შეინახა დეითა.

მისი ოპტიმისტურად გადაკეთებაც შეგვეძლო, თუმცა ასე სჯობს რომ იყოს...

**6. NgRx Data In Depth**

45. NgRx Data CRUD - Optimistic Delete Implementation

რაიმე ენთითის წასაშლელად, რამდენადაც როგორც ქეშში, ასევე ბექშიც უნდა წაიშალოს, გამოიყენება **EntityService**-ის **.delete()** მეთოდი, რომელიც არგუმენტად იღებს წასაშლელ დეითას და როგორც ქაშში, ასევე ბექშიც შლის დეითას. იგი ბუნებით ოპტიმისტურია, ანუ ეგრევე დაააფდეითებს **UI**-ს და ასევე სუბსქრაიბიც შეგვიძლია, რაც საჭირო არაა, თუ არ გვსურს. მაგალითი:

1. this.courseService.delete(course).subscribe(
2. () => console.log("Delete completed"),
3. err => console.log("Deleted failed", err)
4. );

**6. NgRx Data In Depth**

46. Setting Up a new Entity - The Lesson Entity

ახალი ენთითს დასამატებლად, კონკრეტულ მოდულში დეკლარირებულ ენთითიებს უნდა ჩავამატოთ და ვისარგებლოთ ინდივიდუალურ მოთხოვნებითაც, მაგალითად, **sortComparer**-ით და შემდგომ კონკრეტულ ენთითის შევუქმნათ ცალკე სერვისი და იგივენაირად გავმართოთ ყველაფერი, როგორც წინაში.

**module.ts:**

1. const entityMetadata: EntityMetadataMap = {
2. Course: {
3. sortComparer: compareCourses,
4. entityDispatcherOptions: {
5. optimisticUpdate: true
6. }
7. },
8. Lesson: {
9. sortComparer: compareLessons
10. }
11. };

**6. NgRx Data In Depth**

48. Lessons Pagination using NgRx Data

გარკვეული დეითას გამოსაძახებლად, ანუ ლიმიტირებული, ანუ არა მთლიანი დეითას, გამოიყენება **EntityService**-ის მეთოდი**.getWithQuery()**, რომელიც არგუმენტებად მიიღებს შემდეგ პარამეტრებს:**'courseId', 'pageNumber'** და **'pageSize'**, რომლებიც ყველა სტრინგი უნდა იყოს:

1. loadLessonsPage(course: Course) {
2. this.lessonsService.getWithQuery({
3. 'courseId': course.id.toString(),
4. 'pageNumber': this.nextPage.toString(),
5. 'pageSize': '3'
6. });
8. this.nextPage += 1;
9. }

ასევე შეგვიძლია ვისარგებლოთ **loading$**ობზერვებლით, რომელიც **EntityService**-ში ინახებას და ლოგიკურად ბულიანს აბრუნებს:

1. this.loading$ = this.lessonsService.loading$.pipe(delay(0));

**delay() rxjs** ოპერატორი დილეის გაუკეთებს **0** მილიწამით, თუმცა რამდენადაც იგი ასინქრონულია, ბოლოს გაეშვება.

**6. NgRx Data In Depth**

49. Switching an NgRx Application to OnPush Change Detection

**changeDetection: ChangeDetectionStrategy.OnPush** - გამოიყენება **@Component({})**-ის ფროფერთიდ და იგი გულისხმობს იმას, რომ კონკრეტული თემფლეითი მაშინ წაიკითხება, თუ **@Input()**-ით ან **async**პაიპით დაეითა შევა თემფლეითში.

**შესადარებელი კოდები:** [**https://github.com/angular-university/ngrx-course/branches**](https://github.com/angular-university/ngrx-course/branches)