Angular 4 state management with usage of @ngrx/store libary

Michał Czarnecki

2017-09-10

1 Introduction

Having a knowledge about the state of the application is power. Information about current state and changes that are happening is necessary to be sure that everything is working correctly. In the pure angular it is not so easy to keep track of it. You can try different methods like logging, keeping crucial data in shared services or even making additional calls to the back end of the application. Unfortunately none of these are perfect and can bring chaos and unnecessary complexity to the project. That's why we want to use custom libraries to simplify the process. The purpose of this document is to explain how to use ngrx store management tool to keep track of the state and changes made in the application. It will provide basic knowledge how to install and how to develop application with it.

During this tutorial you will be creating simple view of members management for abstract organization. This will include displaying the list of members and creating or editing them with the usage of ngrx store.

Finished code can be found on my Github, although I strongly recommend for you to walk through this tutorial step by step:

```
https://github.com/MCzarnecki/StoreManagement.
```

2 Prerequisites

In order to write application you will need to have following software installed:

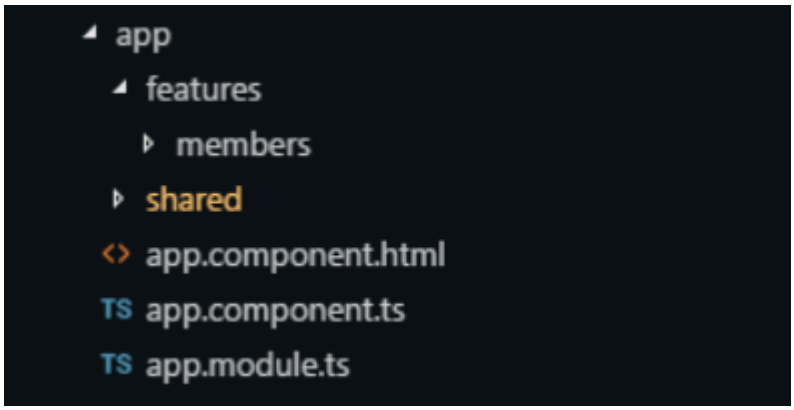
1. NodeJS and npm package manager
2. Angular cli
3. IDE for angular developement (Visual Code, Webstorm etc...)
4. Redux DevTools - extension in the chrome browser

NodeJS can be installed directly from their website and the use npm (or yarn) to install Angular Cli. If it comes to IDE, use whatever you like the most. When you have everything installed we can proceed with the next step.

3 Project setup

Use angular cli tool to create the new project. It will be our base for all work that will be accomplished. Run it to check if everything is working correctly.

It's a good practice to keep our code organized so let's prepare out package structure. In the app folder create two additional directories: shared and features. In most of the times we would also create core directory but since we will not have any core business logic we can skip that part. Now let's create additional package in the app/features - name it *members*. Now our structure should look like this:



Let's make some final preparation. Create two module files: in app/shared *shared.module.ts* and in app/features/members *members.module.ts*. Make SharedModule export *CommonModule*, *RouterModule* and *ReactiveFormsModule*. Import this module to all modules in the app. MembersModule for now should just be declared and imported into AppModule. That's all for the initial preparations.

It's time to install the rest of the required dependencies. Let's start with including bootstrap for styling the application. Type the following command:

```
npm install bootstrap@4.0.0-alpha.6 --save
```

After installation is complete we need to include it in the project. Open *.angular-cli.json* and paste the following path into the styles array:

```
"../node_modules/bootstrap/dist/css/bootstrap.min.css",
```

All that is left to do is to install our state management library. To do so write the following command in the terminal:

```
npm install @ngrx/store @ngrx/effects
@ngrx/store-devtools @ngrx/router-store --save
```

Let's quickly check what's the main purpose of each dependency:

1. Store - the root of the ngrx - contains the logic about state management and basic types which are used in ngrx,
2. Effects - provides possibilities to create effects when the actions are dispatched like loading data from the service or redirecting,
3. Store-devtools - enabling checking the state with ReduxTool
4. router-store - keeping information about the route changes

Before we jump into using those dependencies let's just make some final preparations in the project structure. We have two missing things: routing configuration and backend connection. Let's start with initial routes for the members module. For now it will only include the view for collection of the members. Create *members-routing.module.ts* and import it in the *MembersModule*. Listing below shows how it should look like:

```
import { MembersPreviewComponent } from
'./containers/members-preview/members-preview.component';
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';

const MEMBERS_ROUTES = [
       { path: '', component: MembersPreviewComponent, pathMatch:
           ↪  'full' }
];

@NgModule({
       imports: [RouterModule.forChild(MEMBERS_ROUTES)]
})
export class MembersRoutingModule { }
```

Now we need to declare *MembersPreviewComponent* which will be repsonsible for containing the list of memebers. Create *containers* directory in the *members* and using cli generate the component.

Let's jump to main routes. Create *app-routing.module.ts* and include it in *AppModule*. It should contain following code:

```
import { PageNotFoundComponent } from './shared/components/page-
    ↪ not-found/page-not-found.component';
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';

const APP_ROUTES = [
       { path: '', redirectTo: 'members', pathMatch: 'full' },
       {
         path: 'members',
         loadChildren: './features/members/members.module#
             ↪ MembersModule'
       },
       { path: '**', component: PageNotFoundComponent }
]

@NgModule({
       imports: [RouterModule.forRoot(APP_ROUTES)]
})
export class AppRoutingModule { }
```

Create *PageNotFoundComponent* in SharedModule. Remember to export it!

The last thing we need before we start using state management is a functionality which will provide us members data. For that we will create fake backend provider. Let's start with creating model which will represent members in the application. In the *members* directory create package *models* and inside *member.ts* file. Write down the following definition:

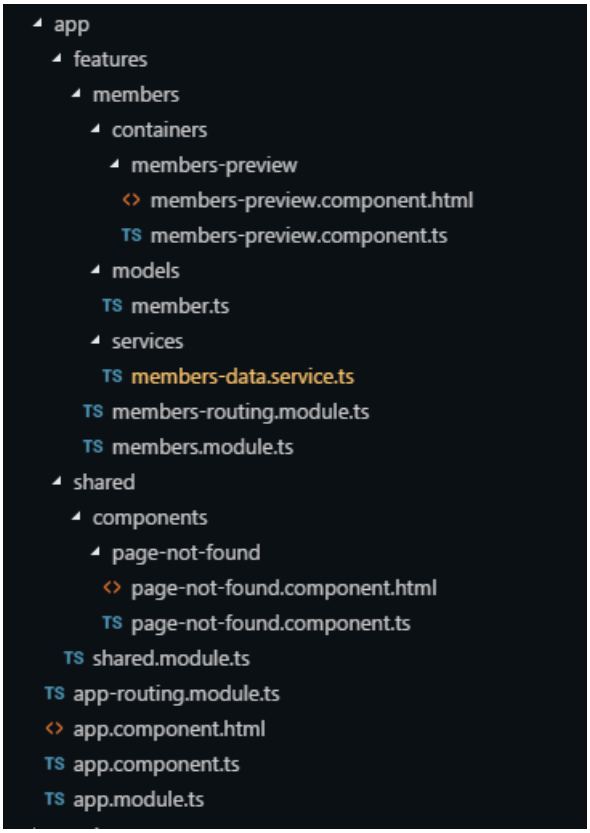
```
export interface Member {
       id: number;
       name: string;
       surname : string;
       status: string;
       address: string;
}
```

Now we just need the service which will provide some initial data and enable us to save additional members and edit existing ones. In *members* create directory *services* and inside *members-data.service.ts*. Write down the code:

```
import { Injectable } from '@angular/core/';
import { Member } from '../models/member';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/of';

// DO NOT DO THIS AT HOME :)
@Injectable()
export class MembersDataService {

	members: Member[] = [
	{ id: 1, name: 'Jake', surname: 'Peralta',
		status: 'Policeman', address: 'Brooklyn' },
	{ id: 2, name: 'Mathew', surname: 'Murdock',
		status: 'Vigilante', address: 'Hells Kitchen' },
	{ id: 3, name: 'Daenerys', surname: 'Targaryen',
		status: 'Stormborn, the Unburnt etc...', address: '
		    ↪ Westeros' },
	];

	addMember(member: Member) {
		const ids = this.members.map(member => member.id);
		const newId = Math.max.apply(Math, ids) + 1;
		this.members.push({...member, id: newId});
	}

	editMember(member: Member) {
		const index = this.members.map(m => m.id).indexOf(
		    ↪ member.id);
		this.members[index] = member;
	}


	getMembers() {
		return Observable.of(this.members);
	}
}
```

Remember to provide it in *MembersModule*. You can play now with this service a little to check if it works correctly. Next we will start looking at state management.

Your package structure should look like this now:



4 @ngrx/store

4.1 State, Action and Reducer

There's always a lot of things going on in the applications. Variables change, events are being sent, data is being loaded from the backend and so on. Single login data can determine what we can and what we cannot do. There is a need to keep crucial data somewhere in the application like session storage or shared services. However, this approach has many flaws. It's almost impossible to check the state of all data, there is no memory of changes, code starts to contain a lot of logic and becomes unreadable and hard to test. This is where the idea of the store comes with help. So what exactly is it? Let's look at official description:

```
@ngrx/store is a controlled state container designed to help
write performant, consistent applications on top of Angular
```

This description does not need additional explanation. There's only one question here: What exactly is the State? To keep it simple: it's a data structure, usually interface, containing information about specified application data parts. It's really simple: let's consider the example of an (Volvo :))employee. The employee has a role (Tester, developer, team leader etc.), the unique identifier and for instance Salary (unfortunately usually too small) :). If we represent him / her as a state it would be something like this:

```
export interface Employee {
       id: string;
       role: string;
       salary: number;
}
```

Of course no one would like to stay at the same posistion and have constant salary. So due to ACTIONS in the company, what we are doing can change and it affects how much we earn. For instance payrise would change the state of our salary. That's all the theory about states :)

I highlighted one word in my explanation: Action. Actions are inseparate parts of the Store. They describe the changes and they often also forward changed data to the store. In application those are classes which implement the Action interface. They always need to have type variable (which is unique identifier of the action) and optionally they can carry a payload. Let's look at payrise action of the employee:

```
export const PAY_RISE = '[Employee] Worked hard and now earns
    ↪ millions.'

export class PayriseAction implements Action {

       readonly type = PAY_RISE;

       // payload represents new salary
       constructor(public payload: number) { }

}

export type Actions = PayriseAction;
```

So what happens when the store sees the action has happened? The state and action are being taken and then the new state is being created. This is done by the function called REDUCER. Reducer should always create new state and newer mutate (change) the old one. Imagine the situation where you receive pay rise and instead of returning new State you mutate the old one. Suddenly the information about how much you earned would be gone. It would be replaced by the current state there would be no record of it. That's why we always want to create new states.

We now know to never mutate the state of the application, so let's check how the reducer looks like:

```
export funtion reducer(state: State = initialState, action:
    ↪ Action) {

        switch(action.type) {

                case PAY_RISE: {
                        return {... state, salary: action.payload};
                }

                default: {
                        return state;
                }

        }

}
```

As you can see reducers are really simple!

We've learned about principles of the ngrx store. We know about the basic flow of data (state - action - reducer - new state). One can have another important question: If we declare such a small state, what about other data in store? Of course the Employee would be just a part of the application root State. Store consists of many states, each consisting having corresponding reducer with the unique identifier. We will later learn how to create them but to simply say: we just need to inform the Store about the pair reducer-state and it will take care of saving it.

There is one very important aspect of using a store. If we want to check the value of part of the state we will not retrieve Objects. We will get OBSERVABLES. Their value can change with every action so it is a necessity to listen for changes. Whole idea of ngrx/store is based on Rx.BehaviourSubjects but this is not a topic of this training. For better understaning how it works underhood I

encourage you to read the following article:
https://gist.github.com/btroncone/a6e4347326749f938510

4.2 Effects

There is one thing which we still need to make use of the Store to the fullest. As you may have noticed, there is no place for logic and operations connected to actions (routing, getting data etc) in the Reducer. Imagine following scenario: You are trying to log in to the application. You write username, password and then the action is being dispatched to the store. Of course Reducer will catch it and change the state but what about handling the success and failure? What about the fact that this action should start auth event? Those are the site effects of the dispatched actions and we already mentioned the tool which handles them: Effects.This library works similar way to the interceptors. Effect listen for actions and then they run some logic based on caught one. Let's look at the example:

```
@Injectable()
export class AuthEffects {

       constructor(
              private actions$: Actions,
              private authService: AuthService,
              private router: Router
       ) {}

       @Effect()
       login$ = this.actions$
              .ofType(Auth.LOGIN)
              .map((action: Auth.Login) => action.payload)
              .exhaustMap(auth => this.authService.login(auth)
              .map(user => new Auth.LoginSuccess({ user }))
              .catch(error => of(new Auth.LoginFailure(error)))
       );

       @Effect({ dispatch: false })
       loginSuccess$ = this.actions$
              .ofType(Auth.LOGIN_SUCCESS)
              .do(() => this.router.navigate(['/']));

       @Effect({ dispatch: false })
       loginRedirect$ = this.actions$
              .ofType(Auth.LOGIN_REDIRECT, Auth.LOGOUT)
              .do(authed => {
                     this.router.navigate(['/login']);
```

```
      });
}
```

Let's analyze what is happening here. There is declaration of Actions in the constructor. It provides the methods enabling us to recognize which type of action has been dispatched. If the type is caught then the logic will be run. Effects will be marked with @Effect() decorator. This annotation will start the chain of actions. The effect will always work for actions specified in ofType([]) method and will always return some action. If there is no return action specified, the method will dispatch the caught one back. Since every dispatched action is being handled by default this may cause some nasty loops. To deal with it we can inform that the effect return action will not be dispatched by adding metadata to annotation: *@Effect(dispatch: false)*. In the chain you can do many standard RX operations, like map, mergeMap, do, etc. What we can do in the chain will not be subject of this tutorial so if you want to know more about possible actions then you can check it for instance on those pages:

```
https://www.learnrxjs.io/
http://rxmarbles.com/
```

Analise the example and see how the new actions are being dispatched and what is happening afterwards.

4.3 Including ngrx in the application

We finally have some basic knowledge about the Store (and it took only 10 pages to learn that!) so we can include it now in the application. There are four modules which will be imported in the application:

1. StoreModule - responsible for registering reducers in the application. The core of store application. Provides us the store,
2. EffectsModule - responsible for registering effects,
3. StoreRouterConnectingModule - keeps router state up-to-date in the store,
4. StoreDevtoolsModule - instrument the store retaining past versions of state and recalculating new states. This enables powerful time-travel debugging.

Let's include them in the AppModule:

```
import { StoreModule } from '@ngrx/store';
import { StoreDevtoolsModule } from '@ngrx/store-devtools';
import { StoreRouterConnectingModule } from '@ngrx/router-store';
import { EffectsModule } from '@ngrx/effects';

// Other imports and logic
```

```
imports: [
       BrowserModule,
       AppRoutingModule,
       SharedModule,
       MembersModule,
       StoreModule.forRoot([]),
       EffectsModule.forRoot([]),
       !environment.production ?
              StoreDevtoolsModule.instrument() : [],
       StoreRouterConnectingModule
],
```

For now we don't have any reducers and effects so we initialize the modules with empty arrays. We usually don't want to use Redux DevTools on production to increase efficency so we only use it on non production environments.

We are almost ready to write our first reducer. Let's start by making the root app state which will include routing state changes. We need to begin with writing our own custom RouterStateSerializer. This is a necessity in the latest version of the Store due to the problem with enormous amount of information provided to us during routing actions. Each of those will provide us with 160 000 lines long json!! We can't keep so much information because it would immediately slow the app.

In the SharedModule create new direcotry named utils and then make file *router-state.ts*. Paste the following code inside:

```
import { RouterStateSerializer } from '@ngrx/router-store';
import { RouterStateSnapshot, Params } from '@angular/router';

/**
* The RouterStateSerializer takes the current
* RouterStateSnapshot * and returns any pertinent
* information needed. The snapshot contains
* all information about the state of the router at the given
    ↪ point in time.
* The entire snapshot is complex and not always needed. In this
    ↪ case, you only
* need the URL and query parameters from the snapshot in the
    ↪ store. Other
* items could be returned such as route parameters and static
    ↪ route data.
*/

export interface RouterStateUrl {
       url: string;
```

```
        queryParams: Params;
}

export class CustomRouterStateSerializer
        implements RouterStateSerializer<RouterStateUrl> {
        serialize(routerState: RouterStateSnapshot):
            ↪ RouterStateUrl {
        const { url } = routerState;
        const queryParams = routerState.root.queryParams;

        return { url, queryParams }; }}
```

With this we will receive only the url and query params of the routing actions. Let's now provide it in AppModule (NOT IN SHARED! (Cause it would not work)):

```
providers: [{ provide: RouterStateSerializer,
        useClass: CustomRouterStateSerializer },],
```

Let's create our first reducer which will override default routing one and use our custom route state serializer. Create the *reducers* directory in the src/app and inside make *root.reducer.ts*. Write down the following code:

```
import { RouterStateUrl } from '../shared/utils/router-state';
import { ActionReducerMap } from '@ngrx/store';

import * as fromRouter from '@ngrx/router-store';

export interface State {
        routerReducer: fromRouter.RouterReducerState<
            ↪ RouterStateUrl>;
}

export const reducers: ActionReducerMap<State> = {
        routerReducer: fromRouter.routerReducer,
};
```

We always start writing reducers by declaring the State on which it will work, here we do this by including the router reducer state as part of our root one. This way we can combine multiple reducers (and states) into one object which will later be declared in the module. In most cases our states will be similar to the one from earlier example (Employee). We can either provide them right into module or combine them into one state ourselves. Next we declare reducers Map. It must much each state with the corresponding reducer. The name of the reducer must be the same as the one from state!!! (Like in this example).

Now let's include the reducers in the AppModule:

```
StoreModule.forRoot(reducers),
```

Now open the Redux DevTools and enjoy the view of working state management :)

5 Members feature development

5.1 Members list

Let's start with creating controller responsible for displaying the list of members. Create *components* directory in *members* and generate members-list component. Now create template for displaying the list, I will use something this one:

```
<table class="table">
	<thead class="thead-inverse">
		<tr>
			<th>Name</th>
			<th>Surname</th>
			<th>Address</th>
			<th>Status</th>
		</tr>
	</thead>
	<tbody>
		<tr *ngFor="let member of members">
			<th>{{member.name}}</th>
			<th>{{member.surname}}</th>
			<th>{{member.address}}</th>
			<th>{{member.status}}</th>
		</tr>
	</tbody>
</table>
```

Data will be provided to component from the parent:

```
export class MembersListComponent{

	@Input()
	members: Member[];

	constructor() { }

}
```

Now inject this view in preview component. You can also check if it is displaying data correctly. Now we will start implementing state management for this view. As we remember we will be working on Observables so create *memebers$* variable in the preview controller and pass it with the async pipe to the list component. For now it's all we can do so let's start with implementation of the Actions. Let's consider two scenarios: Loading Members is initiated and Loading Members ended with success. So create directory *actions* and make file *members-list.actions.ts*. Let's now write down the code:

```
import { Action } from '@ngrx/store';
import { Member } from '../models/member';

export const LOAD_MEMBERS = '[Members-list] Load initiated';
export const LOAD_MEMBERS_SUCCESS = '[Members-list] Load Success
    ↪ ';

export class LoadMembersAction implements Action {
       readonly type = LOAD_MEMBERS;
}

export class LoadMembersSuccessAction implements Action {
       readonly type = LOAD_MEMBERS_SUCCESS;

constructor(public payload: Member[]) { }
}

export type Actions =
       LoadMembersAction |
       LoadMembersSuccessAction;
```

As we mentioned before, each action implements Action interface, have it's unique identifier called type and optionally carries some data as payload. Now let's create reducers.

Let's keep all members reducers in one place. Create the *reducers* directory and create two files: *index.ts* and *members-list.reducer.ts*. Let's start by declaring State in the list reducer:

```
export interface State {
       content: Member[];
}

const initialState: State = {
       content: []
}
```

For now it will only possess information about the loaded members list. Let's now create reducer function:

```
export function reducer(state: State = initialState, action:
    ↪ Actions) {
       switch (action.type) {

       case LOAD_MEMBERS_SUCCESS: {
              return { ...state, content: action.payload };
       }
```

```
        default: {
                return { ...state };
        }

        }
}
```

This reducer will override the content when collection load was successful and return the state on every other action. Let's now create reducer for all member operations. Open index.ts and create members state:

```
import * as List from './members-list.reducer';

export interface State {
        list: List.State;
}

export const reducers: ActionReducerMap<State> = {
        list: List.reducer
};
```

Here we are loading everything we declared in previously written reducer with an alias List. We can now provide list state and reducer to our global members one. This enables as to split out code into smaller parts grouped by their functionality which is a really good practice.

Now we only need to somehow access the values in our state. We can select states from the store by creating selectors. Each of the reducers and state is being kept in store under unique identifier. For this one we will use *'members'*. To access this data we need to create selector for the feature:

```
import { createSelector, createFeatureSelector } from '@ngrx/
    ↪ store';

// Some code

export const getMembersState = createFeatureSelector('members');
```

In similar way we can access the state in the given selection to return values we need. To do that we will use createSelector method. This method takes the state and let's you pull specified values from it. Let's try to get the list content:

```
export const getMembersListState = createSelector(
       getMembersState,
       (state: State) => state.list
       );

export const getMembersListContent = createSelector(
       getMembersListState,
       List.getContent
);
```

To make it work write additional code in *memebrs-list.reducer.ts*:

```
export const getContent = (state: State) => state.content;
```

Of course you could directly access the content in the first createSelector but it's a good practise to keep it split and put the data fetching in proper reducer file.

We can now include our reducer in the *MembersModule*:

```
StoreModule.forFeature('members', reducers)
```

Now we can write some effects for our list loading. Create the *effects* directory and then the file *members-list.effects.ts*. Write down the following code:

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/switchMap';
import { Injectable } from '@angular/core/';
import { Actions, Effect } from '@ngrx/effects'
import { MembersDataService } from '../services/members-data.
    ↪ service';
import { LOAD_MEMBERS,
LoadMembersSuccessAction } from '../actions/members-list.actions
    ↪ ';
import { Member } from '../models/member';

@Injectable()
export class MembersListEffects {

constructor(private actions$: Actions,
           private membersService: MembersDataService) { }

@Effect()
loadList$ = this.actions$.ofType(LOAD_MEMBERS)
.switchMap(() => this.membersService.getMembers())
.map((list: Member[]) => new LoadMembersSuccessAction(list));
```

```
}
```

This effect intercepts action described by LOAD_MEMBERS type and then retrieves collection from the service to return new Action with it's content as the payload.

Let's now include it in the module:

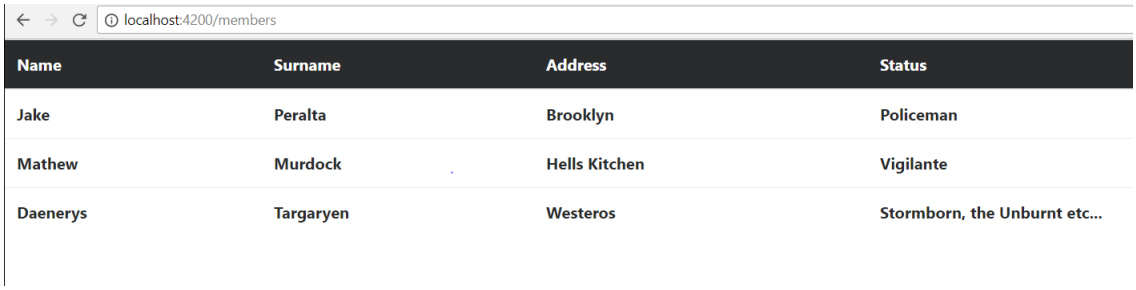
```
EffectsModule.forFeature([MembersListEffects])
```

Let's now modify the *MembersPreviewComponent* a little:

```
import { Component, OnInit } from '@angular/core';
import { MembersDataService } from '../../services/members-data.
    ↪ service';
import { Member } from '../../models/member';

import * as List from '../../actions/members-list.actions';
import * as Members from '../../reducers/index';
import { Store } from '@ngrx/store';
import { LoadMembersAction } from '../../actions/members-list.
    ↪ actions';

@Component({
selector: 'app-members-preview',
templateUrl: './members-preview.component.html'
})
export class MembersPreviewComponent implements OnInit {

       members$;

constructor(private store: Store<Members.State>) {
       this.members$ = store.select(Members.getMembersListContent
           ↪ );
}


ngOnInit() {
       this.store.dispatch(new LoadMembersAction);
}

}
```

To connect to the store we need to provide it in the constructor of our component. As the parameter to provide the State which should be returned. Now we can select the values form it and subscribe it to the proper variables. Thats what

we are doing to *members$*, now each change to to content will also be visible in our component. To initate collection load we dispatch the LoadMembersActions. Now members$ should be updated and the list displayed.



5.2 Adding new member

Let's now create new functionality which will be responsible for adding new member to collection. Let's just start with updating our AppComponent so it contains a navbar with navigation:

```
<nav class="navbar navbar-toggleable-md">
 <div class="collapse navbar-collapse" id="navbarSupportedContent
     ↪ ">
  <ul class="navbar-nav mr-auto">
   <li class="nav-item">
    <a class="nav-link" [routerLink]="['/members']">Members</a>
   </li>
   <li class="nav-item">
    <a class="nav-link" [routerLink]="['/members/form']">New
        ↪ member</a>
   </li>
  </ul>
 </div>
</nav>
```

Let's now create the view which will be shown after we click new member. Use cli to generate *MembersFormPageComponent* in the *containers* directory. This component will be the parent for our form. Go to components and generate *MembersFormComponent*. Write down following code:

```
import { Component, OnInit, Input, Output, EventEmitter } from '
    ↪ @angular/core';
import { FormBuilder } from '@angular/forms';
import { Member } from '../../models/member';
```

```
@Component({
selector: 'app-members-form',
templateUrl: './members-form.component.html'
})
export class MembersFormComponent implements OnInit {

        @Input()
        member: Member;

        @Output()
        submitMember = new EventEmitter<Member>();

        memberForm;

        constructor(private fb: FormBuilder) { }

        ngOnInit() {
                this.createForm();
        }

createForm() {
        this.memberForm = this.fb.group({
        name: [(this.member.name ? this.member.name : '')],
        surname: [(this.member.surname ? this.member.surname : '')
            ↪ ],
        address: [(this.member.address ? this.member.address : '')
            ↪ ],
        status: [(this.member.status ? this.member.status : '')]
        });
}

submit() {
const member = Object.assign(this.member, this.memberForm.value);
this.submitMember.emit(member);
}

}
```

Now let's create a template for it:

```
//Authors comment: SORRY!

<div class="container" style="padding-top:2%;">
<form [formGroup]="memberForm">
<div class="form-group row">
<label for="name-input" class="col-2 col-form-label">Name</label>
```

```
<div class="col-10">
<input class="form-control" formControlName="name"
        type="text" id="name-input">
</div>
</div>
<div class="form-group row">
<label for="surname-input" class="col-2 col-form-label">Surname</
    ↪ label>
<div class="col-10">
<input class="form-control" formControlName="surname"
       type="text" id="surname-input">
</div>
</div>
<div class="form-group row">
<label for="addrress-input" class="col-2 col-form-label">Address
    ↪ </label>
<div class="col-10">
<input class="form-control" formControlName="address"
       type="text" id="addrress-input">
</div>
</div>
<div class="form-group row">
<label for="status-input" class="col-2 col-form-label">Status</
    ↪ label>
<div class="col-10">
<input class="form-control" type="text"
       formControlName="status" id="status-input">
</div>
</div>
<button class="btn btn-primary" (click)="submit()">Submit</button
    ↪ >
</form>
</div>
```

We can now create routing for this view.

```
{ path: 'form', component: MembersFormPageComponent, pathMatch: '
    ↪ full' }
```

Before we start implementing the *MembersFormPageComponent* let's create Actions, Reducer and Effects. Let's start with the actions. We already did that so there is no need to explain anything :) Create *members-form.actions.ts* in actions directory and write following code:

```
import { Action } from '@ngrx/store';
import { Member } from '../models/member';
```

```
export const LOAD_NEW_MEMBER_FORM = '[Member-form] Load new
    ↪ member form';
export const SUBMIT_MEMBER_FORM = '[Member-form] Submit form';

export class LoadNewMemberFormAction implements Action {
       readonly type = LOAD_NEW_MEMBER_FORM;
}

export class SubmitMemberFormAction implements Action {
       readonly type = SUBMIT_MEMBER_FORM;

       constructor(public payload: Member) { }
}

export type Actions =
       LoadNewMemberFormAction |
       SubmitMemberFormAction;
```

We will for now only handle loading for creation of new user and submitting the form. Now let's head to the reducer, let's create it:

```
import { Member } from '../models/member';
import { Actions, LOAD_NEW_MEMBER_FORM, SUBMIT_MEMBER_FORM } from
        '../actions/members-form.actions';

export interface State {
       memberToEdit: Member;
       submitedMember: Member;
}

export const initialState: State = {
       memberToEdit: {} as Member,
       submitedMember: {} as Member
};

export function reducer(state: State = initialState, action:
    ↪ Actions) {
switch(action.type) {

case LOAD_NEW_MEMBER_FORM: {
       return {...state, memberToEdit: { id: -1}, submitedMember:
           ↪  {} }
}

case SUBMIT_MEMBER_FORM: {
```

```
        return {...state, memberToEdit: {}, submitedMember: action
            ↪ .payload }
}

default: {
        return { ...state };
}
}
}
export const getEditedMember = (state: State) => state.
    ↪ memberToEdit;
```

And the simple effects which will be responsible for adding new member to data and redirecting to the table view:

```
import { Injectable } from '@angular/core/';
import { MembersDataService } from '../services/members-data.
    ↪ service';
import { Actions, Effect } from '@ngrx/effects';
import { SUBMIT_MEMBER_FORM, SubmitMemberFormAction } from '../
    ↪ actions/members-form.actions';
import { Router } from '@angular/router';
import 'rxjs/add/operator/do';

@Injectable()
export class MembersFormEffects {

        constructor(private actions$: Actions,
                                private membersService:
                                    ↪ MembersDataService,
                                private router: Router) { }

        @Effect({dispatch: false})
        submit$ = this.actions$.ofType(SUBMIT_MEMBER_FORM)
                .map((action: SubmitMemberFormAction) => action.
                    ↪ payload)
                .do(member => {
                        this.membersService.addMember(member);
                        this.router.navigate(['members']);
        });

}
```

Let's update the code of our member reducers in the index.ts file:

```
import * as Form from './member-form.reducer';
```

```
export interface State {
       list: List.State;
       form: Form.State;
}

export const reducers = {
                                    list: List.reducer,
                                    form: Form.reducer
};

// ...

export const getFormState = createSelector(
       getMembersState,
       (state: State) => state.form
);

export const getFormEditedMember = createSelector(
       getFormState,
       Form.getEditedMember
);
```

We can now write the code for our *MembersFormPageComponent*. Let's do this:

```
import { Component, OnInit } from '@angular/core';
import { Member } from '../../models/member';

import * as Members from '../../reducers/index';
import * as Form from '../../actions/members-form.actions';

import { Store } from '@ngrx/store';
import { LoadNewMemberFormAction, SubmitMemberFormAction } from
    ↪ '../../actions/members-form.actions';

@Component({
selector: 'app-members-form-page',
templateUrl: './members-form-page.component.html'
})
export class MembersFormPageComponent implements OnInit {

member$;

       constructor(private store: Store<Members.State>) {
              this.member$ = store.select(Members.
                  ↪ getFormEditedMember);
```

```
        }

        ngOnInit() {
                this.store.dispatch(new LoadNewMemberFormAction())
        }

        onSubmit(member: Member) {
                this.store.dispatch(new SubmitMemberFormAction(
                    ↪ member));
        }
}
```

And the template:

```
<app-members-form [member]="member$ | async" (submitMember)="
    ↪ onSubmit($event)"></app-members-form>
```

The last thing to do is to add new effects in *MembersModule*:

```
EffectsModule.forFeature([MembersListEffects, MembersFormEffects
    ↪ ])]
```

Now everything should be working correctly!

6 What now?

Congratulation! You've just finished this tutorial and wrote quite a bit of ngrx store code :). I will be leaving some additional tasks if you are interested in continuing working with this tool. Hope you enjoyed it, thanks for reading :)

If you want to learn more about this library i suggest starting with the official Github page of the framework. You can find there a lot of information about how it works but be careful: most of them was created for older versions, not the latest :)

7 Tasks

Task 1: alter struckture a little. Disable navigating from navbar to the form. Adding new member should be availabe after clicking on a button on a list which will dispatch action and redirect to the form view.

Task 2: add new functionality: after double click on a row in table you should be redirected to the form with the chosen member as payload. The event should dispatch new action.

Task 3: add pagination to the table. Changing page should dispatch action

which will load subset of data from our mocked data service.

Enjoy the work :)