## App Ingresos y Egresos

<https://github.com/jsfc2199/IngresosEgresosApp>

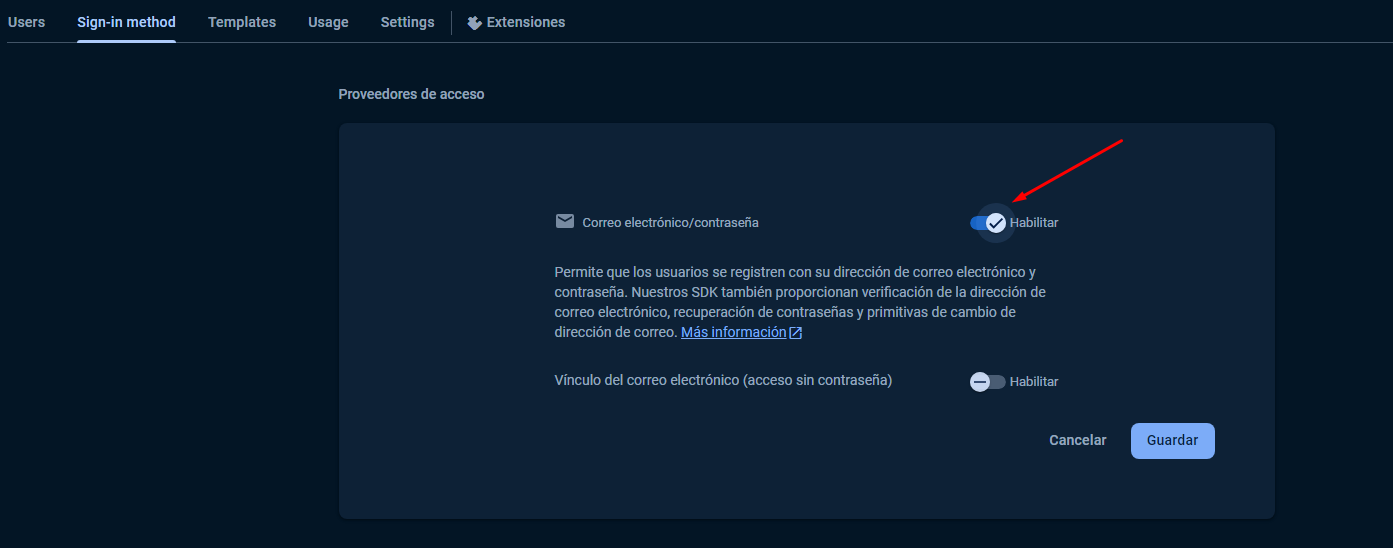
### Estilos y funcionalidad básica

En esta primera parte solo hicimos la organización de los templates, mas no nada de redux, por lo tanto para ver los cambios paso a paso se pueden revisar los commits con títulos “Estilos y Funcionalidad Básica”

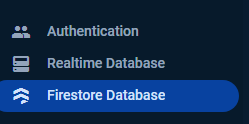
### Autenticación en login y registo

#### Prepración de la base de datos

Para esto usaremos firebase. Creamos un proyecto nuevo común y corriente, y luego en autenticación habilitamos correo electronico y contraseña



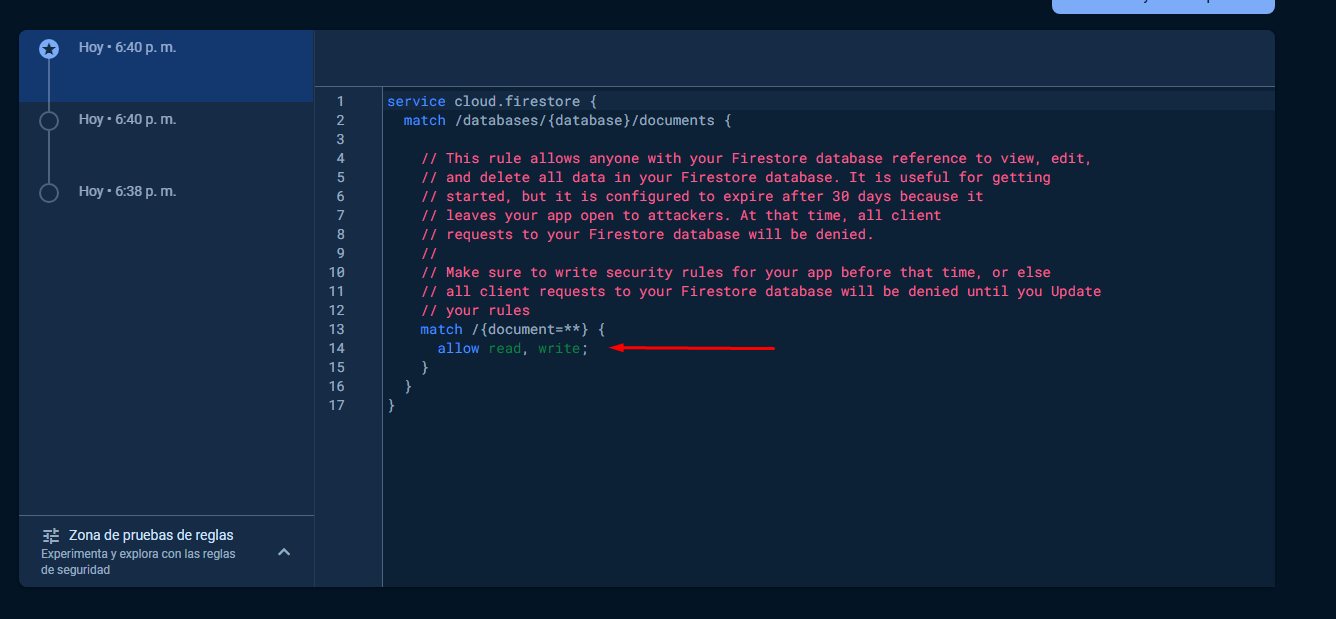
Para la base de datos usamos Firestore database



Iniciamos esta como formato de prueba



Modificamos la regla para que sea por todo el tiempo



#### AngularFire - Comunicar nuestra aplicación de Angular con Firebase

<https://github.com/angular/angularfire>

Solo es seguir la guia  
  
En caso de poner problema usar otra cuenta de gmail

### Crear nuestro primer usuario

Creamos un servicio que reciba el usuario, nombre y contrsaeña ya que tenemos el setup de firebase, y usando metodos del mismo podemos crear un usuario

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

import { Auth, createUserWithEmailAndPassword } from '@angular/fire/auth';

@Injectable({

  providedIn: 'root',

})

export class AuthService {

  constructor(private auth: Auth) {}

  crearUsuario(nombre: string, email: string, password: string) {

    return createUserWithEmailAndPassword(this.auth, email, password);

  }

}

Luego llamamos el servicio en el componente de registro y podemos redirigir al usuario si inició sesion correctamente

  constructor(

    private formBuilder: FormBuilder,

    private authService: AuthService,

    private router: Router

  ) {}

  crearUsuario() {

    if (this.registroForm.invalid) return;

    const { nombre, correo, password } = this.registroForm.value;

    this.authService

      .crearUsuario(nombre, correo, password)

      .then((credencales) => {

        console.log('credenciales', credencales);

        this.router.navigate(['./'])

      })

      .catch((err) => console.log(err));

  }

### Login de nuestra app

Es exactamente lo anterior solo que usando el metodo signInWithEmailAndPassword

### Logout

  logout(){

    return signOut(this.auth)

  }

  onLogout(){

    this.authService.logout().then( () =>{

      this.router.navigate(['/login'])

    })

### Obtener información del usuario activo en Firebase

La idea es tener too el estado del usuario de firebase

En el servicio podemos obtener el estado del usuario

  //nos avisa cuando haya cambios en el estado del usuario

  initAuthListener(){

    authState(this.auth).subscribe(user=>{

      console.log(user);

    })

  }

Luego el lugar mas conveniente de usar esto es en el app component ya que esucha todo a su alrededor

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

import { AuthService } from './services/auth.service';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'IngresoEgresoApp';

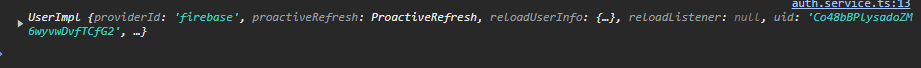
  constructor(private authService: AuthService){

    this.authService.initAuthListener()

  }

}

Al iniciar sesion con un usuario registrado tendremos lo siguiente



Guard - Proteger el dashboard y sus rutas hijas

Creamos en el servicio un metodo que verifique que el usuario de firebase no es nulo

 isAuth(){

    return authState(this.auth).pipe(

      map(firebaseUser => firebaseUser != null)

    )

  }

Lueo creamos un guardian que aplicará este servicio ya que devuelve un observable booleano. Para esto injectamos el servicio al guard, a su vez inyectamos el router porque queremos que el usuario navegue al login en caso de no estar autenticado

export const authGuard: CanActivateFn = (route, state) => {

  const router = inject(Router);

  return inject(AuthService)

    .isAuth()

    .pipe(

      //para es para disparar un efecto secundario

      tap((estado) => {

        if (!estado) router.navigate(['/login']);

      })

    );

};

### Crear documento de base de datos por cada usuario

Creamos un modelo de usuario

export class Usuario {

  constructor(

    public uid: string,

    public nombre: string,

    public email: string

  ) {}

}

### Redux en la aplicación

#### Interacciones del usuario UI State

Crearemos las acciones y el reducer

Instalamos el ngrx con ng add @ngrx/store@latest y luego configuramos las primeras acciones y el reducer

En el archivo ui.actions podemos tener lo siguiente

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

export const isLoading = createAction('[UI Component] Is Loading');

export const stopLoading = createAction('[UI Component] Stop Loading');

Y en el ui.reducer

import { Action, createReducer, on } from '@ngrx/store';

import { isLoading, stopLoading } from './ui.actions';

export interface State {

    isLoading: boolean;

}

export const initialState: State = {

  isLoading: false,

}

const \_counterReducer = createReducer(initialState,

    on(isLoading, *state* => ({ ...*state*, isLoading: true})),

    on(stopLoading, *state* => ({ ...*state*, isLoading: false})),

);

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

    return \_counterReducer(*state*, *action*);

}

#### Estado global Action Reducer Map

Instalamos las dev tools para poder ver nuestro estado en la app

ng add @ngrx/store-devtools@latest

Luego configuramos el app reducer

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

import \* as ui from './shared/ui.reducer';

export interface AppState {

   ui: ui.State

}

export const appReducers: ActionReducerMap<AppState> = {

   ui: ui.uiReducer,

}

#### Dispatch activar y desactivar loading

Para hacer el dispatch inyectamos el servicio en el constructor del login component en primera instancia

private *store*: Store<AppState>,

Luego en el ngOnInit podemos suscribirnos

  ngOnInit(): void {

     this.loginForm = this.formBuilder.group({

      correo: ['', [Validators.required, Validators.email]],

      password: ['', Validators.required],

    });

    this.subcription = this.store.select('ui').subscribe(*ui* => this.cargando = *ui*.isLoading)

  }

Vemos que tenemos una variable subcripscion para usarla en el onDestroy

 ngOnDestroy(){

    this.subcription.unsubscribe()

  }

Luego podemos hacer los dispatch de cambiar el estado a true o false

  iniciarSesion() {

    if (this.loginForm.invalid) return;

*//dispatch*

    this.store.dispatch(ui.isLoading()) *//se cambia el estado a true*

    const { correo, password } = this.loginForm.value;

    this.authService

      .loginUsuario(correo, password)

      .then((*credenciales*) => {

        console.log(*credenciales*);

*// Swal.close()*

        this.store.dispatch(ui.stopLoading()) *//se cambia el estado a false*

        this.router.navigate(['/']);

      })

      .catch((*err*) =>{

        this.store.dispatch(ui.stopLoading()) *//se cambia el estado a false*

        Swal.fire({

          title: 'Oops!',

          text: *err*.message,

          icon: 'error',

        })}

      );

  }

El mismo procedimiento se hace para el register component

#### Auth Actions y Reducer

Primero creamos la accion de setear y quitar un usuario activo en la pagina, el cual para ellos esperamos un usuario en el payload y para el otro no

import { createAction, props } from '@ngrx/store';

import { Usuario } from '../models/usuario.model';

export const setUser = createAction('[Auth Component] Set User', props<{user: Usuario}>());

export const unsetUser = createAction('[Auth Component] Set User');

Luego impleentamos el reducer

import { createReducer, on, Action } from '@ngrx/store';

import \* as auth from './auth.actions';

import { Usuario } from '../models/usuario.model';

export interface State {

    user: Usuario | null;

}

export const initialState: State = {

   user: null,

}

const \_authReducer = createReducer(initialState,

    on(auth.setUser, (*state*, {*user*}) => ({ ...*state*, user: {...*user*}})),

    on(auth.unsetUser, (*state*) => ({ ...*state*, user: null})),

);

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

    return \_authReducer(*state*, *action*);

}

Luego de esto alimentamos nuestro app reducer añadiendo el estado del usuario y su reducer

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

import \* as ui from './shared/ui.reducer';

import \* as auth from './auth/auth.reducer';

export interface AppState {

   ui: ui.State,

   user: auth.State

}

export const appReducers: ActionReducerMap<AppState> = {

   ui: ui.uiReducer,

   user: auth.authReducer

}

##### Almacenar en el store el usuario de firebase

Una vez hecho lo anterior podemos implementar en el servicio en el método **initAuthListener()** el manejo del estado de nuestro usuario

Firebase cuenta con un método llamado **onSnapshot** la cual es de tipo **Unsubscribe** donde esta es una interfaz. El métodopermite obtener actualizaciones en tiempo real, es decir, si modifico algo directamente desde la base de datos obtener inmediatamente esta actualización.

Esta función recibe dos argumentos **el path donde se encuentra nuestro documento** y **callback que ejecuta las actualizaciones**

Este callback se puede decir que es nuestra promesa, por ende allí tentremos la info específica de nuestro documento, es decir, si la definimos con “nombre”, “email” e “id” esto será lo que tengamos

Luego la funcion **onSnapshot,** tal como indica la documentación ***Returns An unsubscribe function that can be called to cancel the snapshot listener.***

Esto anterior implica que si tenemos una varaibla de tipo unsubcribe será a final de cuentas una función que permite desuscribirse de los cambios que se hagan en la base de datos para liberar memoria

De aquí entonces el código es el siguiente

 userSubscription!: Unsubscribe;

*//nos avisa cuando haya cambios en el estado del usuario*

  initAuthListener() {

    authState(this.auth).subscribe(async (*user*) => {

      if (*user*) {

*//se usa onSnapshot para que llegado el caso que se actualice la información desde la base de datos directamente se actualice inmediatamente en la app*

*//este metodo recibe dos argumentos, el path donde esta nuestro documento y un callback que tendrá la información que queremos*

        this.userSubscription = onSnapshot(

*//acceder al documento de firebase pasando como argumento la base de datos y el path del documento*

          doc(this.firestore, `${*user*.uid}/user`),

*//docData es la promesa que nos devuelve en el callback con la información perse del usuario*

          (*docData*) => {

*//Como docData es de tipo DocumentSnapshot se puede obtener la información de esta usando el método .data()*

            const docUserData = *docData*.data() as UsuarioDataFirebase;

*//se setea el usuario para pasarlo al dispatch*

            const newUser = Usuario.fromFirebase(docUserData);

*//dispatch dela accion que setea el usuario*

            this.store.dispatch(auth.setUser({ user: newUser }));

          }

        );

      } else {

*//Aquí se cancela la suscripción al onSnapshot*

*//Se hace de esta forma porque el onSnapshot retorna una function que puede ser llamada para cancelar la suscripcion tal como indica la documentación*

*//@returns*

*//An unsubscribe function that can be called to cancel the snapshot listener.*

*//Por lo anterior estamos preguntando si en la suscripción hay data, y si la hay ejecute la desuscripción a partir del retorno de la misma*

*//Por ello se ejecuta this.userSubscription() como si fuera un método*

        this.userSubscription ? this.userSubscription() : null

        this.store.dispatch(auth.unsetUser());

      }

    });

  }

Donde UsuarioDataFirebase esta definido de la siguiente forma

export class Usuario {

  static fromFirebase(*data*: UsuarioDataFirebase){

    return new Usuario(*data*.uid, *data*.nombre, *data*.email)

  }

  constructor(

    public *uid*: string,

    public *nombre*: string,

    public *email*: string

  ) {}

}

export interface UsuarioDataFirebase{

  nombre: string;

  email: string;

  uid: string;

}

#### Modelo de ingreso egreso

Se crea un modelo de ingreso egreso

export class IngresoEgreso {

  constructor(

    public *descripcion*: string,

    public *monto*: number,

    public *tipo*: string,

    public *uid*?: string

  ) {}

}

Luego está inforamción la usamos en el componente de IngresoEgreso para asociar el formulario

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

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({

  selector: 'app-ingreso-egreso',

  templateUrl: './ingreso-egreso.component.html',

})

export class IngresoEgresoComponent implements OnInit{

  ingresoEgresoForm: FormGroup = new FormGroup('')

  tipoMovimiento: string = 'ingreso'

  constructor(private *formBuilder*: FormBuilder){}

  ngOnInit(): void {

    this.ingresoEgresoForm = this.formBuilder.group({

      descripcion: ['', Validators.required],

      monto: ['', Validators.required]

    })

  }

  guardar(){

    if(this.ingresoEgresoForm.invalid) return

  }

}

Y esto lo asociacions a nuestro template como un form control name

#### Servicio Ingreso Egreso

Para este podemos obtener la información completa del usuario directamente desde el auth service y luego inyectando este servicio al ingreso egreso. Entonces en el auth service, en el listener del usuario

  private \_user!: Usuario | null

  get user(){

    return this.\_user

  }

Luego se asocia el serivcio de ingreso egreso

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

import { Firestore, collection, doc, setDoc } from '@angular/fire/firestore';

import { IngresoEgreso } from '../models/ingreso-egreso.model';

import { AuthService } from './auth.service';

@Injectable({

  providedIn: 'root'

})

export class IngresoEgresoService {

  constructor(

    private *firestore*: Firestore,

    private *authService*: AuthService

    ) { }

  crearIngresoEgreso(*ingresoEgreso*: IngresoEgreso){

    const uid = this.authService.user?.uid

*/\**

*Esta insercion se diferencia de la del auth service*

*En el auth serivce creabamos a partir del id la colección, para luego añador un documento a esta*

*Es decir idColeccion/user donde idColeccion es la colección y user es el documento*

*Aquí lo que queremos hacer es difente, queremos crear un documento ingreso-egreso, y a este añadirle una colección*

*es decir idColeccion/ingreso-egreso/items donde idColeccion es la colección y ingreso-egreso es el documento e items es la colección*

*final a la que añadiremos el nuevo documento*

*Se deja comentada la forma larga para entender el pasó a pasó de lo ocurrido*

*// Teniendo una referencia a la coleccion de usuarios por id*

*const colectionIngresoEgreso = collection(this.firestore, `${uid}`);*

*// se crea referencia al documento, es decir dentro del mismo uid quiero adicional el documento ingreso-egreso*

*const documentRef = doc(colectionIngresoEgreso, 'ingreso-egreso');*

*//se crea la coleccion items dentro del documento ingreso-egreso*

*const coleccionItems = collection(documentRef, 'items');*

*//se crean los documentos pero no se asinan por un nombre sino por un id genérico de firebase*

*const docItems = doc(coleccionItems);*

*setDoc(docItems, {...ingresoEgreso}).then(()=> console.log('documento insertado con exito', documentRef.id))*

*\*/*

*//Forma corta*

*//Como sabemos que items es una colección que queremos tener en el documento ingreso-egreso, asignamos el path directamante*

    const collectionIngresoEgreso = collection(this.firestore, `${uid}/ingreso-egreso/items`);

*//se crean los documentos dentro de la coleccion items pero no se asinan por un nombre sino por un id genérico de firebase*

    const documentRef = doc(collectionIngresoEgreso);

*//se setea el documento a la coleccion definida por id generico*

*//Nota Importante:*

*/\**

*Como comentamos el uid del modelo no es necesario pasarlo explicitamente, es decir, si fuera parte del modelo, debemos enviar { ...ingresoEgreso, uid }, es decir*

*setDoc(documentRef, { ...ingresoEgreso, uid })*

*No es necesario hacer un then para setDoc porque este devuelve un Promise<void>, en caso de requerir informacion se puede hacer de la siguiente manera*

*setDoc(documentRef, { ...ingresoEgreso }).then(()=> console.log('documento insertado con exito', documentRef))*

*\*/*

    return setDoc(documentRef, { ...ingresoEgreso })

  }

}

Luego en el componente en la función de guardar podemos hacer lo que requeramos

  guardar(){

    if(this.ingresoEgresoForm.invalid) return

    const {descripcion, monto} = this.ingresoEgresoForm.value

    const ingresoEgreso = new IngresoEgreso(descripcion, monto, this.tipoMovimiento)

    this.ingresoEresoService.crearIngresoEgreso(ingresoEgreso).then(()=> {

      this.ingresoEgresoForm.reset()

      Swal.fire('Registro creado', descripcion, 'success')

    }).catch(*e* => {

      Swal.fire('Hubo un error registrando la data', e.message, 'error')

    })

  }

#### Acciones y reducer

E crean las acciones

import { createAction, props } from '@ngrx/store';

import { IngresoEgreso } from '../models/ingreso-egreso.model';

export const setItems = createAction('[IngresoEgreso] setItems', props<{items: IngresoEgreso[]}>());

export const unSetItems = createAction('[IngresoEgreso] UnsetItems');

Se crea el reducer

import { Action, createReducer, on } from '@ngrx/store';

import { IngresoEgreso } from '../models/ingreso-egreso.model';

import { setItems } from './ingreso-egreso.actions';

import { unsetUser } from '../auth/auth.actions';

export interface State {

    items: IngresoEgreso[];

}

export const initialState: State = {

   items: [],

}

const \_ingresoEgresoReducer = createReducer(initialState,

    on(setItems, (*state*, {*items*}) => ({ ...state, items: [...items]})),

    on(unsetUser, (*state*) => ({ ...state, items: []})),

);

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

    return \_ingresoEgresoReducer(state, action);

}

Y se actaliza el app reducer

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

import \* as ui from './shared/ui.reducer';

import \* as auth from './auth/auth.reducer';

import \* as ingresoEgreso from './ingreso-egreso/ingreso-egreso-reducer';

export interface AppState {

   ui: ui.State,

   user: auth.State,

   ingresosEgresos: ingresoEgreso.State

}

export const appReducers: ActionReducerMap<AppState> = {

   ui: ui.uiReducer,

   user: auth.authReducer,

   ingresosEgresos: ingresoEgreso.ingresoEgresoReducer

}

#### Cargar items de firebase y obtener su uid

En el servicio creamos un segundo metodo para obtener los items que llevan los montos de ingreso y egreso

  initIngresosEgressosListener(*uid*: string){

*/\**

*tal como se hizo con el usuario con el onSnapshot para escuchar los cambios si se hacen directos en la base de datos, el collectionSnapshosts nos permite hacer lo mismo*

*pero ya para una colección, este retorna un observable al cual podemos suscribirnos*

*collectionSnapshots recibe como argumento la ubicación de la coleccion en firebase*

*Dado que es un observable ya a este podemos usar los operadores de rxjs. Usamos el pipe para pasar la información por el flujo que requeeramos*

*Primero se usa el map para obtener como tal la coleccion items en firebase.*

*Una vez tengamos la coleccion podemos mapear esta información, es decir, mapearemos los documentos dentro de items*

*Dado que por cada documento dentro de items podemos conocer todos sus datos podemos simplemente retornar un objeto como queramos, en este caso uno con el uid del documento*

*dentro de firebase y aparte la data del documento como dal es decir el monto, la descipcion y el tipo*

*\*/*

    this.collectionSubscription = collectionSnapshots(collection(this.firestore, `${uid}/ingreso-egreso/items`))

    .pipe(

      map(*collectionItems* => {

          return collectionItems.map(*documentoItem* => {

            const documentData = documentoItem.data() as itemsData

            return {

              uid: documentoItem.id,

              ...documentData

            }

          })

        }))

        .subscribe(*resp* => console.log(resp))

    return this.collectionSubscription

  }

  ngOnDestroy(): void {

    this.collectionSubscription.unsubscribe()

  }

}

interface itemsData {

  descripcion: string;

  monto: number;

  tipo: string;

}

Luego esta información se usa en el dashboard por lo tanto hacemos la llamada al metodo allí

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

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

import { AppState } from '../app.reducer';

import { Subscription, filter } from 'rxjs';

import { IngresoEgresoService } from '../services/ingreso-egreso.service';

@Component({

  selector: 'app-dashboard',

  templateUrl: './dashboard.component.html',

})

export class DashboardComponent implements OnInit, OnDestroy {

  userSubscription: Subscription = new Subscription()

  constructor(private *store*: Store<AppState>, private *ingresosEgresosService*: IngresoEgresoService){}

  ngOnInit(): void {

    this.userSubscription = this.store.select('user').pipe(

      filter(*auth* => auth.user != null)

    ).subscribe(({*user*}) => {

      this.ingresosEgresosService.initIngresosEgressosListener(user!.uid)

    })

  }

  ngOnDestroy(): void {

    this.userSubscription.unsubscribe()

  }

}

#### Disptch de las acciones de items

Se modifica el auth service par ahacer el unset de los items al cerrar sesion

else {

*//Aquí se cancela la suscripción al onSnapshot*

*//Se hace de esta forma porque el onSnapshot retorna una function que puede ser llamada para cancelar la suscripcion tal como indica la documentación*

*//@returns*

*//An unsubscribe function that can be called to cancel the snapshot listener.*

*//Por lo anterior estamos preguntando si en la suscripción hay data, y si la hay ejecute la desuscripción a partir del retorno de la misma*

*//Por ello se ejecuta this.userSubscription() como si fuera un método*

        this.\_user = null

*//limpiamos los items*

        this.store.dispatch(ingresoEgreso.unSetItems())

        this.userSubscription ? this.userSubscription() : null

        this.store.dispatch(auth.unsetUser());

      }

Y en el detalle component por el contrario nos suscribimos para tener la data de los items

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

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

import { Subscription } from 'rxjs';

import { AppState } from 'src/app/app.reducer';

import { IngresoEgreso } from 'src/app/models/ingreso-egreso.model';

@Component({

  selector: 'app-detalle',

  templateUrl: './detalle.component.html',

})

export class DetalleComponent implements OnInit, OnDestroy {

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

  ingresoEgresoItemsSubs: Subscription = new Subscription();

  ingresosEgresosItems: IngresoEgreso[] = [];

  ngOnInit(): void {

    this.ingresoEgresoItemsSubs = this.store

      .select('ingresosEgresos')

      .subscribe(({ *items* }) => {

        this.ingresosEgresosItems = items;

      });

  }

  borrar(*uid*: string) {

    console.log(uid);

  }

  ngOnDestroy(): void {

    this.ingresoEgresoItemsSubs.unsubscribe();

  }

}

#### Eliminar items de firebase

Nos vamos al servicio de ingreso egreso y creamos un metodo que elimine la informacion del item en firebase

  borrarIngresoEgreso(*uid*: string){

    const userUid = this.authService.user?.uid

*//nos ubicamos en el documento relacionado a cada item*

    const itemDoc = doc(this.firestore, `${userUid}/ingreso-egreso/items/${uid}`)

*//usamos el deleteDoc para borrar el documento que queremos*

    return deleteDoc(itemDoc)

  }

Luego consumimos el servicio

  borrar(*uid*: string) {

    this.ingresoEgresoService.borrarIngresoEgreso(uid).then(()=>{

      Swal.fire({

        icon: 'success',

        title: 'Item eliminado',

        text: `Item eliminado: ${uid}`,

        footer: 'Success'

      });

    }).catch(*error* => {

      Swal.fire({

        icon: 'error',

        title: error.name,

        text: error.message,

        footer: error.code

      });

    })

  }

#### Estadistica básica

Para que funcione debemos primero eliminar la propiedad de uid desde el servicio de ingreso egreso al crear un ingreso

crearIngresoEgreso(*ingresoEgreso*: IngresoEgreso){

    const uid = this.authService.user?.uid

    const collectionIngresoEgreso = collection(this.firestore, `${uid}/ingreso-egreso/items`);

    const documentRef = doc(collectionIngresoEgreso);

    delete ingresoEgreso.uid

    return setDoc(documentRef, { ...ingresoEgreso })

  }

Luego podemos crear la estadistica

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

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

import { AppState } from 'src/app/app.reducer';

import { IngresoEgreso } from 'src/app/models/ingreso-egreso.model';

@Component({

  selector: 'app-estadistica',

  templateUrl: './estadistica.component.html',

})

export class EstadisticaComponent implements OnInit{

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

  ingresos: number = 0

  egresos: number = 0

  totalIngresos: number = 0

  totalEgresos: number = 0

  ngOnInit(): void {

    this.store.select('ingresosEgresos').subscribe(({*items*}) => {

      this.generarEstadistica(items)

    })

  }

  generarEstadistica(*items*:IngresoEgreso[]){

    for (const item of items) {

      if(item.tipo === 'ingreso'){

        this.totalIngresos += item.monto;

        this.ingresos++

      }else{

        this.totalEgresos += item.monto;

        this.egresos++

      }

    }

  }

}

#### Graficas angular

<https://valor-software.com/ng2-charts/#GeneralInfo>

Usamos la siguiente libreria ng2-charts

npm install --save ng2-charts

npm install --save chart.js

Luego usando el temaplte que ofrece la pagina podemos usar todo lo que requerimos

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

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

import { ChartData, ChartType } from 'chart.js';

import { BaseChartDirective } from 'ng2-charts';

import { AppState } from 'src/app/app.reducer';

import { IngresoEgreso } from 'src/app/models/ingreso-egreso.model';

@Component({

  selector: 'app-estadistica',

  templateUrl: './estadistica.component.html',

})

export class EstadisticaComponent implements OnInit{

  @ViewChild(BaseChartDirective) chart: BaseChartDirective | undefined;

  public doughnutChartLabels: string[] = [

    'Ingresos',

    'Egresos',

  ];

  public doughnutChartData: ChartData<'doughnut'> = {

    labels: this.doughnutChartLabels,

    datasets: [

      { data: [] },

    ],

  };

  public doughnutChartType: ChartType = 'doughnut';

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

  ingresos: number = 0

  egresos: number = 0

  totalIngresos: number = 0

  totalEgresos: number = 0

  ngOnInit(): void {

    this.store.select('ingresosEgresos').subscribe(({*items*}) => {

      this.generarEstadistica(*items*)

    })

  }

  generarEstadistica(*items*:IngresoEgreso[]){

    for (const item of *items*) {

      if(item.tipo === 'ingreso'){

        this.totalIngresos += item.monto;

        this.ingresos++

      }else{

        this.totalEgresos += item.monto;

        this.egresos++

      }

    }

    this.doughnutChartData.datasets = [{

      data: [this.totalIngresos, this.totalEgresos]

    }]

    this.chart?.chart?.update();

  }

}

### NgRx con LazyLoad

#### Modulo Auth

Usamos ng g m auth/auth --flat, con el --flat lo obligamos a crear el modulo en ese directorio

Se crea el modulo con lo que necesitamos como el rauter module y los formularios reactivos

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

import { CommonModule } from '@angular/common';

import { LoginComponent } from './login/login.component';

import { RegisterComponent } from './register/register.component';

import { ReactiveFormsModule } from '@angular/forms';

import { RouterModule } from '@angular/router';

@NgModule({

  declarations: [

    LoginComponent,

    RegisterComponent,

  ],

  imports: [

    CommonModule,

    ReactiveFormsModule,

    RouterModule

  ],

})

export class AuthModule { }

Ya solo lo importamos en el app module

#### SharedModule

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

import { CommonModule } from '@angular/common';

import { NavbarComponent } from './navbar/navbar.component';

import { SidebarComponent } from './sidebar/sidebar.component';

import { FooterComponent } from './footer/footer.component';

import { RouterModule } from '@angular/router';

@NgModule({

  declarations: [

    NavbarComponent,

    SidebarComponent,

    FooterComponent

  ],

  imports: [

    CommonModule,

    RouterModule

  ],

*//se añaden los xports porque en este caso el que usa estos componentes es el dashboard y no el app module directamente*

  exports:[

    NavbarComponent,

    SidebarComponent,

    FooterComponent

  ]

})

export class SharedModule { }

#### IngresoEgresoModule

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

import { CommonModule } from '@angular/common';

import { DashboardComponent } from '../dashboard/dashboard.component';

import { OrdenIngresoPipe } from '../pipes/orden-ingreso.pipe';

import { DetalleComponent } from './detalle/detalle.component';

import { EstadisticaComponent } from './estadistica/estadistica.component';

import { IngresoEgresoComponent } from './ingreso-egreso.component';

import { ReactiveFormsModule } from '@angular/forms';

import { NgChartsModule } from 'ng2-charts';

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

@NgModule({

  declarations: [

    DashboardComponent,

    IngresoEgresoComponent,

    EstadisticaComponent,

    DetalleComponent,

    OrdenIngresoPipe,

  ],

  imports: [

    CommonModule,

    ReactiveFormsModule,

    NgChartsModule,

    SharedModule *//Se importa el shared module porque ingreso egreso usa estos elemtos*

  ]

})

export class IngresoEgresoModule { }

#### Dashboard Routing Module

Como tenemos rutas que solo aplican para el dashaborad también podemos tenerlas en su propio modulo

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

import { DashboardComponent } from './dashboard.component';

import { dashBoardRoutes } from './dashboard.routes';

import { RouterModule, Routes } from '@angular/router';

const rutasHijas: Routes = [{

  path: '',

  component: DashboardComponent,

  children: dashBoardRoutes,

*// canActivate: [authGuard], se comenta porque se debe añadir al guardian que no debe tampoco cargar el modulo usando el lazy*

}];

@NgModule({

  declarations: [],

  imports: [

    RouterModule.forChild(rutasHijas)

  ],

  exports:[

    RouterModule

  ]

})

export class DashboardRoutesModule {}

Luego en el modulo de ingreso egreso podemos llamar a este modulo de rutas

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

import { CommonModule } from '@angular/common';

import { DashboardComponent } from '../dashboard/dashboard.component';

import { OrdenIngresoPipe } from '../pipes/orden-ingreso.pipe';

import { DetalleComponent } from './detalle/detalle.component';

import { EstadisticaComponent } from './estadistica/estadistica.component';

import { IngresoEgresoComponent } from './ingreso-egreso.component';

import { ReactiveFormsModule } from '@angular/forms';

import { NgChartsModule } from 'ng2-charts';

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

import { DashboardRoutesModule } from '../dashboard/dashboard-routes.module';

@NgModule({

  declarations: [

    DashboardComponent,

    IngresoEgresoComponent,

    EstadisticaComponent,

    DetalleComponent,

    OrdenIngresoPipe,

  ],

  imports: [

    CommonModule,

    ReactiveFormsModule,

    NgChartsModule,

    SharedModule, *//Se importa el shared module porque ingreso egreso usa estos elemtos*

    DashboardRoutesModule

  ]

})

export class IngresoEgresoModule { }

#### Lazy Load del modulo ingreso egreso

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

import { RouterModule, Routes } from '@angular/router';

import { LoginComponent } from './auth/login/login.component';

import { RegisterComponent } from './auth/register/register.component';

const routes: Routes = [

  { path: 'login', component: LoginComponent },

  { path: 'register', component: RegisterComponent },

  {

    path: '',

    loadChildren:() => import('./ingreso-egreso/ingreso-egreso.module')

                          .then(*m* => *m*.IngresoEgresoModule)

  },

  { path: '\*\*', redirectTo: '' },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule],

})

export class AppRoutingModule {}

#### CanMatch

Queremos prevenir que el usuario si no esta autenticado no puedra entrar a la app.

No podemos utilizar el guardian que tenemos porque aunque nos protege de las rutas, este cargará el modulo de ingreso egreso, y la idea es que solo lo cargue cuando nos autentiquemos

En el guardian creamos un metodo pero que implmente CanMatchFn

export const authMatch: CanMatchFn = (*route*, *state*) => {

  const router = inject(Router);

  return inject(AuthService)

    .isAuth()

    .pipe(

*//para es para disparar un efecto secundario*

      tap((*estado*) => {

        if (!*estado*) router.navigate(['/login']);

      }),

      take(1),*//cada que ingreso al modulo me suscribo a esto pero con el take 1 a su vez cancelo la suscripción*

    );

};

Uego en el app rputing que es donde debemos implementar que nuestro modulo tenga carga peresoza usamos el canMatch

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

import { RouterModule, Routes } from '@angular/router';

import { LoginComponent } from './auth/login/login.component';

import { RegisterComponent } from './auth/register/register.component';

import { authMatch } from './services/auth.guard';

const routes: Routes = [

  { path: 'login', component: LoginComponent },

  { path: 'register', component: RegisterComponent },

  {

    path: '',

    canMatch: [authMatch], *//para implementar el guardian que prevenga la carga del modulo usamos es canMatch*

    loadChildren:() => import('./ingreso-egreso/ingreso-egreso.module')

                          .then(*m* => *m*.IngresoEgresoModule)

  },

  { path: '\*\*', redirectTo: '' },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule],

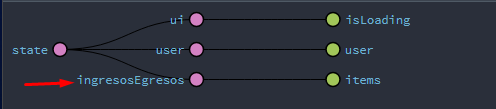
})

export class AppRoutingModule {}

#### Lazy Load Store

Esto lo hacemos también para que el store de nuestro estado en ngrx aplique lazy load

Lo que queremos es que el nodo de ingresosEgresos se cargue con lazy load



Lo primero es quitar de nuestro AppState, el global la propuiedad para cargarla con lazyLoad

Luego en el modulo de ingreso egreso indicamos que queremos cargar con lazy load en el store usando el forFeature

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

import { CommonModule } from '@angular/common';

import { DashboardComponent } from '../dashboard/dashboard.component';

import { OrdenIngresoPipe } from '../pipes/orden-ingreso.pipe';

import { DetalleComponent } from './detalle/detalle.component';

import { EstadisticaComponent } from './estadistica/estadistica.component';

import { IngresoEgresoComponent } from './ingreso-egreso.component';

import { ReactiveFormsModule } from '@angular/forms';

import { NgChartsModule } from 'ng2-charts';

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

import { DashboardRoutesModule } from '../dashboard/dashboard-routes.module';

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

import { ingresoEgresoReducer } from './ingreso-egreso-reducer';

@NgModule({

  declarations: [

    DashboardComponent,

    IngresoEgresoComponent,

    EstadisticaComponent,

    DetalleComponent,

    OrdenIngresoPipe,

  ],

  imports: [

    CommonModule,

    ReactiveFormsModule,

    NgChartsModule,

    SharedModule, *//Se importa el shared module porque ingreso egreso usa estos elemtos*

    DashboardRoutesModule,

    StoreModule.forFeature('ingresosEgresos', ingresoEgresoReducer)

  ]

})

export class IngresoEgresoModule { }

Luego sexxiones de nuestra app fallarán porque estan usando el AppState y este no cuenta con ingresosEgreso, entonces en el reducer de ingresoEgreso podemos crear una nueva interaz que exteienda de la app global

import { Action, createReducer, on } from '@ngrx/store';

import { IngresoEgreso } from '../models/ingreso-egreso.model';

import { setItems } from './ingreso-egreso.actions';

import { unsetUser } from '../auth/auth.actions';

import { AppState } from '../app.reducer';

export interface State {

    items: IngresoEgreso[];

}

export interface AppStateWithIngresoEgreso extends AppState{

  ingresosEgresos: State

}

export const initialState: State = {

   items: [],

}

const \_ingresoEgresoReducer = createReducer(initialState,

    on(setItems, (*state*, {*items*}) => ({ ...*state*, items: [...*items*]})),

    on(unsetUser, (*state*) => ({ ...*state*, items: []})),

);

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

    return \_ingresoEgresoReducer(*state*, *action*);

}

Luego en los componentes detalle y estadistica que requieren esto, ya no usaran el store del AppState sino del AppStateWithIngresoEgreso

constructor(private *store*: Store<AppStateWithIngresoEgreso>) {}

Con solo tenemos un reducer cargado de forma lazy, si queremos cargar mas de un reducer con lazy load podemos seguir aqui

<https://stackoverflow.com/questions/47052282/ngrx-store-combine-multiple-reducers-from-feature-module>

Otra forma es crearlos comun y corriente como si fuera n state global pero para una seccion dedicada igual usando el forFeature

### Desplegar a firebase

<https://firebase.google.com/docs/hosting/quickstart?hl=es>