Лабораторна робота № 5. Сервіси та DI.

Мета: навчитися розділяти представлення та логіку, а також використовувати сервіси у компонентах.

Завдання

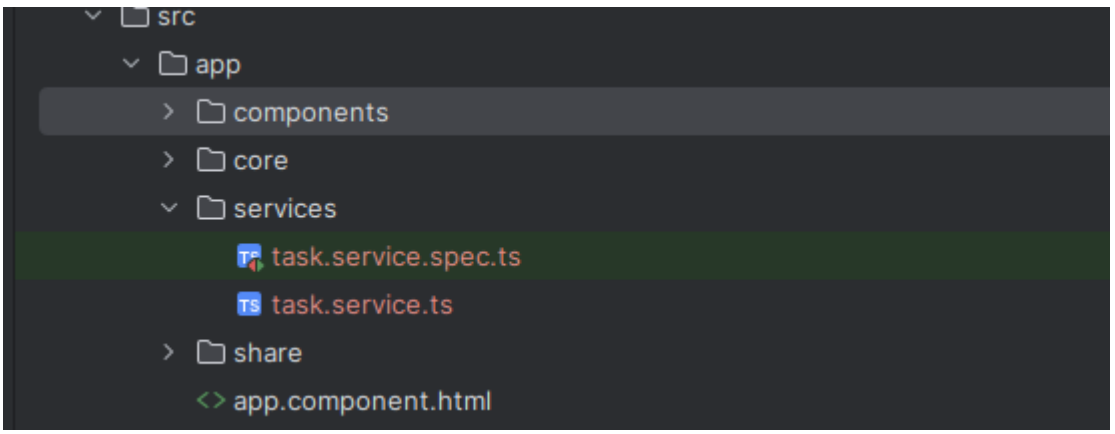
1) Ознайомитися з матеріалом лекції № 5;
2) Виконати пункти 1-6 ходу роботи;
3) У звіті відобразити кроки 1-6 ходу роботи;
4) Надати відповіді на контрольні питання.

Хід роботи

1) Для початку створимо сервіс, який буде містити логіку взаємодії із завданнями. Для цього використовуємо команду: ng g s services/task

```
svarog@svarog:~/projects/angular-course$ ng g s services/task
CREATE src/app/services/task.service.spec.ts (347 bytes)
CREATE src/app/services/task.service.ts (133 bytes)
```

Це створить файл task.service.ts у теці **app/services**



2) Переходимо до реалізації сервісу, так як він буде працювати з mock-даними, то для початку потрібно підключити їх до сервісу та записати у змінну

```
import { Injectable } from '@angular/core';
import {Task} from '../core/models/task.model';
import {tasks} from '../core/mock_data/tasks';

@Injectable({  Show usages
  providedIn: 'root'
})
export class TaskService {

  private tasks: Task[] = [...tasks];

  constructor() { }  no usages
```

Зробимо змінну приватною, щоб унеможливити звертання до неї напряму.

3) Реалізуємо методи по взаємодії з даними, почнемо з додавання даних:
    a) Створюємо метод **addTask(newTask: Task)** та переносимо у нього логіку генерації id та запис до масиву завдань;

```
export class TaskService {

  private tasks: Task[] = [...tasks];

  constructor() { }  no usages

  addTask(newTask: Task): void {  no usages
    const maxId :number  = this.tasks.length > 0 ? Math.max(...this.tasks.map(task :Task  => task.id)) : 0;
    newTask = {
      ...newTask,
      id: maxId + 1,
    }
    this.tasks.push(newTask);
  }
}
```

b) Далі переходимо до логіки оновлення завдання, для цього створюємо метод **updateTask(updateTask: Task)** та переносимо до нього логіку оновлення завдання:

```
  }

  updateTask(updateTask: Task): void {  no usages
    this.tasks = this.tasks.map(t :Task  =>
      t.id === updateTask.id ? { ...updateTask } : t
    );
  }
}
```

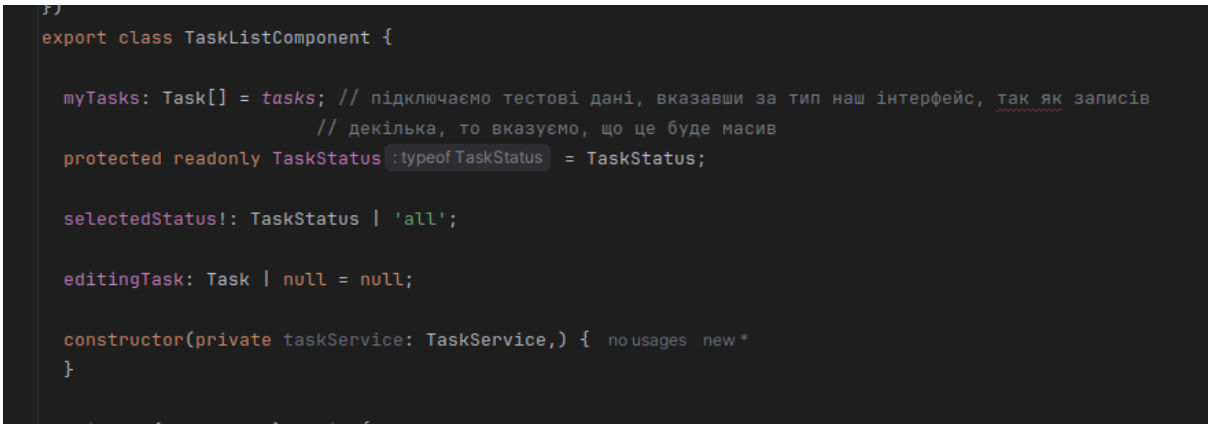
c) Наступний кроком реалізуємо метод видалення завдання, перенесемо логіку видалення до методу **deleteTask(id: number)**:

```
  deleteTask(index: number): void {  no usages
    this.tasks = this.tasks.filter(task :Task  => task.id !== index);
  }
```

d) Останнім кроком буде реалізація методу **getTasks()**, який дозволить отримати поточний список завдань

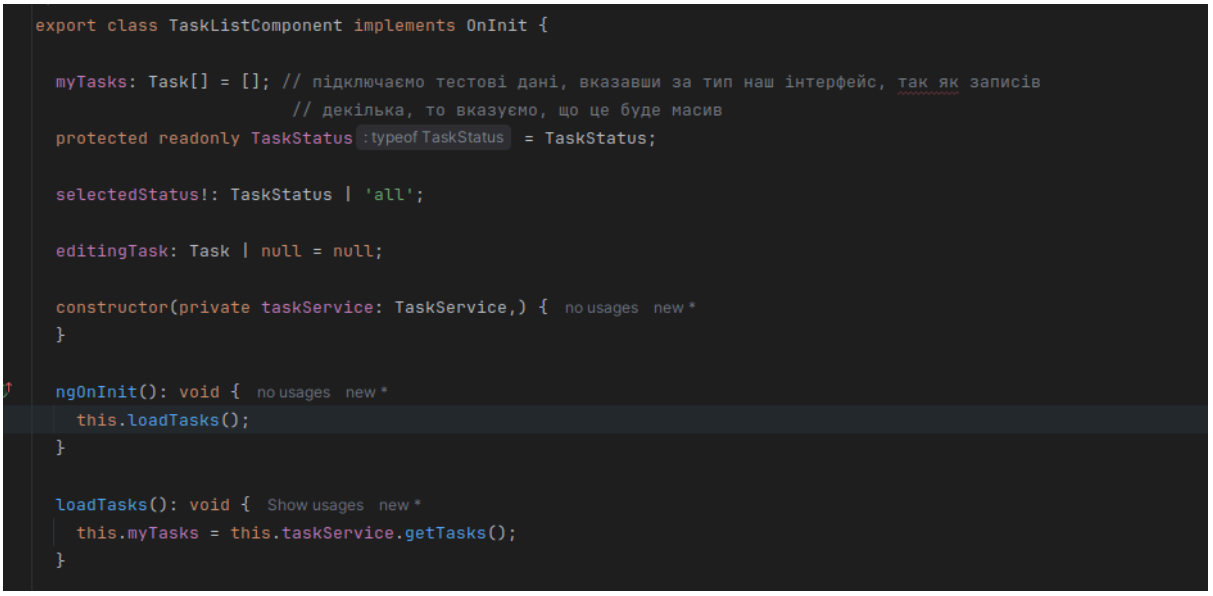
```
  getTasks(): Task[] {  no usages
    return this.tasks;
  }
```

4) Далі підключимо наш сервіс у конструктор компоненту, використовуючи DI. Так як у властивостях **@Injectable** стоїть значення **providedIn: 'root'**, то сервіс є Singleton, що означає, що є один спільний екземпляр, який будуть використовувати усі сервіси, компоненти та/або директиви. Також це означає, що не потрібно прописувати сервіс у масив **providers**. Підключаємо сервіс у файлі **task-list.component.ts**



```
export class TaskListComponent {

  myTasks: Task[] = tasks; // підключаємо тестові дані, вказавши за тип наш інтерфейс, так як записів
                           // декілька, то вказуємо, що це буде масив
  protected readonly TaskStatus :typeof TaskStatus  = TaskStatus;

  selectedStatus!: TaskStatus | 'all';

  editingTask: Task | null = null;

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

5) Наступний кроком змінимо методи компонента, які відповідали за взаємодію з масивом завдань. Для цього потрібно зробити наступні зміни:
    a) Ввести метод, який буде записувати масив завдань, які ми отримали із сервісу у змінну компонента. Також цей метод буде потрібний для оновлення змінної компоненту після взаємодії із сервісом (додавання, оновлення, видалення). Для цього створимо метод **loadTask()**, а токаж змінимо значення змінної myTasks



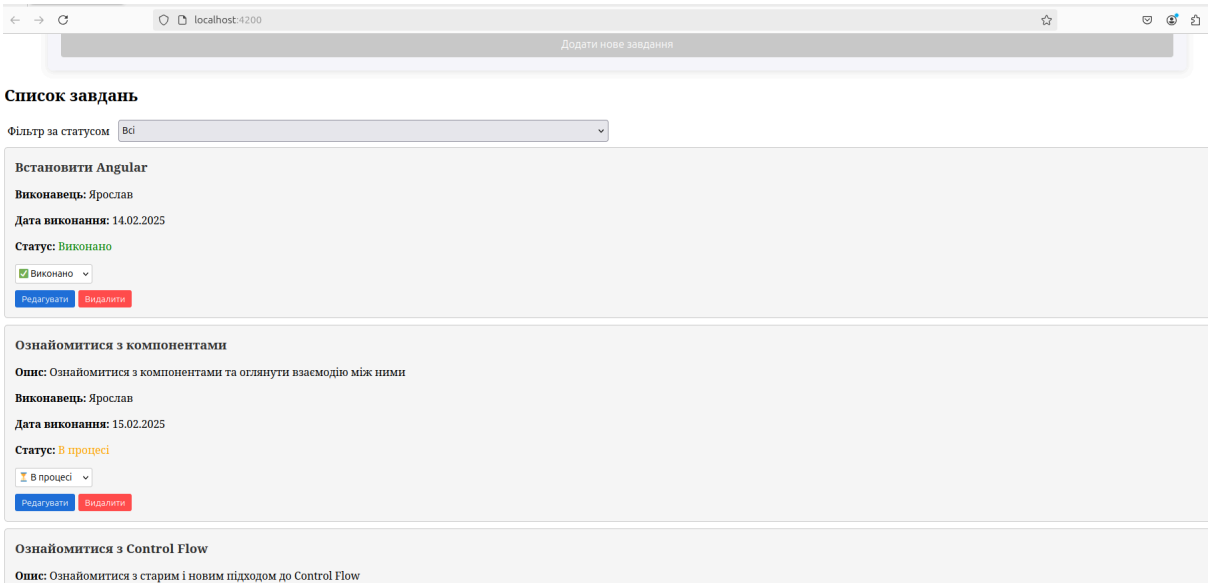
```
export class TaskListComponent implements OnInit {

  myTasks: Task[] = []; // підключаємо тестові дані, вказавши за тип наш інтерфейс, так як записів
                        // декілька, то вказуємо, що це буде масив
  protected readonly TaskStatus :typeof TaskStatus  = TaskStatus;

  selectedStatus!: TaskStatus | 'all';

  editingTask: Task | null = null;

  constructor(private taskService: TaskService,) {  no usages  new *
  }

  ngOnInit(): void {  no usages  new *
    this.loadTasks();
  }

  loadTasks(): void {  Show usages  new *
    this.myTasks = this.taskService.getTasks();
  }
```

Також використовуючи хук **ngOnInit**, який спрацьовує після ініціалізації компонента, викличемо новостворений метод для запису початкового масиву завдань до змінної компонента.

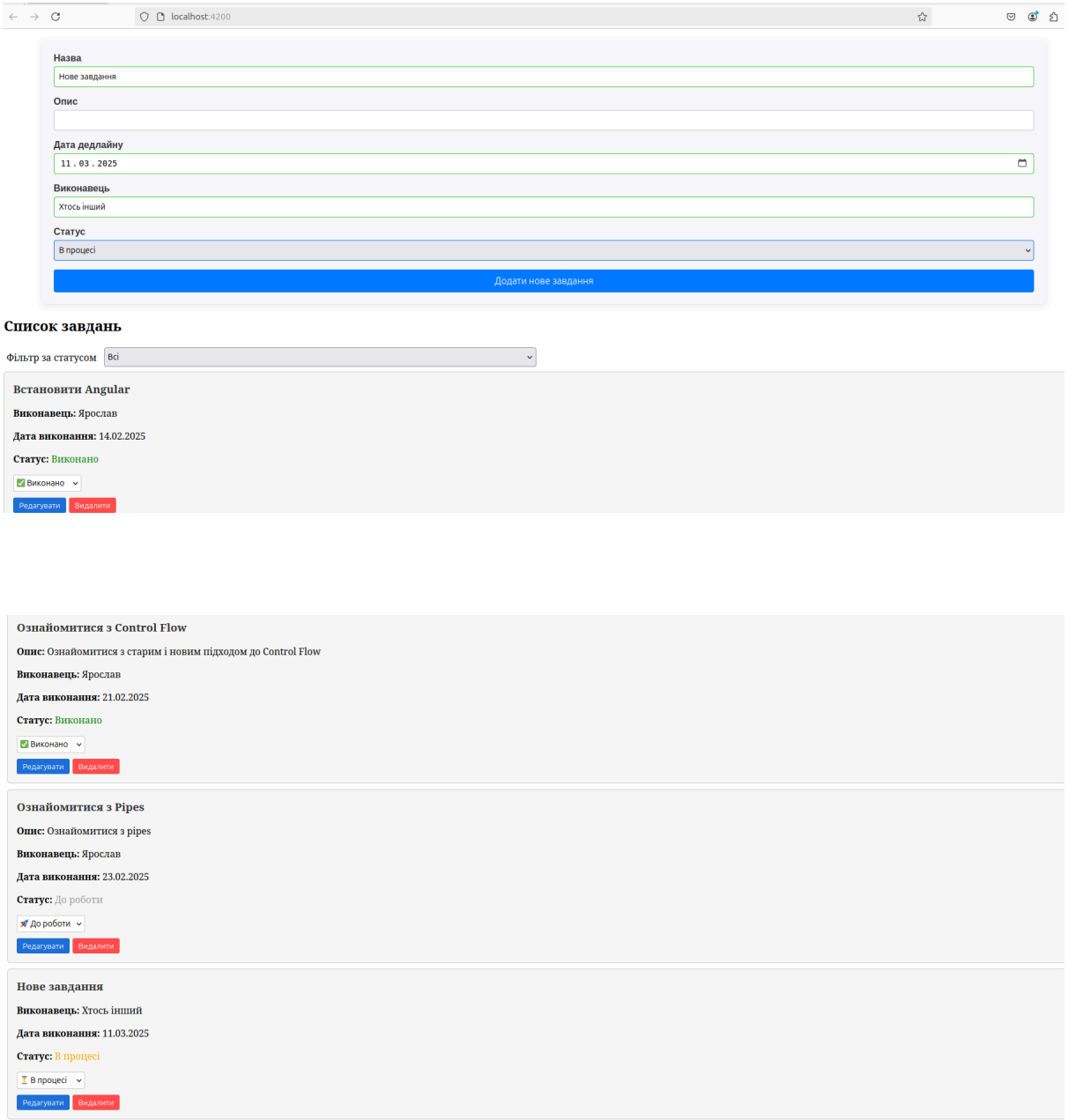
b) За таким же принципом змінимо функції додавання, оновлення та видалення завдань, замінивши логіку взаємодії з ними на виклики відповідних методів сервісу.

```
deleteTask(index: number): void {  Show usages   Kryvyi Yaroslav *
  this.taskService.deleteTask(index);
  this.loadTasks();
}

addTask(task: Task): void {  Show usages   Kryvyi Yaroslav *
  if (this.editingTask) {
    this.taskService.updateTask(task);
    this.editingTask = null;
  } else {
    this.taskService.addTask(task);
  }
  this.loadTasks();
}
```

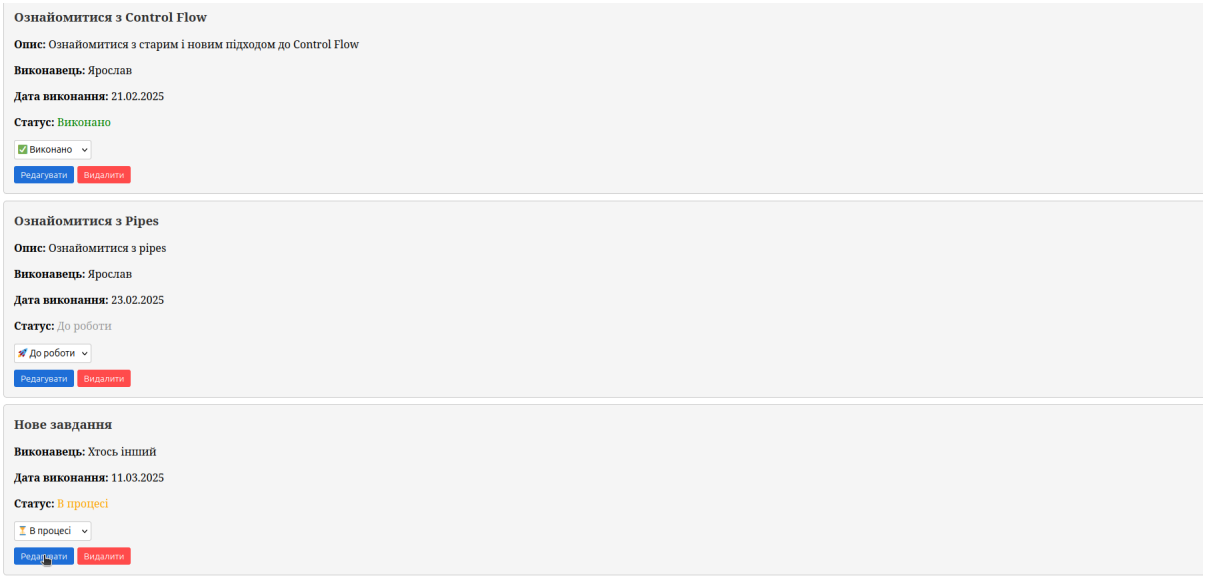
6) Перевіряємо роботу застосунку
   a) Відображення

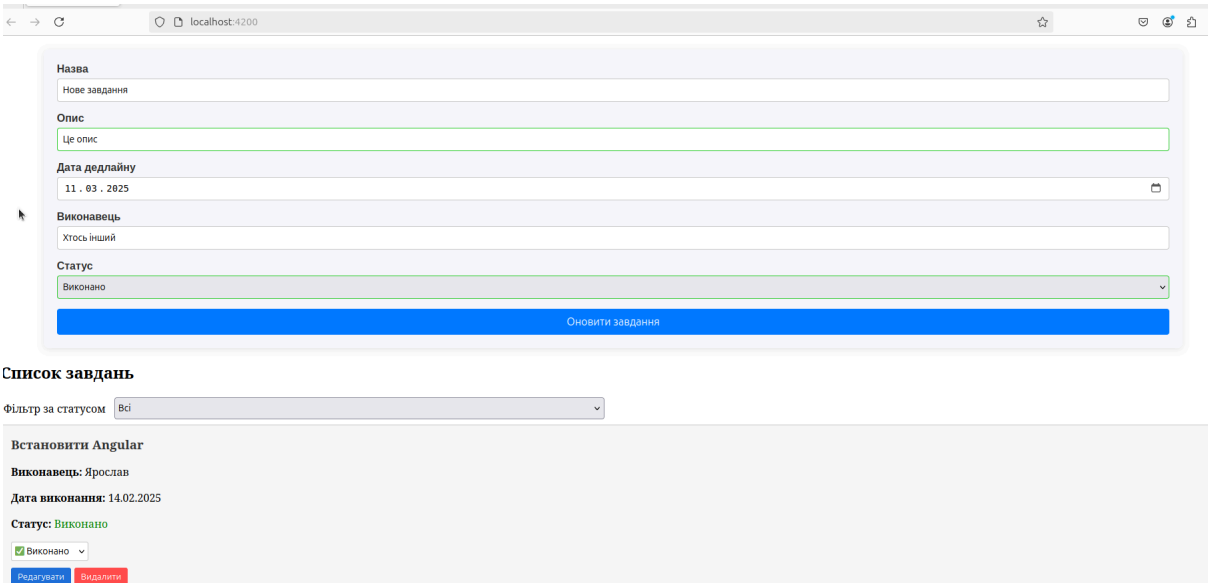


   b) Додавання нового завдання

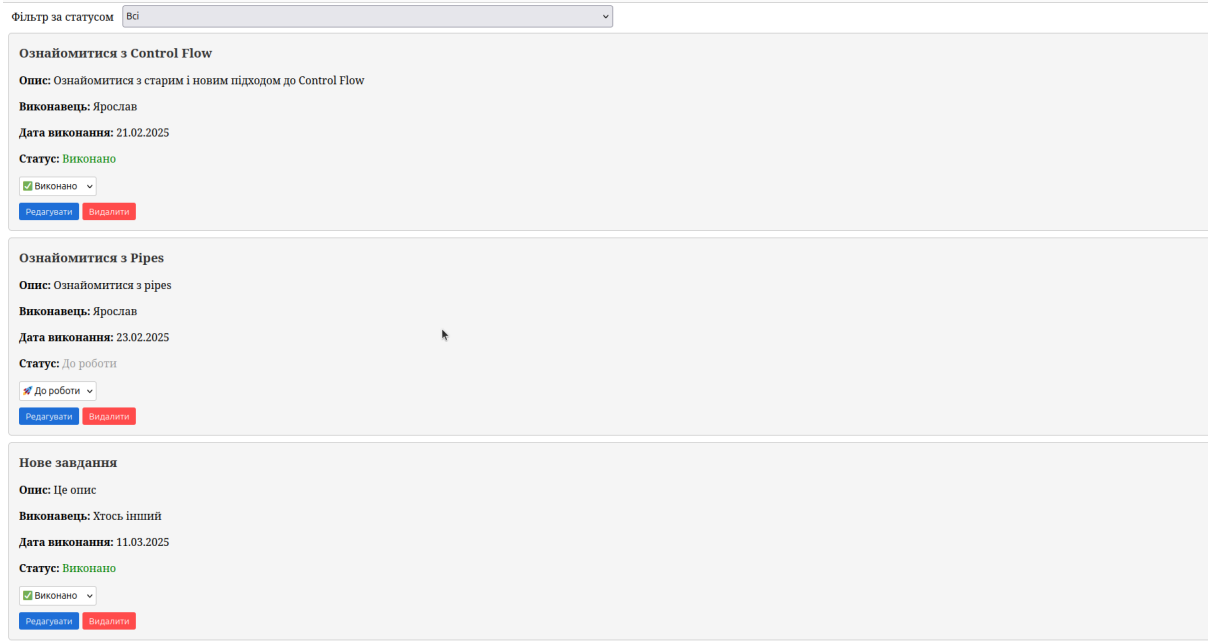


c) Оновимо це завдання

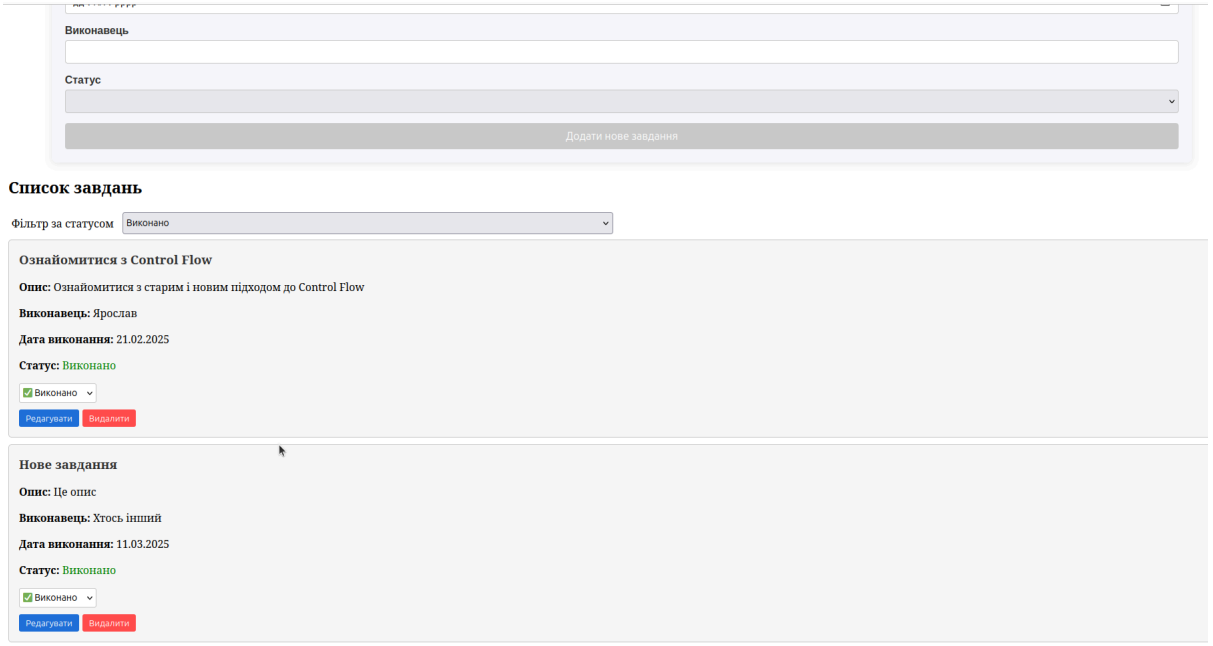

Ознайомитися з Control Flow
Опис: Ознайомитися з старим і новим підходом до Control Flow
Виконавець: Ярослав
Дата виконання: 21.02.2025
Статус: Виконано
✅ Виконано
Редагувати
Видалити
Ознайомитися з Pipes
Опис: Ознайомитися з pipes
Виконавець: Ярослав
Дата виконання: 23.02.2025
Статус: До роботи
🚀 До роботи
Редагувати
Видалити
Нове завдання
Виконавець: Хтось інший
Дата виконання: 11.03.2025
Статус: В процесі
⏳ В процесі
Редагувати
Видалити


localhost:4200
Назва
Нове завдання
Опис
Це опис
Дата дедлайну
11 . 03 . 2025
Виконавець
Хтось інший
Статус
Виконано
Оновити завдання
Список завдань
Фільтр за статусом
Всі
Встановити Angular
Виконавець: Ярослав
Дата виконання: 14.02.2025
Статус: Виконано
✅ Виконано
Редагувати
Видалити

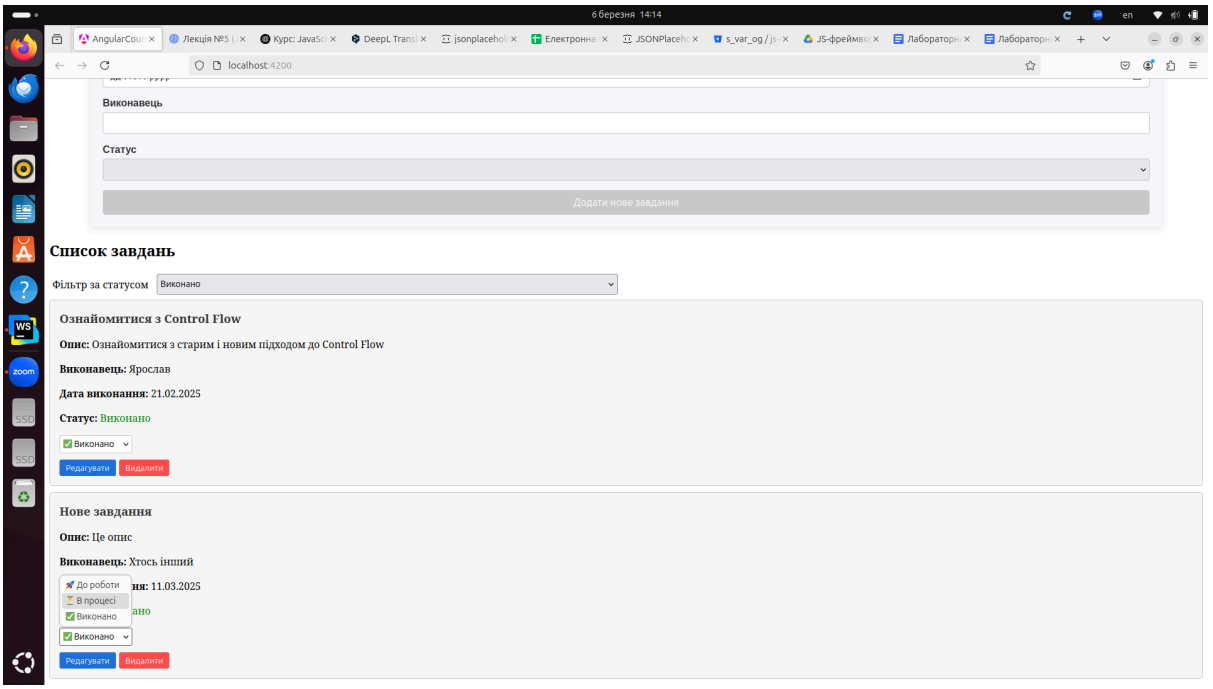
d) Видалимо перше та друге завдання



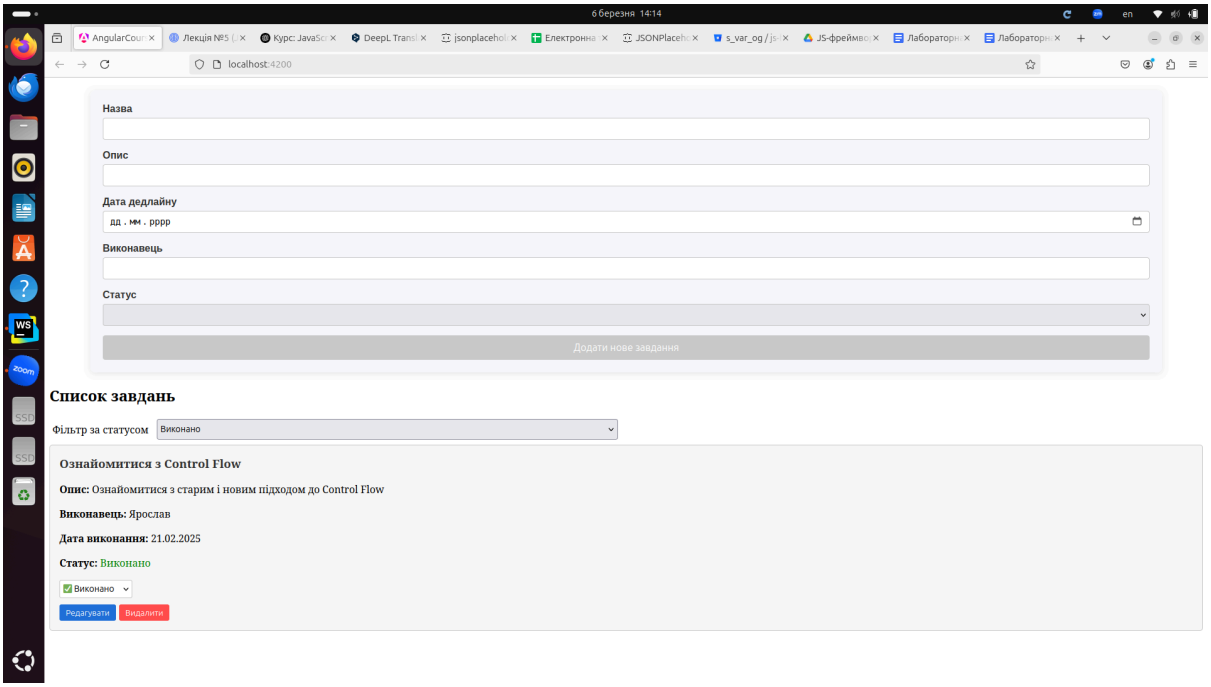
e) Протестуємо роботу фільтрації



Змінимо статус нового завдання



Результат



Як результат, ми отримали розділення логіки та відображення завдань, що зменшило код у компоненті та надало можливість взаємодіяти із завданнями із будь-якого компоненту, сервісу або директиви.

Контрольні питання

1) Який декоратор “повідомляє” Angular, що клас бути інжектований через DI?
2) Що таке DI та для чого воно використовується в Angular?
3) У який масив модуля, сервіса або директиви необхідно додати сервіс, якщо у декораторі @Injectable властивість providedIn: 'root'?
4) Який параметр необхідно передати властивості providedIn, щоб сервіс необхідно було додавати у масив providers?
5) Як працює дерево провайдерів?