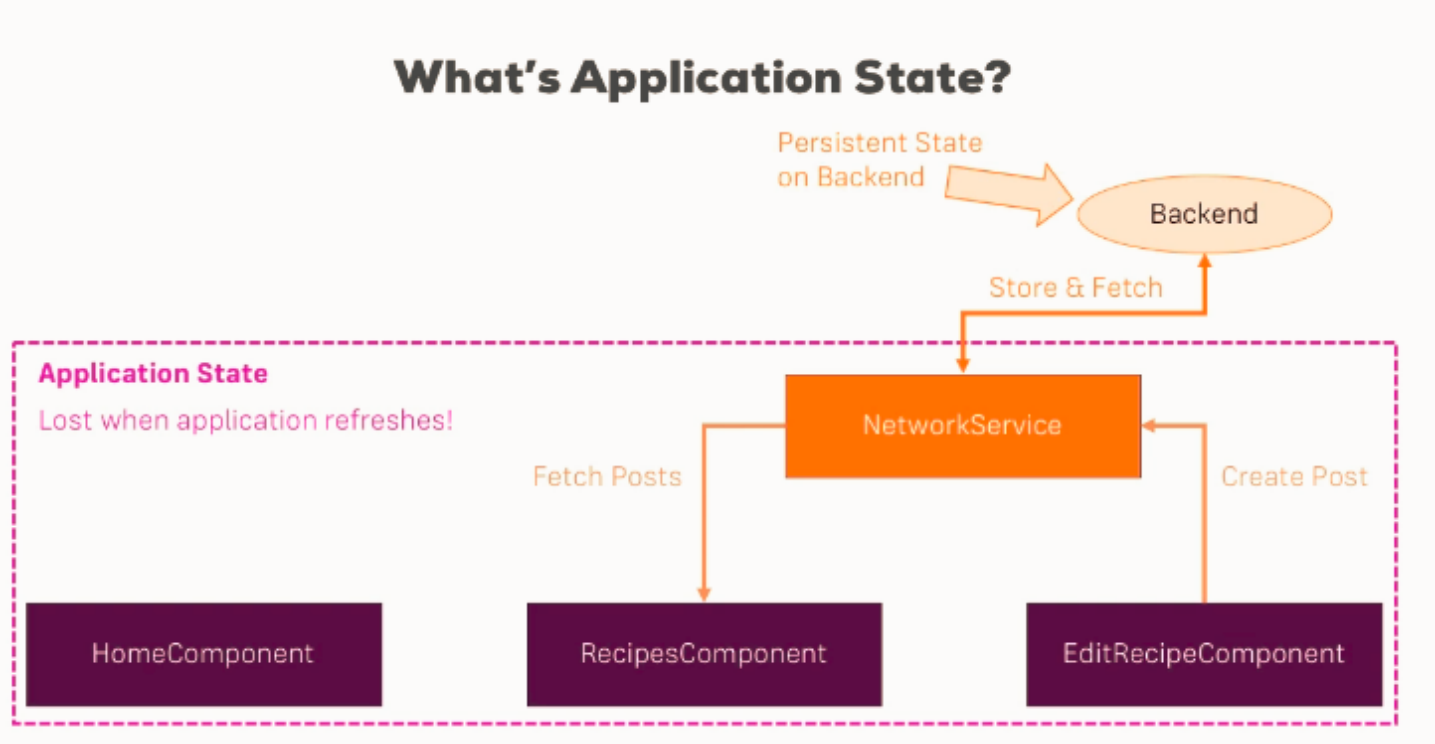
10 Angular NgRx(24)

SourceFiles: https://ide.c9.io/laczor/angular
Information
Official Github Repo with Documentation: https://github.com/ngrx/platform
Angular & NgRx Tutorial: https://blog.nrwl.io/using-ngrx-4-to-manage-state-in-angular-applications-64e7a1f84b7b
NgRx Patterns & Techniques: https://blog.nrwl.io/ngrx-patterns-and-techniques-f46126e2b1e5

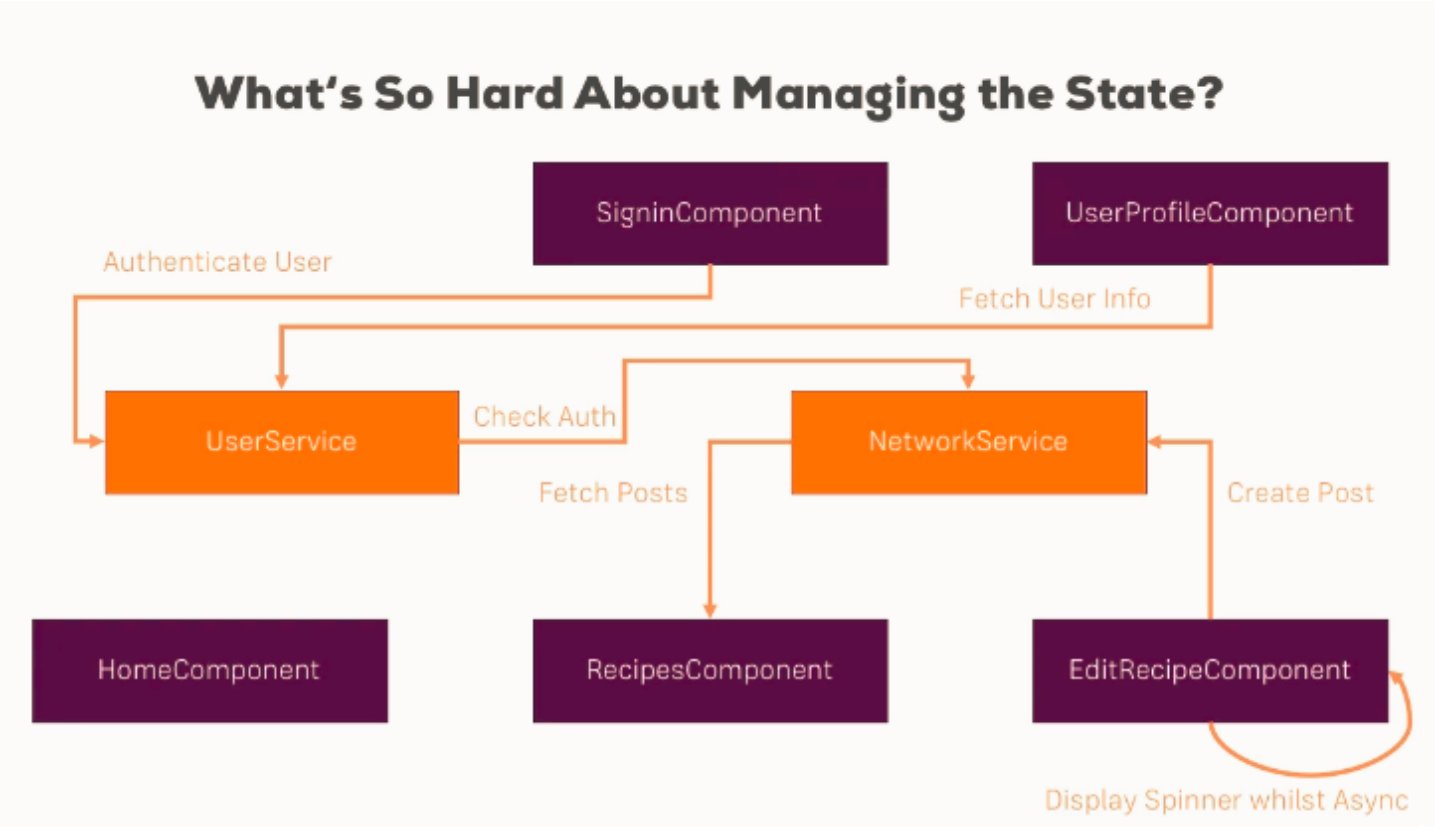
Session_24_NgRx/02-ngrx-basic-sl-setup
301 Application State changes
302 Getting Started with Reducers
307 Dispatch Actions
(03-ngrx-finished-sl-setup)
308 Adding additional Dispatch
309 Update + Delete
310 Expanding App (Interfaces)
311 Start Stop Edit
312 Stop Edit
(04-ngrx-basic-auth-setup)
313 Adding Authentication
319 Dispatch Async Actions
320 Getting State Access in theHttp Interceptor
(05-ngrx-auth-effects)
322 Installing effects
325 Listening to an Action with effects
326. SignIn
329 LogOut
(06-ngrx-router-store-devtools)
333 Installing Router Package
334 Installing DevTools
(07-ngrx-recipes-finished)
335 Recipes ngrx + lazy Loading dynamic injections
339 Using two ngrx in one component
341 Fetching and storing Data with effects

301 Application State changes

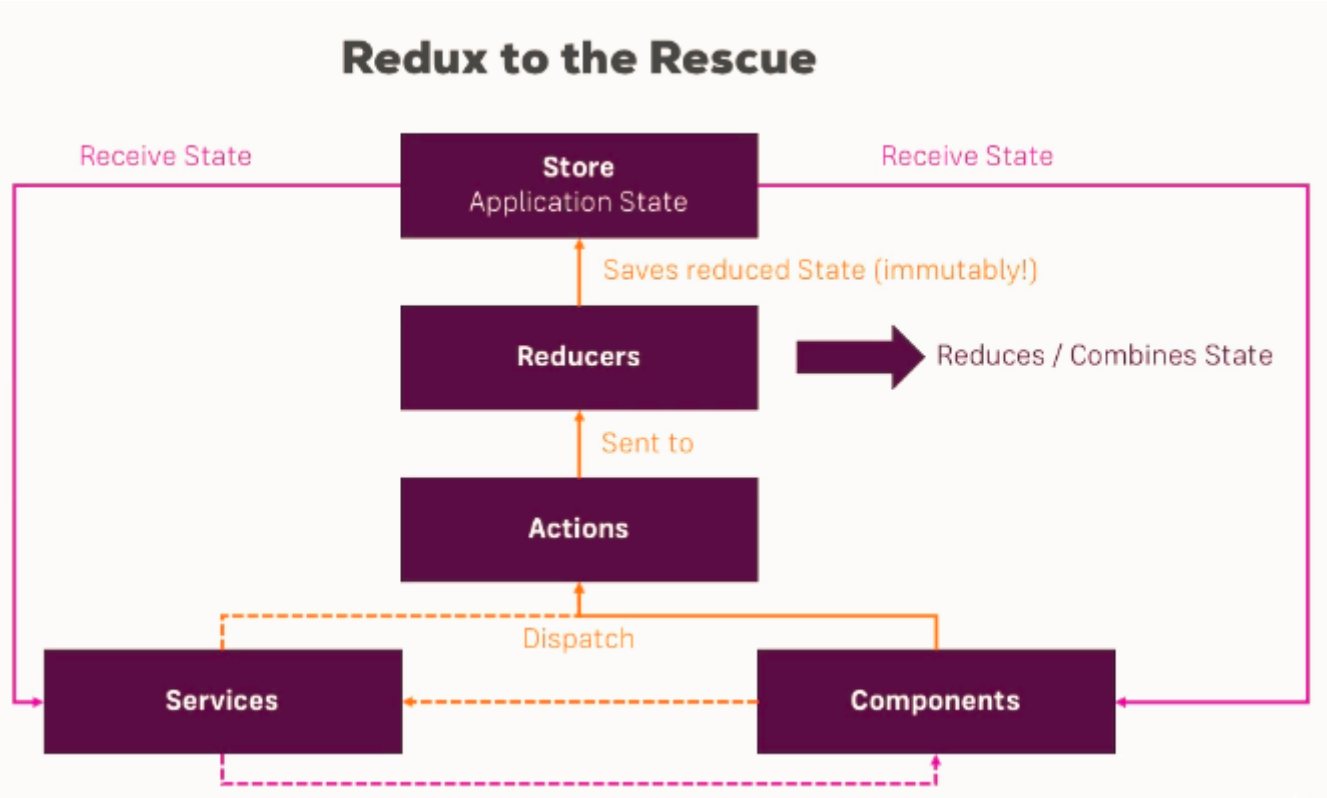
1. First state is when there is a provided service which handles all of the data modification and the storage is centralized in one single source of truth.
1.1 Also at this states subjects are used to notify other component's that some data has been changed



2. Is when the there are more services, which can modify the data at seperate modules, component's harder to maintain the order, track the data.



3. **Redux** approach is to listen for the datastorage modificaiton reqeuests as **'actions'** which will be **immutably** chage the data at the single source of truth



302 Getting Started with Reducers

1. install it **npm install --save @ngrx/store (includes the main package)**

2, Create a reducers.ts file
shopping-list.actions.ts

```
//Importing the Action interface from the store + the ingredient model

//1. We define a string constant to be a property

//2. Create an exportable class which implements the Action inferface

//3. This interface should have  type& payload propety defined.

//3.1 Here the type is the declared const string

//3.2 The payload is the ingredieng object model

//4.0 Lastly we are exporing it as a type, which will be able to include multiple types, export
ed classed


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;
```

shopping-list.reducers.ts

```
//0. We are importing all of the types defined in the shopping-list.actions and exported as a t
ype of ShoppingListActions  + ingredient model
//1.Create an inital state, data
//2.Create an exportable ShoppingListReducer function.
//    2.1 state is the inital object, which contains ingredients
//    2.2 action is the ShoppingListActions.ShoppingListActions (first is referencing the impor
t, second is the exported type class)
//    2.1 ShoppingListActions.AddIngredient  = 'ADD_INGREDIENT'
//3.0 Returning the seperated, newly created array


import * as ShoppingListActions from './shopping-list.actions';         //Exporting all of the
 exported class from the shopping-list.actions typescript file


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


//1.Creating an inital state, with inital values
const initialState = {
  ingredients: [
    new Ingredient('Apples', 5),
    new Ingredient('Tomatoes', 10),
  ]
};
```

```
//Create an exportable ShoppingListReducer function.


export function shoppingListReducer(state = initialState, action: ShoppingListActions.ShoppingL
istActions) {
  switch (action.type) {
    case ShoppingListActions.ADD_INGREDIENT:
      return {
        ...state,                                                    //Destructing the object o
f the initalState, which is an obect wiht 1 property, but in case of severale properties, it wi
ll retirn all of the properties
        ingredients: [...state.ingredients, action.payload]          //Recreating the ingredien
ts array, by separating the old one + add 1 new ingredient which is actually the payload.
      };


    default:
      return state;
  }
}
```

3. Register at the **app.module.ts**
0. We are importing firstly the storeModule, to use the reducer, then we are including the recuder file
1. we are creating an identifier for the reducer function as **shoppingList**

```
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { NgModule } from '@angular/core';
import {StoreModule} from '@ngrx/store';


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


@NgModule({
  declarations: [
    AppComponent
  ],
```

```
  imports: [
    StoreModule.forRoot({shoppingList:shoppingListReducer})
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

4. Using the reducer instead of the service **shopping-list.component.ts**

```
//0. Importing the Store + Observables,
// 1.Modifying the ingredients component variable to shoppingListate and assing a type of obser
vable, since that is what we got from the store, than further defining that we are expecting an
 object with an ingredients property which consist of Ingredient [] array
// 2.Injecting the store, with the defined ShoppingList type in the app.module.ts, which is an
 object with property ingredients and array of Ingredient
// 3. On init, we are get the Inital state, since we are jus returning the initalState defined
 object in the reducer.ts
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs/Subscription';
import { Store } from '@ngrx/store';
import {Observable} from 'rxjs/Observable';

import { Ingredient } from '../shared/ingredient.model';
import { ShoppingListService } from './shopping-list.service';
@Component({
  selector: 'app-shopping-list',
  templateUrl: './shopping-list.component.html',
  styleUrls: ['./shopping-list.component.css']
})
export class ShoppingListComponent implements OnInit, OnDestroy {
  shoppingListState: Observable<{ingredients:Ingredient[]}>;                              //
  private subscription: Subscription;
```

```
  constructor(private slService: ShoppingListService, private store:Store<{shoppingList:{ingred
ients:Ingredient[]}}>) { }


  ngOnInit() {
    this.shoppingListState = this.store.select('shoppingList');
  }
```

307 Dispatch Actions

shopping-edit.component.ts
0. We are importing Store + the ShoppingListActions exported type of class.
1. Inject it specifying exactly the type what a store service will look like
2. Simply use the exported **AddIngredient** class, (which has a basic constructor)
(`new` ShoppingListActions.AddIngredient(newIngredient))

```
import {
  Component,
  OnInit,
  OnDestroy,
  ViewChild
} from '@angular/core';
import { NgForm } from '@angular/forms';
import { Subscription } from 'rxjs/Subscription';
import {Store} from '@ngrx/store';

import { Ingredient } from '../../shared/ingredient.model';
import { ShoppingListService } from '../shopping-list.service';

import * as ShoppingListActions from '../store/shopping-list.actions';

@Component({
  selector: 'app-shopping-edit',
  templateUrl: './shopping-edit.component.html',
  styleUrls: ['./shopping-edit.component.css']
```

```
})
export class ShoppingEditComponent implements OnInit, OnDestroy {
  @ViewChild('f') slForm: NgForm;
  subscription: Subscription;
  editMode = false;
  editedItemIndex: number;
  editedItem: Ingredient;

  constructor(private slService: ShoppingListService,private store:Store<{shoppingList:{ingredi
ents:Ingredient[]}}>) { }

  ngOnInit() {
    this.subscription = this.slService.startedEditing
      .subscribe(
        (index: number) => {
          this.editedItemIndex = index;
          this.editMode = true;
          this.editedItem = this.slService.getIngredient(index);
          this.slForm.setValue({
            name: this.editedItem.name,
            amount: this.editedItem.amount
          })
        }
      );
  }

  onSubmit(form: NgForm) {
    const value = form.value;
    const newIngredient = new Ingredient(value.name, value.amount);
    if (this.editMode) {
      this.slService.updateIngredient(this.editedItemIndex, newIngredient);
    } else {
```

```
      // this.slService.addIngredient(newIngredient);            //This was the old one
      //THis is the injected store service, with dispatch using the ShoppinglistActions, functi
on
      this.store.dispatch(new ShoppingListActions.AddIngredient(newIngredient));
    }
    this.editMode = false;
    form.reset();
  }

  onClear() {
    this.slForm.reset();
    this.editMode = false;
  }

  onDelete() {
    this.slService.deleteIngredient(this.editedItemIndex);
    this.onClear();
  }
}
```

shopping-list.actions.ts
- when we are calling

```
new ShoppingListActions.AddIngredient(newIngredient)
```

Somewhoe the action and store function will know that the passed argument should be used a the newly created class's **payload** poperty

```
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;

  constructor(public payload:Ingredient){}

}

export type ShoppingListActions =  AddIngredient;
```

308 Adding additional Dispatch

1. Add a new action, where we are expecting an array of Ingrediens and the type name is `ADD_INGREDIENTS`

shopping-list.ations.ts

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


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


export const ADD_INGREDIENT = 'ADD_INGREDIENT';

export const ADD_INGREDIENTS = 'ADD_INGREDIENTS';


export class AddIngredient implements Action {

  readonly type = ADD_INGREDIENT;

  constructor(public payload:Ingredient){}

}
export class AddIngredients implements Action {

  readonly type = ADD_INGREDIENTS;

  constructor(public payload:Ingredient[]){}

}

export type ShoppingListActions =  AddIngredient | AddIngredients;
```

2. Creating a new case for our reducer,, which takes an array of ingredients, and spread it with ... into the ingredients [] with ...**action,payload**
export function shoppingListReducer(state = initialState, action: ShoppingListActions.ShoppingListActions) {

```
```

```
  switch (action.type) {
    case ShoppingListActions.ADD_INGREDIENT:
      return {
        ...state,
        ingredients: [...state.ingredients, action.payload]
      };
    case ShoppingListActions.ADD_INGREDIENTS:
      return {
        ...state,
        ingredients: [...state.ingredients, ...action.payload]
      };

    default:
      return state;
  }
}
```

3. Then we can modify the addIngredients funciton on the **recipe.detail.component.ts**

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params, Router } from '@angular/router';

import { Recipe } from '../recipe.model';
import { RecipeService } from '../recipe.service';

//iMPORTING EVERYthing Store, Observable for the type declaration, Ingredient object model, and
 all of the exported class
import { Store } from '@ngrx/store';
import {Observable} from 'rxjs/Observable';
import {Ingredient} from '../../shared/ingredient.model';
import * as ShoppingListActions from '../../shopping-list/store/shopping-list.actions';

@Component({
```

```
  selector: 'app-recipe-detail',
  templateUrl: './recipe-detail.component.html',
  styleUrls: ['./recipe-detail.component.css']
})
export class RecipeDetailComponent implements OnInit {
  recipe: Recipe;
  id: number;

//2. Inject the store
  constructor(private recipeService: RecipeService,
              private route: ActivatedRoute,
              private router: Router,
              private store:Store<{shoppingList:{ingredients:Ingredient[]}}>) {
  }
//Create new dispatch, to update the reducer.
  onAddToShoppingList() {
    this.store.dispatch(new ShoppingListActions.AddIngredients(this.recipe.ingredients));
    // this.recipeService.addIngredientsToShoppingList(this.recipe.ingredients);
  }

}
```

309 Update + Delete

Upgrade:
- You got the index. and the indgredient object
Shopping-edit.component.ts

```
 this.store.dispatch(new ShoppingListActions.UpdateIngredient({index: this.editedItemIndex, ing
redient: newIngredient}))
```

shopping-list.actions.ts

```
export class UpdateIngredient implements Action {

  readonly type = UPDATE_INGREDIENT;


  constructor(public payload: {index: number, ingredient: Ingredient}) {}

}
```

shopping-list.reducers.ts
- Copy & overwrite the properties of the **ingredient object {name:'xxx',amount:'xx'}**
- Destructure the object properties and update by the index number

```
  case ShoppingListActions.UPDATE_INGREDIENT:

      const ingredient = state.ingredients[action.payload.index];        //To store the ingredi
ent

      const updatedIngredient = {

        ...ingredient,                                                 // to copy all of the propert
ies of the ingredient object

        ...action.payload.ingredient                                     //overWrite the properties
 with the new ingredient

      };

      const ingredients = [...state.ingredients];                       //Move apart the old objec
t, update the correct index

      ingredients[action.payload.index] = updatedIngredient;

      return {                                                          //Assign it to the new obje
ct

        ...state,

        ingredients: ingredients

      };
```

Delete:
- You got the index, seperate the array, **splice** the to be deleted one, return the desctrucutred array.

Shopping-edit.component.ts

```
    this.store.dispatch(new ShoppingListActions.DeleteIngredient(this.editedItemIndex));
```

shopping-list.actions.ts

```
export class DeleteIngredient implements Action {
  readonly type = DELETE_INGREDIENT;

  constructor(public payload: number) {}
}
```

shopping-list.reducers.ts
- Copy & overwrite the properties of the **ingredient object {name:'xxx',amount:'xx'}**
- Destructure the object properties and update by the index number

```
  case ShoppingListActions.DELETE_INGREDIENT:
      const oldIngredients = [...state.ingredients];
      oldIngredients.splice(action.payload, 1);
      return {
        ...state,
        ingredients: oldIngredients
      };
```

310 Expanding App (Interfaces)

1.Create interfaces to be used.

shopping-list.recuders.ts

```
// 0.we are defining what properties, methods the component should have when using an interface
// State  is an objet with ingredients array + two other properties
// Appstate is an object with a shoppingList property with a state object

export interface AppState {
  shoppingList: State
}

export interface State {
```

```
  ingredients: Ingredient[];
  editedIngredient: Ingredient;
  editedIngredientIndex: number;
}

const initialState: State = {
  ingredients: [
    new Ingredient('Apples', 5),
    new Ingredient('Tomatoes', 10),
  ],
  editedIngredient: null,
  editedIngredientIndex: -1
};
```

shoppinglist-edit.component.ts

```
//Original declaration, an object having a shoppingList property with an object, containing ing
redients array
constructor(private slService: ShoppingListService, private store: Store<{shoppingList: {ingred
ients: Ingredient[]}}>) { }
//By exporting the defined interfaces classes from the reducers, we can implement them to defin
e the injected ngrx store observvable

import * as fromShoppingList from '../store/shopping-list.reducers';
constructor(private store: Store<fromShoppingList.AppState>) { }
```

311 Start Stop Edit

1.Create actions for them
shoppinglist.actions.ts

```
export class StartEdit implements Action {                          //Has only 1 argument a numb
er
  readonly type = START_EDIT;
```

```
  constructor(public payload: number) {}
}


export class StopEdit implements Action {                          //Has no argument, only a type
  readonly type = STOP_EDIT;
}
```

2. Create the appropriate reducers
- Since we get the the index number (**editedIngredientIndex**) in the store we can reference it upon updating, deleting
- We continously keep track of the index, + the editedINgredient object, so we know centrally which ingredient should be edited.

shoppinglist.reducers.ts

```
    case ShoppingListActions.UPDATE_INGREDIENT:
      const ingredient = state.ingredients[state.editedIngredientIndex];
      const updatedIngredient = {
        ...ingredient,
        ...action.payload.ingredient
      };
      const ingredients = [...state.ingredients];
      ingredients[state.editedIngredientIndex] = updatedIngredient;
      return {
        ...state,
        ingredients: ingredients,
        editedIngredient: null,
        editedIngredientIndex: -1
      };
    case ShoppingListActions.DELETE_INGREDIENT:
      const oldIngredients = [...state.ingredients];
      oldIngredients.splice(state.editedIngredientIndex, 1);
      return {
        ...state,
        ingredients: oldIngredients,
```

```
        editedIngredient: null,
        editedIngredientIndex: -1
      };
    case ShoppingListActions.START_EDIT:
      const editedIngredient = {...state.ingredients[action.payload]};
      return {
        ...state,
        editedIngredient: editedIngredient,
        editedIngredientIndex: action.payload
      };
    case ShoppingListActions.STOP_EDIT:
      return {
        ...state,
        editedIngredient: null,
        editedIngredientIndex: -1
      };
```

3. It is important that we can even subscribe to the store states! not just make a copy of them
shoppinglist-edit.component.ts

```
ngOnInit() {
    this.subscription = this.store.select('shoppingList')
      .subscribe(
        data => {
          if (data.editedIngredientIndex > -1) {
            this.editedItem = data.editedIngredient;
            this.editMode = true;
            this.slForm.setValue({
              name: this.editedItem.name,
              amount: this.editedItem.amount
            })
          } else {
            this.editMode = false;
```

```
          }
        }
      );
  }
```

312 Stop Edit

It is important that, the editing parameters, index, is edit are alwayys cleared when the user navigates away.

So we can create a stop edit **action** + a **reducer** and when the component is **destroyed** you can dispatch this action

shopping-list.actions.ts

```
export class StopEdit implements Action {
  readonly type = STOP_EDIT;
}
```

shopping-list.reducers.ts

```
 case ShoppingListActions.STOP_EDIT:
      return {
        ...state,
        editedIngredient: null,
        editedIngredientIndex: -1
      };
```

shopping-listedit.component.ts

```
  ngOnDestroy() {
    this.store.dispatch(new ShoppingListActions.StopEdit());
    this.subscription.unsubscribe();
  }
```

313 Adding Authentication

1. Create the **auth/store/auth.actions.ts**

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


export const SIGNUP = 'SIGNUP';

export const SIGNIN = 'SIGNIN';

export const LOGOUT = 'LOGOUT';

export const SET_TOKEN = 'SET_TOKEN';


export class Signup implements Action {

  readonly type = SIGNUP;

}


export class Signin implements Action {

  readonly type = SIGNIN;

}


export class Logout implements Action {

  readonly type = LOGOUT;

}


export class SetToken implements Action {

  readonly type = SET_TOKEN;


  constructor(public payload: string) {}

}


export type AuthActions = Signup | Signin | Logout | SetToken;
```

2. Create the **auth.reducers.ts**
//Basically modifying the token, atuhenticated states.

```
import * as AuthActions from './auth.actions';
```

```
export interface State {
  token: string;
  authenticated: boolean;
}

const initialState: State = {
  token: null,
  authenticated: false
};

export function authReducer(state = initialState, action: AuthActions.AuthActions) {
  switch (action.type) {
    case (AuthActions.SIGNUP):
    case (AuthActions.SIGNIN):
      return {
        ...state,
        authenticated: true
      };
    case (AuthActions.LOGOUT):
      return {
        ...state,
        token: null,
        authenticated: false
      };
    case (AuthActions.SET_TOKEN):
      return {
        ...state,
        token: action.payload
      };
    default:
      return state;
```

```
  }
}
```

3. Create a central **store/app.reducer.ts**
- Import the created two reducers,
- Creates a common interface out of the wo reducers
- Export a **ActionReducerMap** which is a class defned with the AppState interface + assigning the reducer exports.

```
import { ActionReducerMap } from '@ngrx/store';

import * as fromShoppingList from '../shopping-list/store/shopping-list.reducers';
import * as fromAuth from '../auth/store/auth.reducers';

export interface AppState {
  shoppingList: fromShoppingList.State,
  auth: fromAuth.State
}

export const reducers: ActionReducerMap<AppState> = {
  shoppingList: fromShoppingList.shoppingListReducer,
  auth: fromAuth.authReducer
};
```

4. Provide it in the **app.module.ts**

```
import { reducers } from './store/app.reducers';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    CoreModule,
    StoreModule.forRoot(reducers)
```

```
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

5. inject the created service to the components.
header.component.ts
1.Import the commong reducers, to use it's interface implementation
2. import authService as well, since the authState is an object whith two properties, defined in the auth reducer as interface
3. Import Observable since store.select() return an observable

```
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs/Observable';

import { DataStorageService } from '../../shared/data-storage.service';
import { AuthService } from '../../auth/auth.service';
import * as fromApp from '../../store/app.reducers';
import * as fromAuth from '../../auth/store/auth.reducers';

@Component({
  selector: 'app-header',
  templateUrl: './header.component.html'
})
export class HeaderComponent implements OnInit {
  authState: Observable<fromAuth.State>;                              //Defining to be an osberva
lbe of a specifc type object

  constructor(private dataStorageService: DataStorageService,
              private authService: AuthService,
              private store: Store<fromApp.AppState>) {                //Using the crreated int
erface
  }
  ngOnInit() {
    this.authState = this.store.select('auth');                       //Returns an observable
```

```
  }
```

6. Can modify the html based on tha authState
header.component.html
1. Since authState is an observable, we are using async pipe to wait for the response
2.When the response recieved we can acces it's proeprties by closing it in **brackets ().property.**

```
<li class="dropdown" appDropdown *ngIf="(authState | async).authenticated">
```

319 Dispatch Async Actions

Since reducers has to assign the values, return the values syncronosly, we have to use the services module, to dispatch actions in the success callbacks of the promises.
When the promises are executed sucessfully, we are updating the global states with the reducers, to maintain data.
auth/auth.services.ts

```
import { Router } from '@angular/router';
import * as firebase from 'firebase';
import { Injectable } from '@angular/core';
import { Store } from '@ngrx/store';

import * as fromApp from '../store/app.reducers';
import * as AuthActions from './store/auth.actions';

@Injectable()
export class AuthService {
  constructor(private router: Router, private store: Store<fromApp.AppState>) {}

  signupUser(email: string, password: string) {
    firebase.auth().createUserWithEmailAndPassword(email, password)
      .then(
        user => {
          this.store.dispatch(new AuthActions.Signup());
          firebase.auth().currentUser.getToken()
```

```
          .then(
            (token: string) => {
              this.store.dispatch(new AuthActions.SetToken(token));
            }
          )
      }
    )
    .catch(
      error => console.log(error)
    )
}

signinUser(email: string, password: string) {
  firebase.auth().signInWithEmailAndPassword(email, password)
    .then(
      response => {
        this.store.dispatch(new AuthActions.Signin());
        this.router.navigate(['/']);
        firebase.auth().currentUser.getToken()
          .then(
            (token: string) => {
              this.store.dispatch(new AuthActions.SetToken(token));
            }
          )
      }
    )
    .catch(
      error => console.log(error)
    );
}

logout() {
```

```
    firebase.auth().signOut();
    this.store.dispatch(new AuthActions.Logout());
  }
}
```

320 Getting State Access in theHttp Interceptor

Important!!

```
this.store.select('auth')     ---> returns data pacakage wrapped in an observable
(this.store.select('auth') | async) returns data unwrapped from the observalbe ,you can referen
ce it's properties (this.store.select('auth') | async) .isAuthenticated
this.store.select('auth').map((authState: fromAuth.State) => { return authState.authenticated;}
  --> Wit using map, we are transforming our observable values and return a new observable, whi
ch is a boolean value.
```

1. You can check if the authentication prior loading a copmonent with **CanActivate.**
Important ! to use the **take(1)** since instead of subscribing to every change, we want to get only the first, returned observable from the store.
take has to be imported!

```
import 'rxjs/add/operator/take';
```

auth/auth.guard.service.ts

```
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { Injectable } from '@angular/core';
import { Store } from '@ngrx/store';

import * as fromApp from '../store/app.reducers';
import * as fromAuth from './store/auth.reducers';

@Injectable()
export class AuthGuard implements CanActivate {
```

```
  constructor(private store: Store<fromApp.AppState>) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
  // It was simply returning a boolean value
  //return this.authService.isAuthenticated();
  //So we are returning a booleadn wrapped in an transformed observable

    return this.store.select('auth')
    .take(1)
    .map((authState: fromAuth.State) => {
      return authState.authenticated;

    });
  }
}
```

auth,interceptor.ts

0.Import the reducers inject interfaces

1. **take(1)** means it will check only for the first subscription, not reamining for constant changes.
2. **switchMap** with it you can transform observable values, in a way that it won't return an observable anymore, but only value

```
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import { Injectable } from '@angular/core';
import { Store } from '@ngrx/store';
import 'rxjs/add/operator/switchMap';

import * as fromApp from '../store/app.reducers';
import * as fromAuth from '../auth/store/auth.reducers';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
```

```
  constructor(private store: Store<fromApp.AppState>) {}


  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    console.log('Intercepted!', req);
    // const copiedReq = req.clone({headers: req.headers.set('', '')});
    return this.store.select('auth')
      .take(1)
      .switchMap((authState: fromAuth.State) => {
        const copiedReq = req.clone({params: req.params.set('auth', authState.token)});
        return next.handle(copiedReq);                                          //In order to
 return handle and not an osbervable we are using switchmap
      })


    // return null;
  }
}
```

322 Installing effects

1. Add effects
npm install --save @ngrx/effects
2. Create the effects file
auth/store/auth.effects.ts

```
import { Injectable } from '@angular/core';            //Since we are using other services
import { Actions, Effect } from '@ngrx/effects';        //Importing the Actions + the effect d
ecorator


@Injectable()
export class AuthEffects {
  @Effect()                                          //Telling angular that this is an effect
  authSignup ;
  //By injecting the Axtions and declaring it as actions$ you can access all of the actions.
  constructor(private actions$: Actions) {
```

```
  }
}
```

3. Add it into the **app.module.ts**
-Import EffectsModule + adding a forRoot() to it where we declared the effects.

```
import { EffectsModule } from '@ngrx/effects';
import { AuthEffects } from './auth/store/auth.effects';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    AppRoutingModule,
    SharedModule,
    ShoppingListModule,
    AuthModule,
    CoreModule,
    StoreModule.forRoot(reducers),
    EffectsModule.forRoot([AuthEffects])
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

325 Listening to an Action with effects

1. Create a TRY_SIGNUP action
2. Inject to signup and dispatch the action
3. Write the effects, what t do with the action and what should be returned

1. **auth.actions.ts**
:-Takes two argument in an object

```
export const TRY_SIGNUP = 'TRY_SIGNUP';

export class TrySignup implements Action {
  readonly type = TRY_SIGNUP;


  constructor(public payload: {username: string, password: string}) {}
}
```

2.**signup.component.ts**
- Dispatching the TrySignUp action

```
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';
import { Store } from '@ngrx/store';

import * as fromApp from '../../store/app.reducers';
import * as AuthActions from '../store/auth.actions';

@Component({
  selector: 'app-signup',
  templateUrl: './signup.component.html',
  styleUrls: ['./signup.component.css']
})
export class SignupComponent implements OnInit {

  constructor(private store: Store<fromApp.AppState>) { }

  ngOnInit() {
  }

  onSignup(form: NgForm) {
```

```
    const email = form.value.email;
    const password = form.value.password;
    this.store.dispatch(new AuthActions.TrySignup({username: email, password: password}));
  }

}
```

3.**auth.effects.ts**

```
//1. FIrst we check the type of the actions, whic has been triggered, if it its TRY_SIGNUP then

//2. We will transform our observable, extract the action.payload {email:'',username:''} and re
turn it as an observable
//3. Get the value from the recieved observable with switchmap and transforming the promise res
ponse from firebase to an observable so we can chain more acitons
//4. with switchmap u can call the getToken action as well
//5. Lastly we are returning 1 merged observable which contains tow actionsto



import { Injectable } from '@angular/core';                    //Since we are using other servic
es, actons, effects,
import { Actions, Effect } from '@ngrx/effects';               //Getting acceses to the ections
 + the Effect decorator
import 'rxjs/add/operator/map';                                //Transform observable value and
 return an observable
import 'rxjs/add/operator/switchMap';                          //Transform observable value and
 return a value
import 'rxjs/add/operator/mergeMap';                           //Tkaes multiple observables and
 merge together
import { fromPromise } from 'rxjs/observable/fromPromise';     //From promise it will make an ob
servable
import * as firebase from 'firebase';                          //Importing firebase for authenti
cation at the backend
```

```
import * as AuthActions from './auth.actions';                //IMporting to get the teypes of
 actions


@Injectable()
export class AuthEffects {
  @Effect()
  //1
  authSignup = this.actions$
    .ofType(AuthActions.TRY_SIGNUP)
 //2
    .map((action: AuthActions.TrySignup) => {
      return action.payload;
    })
//3
    .switchMap((authData: {username: string, password: string}) => {
      return fromPromise(firebase.auth().createUserWithEmailAndPassword(authData.username, auth
Data.password));
    })
//4
    .switchMap(() => {
      return fromPromise(firebase.auth().currentUser.getIdToken());
    })
//5
    .mergeMap((token: string) => {
      return [
        {
          type: AuthActions.SIGNUP
        },
        {
          type: AuthActions.SET_TOKEN,
          payload: token
        }
      ];
```

```
    });
  constructor(private actions$: Actions, private router: Router) {
  }
}
```

326. SignIn

1. Add a SIGN_IN action to the **auth.effects.ts**

```
export const SIGNIN = 'SIGNIN';
export class TrySignin implements Action {
  readonly type = TRY_SIGNIN;

  constructor(public payload: {username: string, password: string}) {}
}
```

2. Insert it into the **signin.component.ts**
Import store and reducer declarations, to implement the inferfaces + adding authActions to dispatch an aciton

```
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';
import { Store } from '@ngrx/store';

import * as fromApp from '../../store/app.reducers';
import * as AuthActions from '../store/auth.actions';

@Component({
  selector: 'app-signup',
  templateUrl: './signup.component.html',
  styleUrls: ['./signup.component.css']
})
export class SignupComponent implements OnInit {
```

```
  constructor(private store: Store<fromApp.AppState>) { }

  ngOnInit() {
  }

  onSignup(form: NgForm) {
    const email = form.value.email;
    const password = form.value.password;
    this.store.dispatch(new AuthActions.TrySignup({username: email, password: password}));
  }
}
```

3. Write the **effect** which is listening to the **SIGN_IN** action
-basically everything is the same as with the signup, however we are **firebase.signInWithEmailAndPassword + navigating** away when signed in

```
 @Effect()
  authSignin = this.actions$
    .ofType(AuthActions.TRY_SIGNIN)
    .map((action: AuthActions.TrySignup) => {
      return action.payload;
    })
    .switchMap((authData: {username: string, password: string}) => {
      return fromPromise(firebase.auth().signInWithEmailAndPassword(authData.username,
authData.password));
    })
    .switchMap(() => {
      return fromPromise(firebase.auth().currentUser.getIdToken());
    })
    .mergeMap((token: string) => {
      this.router.navigate(['/']);
      return [
        {
```

```
          type: AuthActions.SIGNIN
        },
        {
          type: AuthActions.SET_TOKEN,
          payload: token
        }
      ];
    });
```

329 LogOut

auth.effects.ts
- listening for the LGOUT action
- {dispatch: false} means ---> it will not return any observableand dispatch it for aciton
- **.do()** rjx operator will do the following: this is like an intemediary in the observable cahin it will do some stuff, subscribe, will end the chain end we have to return an observable somehow.

```
  @Effect({dispatch: false})
  authLogout = this.actions$
    .ofType(AuthActions.LOGOUT)
    .do(() => {
      this.router.navigate(['/']);
    });
```

333 Installing Router Package

1. Install it with **npm install --save @ngrx/router -store**

2. Add it to the **app.module ts.**

```
import { StoreRouterConnectingModule } from '@ngrx/router-store';

import { AuthEffects } from './auth/store/auth.effects';
```

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    EffectsModule.forRoot([AuthEffects]),
    StoreRouterConnectingModule,

  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

334 Installing DevTools

1install with **npm install --save @ngrx/store-devtools**
2. Install **DevTools** chrome plugin
https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd
3. Include it in the **app.module.ts** but **only in production**!
-Import the StoreDevToolsModule + the enviroment module, and add it only if the the procution is true

```
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { NgModule } from '@angular/core';
import { StoreModule } from '@ngrx/store';
import { reducers } from './store/app.reducers';
import { EffectsModule } from '@ngrx/effects';
import { StoreRouterConnectingModule } from '@ngrx/router-store';
import { StoreDevtoolsModule } from '@ngrx/store-devtools';
import { environment } from '../environments/environment';

import { AuthEffects } from './auth/store/auth.effects';

@NgModule({
```

```
  declarations: [
    AppComponent
  ],
  imports: [
    StoreModule.forRoot(reducers),
    EffectsModule.forRoot([AuthEffects]),
    StoreRouterConnectingModule,
    !environment.production ? StoreDevtoolsModule.instrument() : []
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

335 Recipes ngrx + lazy Loading dynamic injections

1. Create a **recipe.reducers.ts**
- Import the Recipe + Ingredient object model
- This reducer has an inital state an array of recipes
. Important to export hte whoel defined interface of **FeatureState**

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

export interface FeatureState {
  recipes: State
}

export interface State {
  recipes: Recipe[];
}

const initialState: State = {
  recipes: [
    new Recipe(
      'Tasty Schnitzel',
```

```
      'A super-tasty Schnitzel - just awesome!',
      'https://upload.wikimedia.org/wikipedia/commons/7/72/Schnitzel.JPG',
      [
        new Ingredient('Meat', 1),
        new Ingredient('French Fries', 20)
      ]),
    new Recipe('Big Fat Burger',
      'What else you need to say?',
      'https://upload.wikimedia.org/wikipedia/commons/b/be/Burger_King_Angus_Bacon_%26_Cheese_Steak_Burger.jpg',
      [
        new Ingredient('Buns', 2),
        new Ingredient('Meat', 1)
      ])
  ]
};

export function recipeReducer(state = initialState, action: RecipeActions.RecipeActions) {
       return state;
  }
}
```

2. Adding to the **main module, of the feature**
recipes.module.ts
- Import StoreModule + RecipeReducer
- Usign **forFeature**(exoported property, exported funciton in the reducer)

```
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
import { CommonModule } from '@angular/common';
import { StoreModule } from '@ngrx/store';

import { RecipesComponent } from './recipes.component';
import { recipeReducer } from './store/recipe.reducers';
```

```
@NgModule({
  declarations: [
    RecipesComponent,
  ],
  imports: [
    CommonModule,
    StoreModule.forFeature('recipes', recipeReducer),
  ]
})
export class RecipesModule {}
```

339 Using two ngrx in one component

- SInce in our project we have multiple reducers + lazy loaded reducers we have to do the folowing if we want to use both of them in 1 component
1. Extend the root class with the lazay loaded reducer
recipes.reducers.ts

```
import * as fromApp from '../../store/app.reducers';


export interface FeatureState extends fromApp.AppState {
  recipes: State
}
```

2. Inject it as the lazyloaded feature since it has all of the properties attributes of the parent reducer AppState
recipe-detail.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params, Router } from '@angular/router';
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/take';
```

```
import * as ShoppingListActions from '../../shopping-list/store/shopping-list.actions';
import * as fromRecipe from '../store/recipe.reducers';
import * as RecipeActions from '../store/recipe.actions';

@Component({
  selector: 'app-recipe-detail',
  templateUrl: './recipe-detail.component.html',
  styleUrls: ['./recipe-detail.component.css']
})
export class RecipeDetailComponent implements OnInit {
  recipeState: Observable<fromRecipe.State>;
  id: number;

  constructor(private route: ActivatedRoute,
              private router: Router,
              private store: Store<fromRecipe.FeatureState>) {
  }

//Dispacth a calling actions for the recipes
  ngOnInit() {
    this.route.params
      .subscribe(
        (params: Params) => {
          this.id = +params['id'];
          this.recipeState = this.store.select('recipes');
        }
      );
  }

  onAddToShoppingList() {
    this.store.select('recipes')
```

```
      .take(1)
      .subscribe((recipeState: fromRecipe.State) => {
        this.store.dispatch(new ShoppingListActions.AddIngredients(
          recipeState.recipes[this.id].ingredients)
        );
      });

  }

  onEditRecipe() {
    this.router.navigate(['edit'], {relativeTo: this.route});
    // this.router.navigate(['../', this.id, 'edit'], {relativeTo: this.route});
  }

  onDeleteRecipe() {
    this.store.dispatch(new RecipeActions.DeleteRecipe(this.id));
    this.router.navigate(['/recipes']);
  }

}
```

341 Fetching and storing Data with effects

recipes.effects.ts
1. **Fetchrecipe:**
-we listen to the actions, then with **switchmap** without wrapping the modificaiton in an observalbe, since httpcleint will result and observable e execute the request,-Then on the returned observable we make some transformation with map, then return a dispatchable object of **SET_RECIPES**
2.StoreRecipe
-listen for **STORE_RECIPES** action then execute **withLatestFrom** which lets combine the two recieved observable from different actions
-then with switchMap we are executing the hhtp request
Note:
.switchMap(([action, state]) ---> action is coming from the first observable, state is coming from the second observable

```
import { Injectable } from '@angular/core';
import { Actions, Effect } from '@ngrx/effects';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/withLatestFrom';
import { HttpClient, HttpRequest } from '@angular/common/http';
import { Store } from '@ngrx/store';

import * as RecipeActions from '../store/recipe.actions';
import { Recipe } from '../recipe.model';
import * as fromRecipe from '../store/recipe.reducers';

@Injectable()
export class RecipeEffects {
  @Effect()
  recipeFetch = this.actions$
    .ofType(RecipeActions.FETCH_RECIPES)
    .switchMap((action: RecipeActions.FetchRecipes) => {
      return this.httpClient.get<Recipe[]>('https://ng-recipe-book-3adbb.firebaseio.com/recipe
s.json', {
        observe: 'body',
        responseType: 'json'
      })
    })
    .map(
      (recipes) => {
        console.log(recipes);
        for (let recipe of recipes) {
          if (!recipe['ingredients']) {
            recipe['ingredients'] = [];
          }
        }
        return {
```

```
          type: RecipeActions.SET_RECIPES,
          payload: recipes
        };
      }
    );


  @Effect({dispatch: false})
  recipeStore = this.actions$
    .ofType(RecipeActions.STORE_RECIPES)
    .withLatestFrom(this.store.select('recipes'))
    .switchMap(([action, state]) => {
      const req = new HttpRequest('PUT', 'https://ng-recipe-book-3adbb.firebaseio.com/recipes.j
son', state.recipes, {reportProgress: true});
      return this.httpClient.request(req);
    });


  constructor(private actions$: Actions,
              private httpClient: HttpClient,
              private store: Store<fromRecipe.FeatureState>) {}
}
```