* MaterialModule manages the material imports, and keeps the code clean. It exports what it imports, so the AppModule can have access (and the whole app) to the Material Modules.
* To create a component and assigned to a specific module (if there are more than one, like MaterialModule and AppModule), use ng g c auth/signup --module app.module
* Instead of adding classes like boostrap, directives are used to add those classes
* <mat-hint align="end"> {{pwInput.value?.length}} / 6</mat-hint>
* The ? mark is to apply length if it exists.
* Forms: errors take the place of hints if both are to be displayed
* <mat-error \*ngIf="!email.hasError('required')">Invalid email</mat-error>
* Use hasError() method, instead of email.errors.required, <https://angular.io/api/forms/AbstractControl#hasError>
* fxFlex, makes the flex item to take all the available space (like a block element), instead of what is needed. It was used in the toolbar, to move links to the right.
* Applying flex with column direction, makes the elements take only the width they need
* To center elements in the select dropdown of new training, I used fxLayoutAlign="center"
* Applied to children, and not to the flex container as usual. The same centering result can be achieve applying the text-center property to the form
* Try to modify user in the recipe app, because the user object was returned and not the clone
* In mat-select, there’s no need to use a form, and use 2 way data binding, but I loose the error display…
* <mat-select [(value)]="selected">
* <mat-option \*ngFor="let exercise of exercises" [value]="exercise.id">
* {{ exercise.name }}
* </mat-option>
* </mat-select>
* Nunca es Bueno tener dos lugares donde se maneja un estado, como el currentExercise. It’s better to rely on the service an pick up the state from there.

Working with event.target const filterValue = (event.target as HTMLInputElement).value;

<div fxLayoutAlign="center center"> in past-trainings component, it’s not a flex container, but fxLayoutAlign is applied???????

    <mat-form-field  fxFlex="40%">

        <mat-label>Filter</mat-label>

        <input matInput (keyup)="applyFilter($event)" placeholder="Filter by exercise">

    </mat-form-field>

</div>

* Services don’t use ngOnInit(), because are not renderer like components, they have just the constructor. It doesn’t work to call the availableExercises from the constructor, because if it takes long and a the array is needed, an empty array is returned in fetchAvailableExercises()
* If there’s a subscription in the service to a firebase observable, it’s automatically unsubscribed, like the angular built in ones.

Intead of emiting events to the app upon successful auth and logout, there’s the firebase auth observable that reacts to user auth state changes:  initAuthListener() {

    this.initAuthListenerSubs = this.fireAuth.authState.subscribe(user => {

      if (user) {

        this.isAuthenticated = true;

        this.authChange.next(true);

        this.router.navigate(['/training']);

      } else {

        this.user = null;

        this.authChange.next(false);

      }

    })

  }

The reload button should be of type=’button’, otherwise, it submits the form: <mat-card-actions fxLayoutAlign="center">

                <mat-spinner \*ngIf="isLoading && !showButton"></mat-spinner>

                <button  \*ngIf="!isLoading && !showButton" mat-raised-button type="submit" color="primary" [disabled]=!f.valid>Start</button>

                <button \*ngIf="showButton" mat-raised-button color="primary" type="button" (click)="onReload()">Reload</button>

            </mat-card-actions>

* Modules needed in e.g auth.module.ts, should be imported there, no matter if auth module is then imported in app.module, because each module works standalone. It doesn’t matter if the same modules are imported in other modules (like, material.module), angular doesn’t increase the bundle size.
* Routing setup without lazyloading:

// AppRoutingModule

const routes: Routes = [

  { path: '', component: WelcomeComponent },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

// AppModule

@NgModule({

  declarations: [

    AppComponent,

    WelcomeComponent,

    HeaderComponent,

    SidenavListComponent

  ],

  imports: [

    BrowserModule,

    BrowserAnimationsModule,

    AppRoutingModule,

    AngularFireModule.initializeApp(environment.firebase),

    SharedModule,

    AuthModule,

    TrainingModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

//SharedModule

@NgModule({

  declarations: [],

  imports: [

    CommonModule,

    FormsModule,

    MaterialModule,

    FlexLayoutModule,

  ],

  exports: [

    CommonModule,

    FormsModule,

    MaterialModule,

    FlexLayoutModule,

  ]

})

export class SharedModule { }

//AuthRoutingModule

const authRoutes: Routes = [

    { path: 'signup', component: SignupComponent },

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

]

@NgModule({

    imports:[

        [RouterModule.forChild(authRoutes)]

    ],

    exports: [

        [RouterModule]

    ]

})

export class AuthRoutingModule{ }

// AuthModule

@NgModule({

    declarations: [

        LoginComponent,

        SignupComponent

    ],

    imports: [

        SharedModule,

        ReactiveFormsModule,

        AngularFireAuthModule,

        // AngularFireAuthModule is used in a service, so it could be instead imported in app.module because all services are injected at root

        AuthRoutingModule

    ]

})

export class AuthModule { }

// TrainingModule

@NgModule({

    declarations: [

        TrainingComponent,

        CurrentTrainingComponent,

        NewTrainingComponent,

        PastTrainingsComponent,

        StopTrainingComponent

    ],

    imports: [

        RouterModule.forChild([

            { path: 'training', canActivate: [AuthGuardService], component: TrainingComponent }

        ]),

        SharedModule,

        AngularFirestoreModule

        //the scope if AngularFirestoreModule is all services provided in root and the components inside this module

        //if AngularFirestoreModule intended to use in another component, must be imported in the respective module

    ]

})

export class TrainingModule  { }

* Summary: app.module imports each feature module. So all the feature modules are loaded eagerly, when the app loads.
* Pay attention to Modules that are used in services, like AngularFireAuthModule in a specific module, because services are injected at root, and may be used across modules

To add Lazyloading, remove the TrainigModule from the app.module. The setup is:

// AppRoutingModule

const routes: Routes = [

  { path: '', component: WelcomeComponent },

  { path: 'training', loadChildren: () => import('./training/training.module').then(m => m.TrainingModule), canLoad:[AuthGuardService]  },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

//TrainingModule

@NgModule({

    declarations: [

        TrainingComponent,

        CurrentTrainingComponent,

        NewTrainingComponent,

        PastTrainingsComponent,

        StopTrainingComponent

    ],

    imports: [

        RouterModule.forChild([

            { path: '', component: TrainingComponent }

        ]),

        SharedModule,

        AngularFirestoreModule

        //the scope if AngularFirestoreModule is all services provided in root and the components inside this module

        //if AngularFirestoreModule intended to use in another component, must be imported in the respective module

    ]

})

export class TrainingModule  { }

* Move AuthGuard outside the lazyloaded module (training), because even if the guard blocks the user reaching ‘/training’, the module is always loaded.

NGRX

Assigning an argument to use it as default value: export function appReducer (state = initialState, action) {

    return state;

}

* An Action is always an object {type: ‘actionString’}, that’s why action.type is used in the switch statement in the reducers switch(action.type)
* Variables controlled by ngrx are named: variable$, with dolar sign at the end.

Using strings for actions, is error prone, because it’s used in the reducers and when dispatching actions:

switch(action.type) {

        case 'START\_LOADING':

this.store.dispatch({type: 'START\_LOADING'})

It’s better to save the strings in constants like this: export const START\_LOADING = '[UI] Start Loading';

export const STOP\_LOADING = '[UI] Stop Loading';

import \* as UIActions from './ui.actions';

 case UIActions.START\_LOADING:

* But just importing constants doesn’t help with typescript and autocompletion

Always use the spread operator in the reducers: export function uiReducer(state = initialState, action: UiActions.UiActions) {

    switch(action.type) {

        case UiActions.START\_LOADING:

            return {

                ...state,

                isLoading: true

            }

        case UiActions.STOP\_LOADING:

            return {

                ...state,

                isLoading: false

            }

        default:

            return {

                ...state

            }

    }

}

* Value != null covers the undefined as well, less strict!
* TO DO: change showButton logic in services project new-training component
* Delete value from new-exercise component
* To avoid subscriptions to data that need to do specific operations, like add it to a database, use take(1) before subscription, it’s automatically unsubscribed after getting the first result.
* Subscriptions to the store are automatically unsubscribed.