LAB9 - Use Observables

Observables provide support for passing messages between parts of your application. They are used frequently in Angular and are a technique for event handling, asynchronous programming, and handling multiple values. In this lab, we will refactor our code so that instead of using raw values and updates them manually we'll be using RxJS Observables and subjects.

Refactoring Task service

In this section we will refactor the `TaskService` by changing the methods signature and their implementation in the `MockTaskService`

1. Head over `task.service.ts` , change the `getAllTasks` method signature instead of returning the list of tasks directly, return an observable of tasks

   ```
   abstract  getAllTasks(): Observable<Task[]>;
   ```

2. For the `addTask` and `removeTask` you won't need to return the list of tasks since it will be observed

   ```
   abstract  addTask(task: Task):void;
   abstract  removeTask(task: Task):void;
   ```

3. In the `MockTaskService` , fix the mehods signature according to the base calss (`TaskService`)

4. `getAllTasks` need to return an observable now, to do so we'll need the actual observable which is the list of tasks. Add a new property `tasks$` of type `BehaviorSubject` (We use `BehaviorSubject` because it can be initialized with initial data)

   ```
   private  tasks$ = new  BehaviorSubject<Task[]>(this.tasks)
   ```

   > **$ suffix** is a naming convention used for observables, it indicates that the variable is an **Observable** (Subject, BehaviorSubject, ...)

5. Change the implementation of getList so it returns the `tasks$` instead of `tasks`

```
getAllTasks(): Observable<Task[]> {
    return  this.tasks$
}
```

Subscribing to tasks

After refactoring the task service, the list of tasks became an observable, therefore we'll have to observe (subscribe) this later to actually get real data.

1. Go to `tasks-list.component.ts`, in `ngOnInit` hook instead of assinging the tasks property to the observable returned by `getAllTasks` subscribe to it like the following:

   ```
   ngOnInit(): void {
       this.taskService.getAllTasks().subscribe(tasks=>
           this.tasks = tasks
       });
   }
   ```

2. To see the real difference, implement the delete functionality for the tasks, add an event handler on the remove icon button and bind it to method `removeTask` in the `TasksListComponent`

   ```
   <button  (click)="removeTask(task)"  mat-icon-button  aria-label="Delete">
       <mat-icon>delete</mat-icon>
   </button>
   ```

   TS (User the service to remove the task)

   ```
   removeTask(task:Task) {
       this.taskService.removeTask(task)
   }
   ```

   If you run the application ans start deleting tasks, you will notice that the tasks list is automatically updated without calling `getList` method a second time, and that's because the first time returned an observable and now the component is subscribed to it.

Using *async* pipe

Angular provides a handfull pipe called `async` helps to subscribe to the observable directly from

the template without adding extra code in the typescript code.

We use the `async` pipe it like the following:

- Typesript

```
tasks$? : Observable<Task[]>

ngOnInit(): void {
    this.tasks$ = this.taskService.getAllTasks();
}
```

- Template

```
<mat-grid-tile  *ngFor="let  task  of (tasks$ | async)">
    ...
</mat-grid-tile>
```

Unsubscribing

In Angular applications, it's always recommended to unsubscribe the observables to **avoid Memory Leaks** and running uncessary codes (subscribe handler)

We unsubsribe from observables just before the destruction fo the component, meaning in the `ngOnDestory` lifecycle hook

1. Implement the interface `OnDestory`

```
export  class  TasksListComponent  implements  OnInit, OnDestroy {
    ...
}
```

2. To unsubscribe we need to save the `subscription` object, do the necessary changes

```
subscription?: Subscription
ngOnInit(): void {
    this.subscription = this.taskService.getAllTasks().subscribe(tasks=>{
        this.tasks = tasks
    });
}
```

3. Unsubscribe from all the observables used in the component in the `ngOnDestroy` method

```
ngOnDestroy(): void {
    this.subscription?.unsubscribe()
}
```

Using RxJS operators

In this section, we will use some of the most used RxJS operators used to chain and compose observables.

1. First add a new abstract method `searchTasks` n the `TaskService`, it should accept one string parameter `keyword` and return an observable of taks

```
abstract  searchTasks(keyword?:string): Observable<Task[]>;
```

2. Implement the method `searchTasks` in the `MockTaskService` so that it search the filter keyword in the `description` field and return the matching elements, you can use the RxJS operator `map`

```
searchTasks(keyword: string = ""): Observable<Task[]> {
    return  this.tasks$.pipe(
        map(tasks=>tasks.filter(task=>task.description.includes(keyword)))
    )
}
```

3. Go to `tasks-list.component.html` and add a search field in the task list component

```
<mat-form-field  style="margin: 16px;min-width: 300px;"  appearance="outline">
    <mat-label>Search</mat-label>
    <mat-icon  matPrefix>search</mat-icon>
    <input  #searchInput  matInput  type="text"   >
</mat-form-field>
```

4. Add a property `search$` of type `BehaviorSubject` in the `TasksListComponent` class, this property will hold the search field value (initiazed to ("") empty string)

```
search$ = new  BehaviorSubject<string>("")
```

5. Add a private method `searchTasks` in the TasksListComponent, this method call `searchTasks` of the `TaskService` and subscribe to returned observable

```
searchTasks(keyword: string = "") {
    this.subscription = this.taskService.searchTasks(keyword).subscribe(tasks=>{
        this.tasks = tasks
    });
}
```

6. We want to update the search subject as the user start typing on the search input, to do so add a handler on `keyup` event, that will emit the new input value

```
<input ...  (keyup)="search$.next(searchInput.value)"  >
```

7. Now, you need to update the list of tasks as the search value changes (`search$`) in order to perform that observe the `search$` subject and call `searchTasks` .

   Remove the old code inside `ngOnInit` method and replace it by the following:

```
ngOnInit(): void {
    this.search$.subscribe(this.searchTasks)

    //or this.search$.subscribe(keyword=>this.searchTasks(keyword))
}
```

   Notice that we dont need to call `getAllTasks` the first time the page load, because we used a behavior subject for the search value, and when subscribing to a behavior subject it emit also the current value (which is empt string at the frist time). As a result the `seachTasks` is called.

8. If you run and test, the search functionalty should work, but notice that the list reloads each time you type, if you're using a real backend service you would generate too many requests and could slow down your app.

   Forthuntly, RxJS comes with a useful operator called `debounceTime` , this operator is popular in scenarios such as type-ahead where the rate of user input must be controlled which is our case. We use it like the following

```
ngOnInit(): void {
    this.search$.pipe(
        debounceTime(500)
    ).subscribe(keyword=>this.searchTasks(keyword))
```

```
}
```

> `debounceTime(500)` means that the observable should ignore emitted values that takes less that 500ms from the last emit, in our case we ignore all the search input changes until user stop typing and hold 500ms, that way we are minimizing the number of requests as the user type.

9. We could improve even more, and ignore the requests for the same keywords, if the value remains the same no need to make a request second time, we could achieve that using `distinctUntilChanged` operator. We use it like the following

```
ngOnInit(): void {
    this.search$.pipe(
        distinctUntilChanged(),
        debounceTime(500)
    ).subscribe(keyword=>this.searchTasks(keyword))
}
```