**24. Bonus: Working with NgRx in our Project**

344. What is Application State?

**NgRx:**

იგი წარმოადგენს აპლიკაციის მდგომარეობას დარეფრეშებამდე.

**State**არის - ნებისმიერი ინფორმაცია, რომელიც აკონტროლებს, რა უნდა გამოჩნდეს ეკრანზე.

**RxJs**- გვეხმარება დიდი აპლიკაციების შემთხვევაში აპლიკაციის სტეიტის უფრო მარტივ მართვაში, რაც მის გარეშეც შესაძლებელია, თუმცა შედარებით რთული.

**24. Bonus: Working with NgRx in our Project**

345. What is NgRx?

**NgRx Intro 1.1:**

ანგულარმა მისი ობიექტების რეფერენსული ტიპის გამო, შესაძლოა სტეიტის აფდეითი ვერ დააფიქსიროს, ანუ მაგალითად თუ რომელიმე ობიექტში რაიმე ფროფერთი შეიცვლება, ან უბრალოდ **http**რექვესტის პრობლემა წარმოიქმნას, როცა მისი ჰენდლინგი ვერ ხდება, ასევე იგი გვეხმარება სტეიტის ნებისმიერ ადგილზე განახლებაში.

**NgRx**ეკუთვნის **Redux**-სს:

**Redux**თავად წარმოადგენს სტეიტის მენეჯმენტის პატერნს, იგი თავისთავად ბიბლიოთეკაა, რომელიც ამ პატერნის იმპლემენტაციაში გვეხმარება.

იგი ქმნის საცავს, რომელშიც ინახება ყველა ტიპის სტეიტი, რაც ჩვენს აპლიკაციაშია, თავისთავად სერვისები და კომპონენტები ისევ ურთიერთობენ, თუმცა იგინი ინფორმაციას ამ საცავიდან იღებენ.

ცვლილებისას აღნიშნული ცვლილება იგზავნება **Action**-ში, რომელიც უკავშირდება ე.წ. **Reducer**-ს, რომელიც წარმაოდგენს ფუნქციას და იგი აბრუნებს ძველი სტეიტის ახალ ვერსიას, ანუ კოპის, ხოლო აღნიშნული სტეიტი უკვე მიემართება საცავში.

**NgRx**საბოლოო ჯამში კი არის **Redux**-ის **Angular**ვერსია.

**24. Bonus: Working with NgRx in our Project**

345. What is NgRx?

**NgRx Intro 1.2:**

იგი ასევე წარმაოდგენს ერთ დიდ **Observable**-ს რაც იმის საშუალებას გვაძლევს, რომ ყველა ოპერატორი გამოვიყენოთ მიღებული ინფორმაციის ტრანსფორმაციისთვის.

იგი ასევე გვაძლევს ხელსაწყოს **Http**რექვესტის **Side Effect**-ების მოსაგვარებლად.

ზოგად, **Reducer**-ში მხოლოდ სინქრონული ტასკები იგზავნება, თუმცა **NgRx**გვაწვდის ხელსაწყოს, სადაც **Http**-ის ასინქრონული ტასკები წავა.

**24. Bonus: Working with NgRx in our Project**

346. Getting Started with Reducers

**NgRx Installation & Reducer start:**

ინსტალაციისთვის **CLI**-ში: npm install --save @ngrx/store

იგი იქნება მთავარი პაკეჯი (სამომავლოდ სხვებიც დაგვჭირდება, თუმცა ეს ბირთვია).

მის ასამუშავებლად უნდა შევქმნათ ცალკე **ts**ფაილი (**shopping-list.reducer.ts**):

1. import { Ingredient } from "../shared/ingredient.model";
3. const initialState = {
4. ingredients: [
5. new Ingredient('Apples', 5),
6. new Ingredient('Tomatoes', 10),
7. ]
8. };
10. export function shoppingListReducer(state = initialState, action){
12. }

**24. Bonus: Working with NgRx in our Project**

347. Adding Logic to the Reducer

**NgRx Reducer Logic:**

ფუნქციის შიგნით სასურველია შევქმნათ **switch()**, რომელშიც განისაზღვრება **action.type**-ები (ასევე **action**არგუმენტს უნდა გავუწეროთ ტიპი: **Action**, რომელიც იმპორტდება **@ngrx/store**-დან). აღნიშნული ტიპების სახელები ჩვენზეა დამოკიდებული.

შემდეგ, მაგალითად პირველ ქეისში, უნდა დავაბრუნოთ არსებული სტეიტის ახალი ვეერსია (ანუ ზევიდან არ უნდა გადავაწეროთ არსებულ სტეიტს). ამისთვის უბრალოდ გამოვიყენებთ **return**და **spread**ოპერატორით დავაკოპიროთ ყველა ფროფერთი:

1. return{
2. ...state
3. };

შემდეგ ვუმატებთ ფროფერთის, რომელიც წარმოადგენს მასივს და მასში ვინახავთ ჯერ ძველის მაგალითად ინგრენდიენტებს და მეორეს უკვე განახლებულს (**action**), რომელსაც სამომავლოდ განვავითარებთ:

1. switch(action.type){
2. case 'ADD\_INGREDIENT':
3. return {
4. ...state,
5. Ingredient: [...state.ingredients, action]
6. };
7. }

**24. Bonus: Working with NgRx in our Project**

348. Understanding & Adding Actions

**NgRx working on action separately:**

შეგვიძლია **action**ლოგიკა ცალკე **ts**ფაილში გავიტანოთ და ეს ორივე ფაილი ერთ ფოლდერში გავაერთიანოთ, შემდეგ კი **reducer**-ის **ts**ფაილში უკვე აქედან დავაიმპორტებთ მნიშნველოვან ინფორმაციას:

**action.ts:**

1. export const ADD\_INGREDIENT = 'ADD\_INGREDIENT'
3. export class AddIngredient implements Action {
4. readonly type = ADD\_INGREDIENT;
5. payload: Ingredient
6. }

**reducer.ts:**

1. const initialState = {
2. ingredients: [
3. new Ingredient('Apples', 5),
4. new Ingredient('Tomatoes', 10),
5. ]
6. };
8. export function shoppingListReducer(state = initialState, action:Action){
9. switch(action.type){
10. case ADD\_INGREDIENT:
11. return {
12. ...state,
13. Ingredient: [...state.ingredients, action]
14. };
15. }
16. }

**24. Bonus: Working with NgRx in our Project**

349. Setting Up the NgRx Store

**NgRx setting-up store 1.1:**

წინა კოდში მცირე ცვლილებები შევიდა, რაც იმპორტს შეეხებოდა, ანუ action-ის ყველა თემა მის **ts**ფაილში გადავიდა, ხოლო **reducer.ts** ფაილი კი მხოლოდ აიმპორტებს მათ და შემდეგ აექსპორტებს კონკრეტულ ფუნქციას:

1. import { Ingredient } from "../../shared/ingredient.model";
2. import \* as ShoppingListActions from "./shopping-list.actions";
4. const initialState = {
5. ingredients: [
6. new Ingredient('Apples', 5),
7. new Ingredient('Tomatoes', 10),
8. ]
9. };
11. export function shoppingListReducer(state = initialState, action:ShoppingListActions.AddIngredient){
13. switch(action.type){
14. case ShoppingListActions.ADD\_INGREDIENT:
15. return {
16. ...state,
17. Ingredient: [...state.ingredients, action.payload]
18. };
19. }
20. }

**24. Bonus: Working with NgRx in our Project**

349. Setting Up the NgRx Store

**NgRx setting-up store 1.2:**

იმისათვის, რომ ამ **reducer**ფუნქციის გამოყენება შევძლოთ, აღნიშნული უნდა დაიმპორტდეს **appModule**-ში:

1. import { StoreModule } from '@ngrx/store'
3. imports: [
4. StoreModule.forRoot({
5. shoppingList: shoppingListReducer
6. })
7. ]

**24. Bonus: Working with NgRx in our Project**

350. Selecting State

**NgRx component introduction 1.1:**

ჩვენს მიერ შექმნილი**reducer.ts-**ის ლოგიკაში გამოსაყენებლად, ჯერ პირველ რიგში თავად **reducer.ts**-ში ერთი მცირე ცვლილება უნდა შევიტანოთ, რაც დეფაულტ სტეიტს განუსაზღვრავს ჩვენს **switch**-ს:

1. switch(action.type){
2. case ShoppingListActions.ADD\_INGREDIENT:
3. return {
4. ...state,
5. Ingredient: [...state.ingredients, action.payload]
6. };
7. default:
8. return state;
9. }

ხოლო შემდეგ, თავად კომპონენტის ფაილში უკვე არსებული მაგალითად ინგრენდიენტების მასივი უნდა შევცვალოთ, ანუ მისი ტიპი მასივიდან **Observable**-ზე უნდა გადაკეთდეს: ingredients:Observable<{ingredients: Ingredient[]}>;

ხოლო შემდეგ უკვე არსებულ ესაინმენტს და შესაბამისად არსებულ საბსქრიფშენს ვაუქმებთ და კომპონენტს ვაინჯექტებთ შესაბამისად:

constructor( private store: Store<{shoppingList: {ingredients: Ingredient[]}} ){}

**24. Bonus: Working with NgRx in our Project**

350. Selecting State

**NgRx component introduction 1.2:**

ამის შემდეგ კი, **NgOnInit**-ში ინგრენდიენტების ფროფერთის ესაინმენტს ვაკეთებთ შემდეგნაირად:

this.ingredients = this.store.select('shoppingList');

ხოლო შემდეგ აღნიშნული კომპონენტის **html**ფაილში**\*ngFor** ლუპში შეგვაქვს ცვლილება:

1. \*ngFor="let ingredient of (ingredients | async).ingredients; let i = index"

**24. Bonus: Working with NgRx in our Project**

351. Dispatching Actions

**NgRx edit component introduction:**

იმისათვის, რომ აღნიშნული საცავი გამოვიყენოთ რაიმე აითემის დამატებისას ან სხვა მოქმედებისას, რაც სთორს შეეხება, აღნიშნული თავად ნახსენებ კომპონენტშიც უნდა დაინჯექტდეს. ჩვენ შემთხვევაში შეგვიძლია დავიწყოთ**shopping-edit.component.ts**-დან:

ინჯექტდება იგივენაირად, როგორც ლისტში:

1. private store: Store<{shoppingList: {ingredients: Ingredient[]}}>

ხოლო შემდეგ, საბმიტის მეთოდში წინა ვერსია უქმდება. ანუ **addIngredient()**მეთოდი და ემატება ახალი:

1. this.store.dispatch(new ShoppingListActions.AddIngredient(newIngredient))

აღნიშნულის ასამუშავებლად, **action**-ის **ts**ფაილში ერთი ფროფერთი კონსტრუქტორში უნდა გადავიტანოთ და მას შემდეგ ესეთი სახე ექნება:

1. export class AddIngredient implements Action {
2. readonly type = ADD\_INGREDIENT;
3. constructor(public payload: Ingredient){}
4. }

**24. Bonus: Working with NgRx in our Project**

352. Multiple Actions

**NgRx service component introduction 1.1:**

შემდეგი მოქმედების განსახორციელებლად, სადაც რამდენიმე ინგრენდიენტს ვამატებთ სხვა კომპონენტიდან და არა ხელით, შესაბამისად განსხვავებულ **action**-ს წარმაოდგენს და განსხვავებული ობიექტის უნდა შევუქმნათ **action**ფაილში:

1. export const ADD\_INGREDIENTS = 'ADD\_INGREDIENTS';
3. export class AddIngredients implements Action {
4. readonly type = ADD\_INGREDIENTS;
5. constructor(public payload: Ingredient[]){}
6. }

ხოლო იმისათვის, რომ განისაზღვროს სამომავლოდ **reducer**რომელს გამოიყენებს, საჭიროა **typescript**-ის ტრიუკის გამოყენება, რომელიც ცვლადში ადეკლარირებს რამდენიმე ტიპს:

1. export type ShoppingListActions = AddIngredient | AddIngredients

**24. Bonus: Working with NgRx in our Project**

352. Multiple Actions

**NgRx service component introduction 1.2:**

**reducer.ts** ფაილში კი ცვლილება შევა კონსტრუქტორშიც და ახალი **switch case**-იც დაემატება:

1. export function shoppingListReducer(state = initialState, action:ShoppingListActions.ShoppingListActions){
2. case ShoppingListActions.ADD\_INGREDIENTS:
3. return {
4. ...state,
5. ingredients: [...state.ingredients, ...action.payload]
6. }
7. }

ამის შემდეგ თავად სერვისის ფაილს უნდა მივადგეთ, სადაც კონკრეტული მოქმედება ხორციელდება, ძველებურად დავაინჯექტოთ სერვისი, ძველი მოქმედება წავშალოთ და მეთოდში ახალი დავადეკლარიროთ:

1. import \* as ShippingListActions from '../shopping-list/store/shopping-list.actions'
3. addIngredientsToShoppingList(ingredients: Ingredient[]) {
4. this.store.dispatch(new ShippingListActions.AddIngredients(ingredients))
5. }

**24. Bonus: Working with NgRx in our Project**

354. Updating & Deleting Ingredients

**NgRx component introduction (Update & Delete) 1.1:**

ამისთვის, როგორც აქამდე, ჯერ **actions.ts** და **reducer.ts** უნდა განვაახლოთ.

**actions.ts** განახლება:

1. export const UPDATE\_INGREDIENT = 'UPDATE\_INGREDIENT';
2. export const DELETE\_INGREDIENT = 'DELETE\_INGREDIENT'
4. export class UpdateIngredient implements Action {
5. readonly type = UPDATE\_INGREDIENT
6. constructor(public payload: {index: number, ingredient: Ingredient}){}
7. }
9. export class DeleteIngredient implements Action {
10. readonly type = DELETE\_INGREDIENT
11. constructor(public payload: number){}
12. }
14. export type ShoppingListActions = AddIngredient | AddIngredients |UpdateIngredient | DeleteIngredient

**24. Bonus: Working with NgRx in our Project**

354. Updating & Deleting Ingredients

**NgRx component introduction (Update & Delete) 1.2:**

**reducer.ts-**ის განახლებაში უბრალოდ 2 ახალი **case**ჩაემატება, რომლებიც წინებისგან შედარებით განსხვავებული ფორმატის იქნებიან:

1. case ShoppingListActions.UPDATE\_INGREDIENT:
2. const ingredient = state.ingredients[action.payload.index];
3. const updatedIngredient = {
4. ...ingredient,
5. ...action.payload.ingredient
6. };
7. const updatedIngredients = [...state.ingredients];
8. updatedIngredients[action.payload.index] = updatedIngredient;
9. return{
10. ...state,
11. ingredients: updatedIngredients
12. };
14. case ShoppingListActions.DELETE\_INGREDIENT:
15. return{
16. ...state,
17. ingredients: state.ingredients.filter((ig, igIndex) =>{
18. return igIndex !== action.payload
19. })
20. };

**24. Bonus: Working with NgRx in our Project**

354. Updating & Deleting Ingredients

**NgRx component introduction (Update & Delete) 1.3:**

ხოლო თავად კომპონენტის **ts** ფაილში კი, როგორც წინაზე, ძველი ვერსიები უქმდება და ახლები ემატება:

**update:**

1. this.store.dispatch(new ShoppingListActions.UpdateIngredient({index: this.editedItemIndex, ingredient: newIngredient}))

**delete:**

1. this.store.dispatch(new ShoppingListActions.AddIngredient(newIngredient))

თუმცა update მეთოდი, როცა რეცეპტი მოინიშნება, ისევ ძველ სტეიტს აბრუნებს, ანუ ნაწილობრივ მუშაობს მხოლოდ. *ამას სამომავლოდ გამოვასწორებთ...*

**24. Bonus: Working with NgRx in our Project**

355. Expanding the State

**NgRx State Extention 1.1:**

სანამ არსებულ პრობლემას მოვაგვარებდეთ, ერთი საკითხი შეგვიძლია გავამარტივოთ: ტიპის ინჯექშენის გამარტივება.

მაგალითად, თუ ჩვენ გვსურს ჩვენს **reducer.ts**-ში **initialState**-ში ცვლილების შეტანა, ყველა ინჯექტში უნდა შევიტანოთ შემდეგ აღნიშნული, ანუ ახალ ფროფერთის თუ ჩავამატებთ ამ ობიექტს. ამის თავიდან ასარიდებლად, შემდეგნაირად ვიქცევით:

**reducer.ts**-ში მაგალითად ვნახლებთ **initialState**-ის ობიექტს:

1. const initialState: State = {
2. ingredients: [
3. new Ingredient('Apples', 5),
4. new Ingredient('Tomatoes', 10),
5. ],
6. editedIngredient: null,
7. editedIngredientIndex: -1
8. };

განახლებაში უკვე ნახსენებ **(:State)** ტიპის ინტერფეისს ვქმნით ამის შესაბამისად:

1. export interface State {
2. ingredients: Ingredient[];
3. editedIngredient: Ingredient;
4. editedIngredientIndex: number
5. }

**24. Bonus: Working with NgRx in our Project**

355. Expanding the State

**NgRx State Extention 1.2:**

ასევე ვქმნით ინტერფეისს, რომელიც სამომავლოდ შოპინგლისტის გარდა, შეუძლია სხვა მეთოდებიც შეინახოს:

1. export interface AppState {
2. shoppingList: State;
3. }

ხოლო ამის შემდეგ ცვლილება შეგვაქვს თავად ფუნქციაშიც, ანუ ვამატებთ ჩვენს მიერ შეექმნილ ტიპს state-ს:

1. export function shoppingListReducer(state: State = initialState, action:ShoppingListActions.ShoppingListActions)

ბოლოს კი ყველა კომპონენტში, სადაც ინჯექტდება აღნიშნული **store**, ტიპი განახლდება შემდეგნაირად:

ჯერ დაიმპორტდება ეს კონკრეტული**reducer.ts** ლოგიკა, და შემდგომ ტიპში ჩაჯდება ჩვენი უკანასკნელი ინტერფეისი:

1. import \* as fromShoppingList from './store/shopping-list.reducer'
3. constructor(
4. private store: Store<fromShoppingList.AppState>
5. ) {}

**24. Bonus: Working with NgRx in our Project**

356. Managing More State via NgRx

**NgRx Start & Stop Editing arrangement 1.1:**

წინაზე არსებულ ბაგს შემდეგნაირად გამოვასწორებთ:

**actions.ts** განახლება:

1. export const START\_EDIT = 'START\_EDIT';
2. export const STOP\_EDIT = 'STOP\_EDIT';
4. export class StartEdit implements Action {
5. readonly type = START\_EDIT;
6. constructor(public payload: number){}
7. }
9. export class StopEdit implements Action {
10. readonly type = STOP\_EDIT;
11. }
13. export type ShoppingListActions = AddIngredient | AddIngredients |UpdateIngredient | DeleteIngredient | StartEdit | StopEdit

**24. Bonus: Working with NgRx in our Project**

356. Managing More State via NgRx

**NgRx Start & Stop Editing arrangement 1.2:**

**reducer.ts**განახლება (ემატება **case**-ები):

1. case ShoppingListActions.START\_EDIT:
2. return {
3. ...state,
4. editedIngredientIndex: action.payload,
5. editedIngredient: {...state.ingredients[action.payload]}
6. };
8. case ShoppingListActions.STOP\_EDIT:
9. return {
10. ...state,
11. editedIngredient: null,
12. editedIngredientIndex: -1
13. };

ხოლო, **shopping-edit.component.ts**-ში კი:

1. onClear() {
2. this.slForm.reset();
3. this.editMode = false;
4. this.store.dispatch(new ShoppingListActions.StopEdit())
5. }
6. ngOnDestroy() {
7. this.subscription.unsubscribe();
8. this.store.dispatch(new ShoppingListActions.StopEdit())
9. }

**24. Bonus: Working with NgRx in our Project**

356. Managing More State via NgRx

**NgRx Start & Stop Editing arrangement 1.3:**

ხოლო იგივე კომპონენტში, **NgOnInit**-ში ძველი მეთოდი ნადგურდება და ახალია:

1. this.subscription = this.store.select('shoppingList').subscribe(stateData => {
2. if(stateData.editedIngredientIndex > -1){
3. this.editMode = true;
4. this.editedItem = stateData.editedIngredient;
5. this.slForm.setValue({
6. name: this.editedItem.name,
7. amount: this.editedItem.amount
8. })
9. }else {
10. this.editMode = false
11. }
12. })

თუმცა კიდევ ერთი ბაგი გვაქვს, *რომელსაც სამომავლოდ გამოვასწორებთ...*

**24. Bonus: Working with NgRx in our Project**

357. Removing Redundant Component State Management

**NgRx Fixing bug from previous 1.1:**

წინაზე არსებული ბაგის გამოსასწორებლად, შემდეგი ცვლილებები შეგვაქვს:

**actions.ts** არსებულს ვანახლებთ:

1. export class UpdateIngredient implements Action {
2. readonly type = UPDATE\_INGREDIENT
4. constructor(public payload: Ingredient){}
5. }
7. export class DeleteIngredient implements Action {
8. readonly type = DELETE\_INGREDIENT
9. }

**24. Bonus: Working with NgRx in our Project**

357. Removing Redundant Component State Management

**NgRx Fixing bug from previous 1.2:**

**reducer.ts**-ის არსებულს ვანახლებთ **actions.ts**-ის მიხედვით **1.1**:

1. case ShoppingListActions.UPDATE\_INGREDIENT:
2. const ingredient = state.ingredients[state.editedIngredientIndex];
3. const updatedIngredient = {
4. ...ingredient,
5. ...action.payload
6. };
7. const updatedIngredients = [...state.ingredients];
8. updatedIngredients[state.editedIngredientIndex] = updatedIngredient;
10. return{
11. ...state,
12. ingredients: updatedIngredients,
13. editedIngredientIndex: -1,
14. editedIngredient: null
15. };

**24. Bonus: Working with NgRx in our Project**

357. Removing Redundant Component State Management

**NgRx Fixing bug from previous 1.3:**

**reducer.ts**-ის არსებულს ვანახლებთ **actions.ts**-ის მიხედვით **1.2**:

1. case ShoppingListActions.DELETE\_INGREDIENT:
2. return{
3. ...state,
4. ingredients: state.ingredients.filter((ig, igIndex) =>{
5. return igIndex !== state.editedIngredientIndex
6. }),
7. editedIngredientIndex: -1,
8. editedIngredient: null
9. };

ხოლო თავად **shopping-edit.component.ts**-ში:

**editedItemIndex**ფროფერთის ყველგან ვშლით და მის ნაცვლად შეგვაქვს შემდეგი ცვლილებები:

1. this.store.dispatch(new ShoppingListActions.UpdateIngredient(newIngredient))

და **onDelete** მეთდში:

1. this.store.dispatch(new ShoppingListActions.DeleteIngredient())

**24. Bonus: Working with NgRx in our Project**

358. First Summary & Clean Up

**NgRx CleaninUp:**

**shopping-list.service.ts** ფაილი შეგვიძლია ამოვშალოთ აპლიკაციიდან და შესაბამისად მისი იმპორტები და ინექშენები, რადგან აღნიშნული უკვე უსარგებლოა.

**24. Bonus: Working with NgRx in our Project**

359. One Root State

**NgRx another Store and different setup 1.1:**

ახლა უკვე სხვა მოდულისთვის შევქმნათ Store-ი. თუმცა მანამდე სასურველია ერთი რამ გავაკეთოთ:

ამჟამად **Reducer**-ების გამაერთიანებელი ინტერფეისი მხოლოდ ერთ **Reducer**-შია დეკლარირებული, რაც ართულებს სხვა **Store**-ებთან წვდომას.

ამის მოსაგვარებლად, ცალკე ფოლდერი უნდა შევქმნათ, სადაც საერთო **Store**იქნება. მისი კოდი იქნება:

**app.Reducer:**

1. import \* as fromShoppingList from '../shopping-list/store/shopping-list.reducer'
2. import \* as fromAuth from '../auth/store/auth.reducer'
3. import { ActionReducerMap } from '@ngrx/store'

6. export interface AppState {
7. shoppingList: fromShoppingList.State;
8. auth: fromAuth.State
9. }
11. export const appReducer: ActionReducerMap<AppState> = {
12. shoppingList: fromShoppingList.shoppingListReducer,
13. auth: fromAuth.authReducer
14. }

**24. Bonus: Working with NgRx in our Project**

359. One Root State

**NgRx another Store and different setup 1.2:**

ხოლო სადაც აღნიშნული **interface**იყო დეკლარირებული, იქიდან უნდა წავშალოთ.

ახლა უკვე, რამდენადაც **reducer**-ებიც გაერთიანდნენ ერთ ფუნქციაში და მათ ჩაეწოდათ **appState**-ების ინტერფეისი, შეგვიძლია **appModule**-ში **forRoot()** შევცვალოთ შემდეგნაირად:

1. import \* as fromApp from './store/app.reducer'
3. StoreModule.forRoot(fromApp.appReducer)

და ყველა კომპონენტში, სადაც აქამდე ტიპს ვაიმპორტებდით (კონსტრუქტორებში):

1. import \* as fromApp from '../store/app.reducer'
3. constructor(
4. private store: Store<fromApp.AppState>
5. ) {}

**24. Bonus: Working with NgRx in our Project**

360. Setting Up Auth Reducer & Actions

**NgRx another Store:**

როგორც წინაზე, აქაც უბრალოდ უნდა ჩავანაცვლოთ არსებული **Subject**და **Services**სთორები **Redux**-ის სთორით.

ანუ სადაც **.next()** მეთოდია მაგალითად გამოყენებული, უნდა ჩანაცვლდეს.

**24. Bonus: Working with NgRx in our Project**

362. Auth Finished (For Now...)

**NgRx call an reform from Store:**

თუ ჩვენი **Redux**სთრიდან რაიმეს გამოძახება გვსურს ან/და მოდიფიკაცია, შემდეგნაირად ვიზამთ:

1. constructor(
2. private store: Store<fromApp.AppState>
3. ) {}
5. user;
7. ngOnInit(){
8. this.user = this.store.select('user')// თუ შეცვლა გვსურს:
9. this.user = this.store.select('user').pipe(map(user => {
10. return user.user
11. })).subscribe()
12. }

**24. Bonus: Working with NgRx in our Project**

363. And Important Note on Actions

**NgRx Important Note:**

მნიშვნელოვანია ვიცოდეთ, რომ როდესაც **.dispatch()** მეთოდს ვიყენებთ, წვდომა მიდის ყველა **Reducer**-ზე, რომლებიც შექმნილი გვაქვს, ამიტომაც ყველა **Reducer**-ის **Switch**-ში სავალდებულოა გვქონდეს **Default**სტეიტმენტი, რომელიც მხოლოდ **state**-ს აბრუნებს. ასევე მნიშნვნელოვანია, რომ **actions.ts** ფაილში არსებულ ცვლადებში შეყვანილი მნიშნველობები (სტრინგები), სხვა **reducer**-ში არსებულს არ დაემთხვეს, ამიტომაც რეკომენდირებულია, რომ:

1. export const LOGOUT = '[Auth] Logout';
2. export const ADD\_INGREDIENT = '[Shopping List] Add Ingredient';

**24. Bonus: Working with NgRx in our Project**

364. Exploring NgRx Effects

**NgRx Effects**

რამდენადაც სტანდარტულ **store**-ში არაა რეკომენდირებული ასინქრონული (**http**) და სინქრონული, როგორიცაა **localStorage**-ის ჰენდლინგი, **NgRx**გვთავაზობს ამისთვის სპეციალი ბიბლიოთეკას, რომელსაც **CLI**-ში შემდეგნაირად ვაინსტალირებთ:

npm install --save @ngrx/effects

იგი გვაძლევს **Side Effects**-თან დილინგის თულებს.

**24. Bonus: Working with NgRx in our Project**

365. Defining the First Effect

**NgRx Effects Defining:**

მის ასამუშავებლად, შეგვიძლია იგივე ფოლდერში, სადაც reducer და actions არიან, შევქმნათ ახალი ფაილი:**auth.effects.ts**, რომელიც შეიცავს კლასს და ყველა საჭირო იმპორტს, მათ შორის **actions.ts**-ის იმპორტსაც, რამდენადაც იქ დავადეკლარირებთ კონკრეტულ მომქედებებს, რომლებსაც აქ გამოვიყენებთ:

**auth.effects.ts:**

1. import {Actions, ofType} from '@ngrx/effects'
2. import \* as AuthActions from './auth.actions'
4. export class AuthEffects {
5. authLogin = this.actions$.pipe(
6. ofType(AuthActions.LOGIN\_START)
7. )
8. constructor(private actions$: Actions){}
9. }

***actions$*** ეს უბრალოდ რეკომენდირებული სინტაქსია, მაინც ჩვენზეა დამოკიდებული, რას დავარქმევთ.

**LOGIN\_START** და სხვა მოქმედებები, ჩვეულებრივ, სტანდარტულად იქმნება **actions.ts** ფაილში, როგორც უბრალო ცვლადი:

1. export const LOGIN\_START = '[Auth] Login Start';

**24. Bonus: Working with NgRx in our Project**

366. Effects & Error Handling

**NgRx Effects next steps 1.1:**

მას შემდეგ, რაც ახალი ექშენი დადეკლარირდება**actions.ts**-ის ფაილში, ჩვენს ეფექტების ფაილს ვუბრუნდებით და ვიყენებთ **rxjs**-ის ოპერატორს - **switchMap**-ს, რომელიც საშუალებას გვაძლევს შევმნათ ახალი ობზერვებლი ძველი ობზერვებლის დეითაზე დაყრდნობით. მისი არგუმენტის ტიპი იქნება **actions**-ის შესაბამისი კლასი, ხოლო შემდეგ უბრალოდ რექვესტის ბრძანებას ჩავსვამთ კოდის ბლოკში:

1. authLogin = this.actions$.pipe(
2. ofType(authActions.LOGIN\_START),
3. switchMap((authData: authActions.LoginStart) => {
4. return this.http
5. .post<AuthResponseData>(
6. 'https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=' +
7. environment.firebaseAPIKey,
8. {
9. email: authData.payload.email,
10. password: authData.payload.password,
11. returnSecureToken: true
12. }
13. )
14. })
15. );

**24. Bonus: Working with NgRx in our Project**

366. Effects & Error Handling

**NgRx Effects next steps 1.2:**

თუმცა, მოცემული კოდი ჯერ არ იმუშავებს, რადგან ჯერ ჩვენ რედიუსერის სტეიტს არ ვეხებით.

შემდეგ მნიშნველოვანია ერორჰენდლინგი, თუმცა არა ძველებურად, ანუ ობზერვებლის ბოლოს, არამედ თავად ობზერვებლში უნდა მოხდეს ეგ შიგნით, ანუ სადაც რექვესტს ვაკეთებთ, რადგან ერორი მთლიან ობზერვებლს კლავს, ჩვენ შემთხვევაში კი არ შეიძლება ეს მოხდეს. ამიტომ, ერორ ჰენდლინგს თავად რექვესტს მივაბამთ, ხოლო შიგნით გამოვიყენებთ **of()** ოპერატორს, რომელიც იმპორტდება **rxjs**-დან და საშუალებას გვაძლევს ახალი ობზერვებლი შევქმნათ ერორის გარეშე:

1. return this.http
2. .post<AuthResponseData>(
3. 'https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=' +
4. environment.firebaseAPIKey,
5. {
6. email: authData.payload.email,
7. password: authData.payload.password,
8. returnSecureToken: true
9. }
10. ).pipe(catchError(error => {
11. //..
12. of()
13. }), map(resData=> {
14. of()
15. }))

**24. Bonus: Working with NgRx in our Project**

367. Login via NgRx Effects

**NgRx Effects next steps 1.3:**

შემდეგ უკვე, სასურველია ჯერ **map()** ეწეროს და მის მერე უკვე **catchError()**. მათ როცა ჩავანაცვლებთ, შემდგომ უკვე **map()**-ში გადმოვიტან ლოგინის ლოგიკას, ანუ დათას, რომელსაც ლოგინისას ვაწვდით:

1. map((resData) => {
2. const expirationDate = new Date(
3. new Date().getTime() + +resData.expiresIn \* 1000
4. );
5. return of(
6. new authActions.Login({
7. email: resData.email,
8. userId: resData.localId,
9. token: resData.idToken,
10. expirationDate: expirationDate,
11. })
12. );
13. })

და ასევე **@Injectable()**უნდა დავამატოთ კლასის ზევით და სტანდარტულად დავაიმპორტოთ.

ასევე **appModule**-შიც უნდა დავაიმპორტოთ ეფექტი შემდეგნაირად:

EffectsModule.forRoot([AuthEffects])

**@ngrx/effects**-დან.

**24. Bonus: Working with NgRx in our Project**

367. Login via NgRx Effects

**NgRx Effects next steps 1.4:**

შემდეგ უკვე იწყება ამ ეფექტის გამოძახება თავად იმ კომპონენტში, სადაც ეს მოქმედება ხორციელდება, ჩვენ შემთხვევაში **auth.component.ts-**ში.

პირველრიგში ვაინჯექტებთ და დავიმპორტებთ:

1. import { Store } from '@ngrx/store';
2. import \* as fromApp from '../store/app.reducer'
3. import \* as AuthActions from './store/auth.actions'
4. constructor(private store: Store<fromApp.AppState>){}

შემდეგ კი გაგზავნის ფუნქციაში:

1. this.store.dispatch(new AuthActions.LoginStart({email: email, password: password}))

აპი მაინც არ იმუშავებს ჯერ, თუმცა რექვესტი უკვე იგზავნება...

**24. Bonus: Working with NgRx in our Project**

368. Managing UI State in NgRx

**NgRx Effects next steps 1.5:**

შემდეგ ვქმნით ახალ ექშენს, რომელიც ლოგინის ფეილის შემთხვევაში მოხდება, **actions.ts**-ფაილში:

1. export const LOGIN\_FAIL = '[Auth] Login Fail';
2. export class LoginFail implements Action {
3. readonly type = LOGIN\_FAIL;
4. constructor(public payload: string){}
5. }

ხოლო**reducer.ts**-ში ვამატებთ:

1. export interface State {
2. user: User;
3. authError: string;
4. }
5. const initialState: State = {
6. user: null,
7. authError: null
8. };
10. //ახალი ქეისები:
11. case AuthActions.LOGIN\_START:
12. return {
13. ...state,
14. authErro: null
15. }
16. case AuthActions.LOGIN\_FAIL:
17. return {
18. ...state,
19. user: null,
20. authError: action.payload
21. }

**24. Bonus: Working with NgRx in our Project**

368. Managing UI State in NgRx

**NgRx Effects next steps 1.6:**

ახლა უკვე მის დასუბსქრაიბებას შევძლებთ**dispatch()**-ის ფაზაში, რათა დავაკვირდეთ პოტენციურ ერორებს.

შემდეგ ვამატებთ **reducer.ts**-ში ახალ სტეიტს: loading:boolean;

რომელიც **false**იქნება ინიციალ სტეიტში, ლოგინისას და ფეილისას, მხოლოდ ლოგინის დაწყებისას იქნება **true**;

შემდეგ **auth.component.ts**-შ ვამატებთ **ngOnInit()**-ს და მასში ვწერთ დანარჩენს:

1. ngOnInit(){
2. this.store.select('auth').subscribe(authState => {
3. this.isLoading = authState.loading;
4. this.error = authState.authError;
5. })
6. }

**24. Bonus: Working with NgRx in our Project**

369. Finishing the Login Effect

**NgRx Effects next steps 1.7:**

ახლა უკვე რედაირექშენს გავწერთ ლოგინის შემდეგ რომ ხდება:

იგი გაიწერება თავად **auth.effect.ts**-ში:

აღნიშნული მოქმედება დისპატჩს არ გამოიწვევს სხვა ეფექტებისგან განსხვავებით, ამიტომ **@Effect()**-ში უნდა ჩავაწოდოთ**{dispatch: false}** ობიექტი *(@Effect()-ს სხვანაირად გავწერ სამომავლოდ)*:

1. @Effect({dispatch: false})
2. authSuccess = this.actions$.pipe(
3. ofType(authActions.LOGIN),
4. tap(()=>{
5. this.router.navigate(['/']);
6. })
7. )

**შემდეგ ერორს ამოაგდებს, ამის გამოსასწორებლად ეფექტში of() უნდა მოვაშოროთ map()-ს.**

**24. Bonus: Working with NgRx in our Project**

369. Finishing the Login Effect

**NgRx Effects next steps 1.8:**

ამ სექციაში ერორ ჰენდლინგს გავწერთ.

ჯერ ჩვენს **auth.component.ts**-ის **ngOnInit()**-ის **select**-ის სუბსქრაიბში ვამატებთ:

1. ngOnInit(){
2. this.store.select('auth').subscribe(authState => {
3. this.isLoading = authState.loading;
4. this.error = authState.authError;
5. if(this.error){
6. this.showErrorAlert(this.error);
7. }
8. })
9. }

**24. Bonus: Working with NgRx in our Project**

369. Finishing the Login Effect

**NgRx Effects next steps 1.9:**

ხოლო თავად **ეფექტში**:

1. catchError((errorRes) => {
2. let errorMessage = 'An unknown error occurred!';
3. if (!errorRes.error || !errorRes.error.error) {
4. return of(new authActions.LoginFail(errorMessage));
5. }
7. switch (errorRes.error.error.message) {
8. case 'EMAIL\_EXISTS':
9. errorMessage = 'This email exists already';
10. break;
11. case 'EMAIL\_NOT\_FOUND':
12. errorMessage = 'This email does not exist.';
13. break;
14. case 'INVALID\_PASSWORD':
15. errorMessage = 'This password is not correct.';
16. break;
17. }
18. return of(new authActions.LoginFail(errorMessage));
19. })

**24. Bonus: Working with NgRx in our Project**

376. Using the Store Devtools

**NgRx DevTools:**

ქრომს (ასევე სხვა ბრაუზერებს) **Redux**-ის სამართავად უკეთეს *devTools*გვთავაზობენ, ამისთვის საჭიროა **Redux DevTools** ექსთენშენის დამატება:

*https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd/related?hl=en*

შემდეგ კი თავად **CLI**-ში უნდა დავაინსტალიროთ: npm install --save-dev @ngrx/store-devtools

შემდეგ **appModule**-ში ვამატებთ შემდეგ იმპორტს:

1. import { StoreDevtoolsModule } from '@ngrx/store-devtools'
2. import { environment } from '../environments/environment';
4. imports:[
5. StoreDevtoolsModule.instrument({logOnly: environment.production})
6. ]

შემდეგ კი, მაგალითად ქრომის დევთულსში **Redux**-ს გავხსნით, სადაც შეგვიძლია **Redux**-ის სთორის სტეიტებს დავაკვირდეთ.

**24. Bonus: Working with NgRx in our Project**

377. The Router Store

**NgRx Router-Store:**

მასაც **CLI**-დან ვაინსტალირებთ: npm install --save @ngrx/router-store

იგი გვეხმარება როუტინგის მოქმედებებზე რეაგირებაში (**Router**).

ამისთვის იგი **appModule**-შიც უნდა დავაიმპორტოთ შემდეგნაირად:

1. import { StoreRouterConnectingModule} from '@ngrx/router-store';
3. imports: [
4. StoreRouterConnectingModule.forRoot()
5. ]

*\*იგი დაამატებს ახალ მოქმედებებს****DevTools****-ში.*

**24. Bonus: Working with NgRx in our Project**

378. Getting Started with NgRx for Recipes

**NgRx For Recipes:**

აქედან იგივენაირად უნდა ავაწყოთ რეცეპტების მოდულისთვის **NgRx**სთორი...

**ოფიციალური დოკუმენტაცია:** https://ngrx.io/docs