Лабораторна робота № 9. Управління глобальним станом через NgRx Store

Мета: навчитися створювати та використовувати глобальний стан у Angular-додатку за допомогою NgRx.

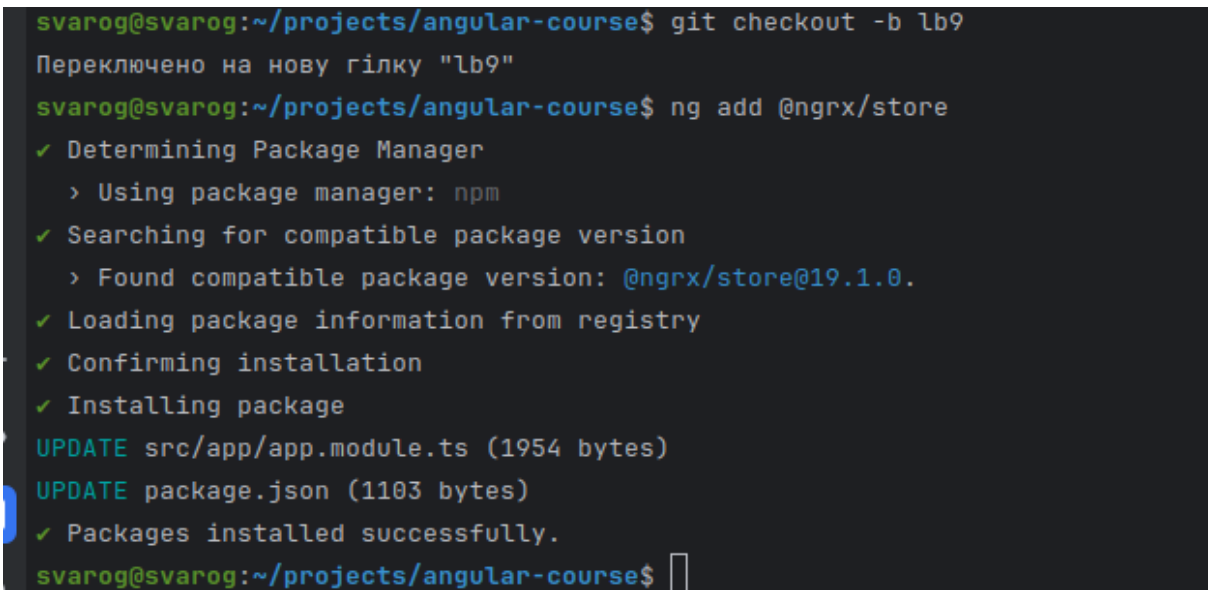
Завдання

1) Ознайомитися з матеріалом лекції № 9;
2) Виконати пункти 1-5 ходу роботи;
3) У звіті відобразити кроки 1-5 ходу роботи та додати скріни роботи застосунку (відображення, створення, оновлення, видалення, пошук, фільтрація);
4) Надати відповіді на контрольні питання.

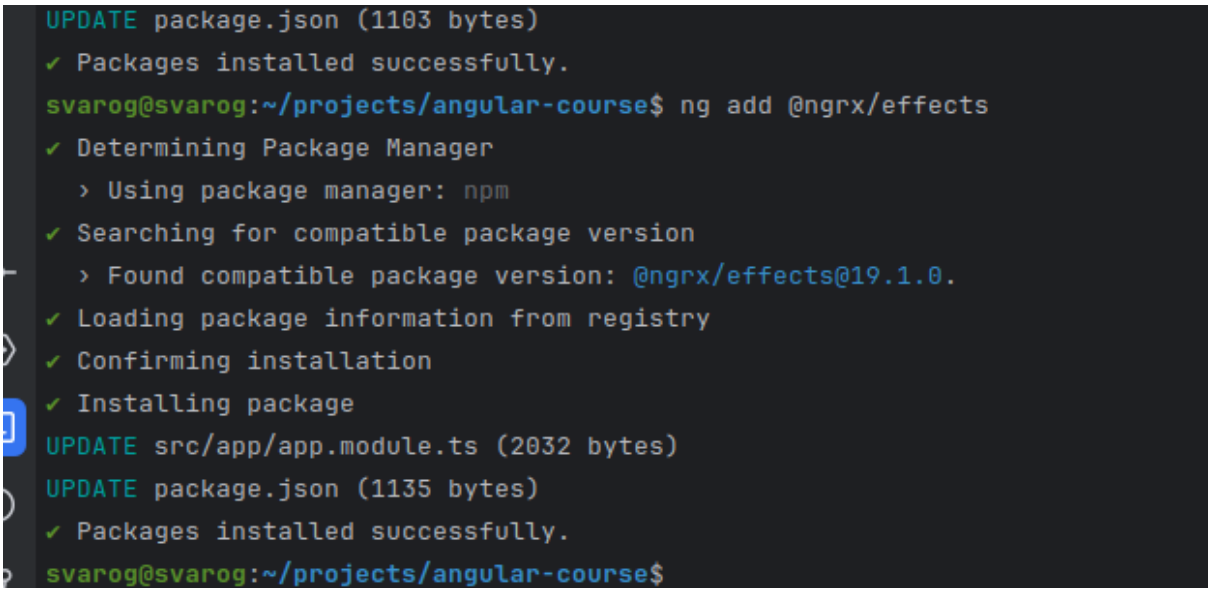
Хід роботи

1) Для початку встановимо необхідні залежності та зареєструємо їх у модулі

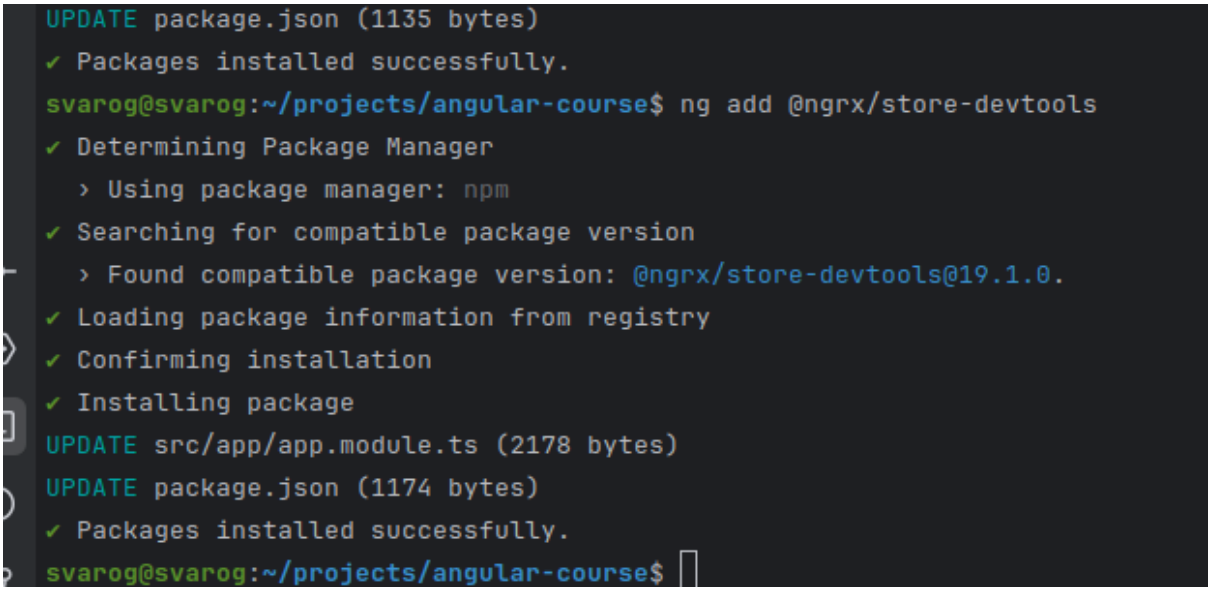
   ng add @ngrx/store



ng add @ngrx/effects



ng add @ngrx/store-devtools



ng add @ngrx/entity

```
✔ Packages installed successfully.
svarog@svarog:~/projects/angular-course$ ng add @ngrx/entity
✔ Determining Package Manager
  › Using package manager: npm
✔ Searching for compatible package version
  › Found compatible package version: @ngrx/entity@19.1.0.
✔ Loading package information from registry
✔ Confirming installation
✔ Installing package
UPDATE package.json (1205 bytes)
✔ Packages installed successfully.
svarog@svarog:~/projects/angular-course$
```

Реєструємо у **app.module.ts** (при встановлені пакетів, вони вже будуть автоматично зареєстровані, на цьому етапі поки залишаємо без змін)

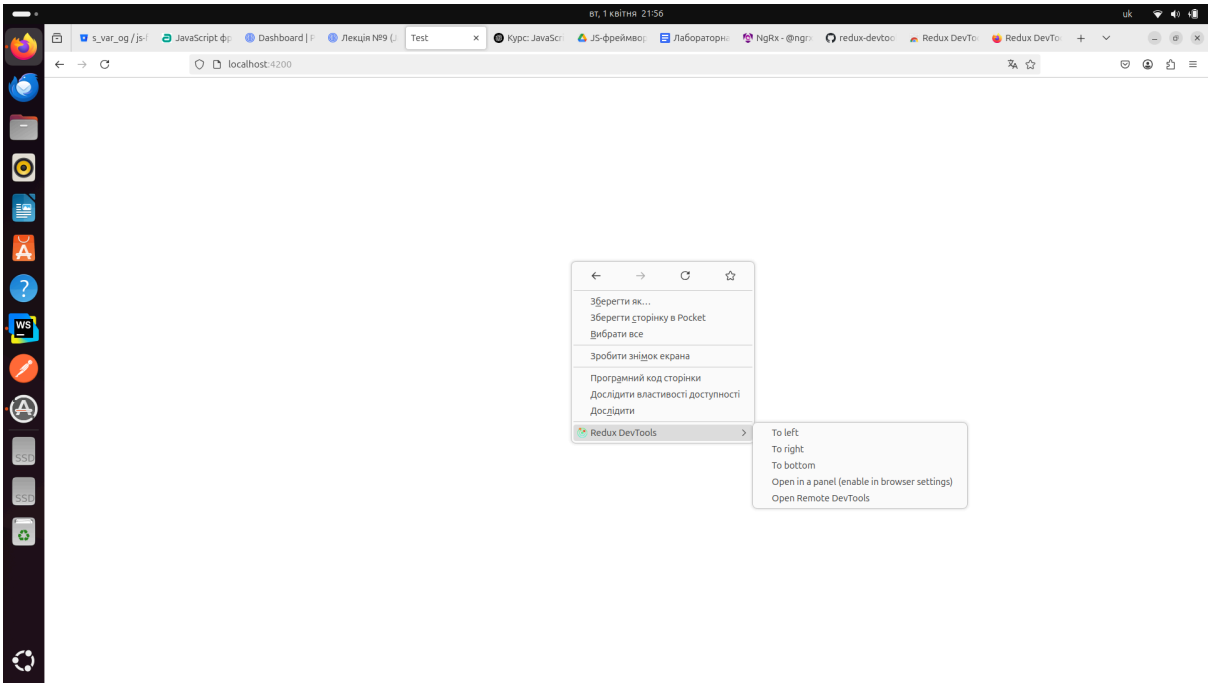
```
47      MatSelectModule,
48      MatIconModule,
49      MatSnackBarModule,
50      MatCardModule,
51      MatProgressSpinnerModule,
52      StoreModule.forRoot({}, {}),
53      EffectsModule.forRoot([]),
54      StoreDevtoolsModule.instrument({ maxAge: 25, logOnly: !isDevMode() }),
55    ],
56    providers: [
57      provideHttpClient(withInterceptorsFromDi()),
58    ],
```

Далі встановлюємо браузерне розширення Redux DevTools:

Chrome Web Store
https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd

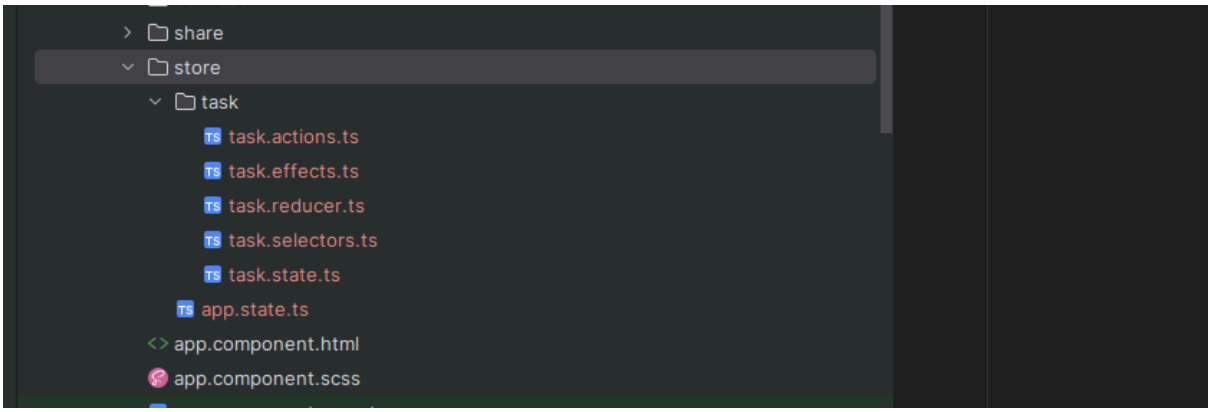
Mozilla Add-ons
https://addons.mozilla.org/en-US/firefox/addon/reduxdevtools/

Якщо встановили успішно, то у браузері можна побачити новий пункт “Redux DevTools”



2) Переходимо до створення структури Store

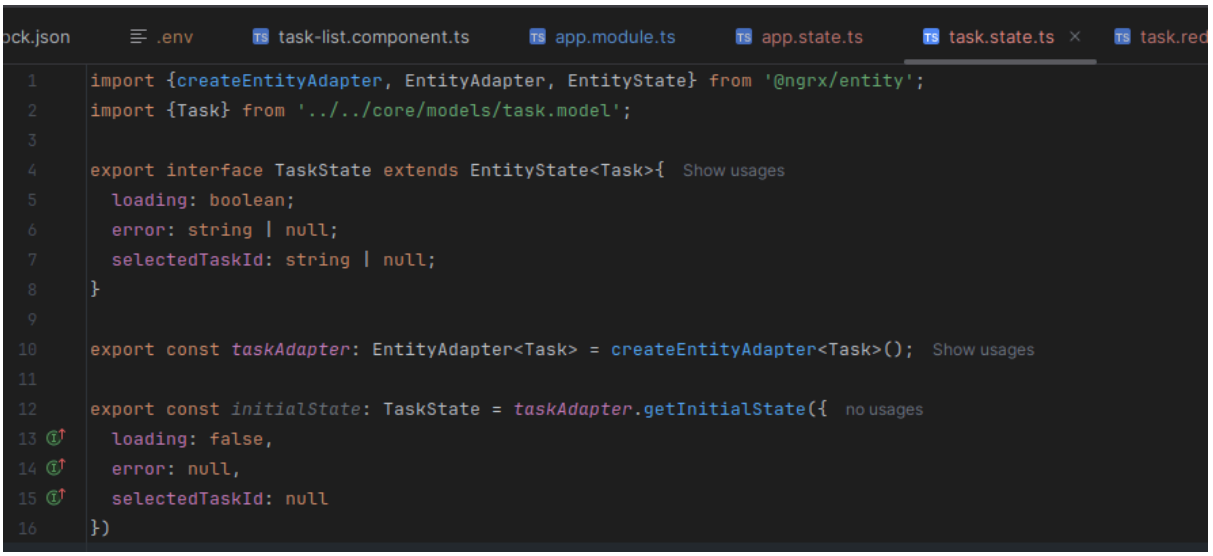
Зараз ми працюємо лише з однією сутністю “Завдання”, тому за структурою було би створити лише файли: actions, reducer, effects, selects. І при цьому не створювати кореневий файл Store (app.state.ts) та сутності (task.state.ts), але цей підхід зменшить наші можливості до масштабування проєкту у майбутньому, а також порушить принцип єдиної відповідальності для reducer (зовсім трохи, але порушить). Тому структуру використовувати feature-based:



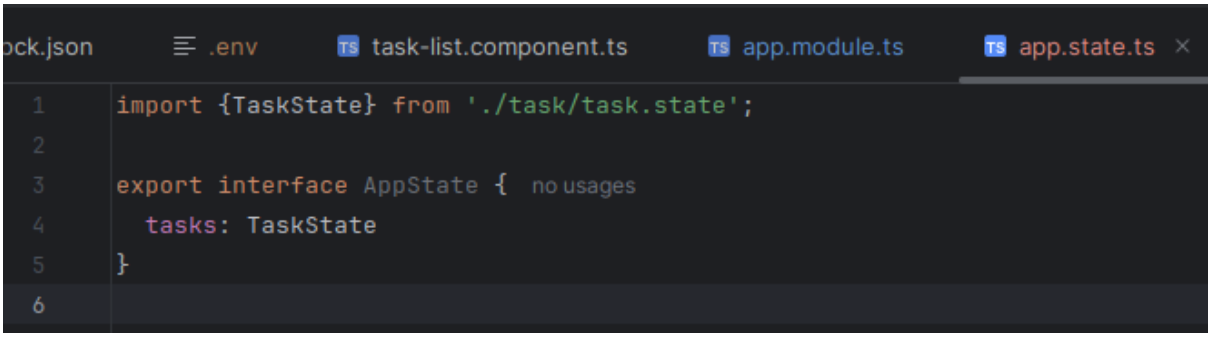
3) Переходимо до файлів Feature Store “task”. Почнемо з файлу task.state.ts у якому створимо інтерфейс TaskState, який розширює інтерфейс EntityState та додає власні властивості (це потрібно для формування структури ids: [], entities: {}), підключимо EntityAdapter

для автоматичного генерування методів (addOne, updateOne, setAll тощо) та за допомогою функції getInitialState, ініціалізуємо початкове значення

task.state.ts



Далі створимо інтерфейс у загальному файлі **app.state.ts**. Цей інтерфейс будемо використовувати для строгої типізації при підключенні Store (у модулі, компоненті, тощо). Такоє цей файл дозволить нам легко масштабувати наш Store при появі нових feature.



Переходимо до файлу **task.actions.ts**, у якому пропишемо Actions для кожної операції:

1) Почнемо з отримання списку завдань, але створимо не один Actions, а три - основний action loadTask (саме на нього буде реагувати effects), а також loadTaskSuccess (якщо завдання завантажено успішно) і loadTasksFailure (у випадку помилки).

```
import {TaskStatus} from '../../core/models/status.enum';

// Load
export const loadTasks : ActionCreator<'[Task] Load All', (props: ...  = createAction(  Show usages
  '[Task] Load All',
  props<{status: TaskStatus}>()
);
export const loadTaskSuccess : ActionCreator<'[Task] Load All Success', ...  = createAction(  Show usages
  '[Task] Load All Success',
  props<{tasks: Task[]}>()
);
export const loadTasksFailure : ActionCreator<'[Task] Load All Failure', ...  = createAction(  Show usages
  '[Task] Load All Failure',
  props<{error: string}>()
)
```

2) Далі переходимо до створення нового завдання. Діємо за аналогією

```
export const createTask : ActionCreator<'[Task] Create', (props: { ...  = createAction(  no usages
  '[Task] Create',
  props<{task: Task}>()
);

export const createTaskSuccess : ActionCreator<'[Task] Create Success', (p...  = createAction(  no usages
  '[Task] Create Success',
  props<{task: Task}>()
);

export const createTaskFailure : ActionCreator<'[Task] Create Failure', (p...  = createAction(  no usages
  '[Task] Create Failure',
  props<{error: string}>()
);
```

3) Далі оновлення та часткове оновлення

```
// Update
export const updateTask : ActionCreator<'[Task] Update', (props: { ...  = createAction(  no usages
  '[Task] Update',
  props<{task: Task}>()
)

export const updateTaskSuccess : ActionCreator<'[Task] Update Success', (p...  = createAction(  no usages
  '[Task] Update Success',
  props<{task: Task}>()
)

export const updateTaskFailure : ActionCreator<'[Task] Update Failure', (p...  = createAction(  no usages
  '[Task] Update Failure',
  props<{error: string}>()
)
```

```
// Patch
export const patchTask : ActionCreator<'[Task] Patch', (props: { i... = createAction( no usages
  '[Task] Patch',
  props<{id: string, changes: Partial<Task>}>()
);

export const patchTaskSuccess : ActionCreator<'[Task] Patch Success', (pr... = createAction( no usages
  '[Task] Patch Success',
  props<{ task: Task }>()
);
export const patchTaskFailure = createAction( no usages
  '[Task] Patch Failure',
  props<{ error: string }>()
);
```

4) Створюємо Actions для видалення

```
// delete
export const deleteTask : ActionCreator<'[Task] Delete', (props: { ... = createAction( no usages
  '[Task] Delete',
  props<{id: string}>()
);

export const deleteTaskSuccess : ActionCreator<'[Task] Delete Success', (p... = createAction( no usages
  '[Task] Delete Success',
  props<{ id: string }>()
);

export const deleteTaskFailure : ActionCreator<'[Task] Delete Failure', (p... = createAction( no usages
  '[Task] Delete Failure',
  props<{ error: string }>()
);
```

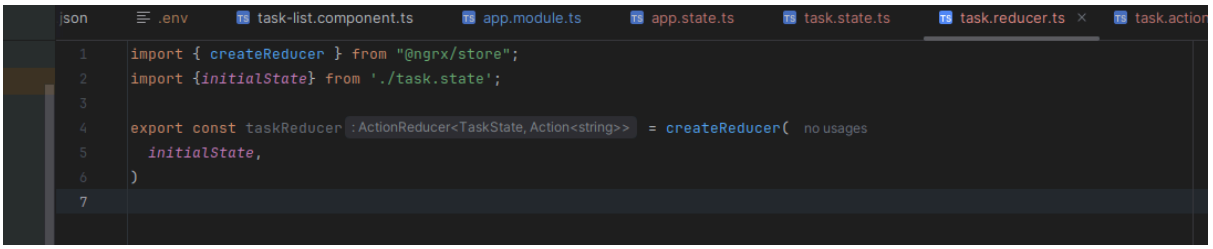
5) Створюємо Actions для вибору завдання

```
// select
export const selectTask : ActionCreator<'[Task] Select', (props: { ... = createAction( no usages
  '[Task] Select',
  props<{ id: string | null }>()
);
```

Action	Коли використовується
loadTasks	при старті сторінки, щоб завантажити всі завдання
createTask	при сабміті нової форми
updateTask	при редагуванні повної форми
patchTask	при зміні лише статусу
deleteTask	при натисканні на "видалити"
selectTask	для відображення у формі або відкриття модального вікна

Далі переходимо до reducer реалізуємо файл **task.reducer.ts**, який буде містити чисту функцію, яка на основі Action оновлює TaskState

Спочатку створюємо функцію за допомогою createReducer та вказуємо початкове значення, підтягуючи initialState з **task.state.ts**



Далі прописуємо обробники наших Actions, при цьому використовуючи функції EntityAdapter

Для початку імпортуємо TaskActions

Далі прописуємо обробку завантаження завдань

Прописуємо обробку створення завдання, але тут є нюанс, якщо нас не потрібно при створенні завдання показувати спінер, то Actions createTask можна не обробляти, його буде “слухати” effects (те саме

стосується і оновлення, і видалення). Але так як ми використовуємо спінер, то будемо прописувати.

```
  on(TaskActions.createTask, (state: TaskState) : { loading: boolean; error: string | null;... => ({
    ...state,
    loading: true,
  })),

  on(TaskActions.createTaskSuccess, (state: TaskState, {task} : { task: Task } & Action<"[Task] Create Su... ) : { loading: boolean; error: string | null;... =>
    taskAdapter.addOne(task, {...state, loading: false}),
  ),

  on(TaskActions.createTaskFailure, (state: TaskState, {error} : { error: string } & Action<"[Task] Create... ) : { loading: boolean; error: string; select... => ({
    ...state,
    loading: false,
    error
  }))
)
```

Оновлення

```
  on(TaskActions.updateTask, (state: TaskState) : { loading: boolean; error: string | null;... => ({
    ...state,
    loading: true,
  })),

  on(TaskActions.updateTaskSuccess, (state: TaskState, {task} : { task: Task } & Action<"[Task] Update Su... ) : { loading: boolean; error: string | null;... =>
    taskAdapter.updateOne({id: task.id, changes: task}, {...state, loading: false}),
  ),

  on(TaskActions.updateTaskFailure, (state: TaskState, {error} : { error: string } & Action<"[Task] Update... ) : { loading: boolean; error: string; select... => ({
    ...state,
    loading: false,
    error
  }))
)
```

Часткове оновлення

```
  on(TaskActions.patchTask, (state: TaskState) : { loading: boolean; error: string | null;... => ({
    ...state,
    loading: true,
  })),

  on(TaskActions.patchTaskSuccess, (state: TaskState, {task} : { task: Task } & Action<"[Task] Patch Suc... ) : { loading: boolean; error: string | null;... =>
    taskAdapter.updateOne({id: task.id, changes: task}, {...state, loading: false}),
  ),

  on(TaskActions.patchTaskFailure, (state: TaskState, {error} : { error: string } & Action<"[Task] Patch ... ) : { loading: boolean; error: string; select... => ({
    ...state,
    loading: false,
    error
  }))
)
```

Видалення

```
71
72  on(TaskActions.deleteTask, (state: TaskState) : { loading: boolean; error: string | null;... => ({
73    ...state,
74    loading: true,
75  })),
76
77  on(TaskActions.deleteTaskSuccess, (state: TaskState, {id} : { id: string } & Action<"[Task] Delete Su... ) : { loading: boolean; error: string | null;... =>
78    taskAdapter.removeOne(id, {...state, loading: false}),
79  ),
80
81  on(TaskActions.deleteTaskFailure, (state: TaskState, {error} : { error: string } & Action<"[Task] Delete... ) : { loading: boolean; error: string; select... => ({
82    ...state,
83    loading: false,
84    error
85  }))
86 )
87
```

Вибір завдання

```
86
87  on(TaskActions.selectTask, (state: TaskState, {id} : { id: string | null } & Action<"[Task] Se... ) : { selectedTaskId: string | null; loading:... => ({
88    ...state,
89    selectedTaskId: id,
90  }))
91 )
92
```

Використання адаптера у Reducer надає нам певні переваги, а саме:

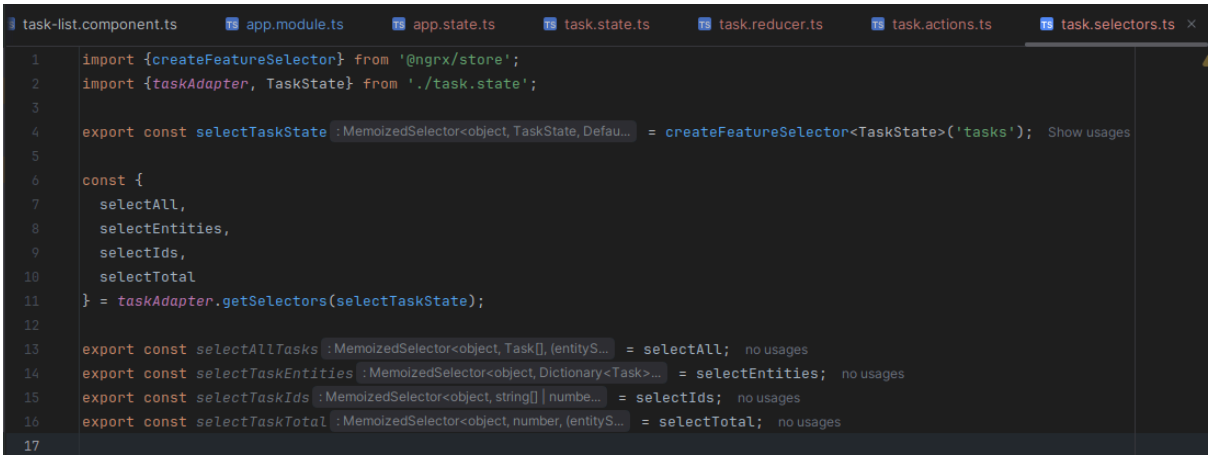
1) всі дії виконуються іммутабельно;
2) ми уникаємо ручної зміни state (state.tasks = [...tasks]), натомість все робить адаптер;
3) якщо loading, error, selectedTaskId - кастомні поля, адаптер їх не чіпає.

Далі переходимо до **task.selectors.ts**
Спочатку створимо глобальний селектор для feature “tasks”

task-list.component.ts | app.module.ts | app.state.ts | task.state.ts | task.reducer.ts | task.actions.ts | task.selectors.t

```
1  import {createFeatureSelector} from '@ngrx/store';
2  import { TaskState } from './task.state';
3
4  export const selectTaskState : MemoizedSelector<object, TaskState, Defau... = createFeatureSelector<TaskState>('tasks'); no usages
5
```

Далі підключимо Entity селектори (getAll, getEntities, getIds, getTotal)



Пропишемо додаткові селектори

Далі переходимо до **task.effects.ts**, він відповідає за побічні ефекти, (HTTP-запити, пов'язані з Task). Також не забуваємо зробити його Injectable

Через inject() інжектимо наш HTTP-сервіс та Actions для “прослуховування” та реакції на дії. Не використовуємо традиційну інжекцію через конструктор із-за особливостей ngrx effects.

Реалізуємо завантаження завдань

```
loadTasks$ : Observable<({ tasks: Task[] } & Action<'[... = createEffect(() : Observable<({ tasks: Task[] } & Action<'[... =>
  this.action$.pipe(
    ofType(TaskActions.loadTasks),
    switchMap(({status} : { status: TaskStatus } & Action<'[Task] L... ) : Observable<({ tasks: Task[] } & Action<'[... =>
      this.taskService.getTasks(status).pipe(
        map((tasks: Task[]) : { tasks: Task[] } & Action<'[Task] Load A... => TaskActions.loadTaskSuccess({tasks})),
        catchError(err : any => {
          let error : any = (err.error.errors) ? err.error.message +  '. ' + err.error.errors : err.error.message;
          return of(TaskActions.loadTasksFailure({error}));
        })
      )
    )
  )
)
```

loadTasks$ це кастомна змінна, вона не пов’язана з іншими частинами Store а слугує для розуміння за що відповідає effects.

За аналогією робимо інші effects.

Створення завдання

```
createTask$ : Observable<({ task: Task } & Action<'[Tas... = createEffect(() : Observable<({ task: Task } & Action<'[Tas... =>
  this.action$.pipe(
    ofType(TaskActions.createTask),
    mergeMap(({task} : { task: Task } & Action<'[Task] Create'> ) : Observable<({ task: Task } & Action<'[Tas... =>
      this.taskService.createTask(task).pipe(
        map((createdTask: Task) : { task: Task } & Action<'[Task] Create Su... => TaskActions.createTaskSuccess({task: createdTask}),
        catchError(err : any => {
          let error : any = (err.error.errors) ? err.error.message + ', ' + err.error.errors : err.error.message;
          return of(TaskActions.createTaskFailure({error}));
        })
      )
    ),
  )
)
}
```

Оновлення завдання

```
updateTask$ : Observable<({ task: Task } & Action<'[Tas... = createEffect(() : Observable<({ task: Task } & Action<'[Tas... =>
  this.action$.pipe(
    ofType(TaskActions.updateTask),
    switchMap(({task} : { task: Task } & Action<'[Task] Update'> ) : Observable<({ task: Task } & Action<'[Tas... =>
      this.taskService.updateTask(task.id, task).pipe(
        map((updatedTask: Task) : { task: Task } & Action<'[Task] Update Su... => TaskActions.updateTaskSuccess({task: updatedTask})),
        catchError(err : any => {
          let error : any = (err.error.errors) ? err.error.message + ', ' + err.error.errors : err.error.message;
          return of(TaskActions.updateTaskFailure({error}));
        })
      )
    )
  )
)
```

Оновлення статусу (часткове оновлення)

```
patchTask$ : Observable<({ task: Task } & Action<'[Tas... = createEffect(() : Observable<({ task: Task } & Action<'[Tas... =>
  this.action$.pipe(
    ofType(TaskActions.patchTask),
    switchMap(({id, changes} : { id: string; changes: Partial<Task> } & ... ) : Observable<({ task: Task } & Action<'[Tas... =>
      this.taskService.patchTask(id, changes).pipe(
        map((updatedTask: Task) : { task: Task } & Action<'[Task] Patch Suc... => TaskActions.patchTaskSuccess({task: updatedTask})),
        catchError(err : any => {
          let error : any = (err.error.errors) ? err.error.message + ', ' + err.error.errors : err.error.message;
          return of(TaskActions.patchTaskFailure({error}));
        })
      )
    )
  )
);
```

Видалення

```
deleteTask$ : Observable<({ id: string } & Action<'[Tas...  = createEffect(() : Observable<({ id: string } & Action<'[Tas...  =>
  this.action$.pipe(
    ofType(TaskActions.deleteTask),
    mergeMap(({id} : { id: string } & Action<'[Task] Delete'> ) : Observable<({ id: string } & Action<'[Tas...  =>
      this.taskService.deleteTask(id).pipe(
        map(() : { id: string } & Action<'[Task] Delete Su...  => TaskActions.deleteTaskSuccess({id})),
        catchError(err : any  => {
          let error : any  = (err.error.errors) ? err.error.message +  '. ' + err.error.errors : err.error.message;
          return of(TaskActions.deleteTaskFailure({error}));
        })
      )
    )
  )
);
}
```

4) Переходимо до файлу модуля, оновлюємо його, підключаючи наш reducer та effects

```
50      MatSelectModule,
51      MatIconModule,
52      MatSnackBarModule,
53      MatCardModule,
54      MatProgressSpinnerModule,
55      StoreModule.forRoot<AppState>({tasks: taskReducer}),
56      EffectsModule.forRoot([TaskEffects]),
57      StoreDevtoolsModule.instrument({ maxAge: 25, logOnly: !isDevMode() }),
58    ],
59    providers: [
60      provideHttpClient(withInterceptorsFromDi()),
61    ],
62    bootstrap: [AppComponent]
63  })
64  export class AppModule { }
65
```

5) Переходимо до підключення у компонентах, почнемо з **task-list.component.ts**

Підключаємо Store у конструкторі

```
34    myTasks$!: Observable<Task[]>;
35
36    hasLoading: boolean = false;
37
38    filterControl : FormControl<string | null>  = new FormControl('');
39
40    constructor(  no usages  Kryvyi Yaroslav *
41      private store: Store<AppState>,
42      public dialog: MatDialog,
43      private snackBar: MatSnackBar
44      ) {
45    }
```

Оновлюємо **ngOnInit**

```
ngOnInit(): void {
 this.store.dispatch(TaskActions.loadTasks({}));

 this.loading$ = this.store.select(TaskSelectors.selectTaskLoading);
 this.error$ = this.store.select(TaskSelectors.selectTaskError);

 this.myTasks$ = combineLatest([
   this.store.select(TaskSelectors.selectAllTasks),
   this.filterControl.valueChanges.pipe(
     startWith(''),
     debounceTime(300),
     distinctUntilChanged()
   )
 ]).pipe(
   map(([tasks, filter]) =>
     tasks.filter(task =>
       task.title.toLowerCase().includes(filter ?? ''.toLowerCase()) ||
       task.description?.toLowerCase().includes(filter ?? ''.toLowerCase()) ||
       task.assignee.toLowerCase().includes(filter ?? ''.toLowerCase())
     )),
 );

 this.error$.pipe(takeUntil(this.destroy$)).subscribe(error => {
   if (error) {
     this.snackBar.open(error, 'Закрити', { duration: 4000, panelClass: ['error-snackbar'] });
   }
 });

 this.loading$.pipe(takeUntil(this.destroy$)).subscribe((loading) => {
   this.hasLoading = loading;
 })
}
```

Так як Selectors це Observable, то і правила роботи залишаються незмінними (все що через subscribe - явна відписка, якщо використовується у зв’язці з AsyncPipe, то явно відписуватися не треба).

Оновлюємо editTask, тепер він відповідає лише за відкриття форми, коли емітиться подія редагування завдання.

```
88
89      editTask(): void {  Show usages   Kryvyi Yaroslav *
90        this.openDialog();
91      }
92
```

Також модернізуємо під ці зміни наш html

```
23      <div class="task-list">
24        @if (myTasks$ | async; as data) {
25          @for(task of data; track task.id; let index = $index) {
26            <app-task-item
27              [task]="task"
28              (taskEdited)="editTask()"
29            >
30            </app-task-item>
31          } @empty {
32            <p>Завдання відсутні</p>
33          }
34        }
```

Переходимо до **task-item.component.ts**

Змінюємо @Output, тепер він буде типу void. Також оновлюємо функції під взаємодію зі Store.

```
14  })
15  export class TaskItemComponent {
16
17    @Input() task!: Task; // отримуємо об'єкт завдання
18    @Output() taskEdited: EventEmitter<void>  = new EventEmitter<void>(); // подія для редагування завдання
19
20    constructor(private store: Store<AppState>) {  no usages  new *
21    }
22
23    deleteTask(id: string): void {  Show usages   Kryvyi Yaroslav *
24      this.store.dispatch(TaskActions.deleteTask({id}));
25    }
26
27    editTask(): void {  Show usages   Kryvyi Yaroslav *
28      const id : string  = this.task.id;
29      this.store.dispatch(TaskActions.selectTask({id}))
30      this.taskEdited.emit();
31    }
32
33    updateStatus(event: MatSelectChange) : void  {  Show usages   Kryvyi Yaroslav *
34      const selectedValue : any  = event.value;
35      const id : string  = this.task.id;
36      this.store.dispatch(TaskActions.patchTask({id: id, changes: {status: selectedValue}}));
37    }
```

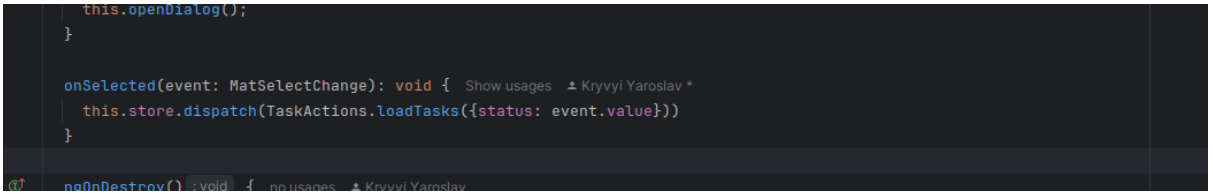
Переходимо до **task-form.component.ts**

Оновимо його, використовуючи Store, а також переробимо створення форми, використавши FormBuilder

```
export class TaskFormComponent implements OnInit, OnDestroy {

 destroy$ = new Subject<void>();
 selectedTask$!: Observable<Task | null>;
 taskForm!: FormGroup;
 editMode: boolean = false;

 constructor(
   private store: Store<AppState>,
   private fb: FormBuilder,
   public dialogRef: MatDialogRef<TaskFormComponent>,
 ) {
 }

 ngOnInit(): void {
   this.selectedTask$ = this.store.select(TaskSelectors.selectSelectedTask);

   this.taskForm = this.fb.group({
     id: [''],
     title: ['', Validators.required],
     description: ['', TaskFormValidator.forbiddenWordsValidator(['React', 'Vue'])],
     dueDate: ['', [Validators.required, TaskFormValidator.dateValidator]],
     assignee: ['', Validators.required],
     status: [TaskStatus.TODO, Validators.required]
   });

   this.selectedTask$.pipe(takeUntil(this.destroy$)).subscribe((task) => {
     if (task) {
       this.taskForm.patchValue(task);
       this.editMode = true;
     } else {
       this.taskForm.reset({status: TaskStatus.TODO});
       this.editMode = false;
     }
   })
 }

 onSubmit(): void {
   if (this.taskForm.valid) {
     if (this.editMode) {
       this.store.dispatch(TaskActions.updateTask({task: {...this.taskForm.value}}));
     } else {
       this.store.dispatch(TaskActions.createTask({task: {...this.taskForm.value}}));
     }
     this.store.dispatch(TaskActions.selectTask({ id: null }));
     this.dialogRef.close();
   }
 }
```

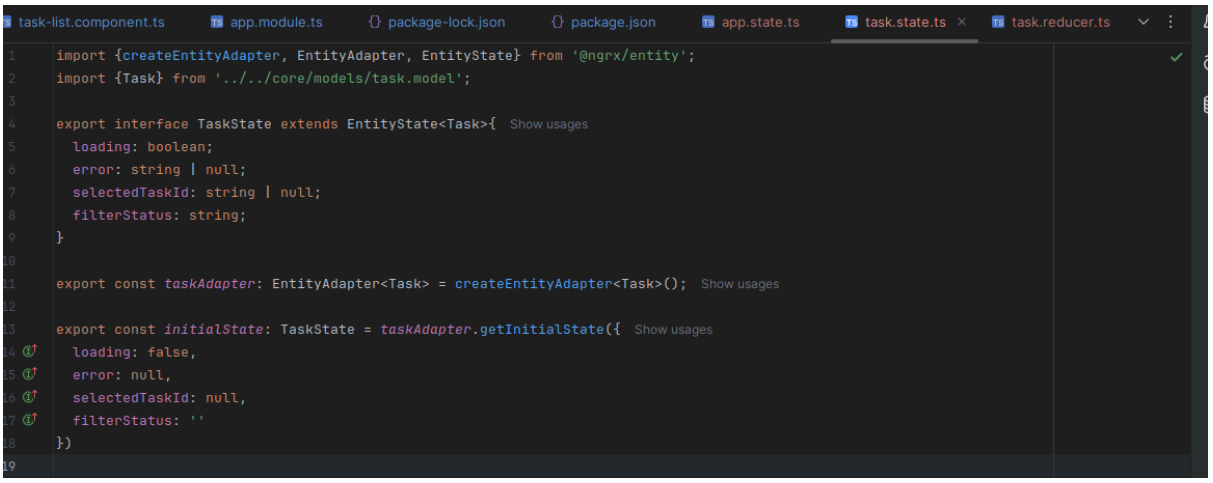
```
ngOnDestroy() {
  this.store.dispatch(TaskActions.selectTask({ id: null }));
  this.destroy$.next();
  this.destroy$.complete();
}

protected readonly TaskStatus = TaskStatus;
}
```

Далі попрацюємо з фільтрацією, зараз вона виглядає так



Своє завдання вона виконує, але давайте зробимо фільтрацію через NgRx Store. Це дасть нам багато переваг: централізоване зберігання фільтру у стані, можемо робити фільтрацію доступною для будь-яких компонентів, можемо оновлювати лише фільтр без зміни основного списку.

Почнемо, оновимо наш **task.state.ts**



Додаємо новий Actions у **task.action.ts**

```
90
91  // filter
92  export const setFilterStatus : ActionCreator<'[Task] Set Filter Status',... = createAction( no usages
93    '[Task] Set Filter Status',
94    props<{ status: string }>()
95  );
96
```

Додамо обробник у **task.reducer.ts**

```
89      selectedTaskId: id,
90    })),
91
92    on(TaskActions.setFilterStatus, (state: TaskState, { status } : { status: string } & Action<"[Task] Set F... ) : { filterStatus: string; loading: boolean;... => ({
93      ...state,
94      filterStatus: status
95    }))
96  )
```

Додамо селектори у **task.selectors.ts**

```
38
39  export const selectFilterStatus : MemoizedSelector<object, string, (s1: Tas... = createSelector( Show usages
40    selectTaskState,
41    state : TaskState => state.filterStatus
42  );
43
44  export const selectFilteredTasks : MemoizedSelector<object, Task[], (s1: Tas... = createSelector( no usages
45    selectAllTasks,
46    selectFilterStatus,
47    (tasks : Task[] , status : string ) : Task[] => {
48      if (!status) return tasks;
49      return tasks.filter(task : Task => task.status === status);
50    }
51  );
```

І оновлюємо **task-list.component.ts**

```
52      this.loading$ = this.store.select(TaskSelectors.selectTaskLoading);
53      this.error$ = this.store.select(TaskSelectors.selectTaskError);
54
55      this.myTasks$ = combineLatest([
56        this.store.select(TaskSelectors.selectFilteredTasks),
57        this.filterControl.valueChanges.pipe(
58          startWith(''),
59          debounceTime(300),
60          distinctUntilChanged()
61        )
62      ]).pipe(
```

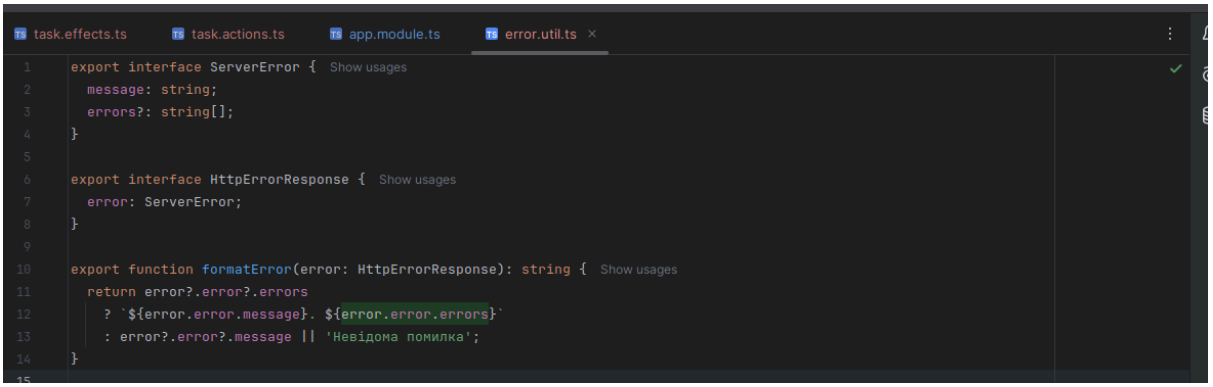
```
92
93    onSelected(event: MatSelectChange): void {  Show usages  Kryvyi Yaroslav *
94      this.store.dispatch(TaskActions.setFilterStatus({status: event.value}))
95    }
96
97    ngOnDestroy() : void {  no usages  Kryvyi Yaroslav
```

Опціонально можна додати функцію скидання статусу або зберігання статусу у localStorage.

Далі нам потрібно вирішити ще одну потенційну проблему та надлишковий код. А саме обробка помилок, зараз у нас дві проблеми:

1) Дублююча логіка у catchError;
2) Зберігання у state попередньої помилки, навіть якщо подальші дії успішні.

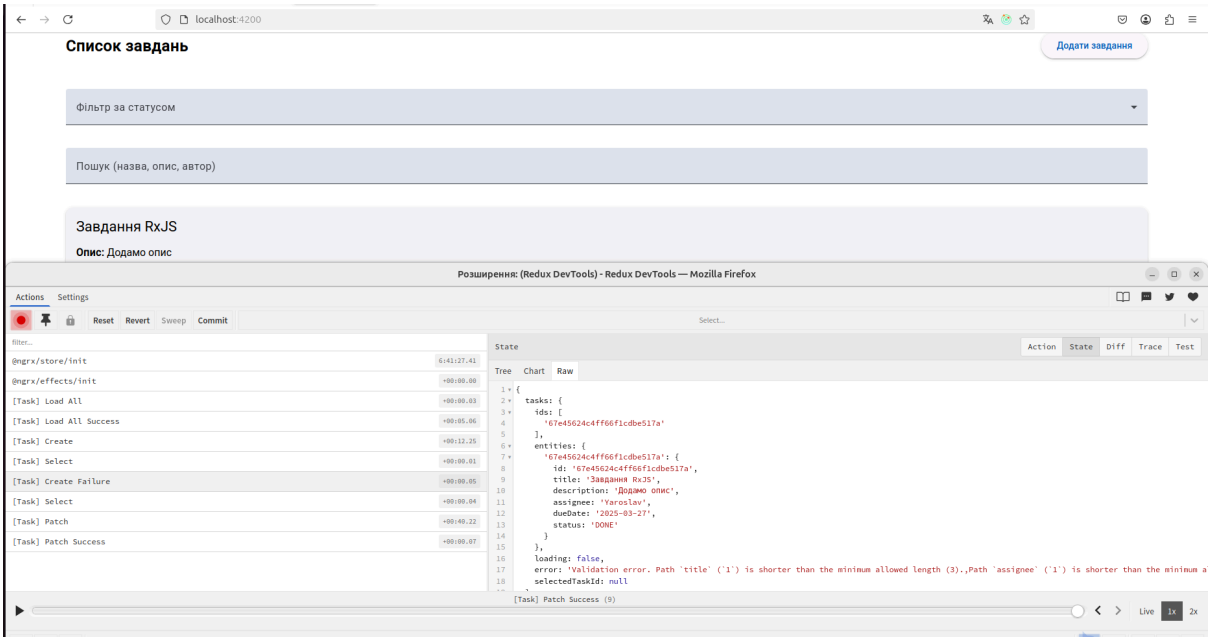
Першу проблему ми можемо вирішити багатьма способами, ми підемо найпростішим, а саме створимо utils у теці share/utils.

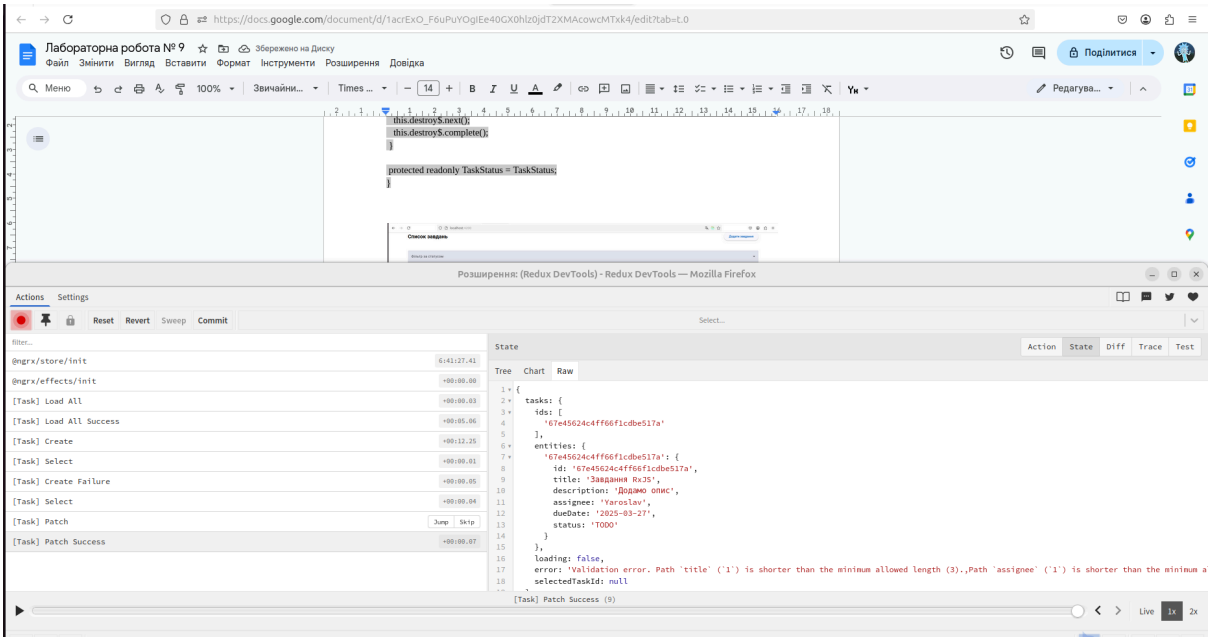


У effects змінюємо catchErrors для кожної функції

```
catchError(err => {
  return of(TaskActions.patchTaskFailure({error: formatError(err)}));
})
```

Тепер переходимо до другої проблеми. На скріншотах можна побачити, що після помилки при створенні завдання, помилка все одно залишається у Store, навіть якщо подальші дії успішні





Очистимо стан помилки, для цього використаємо effects, а саме створимо “невидиму дію”, коли відбувається помилка

```
);

resetError$ : Observable<Action<'[Task] Clear Error'>> ... = createEffect(() : Observable<Action<'[Task] Clear Error'>> =>
  this.actions$.pipe(
    ofType(
      TaskActions.createTaskFailure,
      TaskActions.updateTaskFailure,
      TaskActions.loadTasksFailure,
      TaskActions.deleteTaskFailure,
      TaskActions.patchTaskFailure),
    map(() : Action<'[Task] Clear Error'> => TaskActions.clearTaskError())
  )
);
```

```
// clear
export const clearTaskError : ActionCreator<'[Task] Clear Error', () =>... = createAction( Show usages
  '[Task] Clear Error',
)

```

```
  on(TaskActions.clearTaskError, state : TaskState => ({
    ...state,
    error: null
  }))

```

Тепер помилка буде скидатися, після відображення користувачу

Контрольні питання

1) Яка роль Reducer у NgRx Store?
2) Яка роль Actions у NgRx Store?
3) Яка роль Selectors у у NgRx Store?
4) Для чого використовуються Effects?
5) У чому перевага використання EntityAdapter?