**3. Reactive Component Interaction**

14. Reactive Component Interaction using Custom Observables and Behavior Subject

ობზერვებლის დეკლარირების საუკეთესო გზაა, შევქმნათ საბჯექტი და ასევე ცვლადი, რომელიც ობზერვებლის ტიპის იქნება, და ეს ორი ერთანეთს გავუტოლოთ, რადგან საბჯექტის დაემიტება სხვა კომპონენტიდან არ მოხდეს (ფრაივეთი იქნება) და ასევე, ცვლადის დასუბსქრაიბება შეიძლებოდეს და იმის გაგება, საბჯექტში რა წერია:

1. private loadingSubject = new BehaviorSubject<boolean>(false);
3. loading$:Observable<boolean> = this.loadingSubject.asObservable()
5. loadingOn(){
6. this.loadingSubject.next(true)
7. }

**3. Reactive Component Interaction**

14. Reactive Component Interaction using Custom Observables and Behavior Subject

**finalize()** ოპერატორი გამოიძახება მაშინ, როცა ობზერვებლი დაკომპლიტდება ან დაერორდება, ამიტომაც კარგი გზაა მანდ ლოადერის ლოგიკის ჩასმა:

1. finalize(() => {
2. this.loadingService.loadingOff()
3. })

**3. Reactive Component Interaction**

15. Loading Indication Service - Reactive Implementation Finished

მეორე და უკეთესი გზაა, თუ სერვისში შევქმნით ფუნქციას, რომელიც არგუმენტად მიიღებს ობზერვებლს და დააბრუნებს ახალ ობზერვებლს **of()**-ის გამოყენებით, რომელსაც არგუმენტად ექნება null, ხოლო მასზე მივაბამთ პაიპს და შემდეგ tap() ოპერატორს, რომელიც ლოადინგს ჩართავს, ხოლო შენდეგ **concatMap()**-ს, რომელიც სხვა ობზერვებლს გამოიძახებს და მას არგუმენტად ჩავაწვდით ამ ფუნქციის მიერ მიღებულ ობზერვებლს და მესამე ოპერატორად ჩაეწოდება**finilize(),** რომელიც გათიშავს ლოადინგს.

ეს კონკრეტული ფუნქცია კომპონენტში გამოიძახება და სხვა ცვლადებს გაუტოლდება, რამდენადაც იგივე ობზერვებლს დააბრუნებს, რასაც აქამდე სხვა აბრუნებდა:

**servise.ts:**

1. private loadingSubject = new BehaviorSubject<boolean>(false);
2. loading$:Observable<boolean> = this.loadingSubject.asObservable();
4. showLoaderUntilCompleted<T>(obs$: Observable<T>): Observable<T>{
5. return of(null)
6. .pipe(
7. tap(() => this.loadingOn()),
8. concatMap(() => obs$),
9. finalize(() => this.loadingOff())
10. )
11. }

**3. Reactive Component Interaction**

15. Loading Indication Service - Reactive Implementation Finished

**component.ts:**

1. realoadCourses() {
2. const courses$ = this.coursesService
3. .loadAllCourses()
4. .pipe(
5. map((courses) => courses.sort(sortCoursesBySeqNo)),
6. );
8. const loadCourses$ = this.loadingService.showLoaderUntilCompleted(courses$)
10. this.beginnerCourses$ = loadCourses$.pipe(
11. map((courses) =>
12. courses.filter((course) => course.category == "BEGINNER")
13. )
14. );
16. this.advancedCourses$ = loadCourses$.pipe(
17. map((courses) =>
18. courses.filter((course) => course.category == "ADVANCED")
19. )
20. );
21. }

**3. Reactive Component Interaction**

18. Error Handling with the catchError RxJs operator

ერორის ჰენდლინგისას საურველია გამოვიყენოთ ახალი ერორის ობზერვებლის შექმნის მეთდი, რომელიც ჯერ **catchError(err)**-ს გამოიყენებს, ხოლო აღნიშნული დააბრუნებს **throwError(err)**-ს, რომელიც თავად ობზერვებლია, თუმცა ეგრევე ერორს აგდებს და აჩერებს:

1. const courses$ = this.coursesService
2. .loadAllCourses()
3. .pipe(
4. map((courses) => courses.sort(sortCoursesBySeqNo)),
5. catchError(err => {
6. const message = 'Could not load courses';
7. this.messagesService.showErrors(message);
8. console.log(message, err);
9. return throwError(err)
10. })
11. );

**3. Reactive Component Interaction**

24. Store Optimistic Data Modification Operations - API Design

**Partial<ინტერფეისი>** - საშუალებას გვაძლევს ინტერფეისის მოთხოვნების არასრული შესრულების შემთხვევაშიც კი ტაიპერორი არ წარმოიშვას. ჩვენს შემთხვევაში:

1. saveCourse(courseId:string, changes:Partial<Course>){}

**3. Reactive Component Interaction**

25. Store Optimistic Data Modifications - Step-By-Step Implementation

აქამდე რასაც ვაკეთებდი იყო სტეიტლესი, ამის შემდეგ გადავაწყვეთ სტეიტფულზე, რომელიც თავად კოდში შემიძლია ვნახო.