Лабораторна робота № 8. Створення state-сервісу

Мета: ознайомитися та навчитися використовувати Subject та оператори RxJS.

Завдання

1) Ознайомитися з матеріалом лекції № 8;
2) Виконати пункти 1-6 ходу роботи;
3) У звіті відобразити кроки 1-6 ходу роботи;
4) Надати відповіді на контрольні питання.

Хід роботи

1) Для початку вирішимо нашу “улюблену” задачу, а саме трансформацію даних. Зараз ми це робимо за допомогою **TaskSchema.set("toJSON", {...})** у файлі моделі завдання. Цей підхід працює, але має певні недоліки:
    a) Будь-яка операція конвертування в json буде повертати оновленні назви та/або значення полів, що може бути недоречним у деяких частинах коду сервера;
    b) Також ця конвертація вузьконаправлена на взаємодію з одни конкретним клієнтом, що також може буде недоречно, так як з нашим API-сервером можуть “спілкуватися” інші сервіси, які очікують дані у стандартному форматі.

Тому ми вирішимо цю задачу зі сторони клієнта, для цього виростаємо адаптери. Перед створення адаптеру необхідно визначити інтерфейси запитів та відповідей до/з сервера.

Для початку проаналізуємо, які у нас є відмінності:

1) Angular використовує id, тоді як сервер - _id;
2) Angular працює з датою у браузерному форматі (строка YYYY-MM-DD), сервер з Date (ISO-string);

3) Angular не відправляє _id у тілі запиту, а тільки у URI. Але очікує отримати id у відповідь.

Виходячи з цього створимо для початку необхідні інтерфейси. Так як інтерфейс завдання у нас вже є, нам необхідно створити інтерфейс для серверу.

task-api.model.ts

```
export interface TaskApi {  no usages
  _id: string;
  title: string;
  description?: string;
  assignee: string;
  dueDate: string;
  status: string;
}
```

Далі переходимо до створення адаптеру. Створимо його у вигляді звичайного класу.

app/share/adapters/task.adapter.ts

```
export class TaskAdapter {
  static fromAPI(response: TaskApi): Task {
        return {
        id: response._id,
        title: response.title,
        description: response.description || '',
        assignee: response.assignee,
        dueDate: response.dueDate?.split('T')[0],
        status: response.status as TaskStatus
        }
  }

  static toAPI(task: Task): Omit<TaskApi, '_id'> {
        return {
```

```
        title: task.title,
        description: task.description || '',
        assignee: task.assignee,
        dueDate: new Date(task.dueDate).toISOString(),
        status: task.status
        }
  }

  static toPartialApi(partialTask: Partial<Task>): Partial<Task> {
        const apiTask: Partial<Task> = {};

        if (partialTask.dueDate) {
        apiTask.dueDate = new Date(partialTask.dueDate).toISOString();
        }

        if (partialTask.status) {
        apiTask.status = partialTask.status;
        }

        return apiTask;
  }
}
```

Ми створили адаптери, які на даний момент часу забезпечують коректну трансформацію даних, що дає нам кілька переваг:

1) Видалення **TaskSchema.set("toJSON", {...})** у файлі моделі;
2) Тепер сервер відправляє дані у стандартному форматі, а процес трансформації даних перекладено на клієнта.

Наведемо приклад використання у нашому сервісі

```
getTasks(status?: string): Observable<Task[]> {  Show usages  Kryvyi Yaroslav *
  let params :HttpParams  = new HttpParams();

  if (status) params = params.set('status', status);

  return this.http.get<TaskApi[]>(`${this.config.apiUrl}/v2/tasks`, {params: params}).pipe(
    map((tasks: TaskApi[]): Task[] => tasks.map(task :TaskApi  => TaskAdapter.fromAPI(task)))
  );
}
```

У методі GET нам потрібно тільки обробити вхідні дані з TaskApi у Task.

```
createTask(newTask: Task): Observable<Task> {  Show usages  Kryvyi Yaroslav *
  return this.http.post<TaskApi>(`${this.config.apiUrl}/v2/tasks`, TaskAdapter.toAPI(newTask)).pipe(
    map((task: TaskApi): Task => TaskAdapter.fromAPI(task)));
}

updateTask(id: string, updateTask: Task): Observable<Task> {  Show usages  new *
  return this.http.put<TaskApi>(`${this.config.apiUrl}/v2/tasks/${id}`, TaskAdapter.toAPI(updateTask)).pipe(
    map((task: TaskApi): Task => TaskAdapter.fromAPI(task)));
}

patchTask(id: string, updateTask: Partial<Task>): Observable<Task> {  Show usages  new *
  const formattedTask :Partial<Task>  = TaskAdapter.toPartialApi(updateTask);
  return this.http.patch<TaskApi>(`${this.config.apiUrl}/v2/tasks/${id}`,{...updateTask, ...formattedTask}).pipe(
    map((task: TaskApi): Task => TaskAdapter.fromAPI(task)));
}

deleteTask(id: string): Observable<void> {  Show usages  new *
  return this.http.delete<void>(`${this.config.apiUrl}/v2/tasks/${id}`);
}
```

Інші методи робимо за аналогією, єдина відмінність полягає у тому, що:

1) видалення завдання не потребує трансформації;
2) при створенні ми конвертуємо у обидві сторони, також зверніть увагу, що Omit переїхав до адаптеру, а функція тепер приймає просто Task;
3) так само робимо і з повним оновленням;
4) для часткового оновлення окремий адаптер, який трансформує тільки певні дані. Оператор spread використовуємо, щоб не втратити дані, які ми не трансформує у адаптері (у нашому застосунку таких ситуацій не відбувається, але перестрахуємося).

2) Далі реалізуємо state-сервіс, який буде за допомогою BehaviorSubject зберігати останній стан наших завдань.

Для початку створимо сервіс **src/app/share/state/task-state.service.ts**

```
svarog@svarog:~/projects/angular-course$ ng g s share/state/task-state
CREATE src/app/share/state/task-state.service.spec.ts (373 bytes)
CREATE src/app/share/state/task-state.service.ts (138 bytes)
```

Для початку створимо приватні змінні, які будуть зберігати наш стан, нам потрібні змінні для:

1) списку завдань;
2) редагуємого завдання;
3) стан завантаження даних;
4) помилки.

```
})
export class TaskStateService {

  private _tasks$ :BehaviorSubject<Task[]> = new BehaviorSubject<Task[]>([]);
  private _selectedTask$ :BehaviorSubject<Task | null> = new BehaviorSubject<Task | null>(null);
  private _loading$ :BehaviorSubject<boolean> = new BehaviorSubject<boolean>(false);
  private _error$ :BehaviorSubject<string | null> = new BehaviorSubject<string | null>(null);
```

Далі створимо публічний інтерфейс для кожної змінної і конвертуємо їх із Subject у Observable

```
  private _error$ :BehaviorSubject<string | null> = new BehaviorSubject<string | null>(null);

  public readonly task$ :Observable<Task[]> = this._tasks$.asObservable();
  public readonly selectedTask$ :Observable<Task | null> = this._selectedTask$.asObservable();
  public readonly loading$ :Observable<boolean> = this._loading$.asObservable();
  public readonly error$ :Observable<string | null> = this._error$.asObservable();

  constructor() { } no usages
```

Далі підключимо TaskService у конструкторі

```
  constructor(private taskService: TaskService) { } no usages
}
```

Почнемо реалізовувати функції

loadTasks() - завантаження всіх завдань та запис у _tasks$

```
loadTasks(status?: string): void {  no usages
  this._loading$.next(true);
  this._error$.next(null);

  this.taskService.getTasks(status)
    .pipe(
      tap((tasks: Task[]) :void => this._tasks$.next(tasks)),
      catchError(err :any => {
        this._error$.next(err.message);
        return throwError(() :any => err.message);
      }),
      finalize(() :void => this._loading$.next(false)),
    ).subscribe()
}
```

createTask() - створення завдання, отримання останнього значення з _tasks$ та повертаємо поток через of, який об'єднує поточне значення та створене.

```
createTask(task: Task): void {  no usages
  this._loading$.next(true);
  this._error$.next(null);

  this.taskService.createTask(task)
    .pipe(
      switchMap((added: Task) :Observable<Task[]> => {
        const current :Task[] = this._tasks$.getValue();
        return of([...current, added]);
      }),
      tap((updatedTasks: Task[]) :void => this._tasks$.next(updatedTasks)),
      catchError(err :any => {
        this._error$.next(err.message);
        return throwError(() :any => err.message);
      }),
      finalize(() :void => this._loading$.next(false))
    )
    .subscribe()
}
```

За аналогією робимо інші функції

```
updateTask(task: Task): void {  no usages
  this._loading$.next(true);
  this._error$.next(null);

  this.taskService.updateTask(task.id, task)
    .pipe(
      switchMap(updated => {
        const current = this._tasks$.getValue();
        const updatedTasks = current.map(t => t.id === updated.id ? updated : t);
        return of(updatedTasks);
      }),
      tap(updatedTasks :Task[] => this._tasks$.next(updatedTasks)),
      catchError(err :any => {
        this._error$.next(err.message);
        return throwError(() :any => err.message);
      }),
      finalize(() :void => this._loading$.next(false))
    )
    .subscribe()
}
```

```
patchTask(id: string, task: Partial<Task>): void {  Show usages
  this._loading$.next(true);
  this._error$.next(null);

  this.taskService.patchTask(id, task)
    .pipe(
      switchMap(patched :Task => {
        const updated :Task[] = this._tasks$.getValue().map(t :Task =>
          t.id === patched.id ? Object.assign({}, t, patched) : t
        );
        return of(updated);
      }),
      tap(updatedTasks :Task[] => this._tasks$.next(updatedTasks)),
      catchError(err :any => {
        this._error$.next(err.message);
        return throwError(() :any => err.message);
      }),
      finalize(() :void => this._loading$.next(false))
    )
    .subscribe()
}
```

```
  deleteTask(id: string): void {  no usages
    this._loading$.next(true);
    this._error$.next(null);

    this.taskService.deleteTask(id)
      .pipe(
        switchMap(() :Observable<Task[]> => {
          const filtered :Task[] = this._tasks$.getValue().filter(t :Task => t.id !== id);
          return of(filtered);
        }),
        tap(updatedList :Task[] => this._tasks$.next(updatedList)),
        catchError(err :any => {
          this._error$.next(err.message);
          return throwError(() :any => err.message);
        }),
        finalize(() :void => this._loading$.next(false))
      )
      .subscribe()
  }
}
```

Також робимо функцію, яка буде взаємодіяти з обраним для редагування завданням

```
  selectTask(task: Task | null): void {  no usages
    this._selectedTask$.next(task);
  }
```

3) Далі інтегруємо наш сервіс у компоненти:
    a) Почнемо з **task-list.component.ts**

Потрібно його ідейно переробити, бо зараз він порушує принцип єдиної відповідальності. Залишимо у компоненті тільки наступні функції: завантаження списку завдань, фільтрація/пошук, виклик форми (далі зробимо її модальною) для додавання завдання (це не буде дією запису до БД та state , а тільки керування відображенням) , а також реакція на еміт події редагування завдання (запис до _selectedTask$ та виклик форми). У html приберемо явний виклик форми, додамо поле для пошуку, а також кнопку, яка відкриває форму.

Починаємо з завантаження даних, а також відразу створимо змінну destroy$ для майбутніх відписок

```
export class TaskListComponent implements OnInit, OnDestroy {

  destroy$ :Subject<void> = new Subject<void>();

  myTasks$!: Observable<Task[]>;

  constructor(private taskStateService: TaskStateService,) {  no usages  new *
  }

  ngOnInit(): void {  no usages  Kryvyi Yaroslav *
    this.taskStateService.loadTasks();
    this.myTasks$ = this.taskStateService.task$;
  }

  ngOnDestroy() :void  {  no usages  new *
    this.destroy$.next();
    this.destroy$.complete();
  }
```

Так як myTasks$ ми використовуємо у зв’язці з async pipe то від нього явно відписуватися не потрібно.

Далі необхідно додати логіку відображення модального вікна та створити кнопку, яка буде його відкривати. Для цього ми використаємо Angular Material. Встановлюємо його за допомогою команди:

ng add @angular/material

Відповідаємо на питання

```
svarog@svarog:~/projects/angular-course$ ng add @angular/material
✔ Determining Package Manager
  › Using package manager: npm
✔ Searching for compatible package version
  › Found compatible package version: @angular/material@19.2.7.
✔ Loading package information from registry
✔ Confirming installation
✔ Installing package
✔ Choose a prebuilt theme name, or "custom" for a custom theme: Azure/Blue        [Preview: https://material.angular.io?theme=azure-blue]
✔ Set up global Angular Material typography styles? Yes
UPDATE package.json (1073 bytes)
✔ Packages installed successfully.
UPDATE angular.json (3194 bytes)
svarog@svarog:~/projects/angular-course$
```

Для відображення модального вікна будемо використовувати MatDialog. Для цього імпортуємо у app.module.ts MatDialogModule (тимчасово)

```
import { MatDialogModule } from '@angular/material/dialog';

@NgModule({  Show usages  Kryvyi Yaroslav
  declarations: [
    AppComponent,
    TaskListComponent,
    TaskItemComponent,
    StatusFilterPipe,
    TaskFormComponent,
    TaskStatusPipe
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule,
    MatDialogModule
```

Далі інжектимо його у конструктор

```
  constructor(  no usages  new *
    private taskStateService: TaskStateService,
    public dialog: MatDialog,
    ) {
  }
```

Створюємо функцію, яка при натисканні на кнопку, відкриє модальне вікно з формою. Для цього першим аргументом передаємо компонент, який хочемо відкрити, а другим опціонально налаштування

```
  }

  openDialog() : void {  Show usages  new *
    this.dialog.open(TaskFormComponent, {
      height: '70vh',
      width: '80vw',
    });
  }
```

Змінимо html під Material. Для цього використовуємо MatCardModule, MatFormFieldModule, MatButtonModule, MatInputModule, MatSelectModule (у інших компонентах також).

Спочатку змінимо html

```
<section class="task-list-container">
  <div class="task-header">
        <h2>Список завдань</h2>
        <button mat-raised-button (click)="openDialog()">Додати завдання</button>
  </div>

  <mat-form-field>
        <mat-label>Фільтр за статусом</mat-label>
        <mat-select (selectionChange)="onSelected($event)">
        <mat-option [value]="">Всі</mat-option>
        <mat-option [value]="TaskStatus.TODO">До роботи</mat-option>
        <mat-option [value]="TaskStatus.IN_PROGRESS">В процесі</mat-option>
        <mat-option [value]="TaskStatus.DONE">Виконано</mat-option>
        </mat-select>
  </mat-form-field>

  <div class="task-list">
        @if (myTasks$ | async; as data) {
        @for(task of data; track task.id; let index = $index) {
        <app-task-item
        [task]="task"
        (taskDeleted)="deleteTask($event)"
        (taskEdited)="editTask($event)"
        (statusChange)="statusChange($event)"
        >
        </app-task-item>
        } @empty {
        <p>Завдання відсутні</p>
        }
        }
  </div>
</section>
```

Далі оновимо стилі

```
.task-list-container {
```

```
  display: flex;
  flex-direction: column;
  gap: 1rem;
  width: 90vw;
  margin: 0 auto;

  .task-header {
        display: flex;
        justify-content: space-between;
        align-items: center;
        margin-bottom: 1rem;
  }

  .task-filter {
        display: flex;
        flex-direction: row;
        flex-wrap: wrap;
        gap: 0.5rem;
        margin-bottom: 0.5rem;


        mat-form-field {
        width: 40%;
        font-size: 14px;

        }
  }

  .task-list {
        display: flex;
        flex-direction: column;
        gap: 1rem;
        justify-content: center;
  }
}
```

Отримаємо результат

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

Для початку оновлюємо логіку компонента. Тепер саме цей компонент буде відповідати за створення та оновлення завдання.

Додамо id до форми для оновлення завдання.

При ініціалізації компонента, підпишемося на selectedTask$ щоб можна було реагувати, чи є завдання для оновлення або ми додаємо нове.

Так як від того чи є завдання залежить не тільки перевірка перед виконанням дії, а і відображення - створимо змінну editMode, яка буде зберігати стан.

```
export class TaskFormComponent implements OnInit, OnDestroy {

  destroy$ = new Subject<void>();
  editMode: boolean = false;

  taskForm = new FormGroup({
        id: new FormControl(''),
        title: new FormControl('', Validators.required),
        description: new FormControl('', TaskFormValidator.forbiddenWordsValidator(['React', 'Vue'])),
        dueDate: new FormControl('', [Validators.required, TaskFormValidator.dateValidator]),
        assignee: new FormControl('', Validators.required),
        status: new FormControl<TaskStatus>(TaskStatus.TODO, Validators.required),
  })

  protected readonly TaskStatus = TaskStatus;
```

```
 constructor(
        private taskStateService: TaskStateService,
        public dialogRef: MatDialogRef<TaskFormComponent>,
 ) {
 }

 ngOnInit(): void {
        this.taskStateService.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.taskStateService.updateTask(this.taskForm.value as Task);
        this.taskStateService.selectTask(null);
        } else {
        this.taskStateService.createTask(this.taskForm.value as Task);
        }
        this.dialogRef.close();
        }
 }

 ngOnDestroy() {
        this.taskStateService.selectTask(null);
        this.destroy$.next();
        this.destroy$.complete();
 }
}
```

Оновлюємо html, переробивши форму на Material. Також додаємо кнопку скасувати з атрибутом **mat-dialog-close**, щоб закрити модальне вікно.

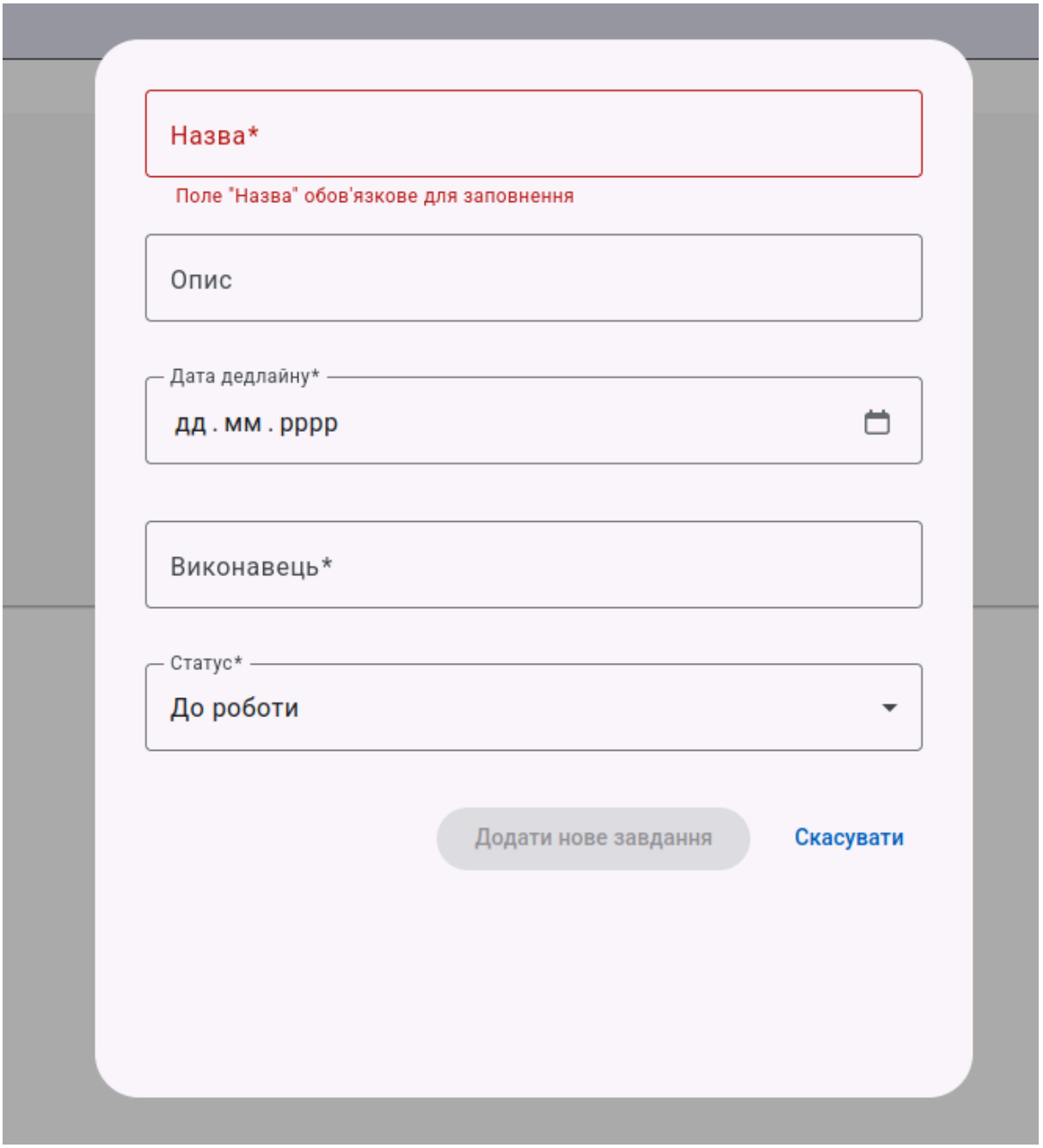
```
<form [formGroup]="taskForm" class="task-form">
  <mat-form-field appearance="outline">
        <mat-label>Назва</mat-label>
        <input matInput type="text" id="title" formControlName="title">
        <mat-error *ngIf="taskForm.get('title')?.hasError('required') &&
taskForm.get('title')?.dirty">
        Поле "Назва" обов'язкове для заповнення
        </mat-error>
  </mat-form-field>

  <mat-form-field appearance="outline">
        <mat-label>Опис</mat-label>
        <input matInput type="text" id="description" formControlName="description">
        <mat-error *ngIf="taskForm.get('description')?.hasError('forbiddenWord')">
        Вводити слова React та Vue заборонено
        </mat-error>
  </mat-form-field>

  <mat-form-field appearance="outline">
        <mat-label>Дата дедлайну</mat-label>
        <input matInput type="date" id="dueDate" formControlName="dueDate">
        <mat-error *ngIf="taskForm.get('dueDate')?.hasError('required') &&
taskForm.get('dueDate')?.dirty">
        Поле "Дата дедлайну" обов'язкове для заповнення
        </mat-error>
        <mat-error *ngIf="taskForm.get('dueDate')?.hasError('dateVal')">
        Рік не може бути меншим за поточний
        </mat-error>
  </mat-form-field>

  <mat-form-field appearance="outline">
        <mat-label>Виконавець</mat-label>
        <input matInput type="text" id="assignee" formControlName="assignee">
        <mat-error *ngIf="taskForm.get('assignee')?.hasError('required') &&
taskForm.get('assignee')?.dirty">
        Поле "Виконавець" обов'язкове для заповнення
        </mat-error>
  </mat-form-field>

  <mat-form-field appearance="outline">
        <mat-label>Статус</mat-label>
        <mat-select id="status" formControlName="status">
        <mat-option [value]="TaskStatus.TODO">До роботи</mat-option>
        <mat-option [value]="TaskStatus.IN_PROGRESS">В процесі</mat-option>
```

```
        <mat-option [value]="TaskStatus.DONE">Виконано</mat-option>
        </mat-select>
        <mat-error *ngIf="taskForm.get('status')?.hasError('required') &&
taskForm.get('status')?.dirty">
        Поле "Статус" обов'язкове для заповнення
        </mat-error>
  </mat-form-field>

  <div class="form-actions">
        <button mat-raised-button color="primary" type="submit"
[disabled]="!taskForm.valid" (click)="onSubmit()">
        {{ editMode ? 'Оновити завдання' : 'Додати нове завдання'}}
        </button>
        <button mat-button mat-dialog-close>Скасувати</button>
  </div>
</form>
```

Оновлюємо стилі

```
.task-form {
  display: flex;
  flex-direction: column;
  gap: 1rem;
  padding: 2rem;

  .form-actions {
        display: flex;
        gap: 1rem;
        justify-content: flex-end;
  }
}
```

Перевіримо відображення форми



c) Переходимо до **task-itemcomponent.ts**

Цей компонент відповідає за: видалення завдання, еміт завдання для оновлення (обробляємо у task-list.component.ts тому що саме він керує модальним вікном), часткове оновлення (статус).

```
export class TaskItemComponent {

  @Input() task!: Task; // отримуємо об'єкт завдання
```

```
  @Output() taskEdited: EventEmitter<Task>  = new EventEmitter<Task>(); // подія для редагування завдання

  constructor(private taskStateService: TaskStateService) {
  }

  deleteTask(id: string): void {
        this.taskStateService.deleteTask(id);
  }

  editTask(): void {
        this.taskEdited.emit(this.task);
  }

  getStatusClasses() {
        return {
        'done': this.task.status === TaskStatus.DONE,
        'todo': this.task.status === TaskStatus.TODO,
        'in-progress': this.task.status === TaskStatus.IN_PROGRESS
        };
  }

  updateStatus(event: MatSelectChange) {
        const selectedValue = event.value;
        this.taskStateService.patchTask(this.task.id, {status: selectedValue});
  }

  protected readonly TaskStatus = TaskStatus;

}
```

Оновимо html, для цього використаємо Material

```
<mat-card class="task-item">
  <mat-card-header>
        <mat-card-title>{{ task.title }}</mat-card-title>
  </mat-card-header>

  <mat-card-content>
        @if(task.description) {
        <p><strong>Опис:</strong> {{ task.description }}</p>
        }
        <p><strong>Виконавець:</strong> {{ task.assignee }}</p>
```

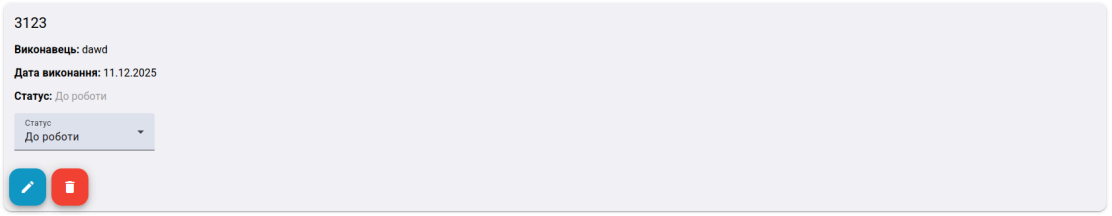
```
        <p><strong>Дата виконання:</strong> {{ task.dueDate | date:'d.MM.yyyy'}}</p>
        <p><strong>Статус: </strong><span [ngClass]="getStatusClasses()">{{ task.status | taskStatus}}</span></p>

        <mat-form-field>
        <mat-label>Статус</mat-label>
        <mat-select [value]="task.status" (selectionChange)="updateStatus($event)">
        <mat-option [value]="TaskStatus.TODO">До роботи</mat-option>
        <mat-option [value]="TaskStatus.IN_PROGRESS">В процесі</mat-option>
        <mat-option [value]="TaskStatus.DONE">Виконано</mat-option>
        </mat-select>
        </mat-form-field>
  </mat-card-content>

  <mat-card-actions>
        <button mat-fab class="edit-button" (click)="editTask()"><mat-icon>edit</mat-icon></button>
        <button mat-fab class="delete-button" (click)="deleteTask(task.id)"><mat-icon>delete</mat-icon></button>
  </mat-card-actions>
</mat-card>
```

Також оновимо стилі

```
.task-item {
  width: 100%;

  mat-card-actions {
        display: flex;
        justify-content: flex-start;
        gap: 0.5rem;

        .edit-button {
        background-color: #0f9ac5;
        color: white;
        }

        .delete-button {
        background-color: #f44336;
        color: white;
        }
  }
}
```

```
.todo {
  color: #999;
}

.done {
  color: green;
}

.in-progress {
  color: orange;
}
```

Тепер компонент має наступний вигляд



Далі необхідно у компоненті task-list.component.ts оновити функції, які відповідають за запис завдання для оновлення у state, а також фільтрацію за статусами

```
editTask(task: Task): void {  Show usages  new *
  this.taskStateService.selectTask(task);
  this.openDialog();
}

onSelected(event: MatSelectChange): void {  Show usages  new *
  this.taskStateService.loadTasks(event.value);
}
```

4) Також важливо додати не тільки обробку помилок сервера, а і їх відображення користувачу. Для цього у всіх методах state-сервісу трохи змінимо логіку catchError

```
    catchError(err => {
      this._error$.next((err.error.errors) ? err.error.message +  '. ' + err.error.errors : err.error.message);
      return throwError(() => err.message);
    }),
```

Ми змінило логіку під помилки сервера. Якщо є тільки повідомлення, то виводимо його, якщо прийшла ще додаткова інформація, то виводимо після крапки основного повідомлення.

Далі у task-list.component.ts при ініціалізації прописуємо наступний код

```
ngOnInit(): void {  no usages  Kryvyi Yaroslav *
  this.taskStateService.loadTasks();
  this.myTasks$ = this.taskStateService.task$;

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

Для відображення помилок, ми підписуємося на errors$ і якщо помилка є, то виводимо її користувачу за допомогою MatSnackBar з Material.

Для цього інжектимо його

```
  constructor(  no usages  new *
    private taskStateService: TaskStateService,
    public dialog: MatDialog,
    private snackBar: MatSnackBar
    ) {
  }

  ngOnInit(): void {  no usages  Kryvyi Yaroslav *
```

5) Далі реалізуємо очікування результатів запиту. Для цього у компоненті **task-list.component.ts** об’явимо змінну hasLoading, яка буде відповідати за стан loading$ зі state-сервісу. Підписуємося при ініціалізації

```
export class TaskListComponent implements OnInit, OnDestroy {

  destroy$ : Subject<void> = new Subject<void>();

  myTasks$!: Observable<Task[]>;

  hasLoading: boolean = false;

  constructor( no usages  new *
```

```
ngOnInit(): void {  no usages  Kryvyi Yaroslav *
  this.taskStateService.loadTasks();
  this.myTasks$ = this.taskStateService.task$;

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

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

У html обгортаємо у @if…@else логіку ітерації завдань

```
    </mat-select>
  </mat-form-field>
  @if (hasLoading) {
    <mat-spinner></mat-spinner>
  } @else {
    <div class="task-list">
      @if (myTasks$ | async; as data) {
        @for(task of data; track task.id; let index = $index) {
          <app-task-item
            [task]="task"
            (taskEdited)="editTask($event)"
          >
          </app-task-item>
        } @empty {
          <p>Завдання відсутні</p>
        }
      }
    </div>
  }
</section>
```

Якщо hasLoading - true, то за допомогою mat-spinner демонструємо анімацію завантаження, якщо ні - список завдань.

Для імітації затримки, додамо до GET-запиту імітацію затримки через delay

```
    @Inject(CONFIG_TOKEN) private config: AppConfig
  ) { }

  getTasks(status?: string): Observable<Task[]> {  Show usages  Kryvyi Yaroslav *
    let params :HttpParams  = new HttpParams();

    if (status) params = params.set('status', status);

    return this.http.get<TaskApi[]>(`${this.config.apiUrl}/v2/tasks`, {params: params}).pipe(
      map((tasks: TaskApi[]): Task[] => tasks.map(task :TaskApi  => TaskAdapter.fromAPI(task))), delay(5000)
    );
  }
```

Оновимо сторінку та перевіримо роботу

Як результат, бачимо відображення анімації очікування завантаження.

6) Далі додамо можливість пошуку за назвою, описом або автором. Для цього у хуку ініціалізації **task-list.component.ts** змінимо отримання списку завдань на наступний код:

```
hasLoading: boolean = false;

filterControl :FormControl<string | null>  = new FormControl('');

constructor(  no usages  new *
  private taskStateService: TaskStateService,
```

```
ngOnInit(): void {  no usages  Kryvyi Yaroslav *
  this.taskStateService.loadTasks();

  this.myTasks$ = combineLatest([
    this.taskStateService.task$,
    this.filterControl.valueChanges.pipe(
      startWith(''),
      debounceTime(300),
      distinctUntilChanged()
    )
  ]).pipe(
    map(([tasks, filter] : [Task[], string | null] ) : Task[] =>
      tasks.filter(task : Task =>
        task.title.toLowerCase().includes(filter ?? ''.toLowerCase()) ||
        task.description?.toLowerCase().includes(filter ?? ''.toLowerCase()) ||
        task.assignee.toLowerCase().includes(filter ?? ''.toLowerCase())
      )),
  );
```

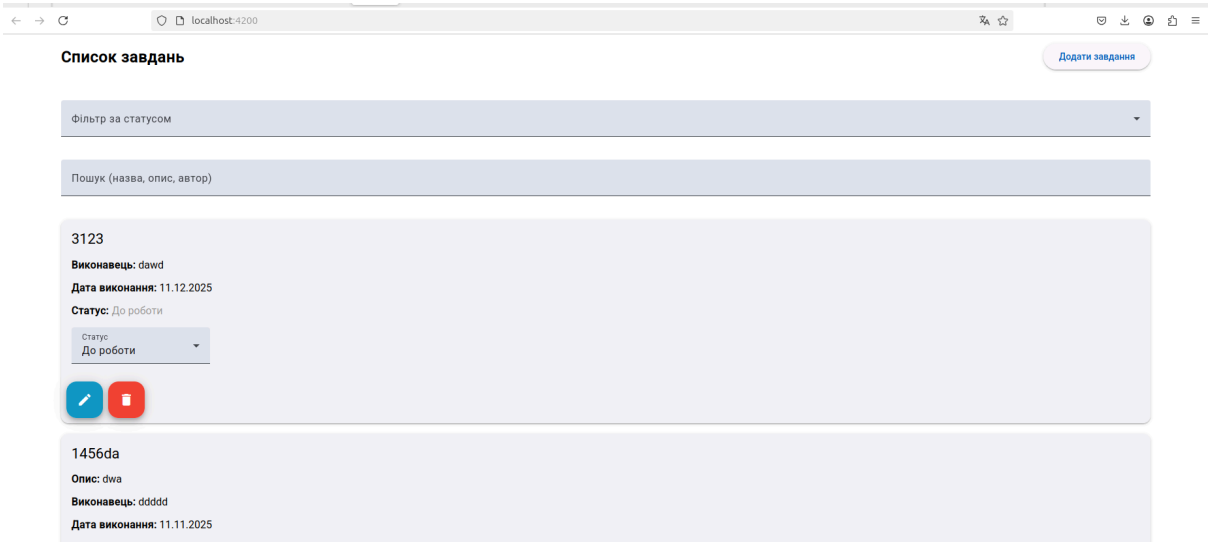
Використовуємо combineLatest щоб об’єднати значення двох потоків і емітити масив, коли хтось з них оновлюється. Перший поток це вже знайомий нам this.taskStateService.task$, який є Observable змінної _tasks$, а другий це підписка на значення FormControl, у ньому ми використовуємо наступні опції:

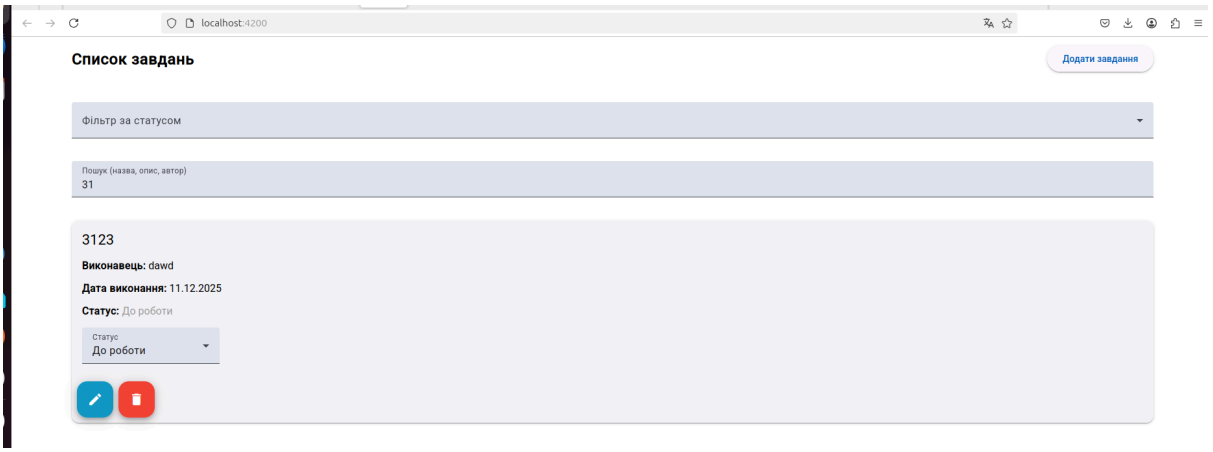
- startWith('') - оператор з RxJS використовується для випуску початкового значення перед тим, як Observable почне випускати значення, що надходять від елемента керування;
- debounceTime(300) - оператор з RxJS використовується для затримки випуску значень від Observable. У нашому випадку він чекає 300 мілісекунд (0.3 секунди) після останньої зміни значення, перш ніж випустити нове значення. Корисно, щоб запобігти швидкому вводу;
- distinctUntilChanged() - оператор з RxJS використовується для випуску лише тих значень, які відрізняються від попереднього значення.

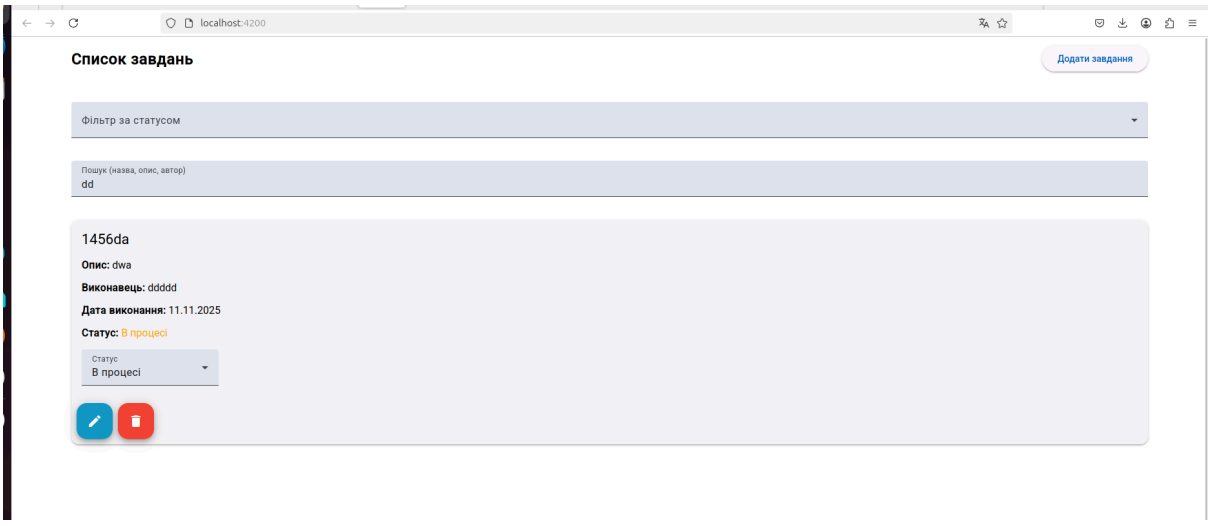
Додамо нове поле у html

```
</mat-form-field>
<mat-form-field>
  <mat-label>Пошук (назва, опис, автор)</mat-label>
  <input matInput type="text" id="filter" [formControl]="filterControl">
</mat-form-field>
@if (hasLoading) {
  <mat-spinner></mat-spinner>
```

Перевіримо роботу

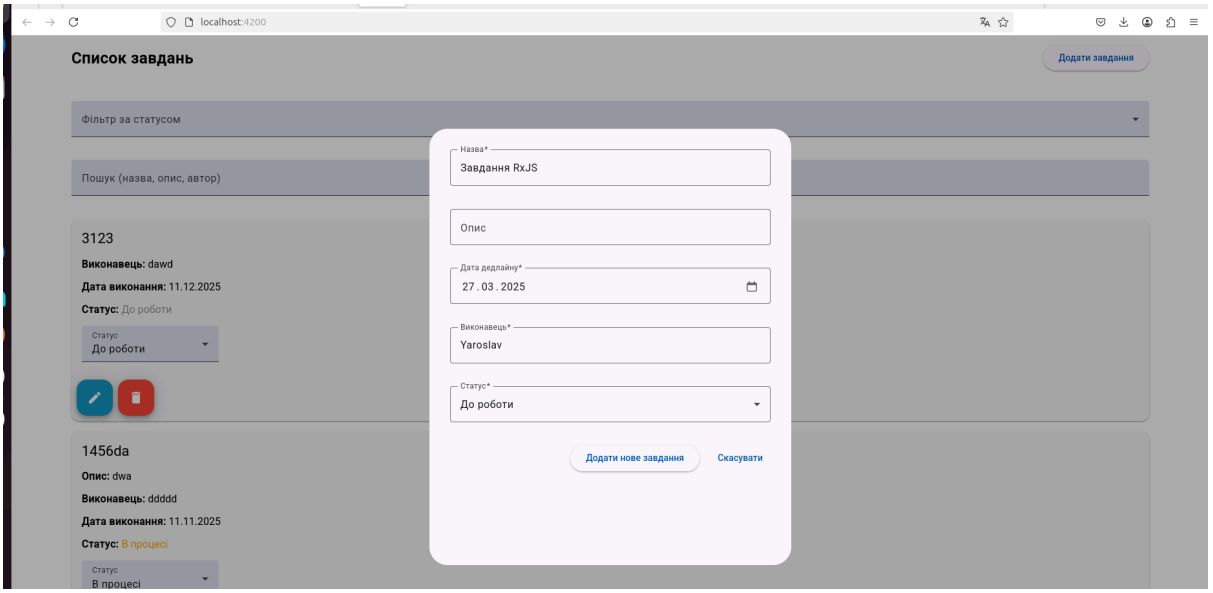

localhost:4200
Список завдань
Додати завдання
Фільтр за статусом
Пошук (назва, опис, автор)
3123
Виконавець: dawd
Дата виконання: 11.12.2025
Статус: До роботи
Статус
До роботи
1456da
Опис: dwa
Виконавець: ddddd
Дата виконання: 11.11.2025


localhost:4200
Список завдань
Додати завдання
Фільтр за статусом
Пошук (назва, опис, автор)
31
3123
Виконавець: dawd
Дата виконання: 11.12.2025
Статус: До роботи
Статус
До роботи


localhost:4200
Список завдань
Додати завдання
Фільтр за статусом
Пошук (назва, опис, автор)
dd
1456da
Опис: dwa
Виконавець: ddddd
Дата виконання: 11.11.2025
Статус: В процесі
Статус
В процесі

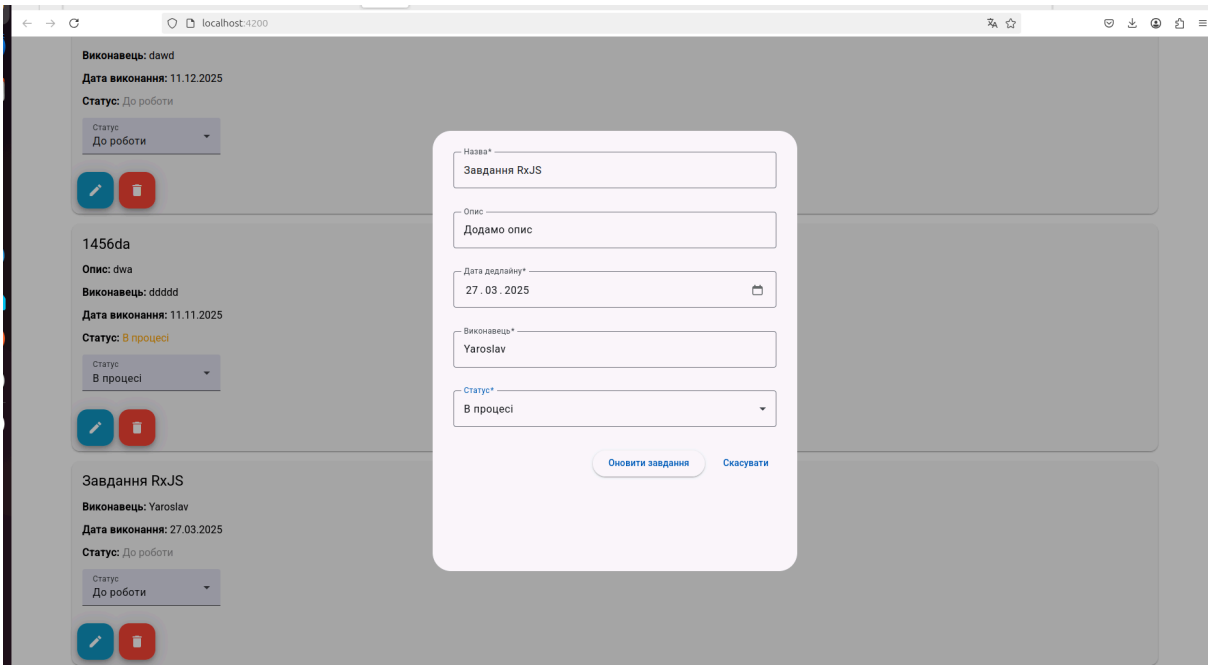
Перевіримо роботу додавання, оновлення, видалення, а також реакції на помилку

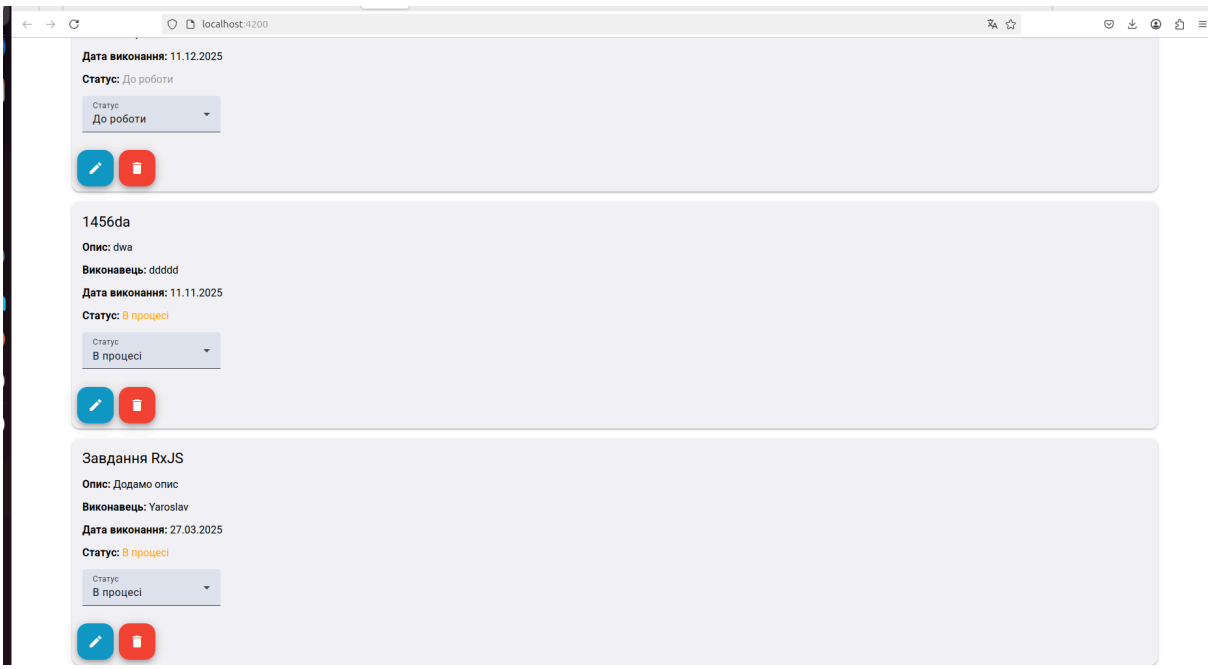
Створення



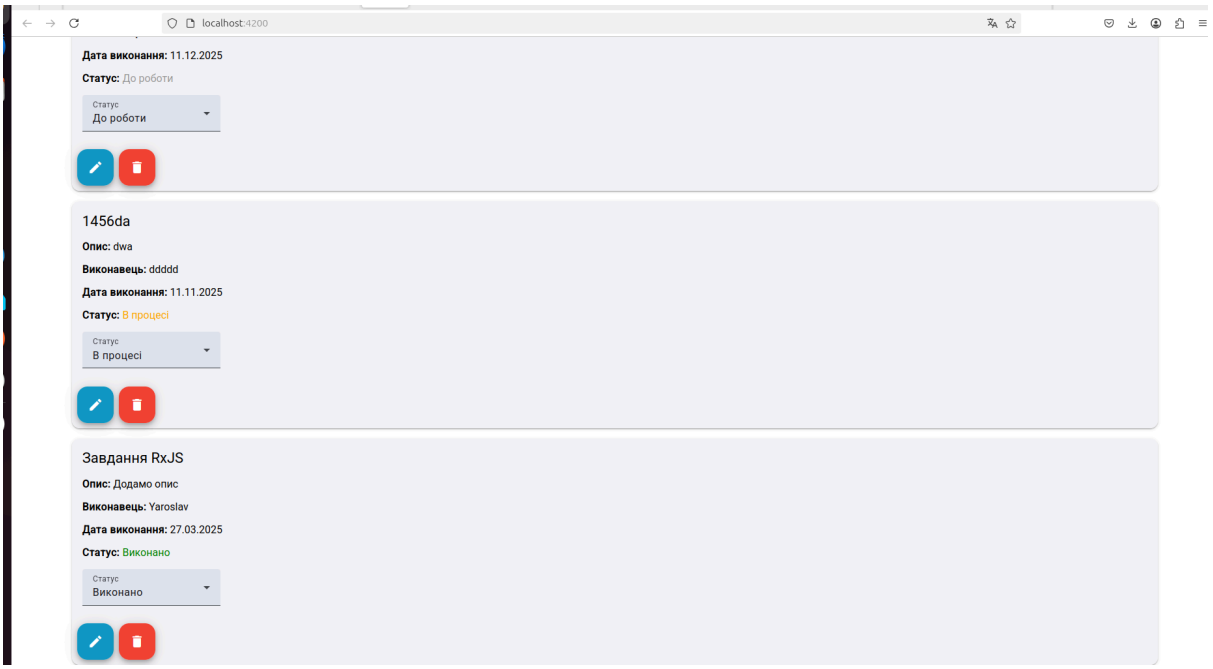

localhost:4200
Виконавець: dawd
Дата виконання: 11.12.2025
Статус: До роботи
Статус
До роботи
1456da
Опис: dwa
Виконавець: ddddd
Дата виконання: 11.11.2025
Статус: В процесі
Статус
В процесі
Завдання RxJS
Виконавець: Yaroslav
Дата виконання: 27.03.2025
Статус: До роботи
Статус
До роботи

Оновлення

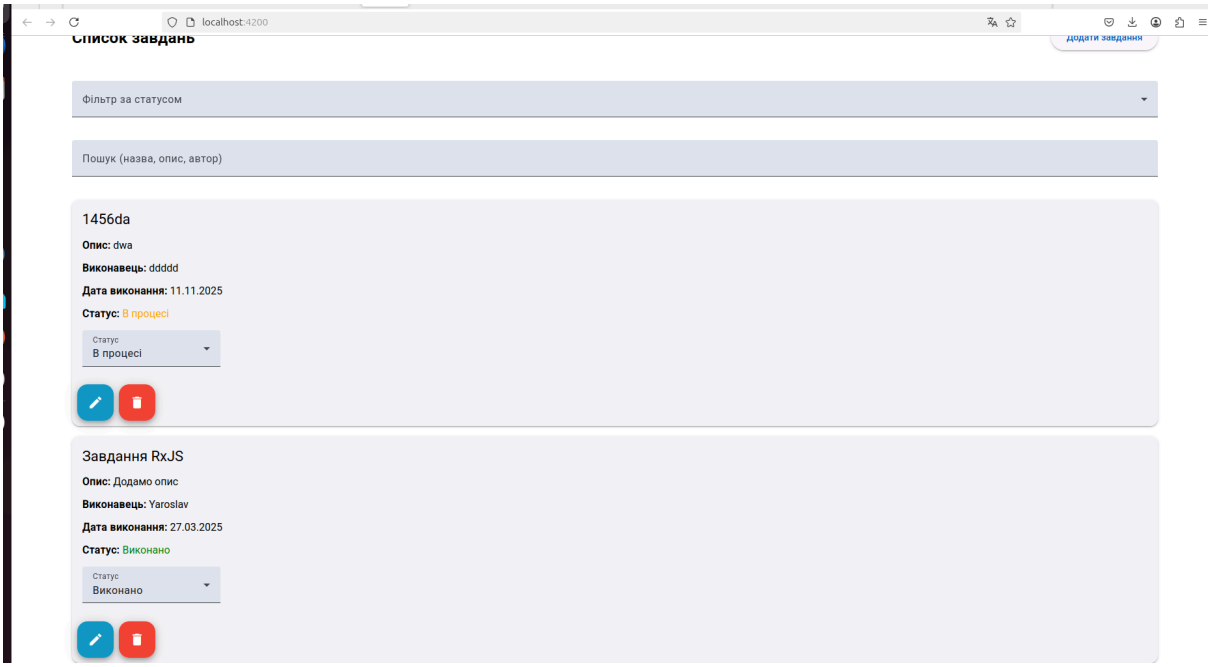

localhost:4200
Виконавець: dawd
Дата виконання: 11.12.2025
Статус: До роботи
Статус
До роботи
1456da
Опис: dwa
Виконавець: ddddd
Дата виконання: 11.11.2025
Статус: В процесі
Статус
В процесі
Завдання RxJS
Виконавець: Yaroslav
Дата виконання: 27.03.2025
Статус: До роботи
Статус
До роботи
Назва*
Завдання RxJS
Опис
Додамо опис
Дата дедлайну*
27 . 03 . 2025
Виконавець*
Yaroslav
Статус*
В процесі
Оновити завдання
Скасувати



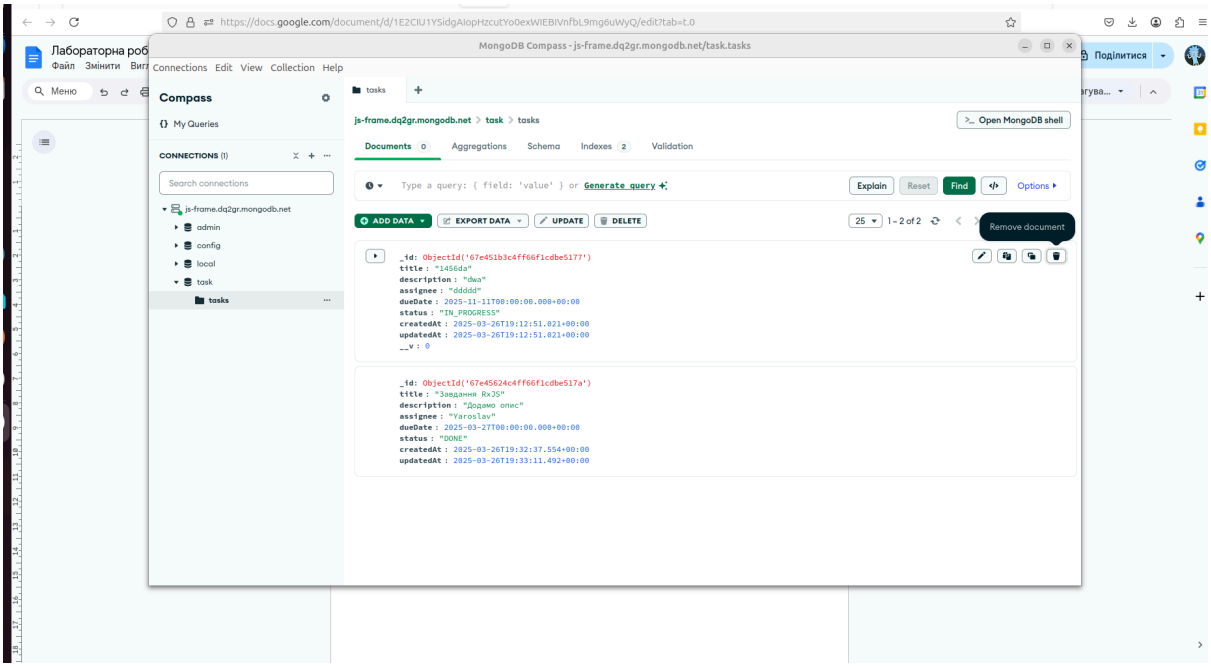
Оновлення статусу на DONE

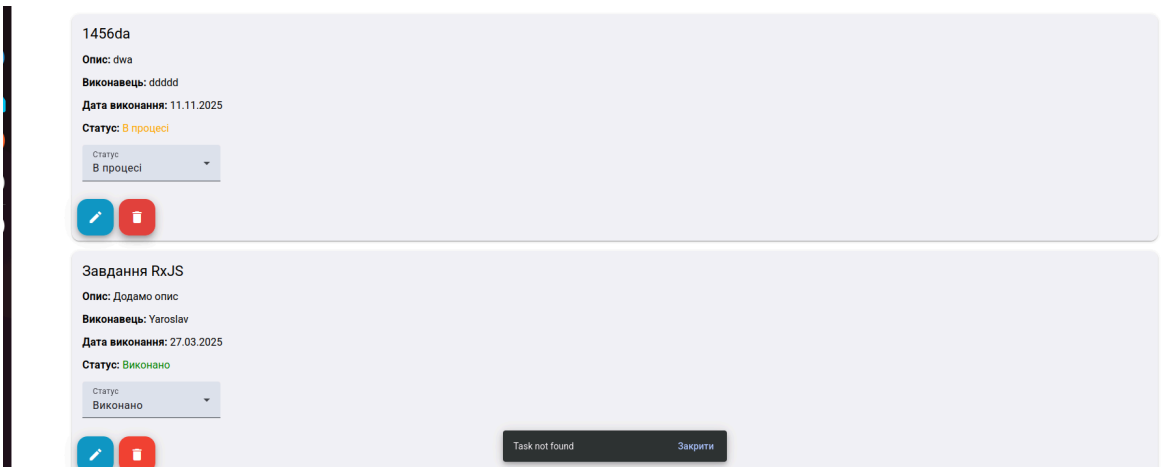


Видалення першого запису

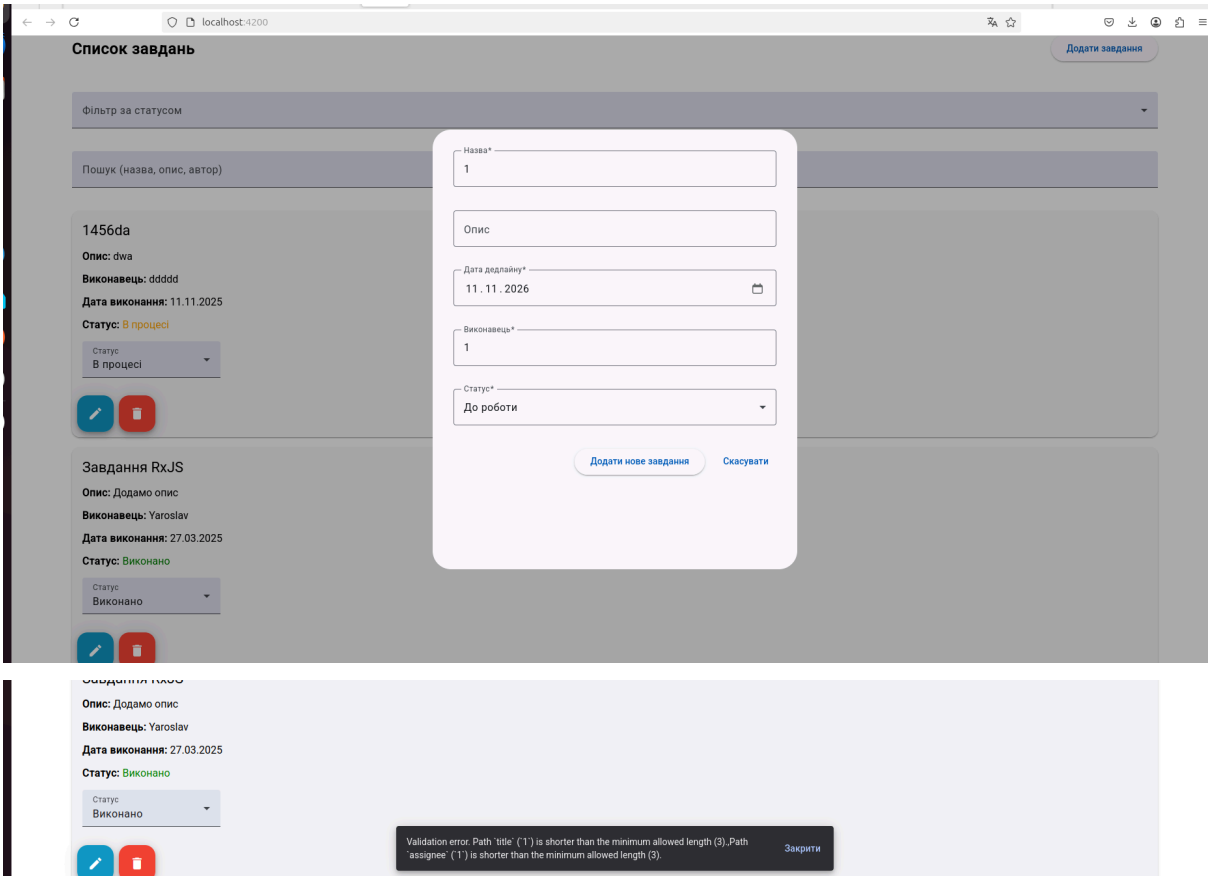


Імітуємо помилку (видалимо 2-й запис напряму з БД, а потім спробуємо видалити з інтерфейсу)





Імітуємо помилку (створення з невалідними полями, на стороні сервера є перевірка на мінімальну довжину, зі сторони клієнта - відсутня)



Як результат ми розділили зони відповідальності між компонентами, створили state-сервіс, який є загальним сховищем для списку завдань, створили адаптери, тим самим зменшивши код моделі серверу та уникнули потенціальних помилок.

Контрольні питання

1) У чому різниця між Subject та BehaviorSubject?
2) Коли спрацьовує AsyncSubject?
3) Які є способи відписки? Чи потрібно відписуватися від потоку, якщо він використовується у зв’язці з AsyncPipe у шаблоні?
4) Для чого використовуються адаптери у Angular?
5) Яка різниця між hot та cold Observable?