300)Module Introduction

In this course we learned a lot about angular and also about state management. What is state? State simply defines state of your application. Is user authenticated? Are receipes loaded? this is state. So far we managed state by using services. We stored authentication state in services, we stored recipes in services. We used services and methods in services to interact with our state , to load more recipes, or add a new one. There is nothing wrong with this approach especially with our setup that we used here, where we also used Rxjs subjects as event emitters. So this setup is good , you can really use it.

However if you are diving into bigger applications or you just feel like you want a little bit of more elaborate approach possibly also using immutable structures. Then this section is for you.

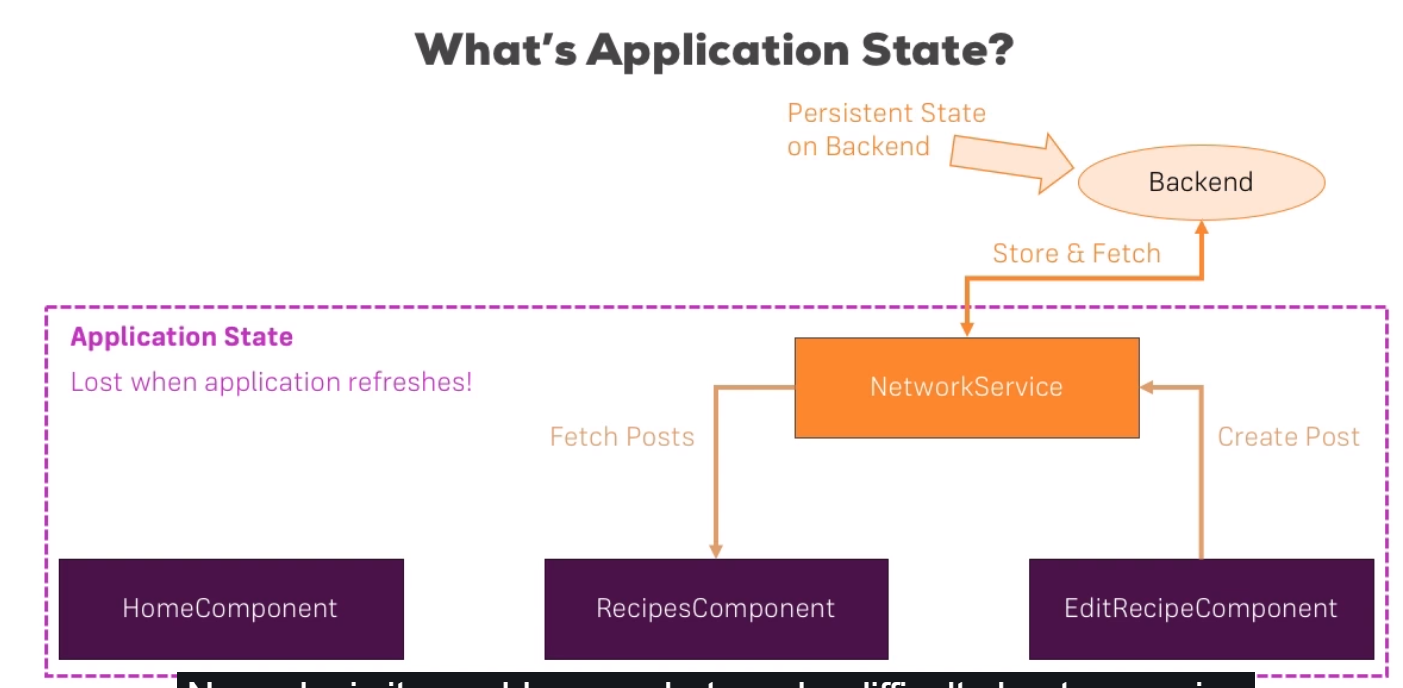
In this section we will dive into NgRx whch is additional package that you can use in your angular project. It is not part of angular. but you can install it. It’s all about state management and forcing a redux like pattern. If you are from react then you will know what redux, if not we will learn it here.

So in our project instead of using services for state management , now we will use NgRx.

302)State Challenges

Here we will understand what application state really is. What do we mean by this term. Lets consider this application which is close to our application, missing some components.

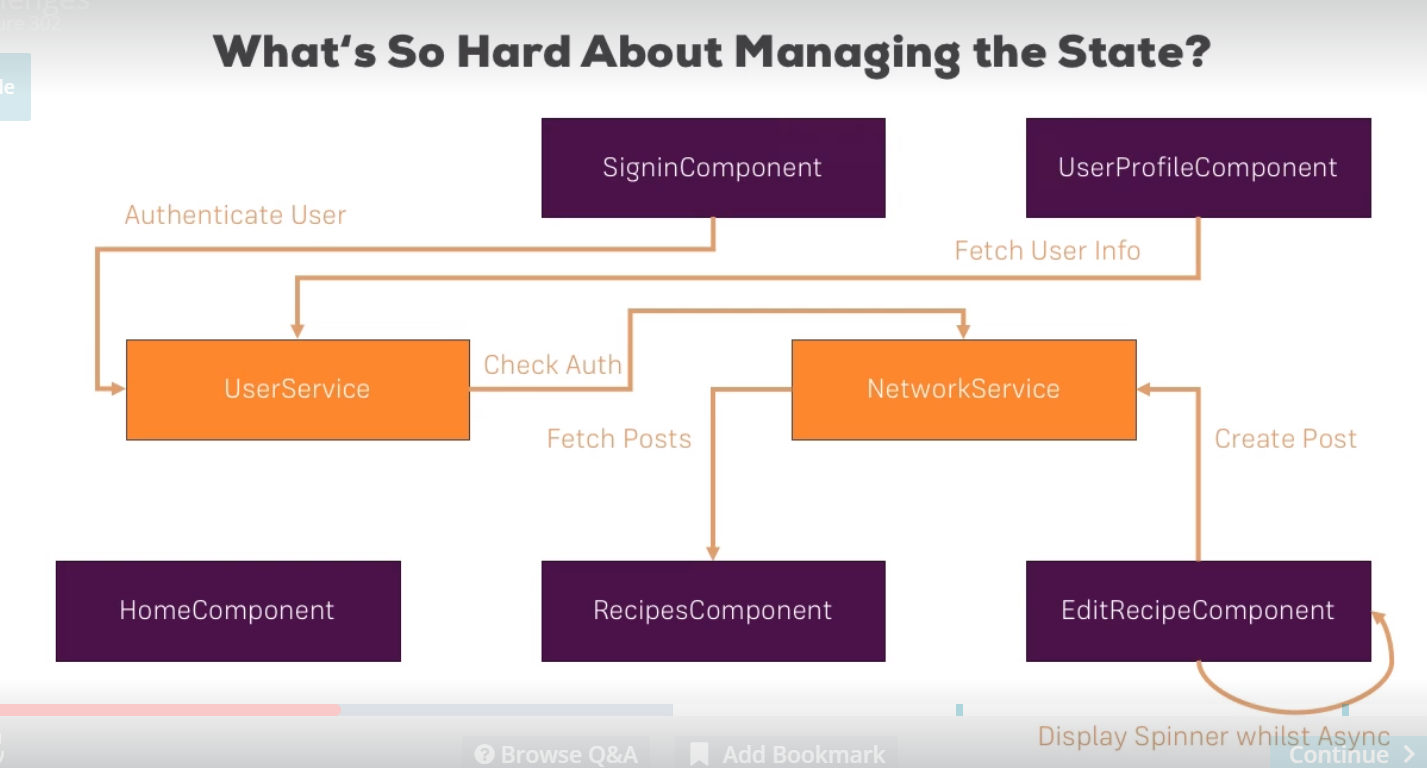
Network service is datastorage service in our application which connects us to to some backend. there may be more components and services in your application but lets stick to this basic one here.



Now we fetch posts in RecipesCoponent and create posts in EditRecipeComponent and everything that happens here is our application state. Its managed entirely thrugh angular app through javascript. Whenever we reload application, we get new index.html and we start from scratch. So all application state is lost when we refresh from application and that is how it is. You can ofcourse store parts of state in local storage if you want to persist it and check for existance whenever your application starts. But in general its’s lost and that is not a problem.

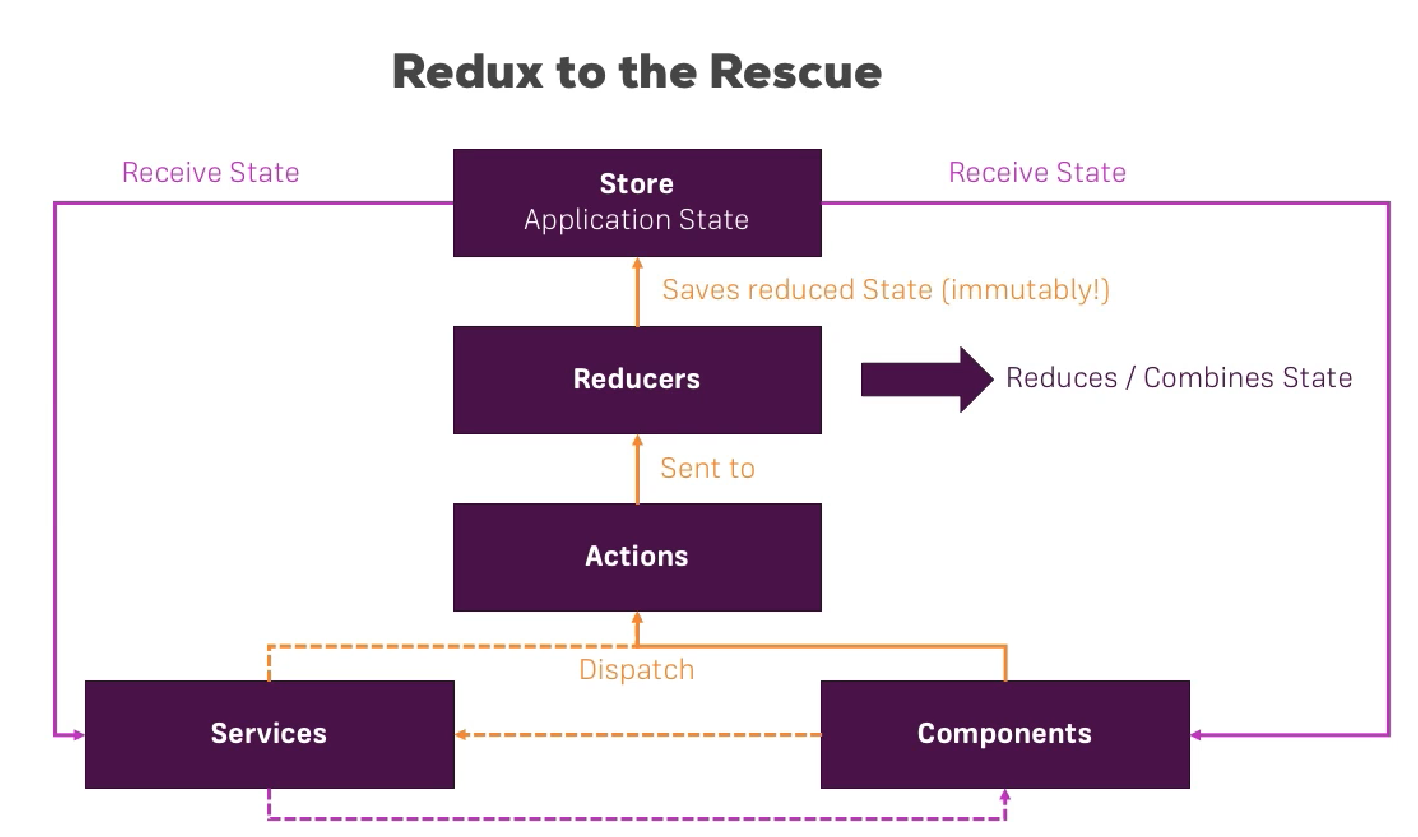
We also have backend from where we store and fetch data. In our app that is firebase. This is persiistent state. So that is state. State is general is describing what user sees and what user did. It keeps track of happened in our application and some things or some pieces of state are things that we want to store in backend. For example the recipes the user created. Other parts of state can be lost. So this is state in general.

Now why is it problem or what can be difficult about managing state. Consider this image-



This is just a random application. But you see there are lot of connections even through this is still a relatively simple application with just couple of components. But we got lot of things going on here and state of our application is actually edited or queried from all over our app and that can lead to issues bcoz as our application grows maintaining our app and always knowing where our state is edited and which service in our app can potentially override existing state can difficult. It can hard to keep track of where state gets changed and queried in our application. This is why we want to write applications where there are not too many places where we can edit our state and we actually already did this in our course project. Lets have a look at it. Lets have a look at it. In our course project there is one thing that we already have, we are using services and in some of these services like recipes service we are using subjcts, so basically event emitters to inform our application whenever something changed. By this pattern, by using subject in service we ensure that we actually only edit our state(in this case recipes) inside of this service and we inform rest of this application about this chage. The alternative would habe been to edit the recipes wherever we will find it appropriate, for ex in recipe –edit component we don’t have to reach out to service there . we could have just edited the recipe array there and then go from there. Using service as a central place is a good practice and using a subject to inform the rest of the app about changes so that we can update the UI is also a very good practice.so we are already doing many things right. That is important takeaway, you dnt need to optimize the state handling in an applications like this if you are already using services and subjects, if it is not getting a problem for you because these applications to be honest isn’t super big. So for application of this size there might not be a need to improve the way we handle our state. As your application grows and grows, even this service driven combined with subject approach might get hard to maintain through. Now to avoid running into issues or applications which are harder to maintain as they grow, you can use a different pattern, whoch is pretty common in react community- redux.

Lets have a look at how would you manage the application state when using redux.



In redux approach, you have one central store in your application, so one central place where you manage your application state. Right now keep in mind with multiple services right now keep in mind with multiple services, so you go back to one central place to really ensure that there, well is only one place where your state lives ,this single source of truth. Your components and services, (you might still have services which for example centralize some methods you use for multiple components), so these 2 pieces still have access to this store and also to each other. so they receive the state from store but ofcourse you can still communicate between components and services or between components and components. This is not interrupted, we just change the way we store our store. So that is one thing. However there’s also another important thing we have a way of accessing our store and state but how do we change it. We dispatch so called actions for that. Actions are clearly defined in this approach. For example we could have action which is save post. This action would next reach so called reducer. Now reducer are functions , you write yourself, which take action and a potential payload as an input and then do something with it to manipulate the original state and they do this in immutable way which means they don’t edit the old store they overwrite it with new one, by simply taking the old one and then editing something but in new javascript object and then saving it. This is redux setup. actions, reducer and store are keything here. We can use this in angular app too, it is not exclusive to react. There is special implementation for angular called NgRx which basically adapts this redux approach to fit really nice into an angular app, even supporting features like lazy loading where you might add things to your single store only once certain module has been loaded.

303)Starting with reducers

Lets start adding ngrx and I dnt want to start in recipe component because we are loading this component lazily.that is something that I want to keep for later. We want to start with shopping list component. We are already using shoppinglist service and in this this service we use subject. We are already use good pattern but we want to replace it with ngrx. Run this command-

**npm install –save @ngrx/store**

this imports main package of ngrx eco system. We will learn about other packages later in this module. So this install @ngrx/store. What do we do next?

We need so called reducers to edit the state and we need this store. Store is provided when we got some reducers. So lets start with them. We will add shopping list realted reducers in shopping-list folder. So we add new file- shopping-list.reducers.ts. in this file I want to export a function. make sure you use fuction keyword and not es6 arrow function syntax. That wnt work. We can name this function anything we name it **shoppingListReducer** to fit file name.This function receives 2 arguments , these will be automatically passed by ngRx. I will show you how to dispatch an action to trigger reducer throughout this module.

That is important takeaway- a reducer function will be triggered whenever action is dispatched. So we get 2 things in reducer, state(current state of application and keep in mind this is passed automatically by ngrx) and we get an action, this is action that we dispatched. I will show you how to displatch a action. Now the state is current state of application, the first time this runs through(reducer) there will be no current state. Which is why we cn define some initial state , we name it initialState. For indregriends this will actually be an object ,where if we have a look at shopping-list service, I want to take this array of ingredonets-

private ingredients: Ingredient[] = [

new Ingredient('Apples', 5),

new Ingredient('Tomatoes', 10),

];

This shall be my initial state for ingredients property. Back to our reducer we add property named ingredients to our initialState object.

with this we have initial state which is an object with ingredients property, which is an array of ingredients, just like service.

State then get default value of initial state, which means if state is not present we will use **initialState** which is important for first time we dispatch an action, when we reached this reducer without having current state.

Shopping-list.reducers.ts-

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

import {Ingredient} from '../shared/ingredient.model';

const initialState = {

ingredients: [

new Ingredient('Apples', 5),

new Ingredient('Tomatoes', 10),

]

};

export function shoppingListReducer(state = initialState, action: Action) {}

now inside this reducer there is one task we have to do, we have to return an updated version of our state or s=actually not even updated one. We have to return state of our application , the new state. Because remember what I said when we walked through slide, I told you that reducers update our satte not be editing it but by simply setting a new one. Now simplest way of doing it is to simply return state and behind the scene, ngrx will replace old state with new one, even though in this case they are equal. Code-

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

import {Ingredient} from '../shared/ingredient.model';

const initialState = {

ingredients: [

new Ingredient('Apples', 5),

new Ingredient('Tomatoes', 10),

]

};

export function shoppingListReducer(state = initialState, action: Action) {

return state;

}

Now typically you want to do more than just returning the unchanged state ofcourse. And for that you typically use a switch statement to determine which kind of action was dispatched because we get acton as second argument. But ofcourse there could be multiple actions which can get displatched.

So lets switch action tyoe, type is property provided on action object. So we switch action type, now we want to check different cases. Now what is value of type though. We can actually set this when we dispatch an action. You typically use string describing the action. And to make sure that you do not mistype anywhere in your app, you typically also store this in constant. We export this constant because later I also want to use it for other files.

Here we can simply have a look at our service to find out which actions we have. Open shoppig-list.serive.ts. getting is special because we dnt change state here. So we ignore these functions-

getIngredients() {

return this.ingredients.slice();

}

getIngredient(index: number) {

return this.ingredients[index];

}

So lets go with add now,add ingredient-

addIngredient(ingredient: Ingredient) {

this.ingredients.push(ingredient);

this.ingredientsChanged.next(this.ingredients.slice());

}

For this we define this constant in our reducer-

export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';

this is for type of action. we talked about defining constants for that. In our function body, in switch, we check whether case is ADD\_INGREDIENT. In that case I want to edit my state by adding ingredient. I can do that by returning a new object because that should be immutable. Taking all the old state parameters. We use spread operator. It is simply expanding the old sate object so that all properties of that object are added to new object that we are returning here. Now I want to overwrite one single property – ingredients property, this shall now be array where we also have this new ingredient being added to it. This is something that we can easily do, again by using spread operator, to also edit this ingredient array (which would also be a refrence type) in imuutable way. Code-

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

import {Ingredient} from '../shared/ingredient.model';

export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';

const initialState = {

ingredients: [

new Ingredient('Apples', 5),

new Ingredient('Tomatoes', 10),

]

};

export function shoppingListReducer(state = initialState, action: Action) {

switch(action.type) {

case ADD\_INGREDIENT:

return {

...state,

ingredients: [...state.ingredients, action.]

}

}

return state;

}

So now this will distribute all elements in this array over this newly created array. Then we add new element. this new element should be one that we receive as a part of action. Now how to retrieve this elemnt.

We can use action. ? what after action.

We want to retrieve some payload from action.because we expect when we add new ingredient, we dispatch an action with some extra information namely new ingredient to be added. By default actions in ngrx are payloadless, so they do not have one. They only have type property, we can also see it by autocompletion. Now to get this extra information we have to define our own clearly typed actions. We will do it next lecture.

304)Adding Actions

In last lecture we are working on reducer but we were stuck when we wanted to retrieve the data, we probably passed with action, once we dispatch it.for this we will add new file in shopping-list folder- shopping-lists.actions.ts . to keep it more organized we added new folder in shopping-list folder. Then we moved our dispatcher and action files in it. We also have to adjust the import of ingredient model in reducer.

In actions file we move constant from reducer to action.

export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';

in actions file we export class that implements Action. Since we implement Action we are forced to define a readonly property , type in class. Value of this type is const that we already define din this class. Now our problem was that our Action should payload, an additional data that I can pass with action, for this we have to add it manually because evry action might not requie payload and ngrx does’nt want to force me to have one. If you do need one like we do here, you can simple add it by adding a another property to this class. I will name it payload. It should be of type ingredient, because that is what we want to send along. Now we have payload property and I am exporting this class.

Finally I want to bundle all actions I set up in this file (right now it’s one but will become more) in one single export type. Type is ts feature that we can use, we can define our own type. In this case it will be ShoppingListActions type. That sound like fitting name. this will be useful wonce we start adding more actions.

Shopping-lists.reducers.ts-

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

import { Ingredient } from '../../shared/ingredient.model';

export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';

export class AddIngredient implements Action {

readonly type = ADD\_INGREDIENT;

payload: Ingredient

}

export type ShoppingListActions = AddIngredient;

now lets work on reducer.

305)Finishng the Reducer

So we created action with payoad, now in reducer we can take advantage of that, by being clear about type of our action. It’s no longer of type Action from@ngrx/store. We will use our own action.

I import everything that I export in my actions file. we store it in **ShoppingListActions** variable basically. Import \* will bundle all exports from that file in one js object.

import \* as ShoppingListActons from './shopping-lists.actions';

then we define type of our actions-

action: ShoppingListActons.ShoppingListActions

first Shoppinglistactions is js objected in which imported al exports from our actions file. second one is type that we exported from actions file.

then we use payload property of action to add new ingredient in our array.

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

import \* as ShoppingListActons from './shopping-lists.actions';

import {Ingredient} from '../../shared/ingredient.model';

const initialState = {

ingredients: [

new Ingredient('Apples', 5),

new Ingredient('Tomatoes', 10),

]

};

export function shoppingListReducer(state = initialState, action: ShoppingListActons.ShoppingListActions) {

switch(action.type) {

case ShoppingListActons.ADD\_INGREDIENT:

return {

...state,

ingredients: [...state.ingredients, action.payload]

}

}

return state;

}

Now that was quite complicated , just to add new ingredient to array. but cool thing about using reducers and ngx in general is once you got basic setup(we are still in that part unfortunately), it’s super easy to work with it and you can quickly change and adjust stuff. With that we got our reducer almost finished, we will add another case here. Default case. Here we want to return just state. Now with that we got reducer finished, we got some actions. Now its time to register this reduced in our app, so that we can actually use it. And its also time to make sure or to also add possibility of dispatching the action.

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

import \* as ShoppingListActons from './shopping-lists.actions';

import {Ingredient} from '../../shared/ingredient.model';

const initialState = {

ingredients: [

new Ingredient('Apples', 5),

new Ingredient('Tomatoes', 10),

]

};

export function shoppingListReducer(state = initialState, action: ShoppingListActons.ShoppingListActions) {

switch(action.type) {

case ShoppingListActons.ADD\_INGREDIENT:

return {

...state,

ingredients: [...state.ingredients, action.payload]

}

default:

return state;

}

return state;

}

306)Registering the Application store

Lets provide this reducer in our application so that we have a store. Right now we don’nt have. So lets register 2 files that we have created. We do that in app.module, for this eagerly loaded module., which shopping list is. We have to import **StoreModule,** this a module. Now adding is alone does’nt do it. We have to call forRoot method there because it is for our main application, for eagerly loaded module not lazily loaded one. forRoot takes one argument, a js object where you now simply add all your different reducers.

In this case we have shopping list part of our application and I want to point to shopping listReducer(name of reducer function).

So this line-

StoreModule.forRoot({shoppingList: shoppingListReducer})

Registers our reducer.

Code-

import { BrowserModule } from '@angular/platform-browser';

import { HttpClientModule } from '@angular/common/http';

import { NgModule } from '@angular/core';

import {StoreModule} from '@ngrx/store';

import { AppComponent } from './app.component';

import { AppRoutingModule } from './app-routing.module';

import { SharedModule } from './shared/shared.module';

import { ShoppingListModule } from './shopping-list/shopping-list.module';

import { AuthModule } from './auth/auth.module'

import { CoreModule } from './core/core.module';

import { shoppingListReducer } from './shopping-list/store/shopping-list.reducers';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

HttpClientModule,

AppRoutingModule,

SharedModule,

ShoppingListModule,

AuthModule,

CoreModule,

StoreModule.forRoot({shoppingList: shoppingListReducer})

],

bootstrap: [AppComponent]

})

export class AppModule { }

by this one line-

StoreModule.forRoot({shoppingList: shoppingListReducer})

Ngrx will setup a store, register shoppinglistreducer as one thing that can edit the store and initial state as one piece of overall application state.

const initialState = {

ingredients: [

new Ingredient('Apples', 5),

new Ingredient('Tomatoes', 10),

]

};

Now lets see if that worked by getting that state(initialState constant) and dispatching an action.

307)Selecting Data from State

So we registered our store , now we want to get our state. In shopping-list.service.ts we have a method , where we return copy of our state.

getIngredients() {

return this.ingredients.slice();

}

Well lets now no longer use internal ingredients array(in service) but out global store for that. for that I go straight, where I want to use that information, shopping-list.compoent. this is where I use ingredients in the end, right. Since I use them there I can also fetch them there. For that we inject a service store of type Store in this component, because I want to get access to my store which holds my state, right. The store is generic type, so we have to tell which type of data we want to retrieve here. The type of pass here actually needs to fit your global state. Now we define state here in app.module where we set up a store.

StoreModule.forRoot({shoppingList: shoppingListReducer})

We will have **shoppingList** part of the state. So lets add shoppinglist to state object that we are describing in component. And this in turn will a js object where we will have ingredients array with ingredient model. This is our global state and I will come bck to ways of managing this state a bit more elegantly later. For now that is our state. Code of component-

import { Component, OnInit, OnDestroy } from '@angular/core';

import { Subscription } from 'rxjs/Subscription';

import { Ingredient } from '../shared/ingredient.model';

import { ShoppingListService } from './shopping-list.service';

import { Store } from '@ngrx/store';

@Component({

selector: 'app-shopping-list',

templateUrl: './shopping-list.component.html',

styleUrls: ['./shopping-list.component.css']

})

export class ShoppingListComponent implements OnInit, OnDestroy {

ingredients: Ingredient[];

private subscription: Subscription;

constructor(private slService: ShoppingListService,

private store:Store<{shoppingList:

{ingredients: Ingredient[]}}>) { }

ngOnInit() {

this.ingredients = this.slService.getIngredients();

this.subscription = this.slService.ingredientsChanged

.subscribe(

(ingredients: Ingredient[]) => {

this.ingredients = ingredients;

}

);

}

onEditItem(index: number) {

this.slService.startedEditing.next(index);

}

ngOnDestroy() {

this.subscription.unsubscribe();

}

}

Now here we can query this state, by reaching out to store and calling select. Now select, selects slice of our global state and in this case I am interested in shoppingList.

So instead of-

this.ingredients = this.slService.getIngredients();

we do-

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

so here I am selecting this(shoppingList) part of state. However we now have an issue, this now is an observable. it’s not array of items, it observable so above line won’t work. We have to change the type of variable ingredients of component., observable.

Now observable again is generic type and this it will eventually resolve to ingredients, you could say, but that will be wrong. It will reolsve to object, which has ingredients property which then holds then ingredients array. So this is what observable will eventuallt reolsve to.

ingredients: Ingredient[];

we change it to-

so its an observable