Лабораторна робота № 3. Control Flow та pipes.

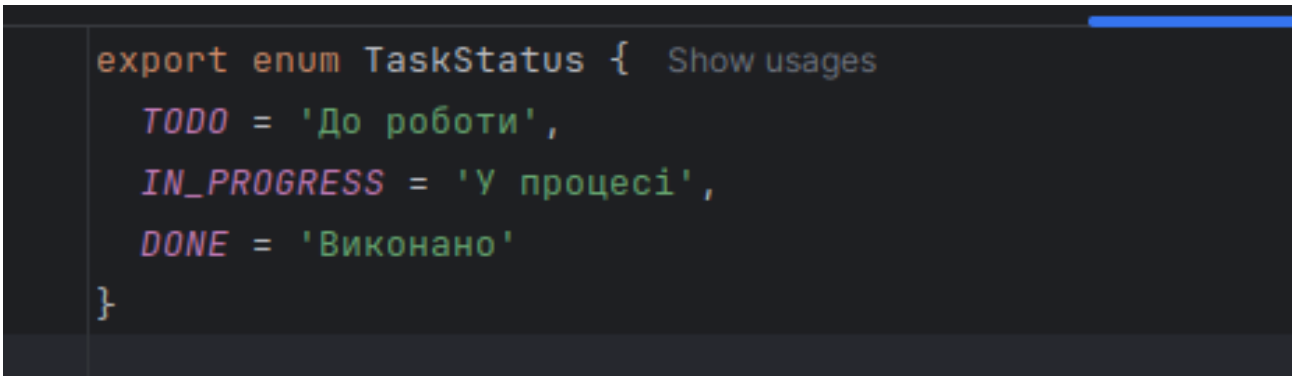
Мета: навчитися використовувати @if, @for, @switch, а також створювати власні pipes.

Завдання

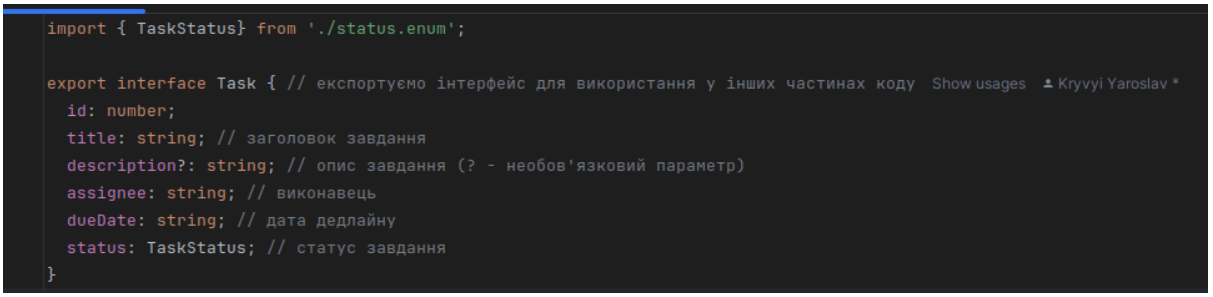
1) Ознайомитися з матеріалом лекції № 3;
2) Виконати пункти 1- ходу роботи;
3) У звіті відобразити кроки 1- ходу роботи;
4) Надати відповіді на контрольні питання.

Хід роботи

1) Для початку розширимо інтерфейс завдання додавши до нього нову властивість “status”, для цього виконаємо наступні кроки:
    a) Створимо файл **core/models/status.enum.ts** і пропишемо у ньому наступні статуси:



    b) Імпортуємо цей enum у файл інтерфейсу завдання



    c) Редагуємо тестові дані

```ts
import {Task} from '../models/task.model';
import {TaskStatus} from '../models/status.enum';

export const tasks: Task[] = [  Show usages  Kryvyi Yaroslav *
  {
    id: 0,
    title: 'Встановити Angular',
    assignee: 'Ярослав',
    dueDate: '14.02.2025',
    status: TaskStatus.DONE
  },
  {
    id: 1,
    title: 'Ознайомитися з компонентами',
    description: 'Ознайомитися з компонентами та оглянути взаємодію між ними',
    assignee: 'Ярослав',
    dueDate: '15.02.2025',
    status: TaskStatus.IN_PROGRESS
  },
  {
    id: 2,
    title: 'Ознайомитися з Control Flow',
    description: 'Ознайомитися з старим і новим підходом до Control Flow',
    assignee: 'Ярослав',
    dueDate: '25.02.2025',
    status: TaskStatus.DONE
```

2) Далі необхідно переписати шаблони **task-list.component.ts** та **task-item.component.ts**, почнемо з першого.

Як можна побачити, зараз для відображення нового завдання необхідно викликати тег <app-task-item> та передати у нього відповідні аргументи.

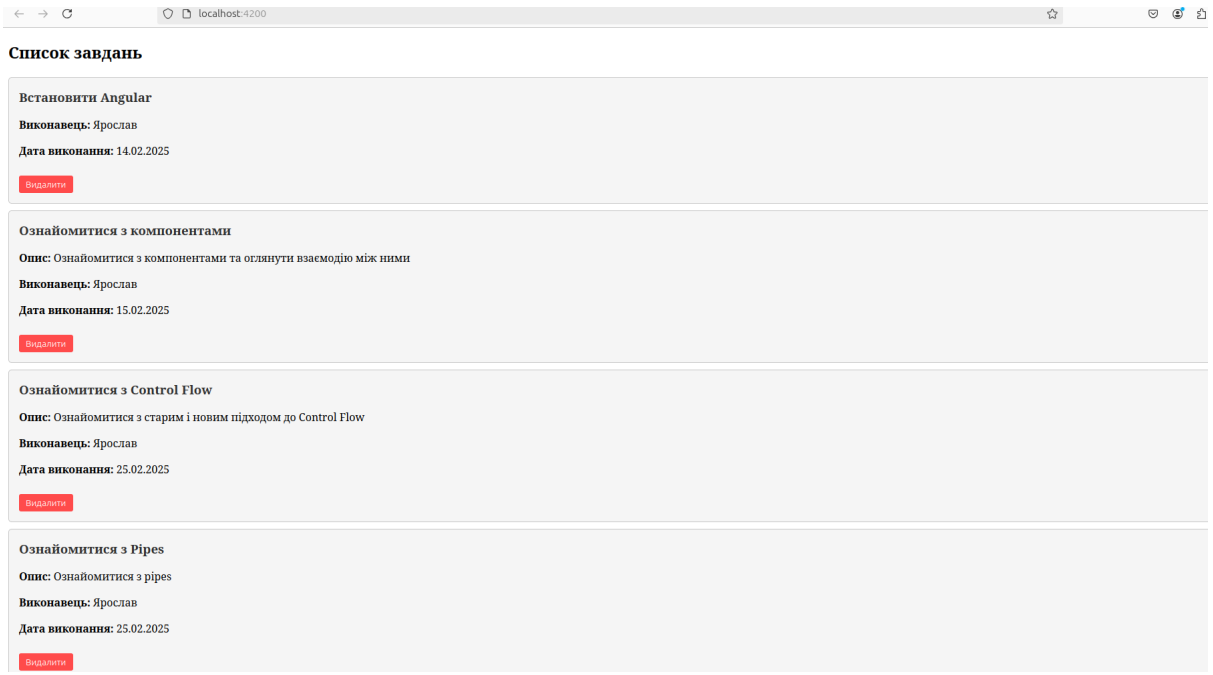
```
s  task-item.component.ts  task-item.component.scss  task-list.component.html  task-list.component.ts  app.component.html
1  <h2>Список завдань</h2>
2  <app-task-item
3    [task]="myTasks[0]"
4    >
5  </app-task-item>
6
7  <app-task-item
8    [task]="myTasks[1]"
9  >
10 </app-task-item>
```

Це незручно тим, що список завдань може бути великий, і прописувати виклик під кожне буде займати багато часу, шаблон буде містити багато строк коду, стане незручним для редагування, а також змушує при кожній зміні списку завдань вносити зміни вручну. Для автоматичної генерації карток завдань на основі масиву даних можна

використовувати цикли *ngFor або @for. Перепишемо наш код використовуючи цикл @for

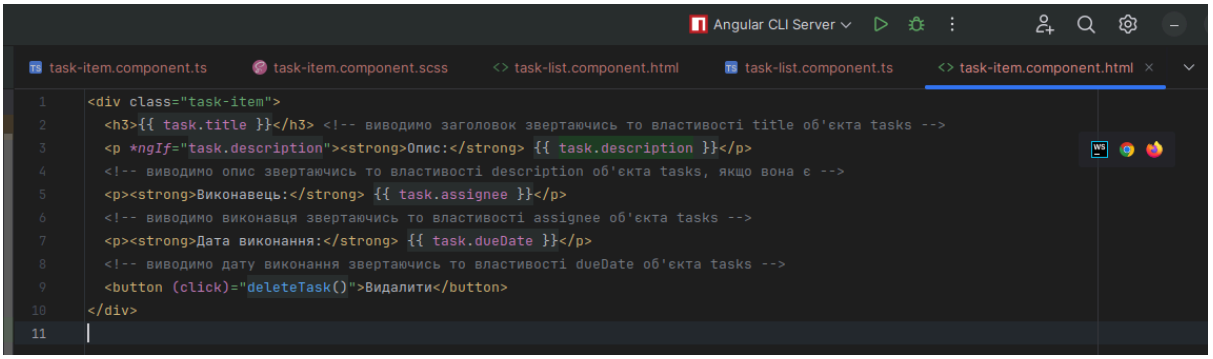
```
<h2>Список завдань</h2>
@for(task of myTasks; track task.id; let index = $index) {
  <app-task-item
    [task]="task"
    (taskDeleted)="deleteTask($event)"
  >
  </app-task-item>
} @empty {
  <p>Курси відсутні</p>
}
```

Ми розклали колекцію myTasks на окремі елементи (завдання), та передали кожне із них у вигляді @Input для дочірнього елементу. Також можна побачити вираз **track task.id**, він допомагає Angular не рендерити увесь список завдань при додаванні або вилучені завдання, а орієнтуючись на унікальну властивість рендерити або прибирати тільки конкретний елемент. Перевіримо, що з переходом на новий синтаксис не змінилося відображення списку завдань.



Як бачимо відображення не змінилося, але тепер ми можемо вивести всі елементи колекції за раз.

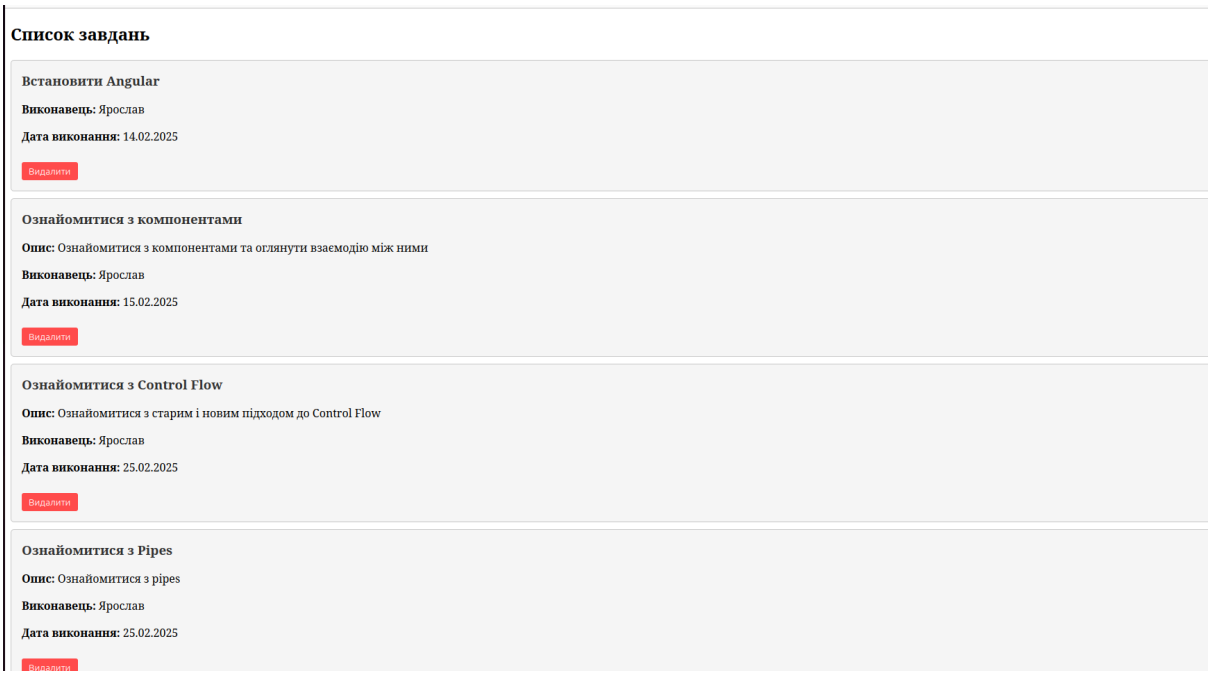
Перейдемо до файлу **task-item.component.ts**, та ознайомимося з його поточним вмістом.



Можемо побачити вираз ***ngIf=”task.description”** у 3 строці, який перевіряє, чи є поточного завдання опис, і якщо є, то відображає його. Змінимо *ngIf на @if

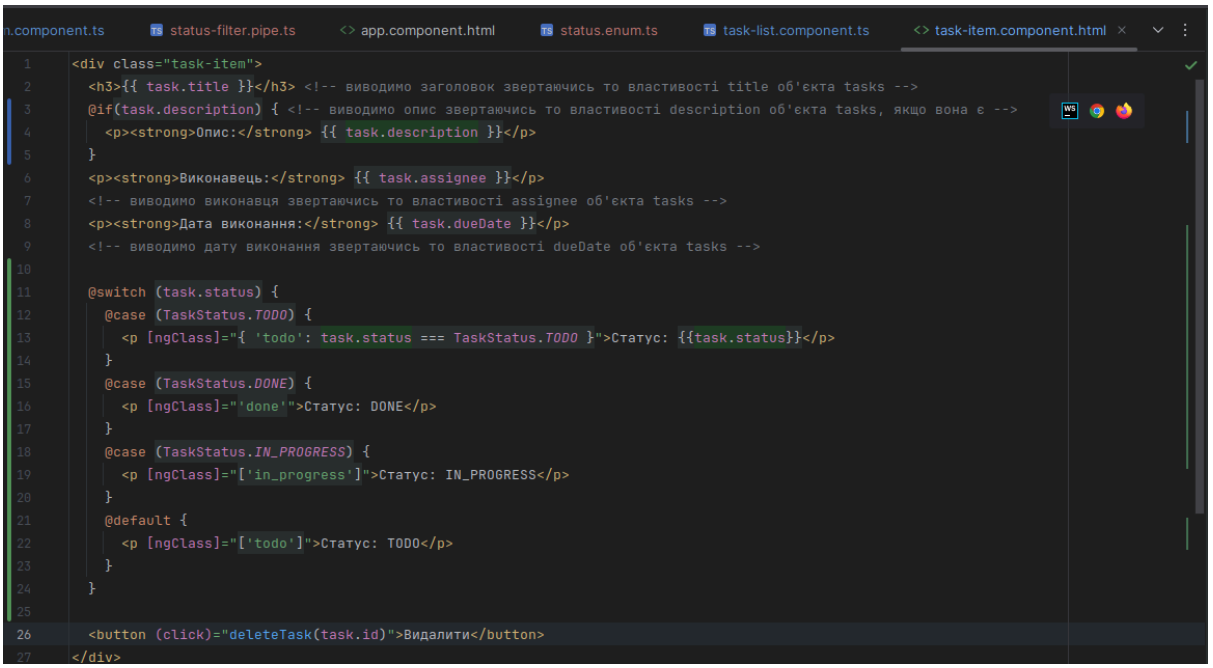
```
<div class="task-item">
  <h3>{{ task.title }}</h3> <!-- виводимо заголовок звертаючись то властивості title об'єкта tasks -->
  @if(task.description) { <!-- виводимо опис звертаючись то властивості description об'єкта tasks, якщо вона є -->
    <p><strong>Опис:</strong> {{ task.description }}</p>
  }
  <p><strong>Виконавець:</strong> {{ task.assignee }}</p>
  <!-- виводимо виконавця звертаючись то властивості assignee об'єкта tasks -->
  <p><strong>Дата виконання:</strong> {{ task.dueDate }}</p>
  <!-- виводимо дату виконання звертаючись то властивості dueDate об'єкта tasks -->
  <button (click)="deleteTask(task.id)">Видалити</button>
</div>
```

Перевіряємо відображення

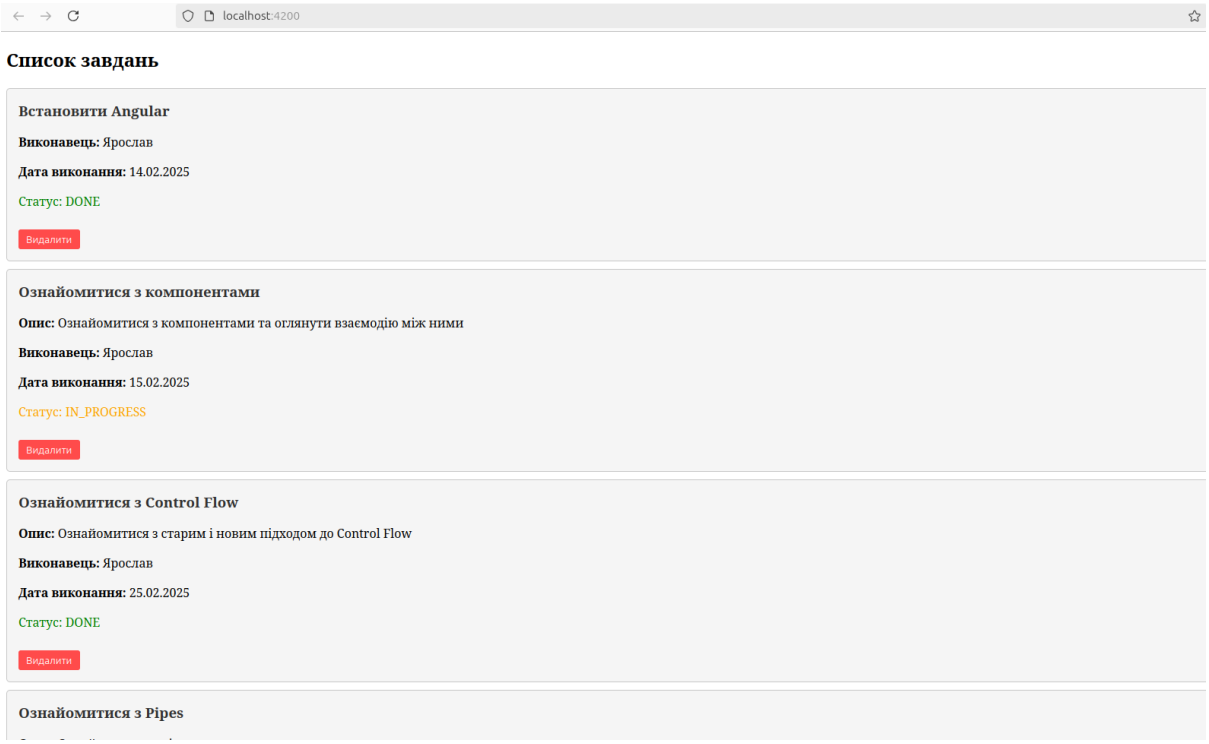


Можемо побачити, що відображення працює коректно і обробляє випадки, коли відсутній опис. Також у завдання з’явилася нова властивість. Її можна відобразити декількома способами, давайте їх розглянемо.

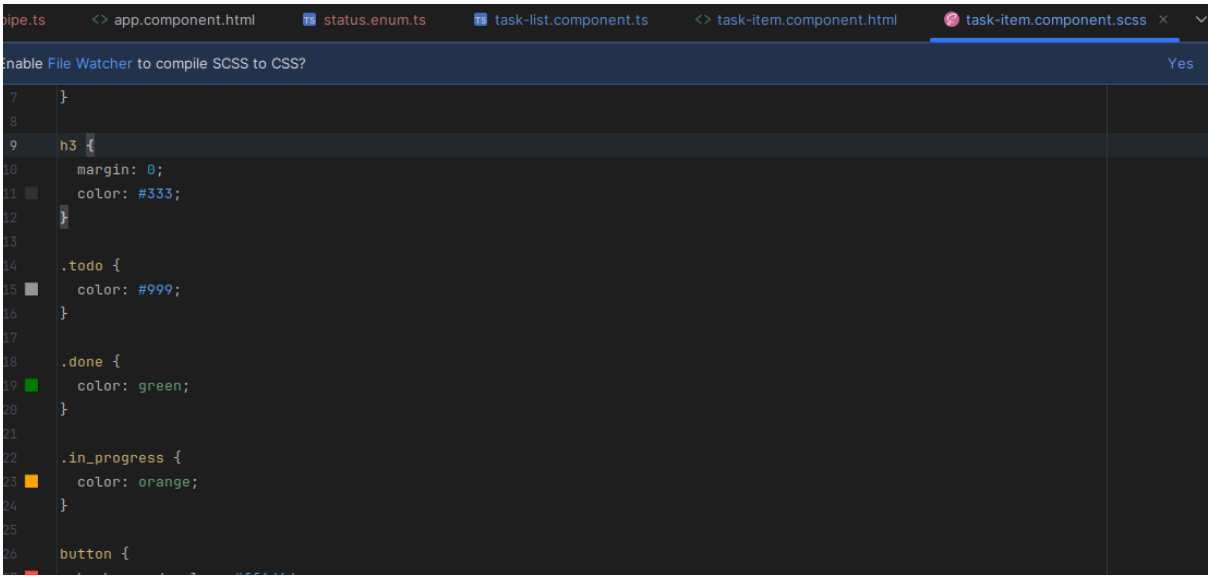
Спочатку використаємо @switch, ідея полягає у тому, що ми отримуємо на вхід статус поточно завдання, тоді як @case будуть підв’язані під список завдань зі створеного enum



Перевіримо відображення на сторінці



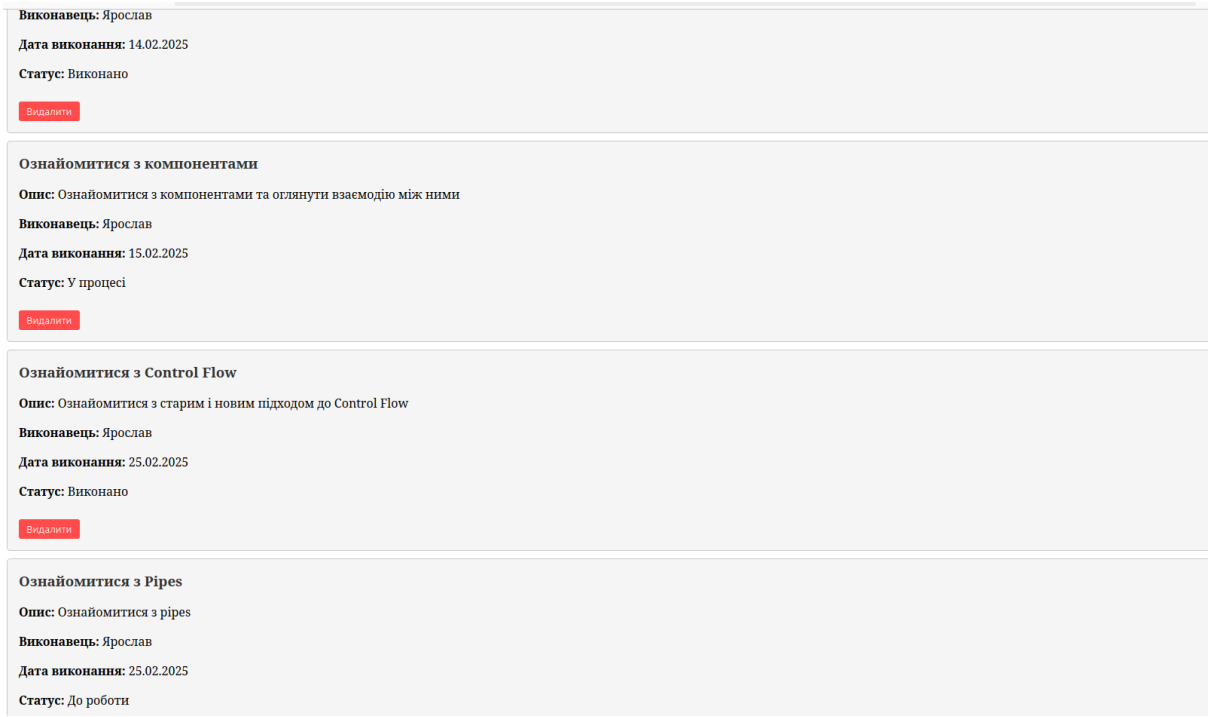
Можемо побачити, що у картці з’явилося нове поле статусу, також ми додали динамічні стилі через [ngClass]



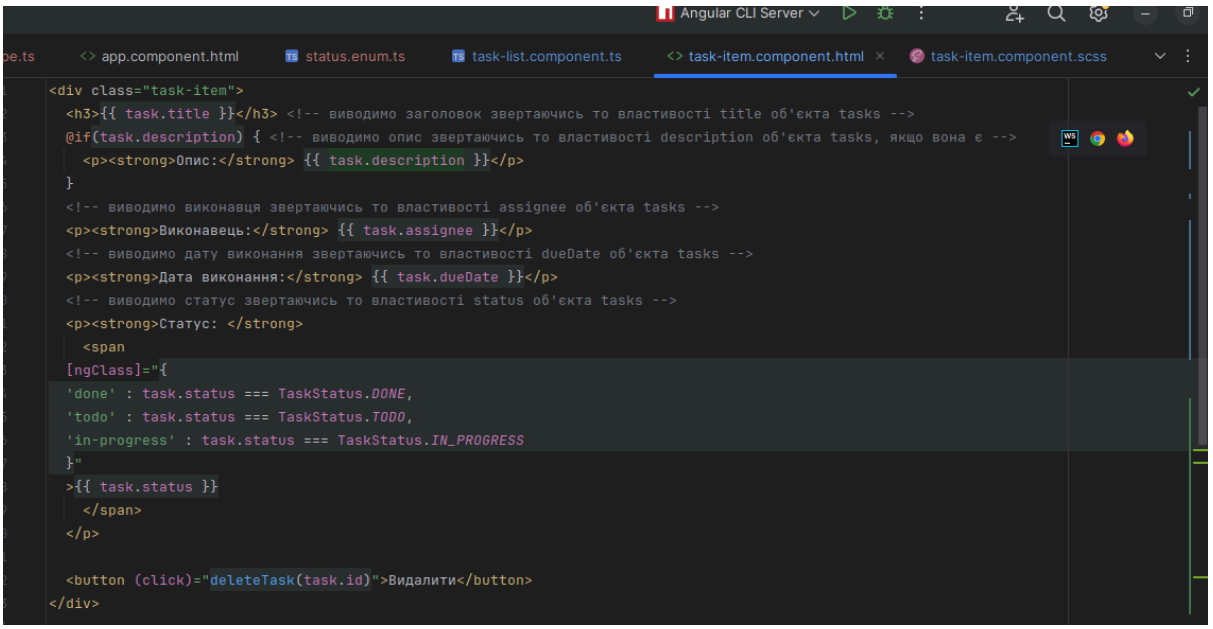
Розглянемо другий спосіб, пропишемо властивість статусу, як і інші у картці завдання

```
<div class="task-item">
  <h3>{{ task.title }}</h3> <!-- виводимо заголовок звертаючись то властивості title об'єкта tasks -->
  @if(task.description) { <!-- виводимо опис звертаючись то властивості description об'єкта tasks, якщо вона є -->
    <p><strong>Опис:</strong> {{ task.description }}</p>
  }
  <!-- виводимо виконавця звертаючись то властивості assignee об'єкта tasks -->
  <p><strong>Виконавець:</strong> {{ task.assignee }}</p>
  <!-- виводимо дату виконання звертаючись то властивості dueDate об'єкта tasks -->
  <p><strong>Дата виконання:</strong> {{ task.dueDate }}</p>
  <!-- виводимо статус звертаючись то властивості status об'єкта tasks -->
  <p><strong>Статус: </strong> <span>{{ task.status }}</span></p>

  <button (click)="deleteTask(task.id)">Видалити</button>
</div>
```

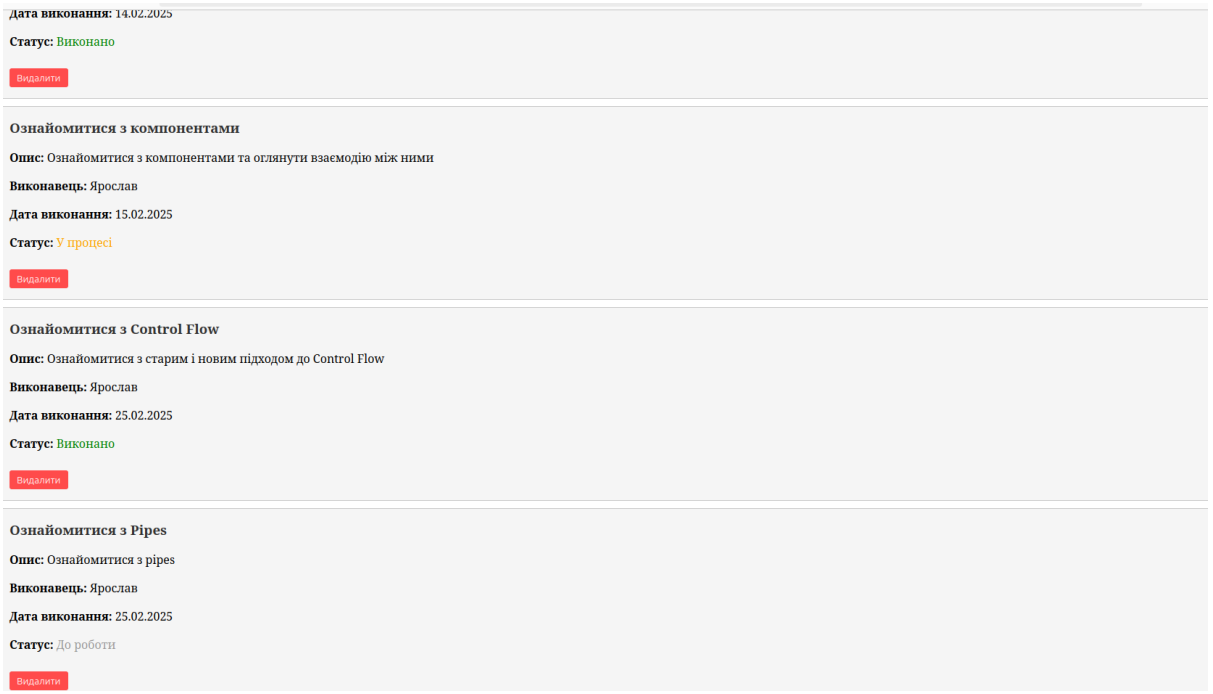
Перевіримо відображення



Бачимо, що статус виводиться згідно enum, але ми втратили стилі. Для того щоб їх повернути, використаємо [ngClass], передавши йому, як об’єкт, наші класи

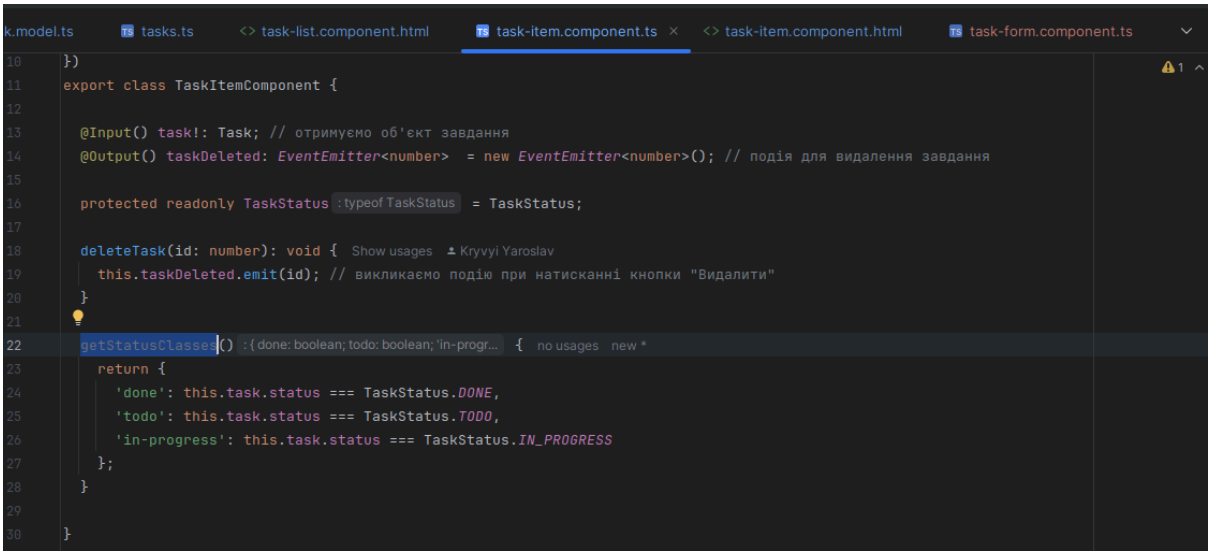


Перевіримо відображення

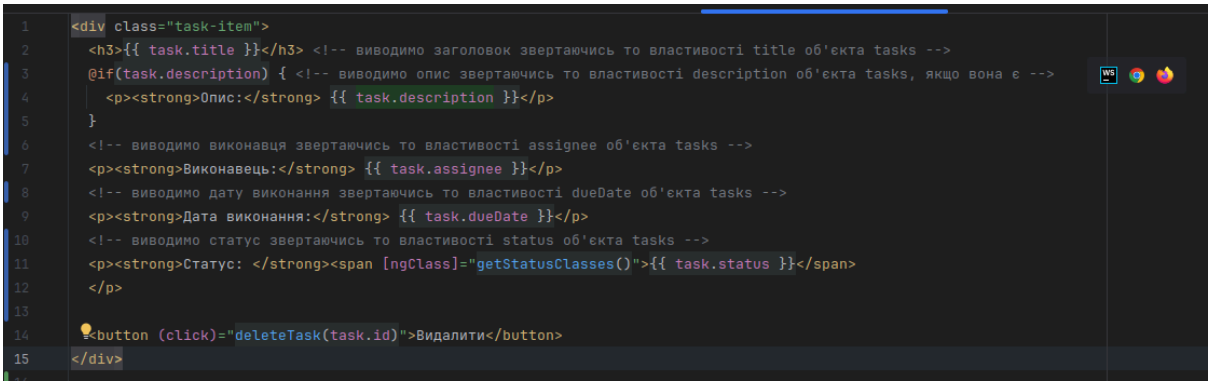


Бачимо, що тепер стилі коректно працюють залежно від статусу. Але що робити, якщо статусів буде 10-15 або потрібно буде цю логіку використовувати повторно у шаблоні. Можна звісно прописувати все вручну, але стикаємося з тією же проблемою, що була до використання цикла @for - незручність та надлишковість. Щоб це виправити, ми можемо повертати класи стилів за допомогою функції, давайте це реалізуємо. У

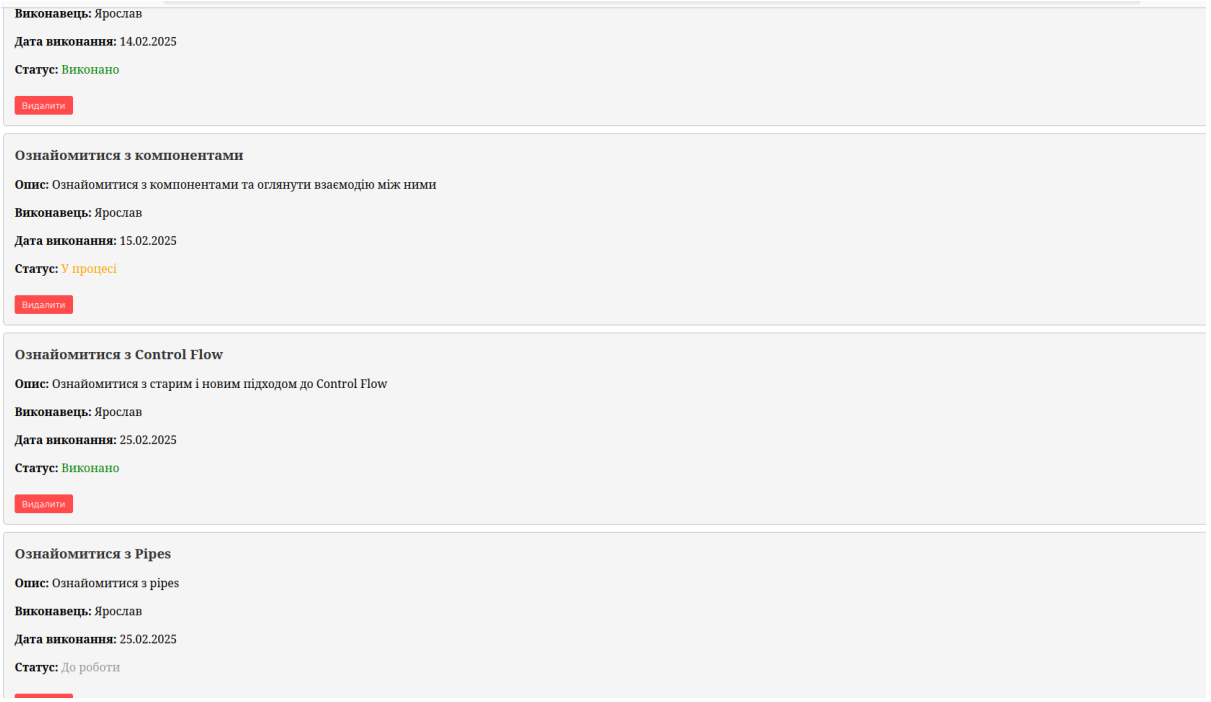
task-item.component.ts створимо нову функцію **getStatusClasses()**, яка буде повертати об’єкт з css-класами



У **task-item.component.html** змінимо об’єкт у [ngClass] на виклик функції

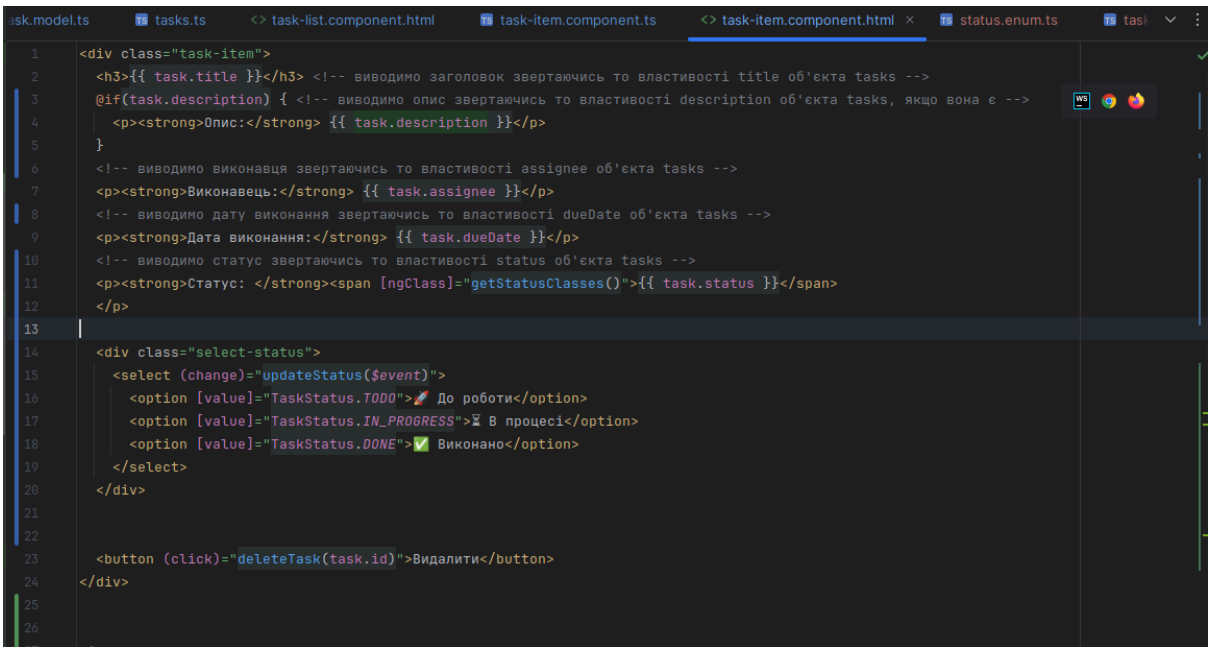


Перевіряємо відображення

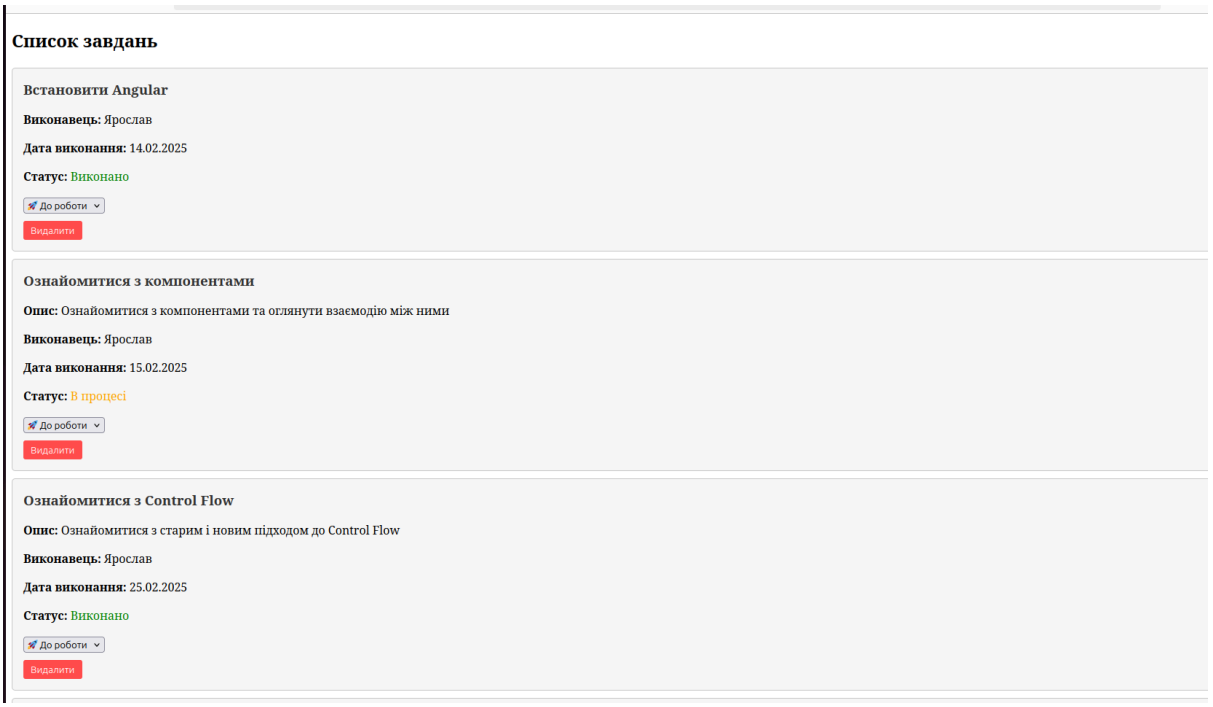


Можемо побачити, що стилі працюють, як і у випадку прямої передачі, але тепер код у шаблоні став більш читабельним та з’явилася можливість повторно використовувати класи стилів.

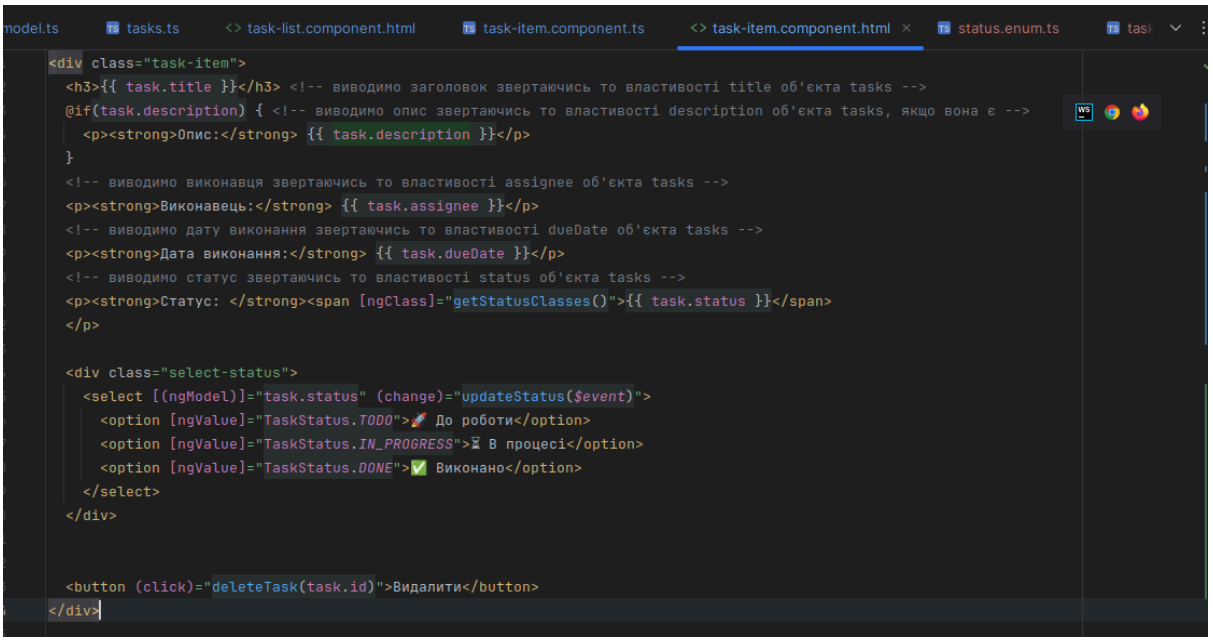
Що робити, якщо завдання було виконане або потрібно просто змінити статус, для цього внесемо корективи у **task-item.component.html**, додавши можливість змінювати статус кожного елементу окремо. Для цього використаємо **select** та івент **(change)**



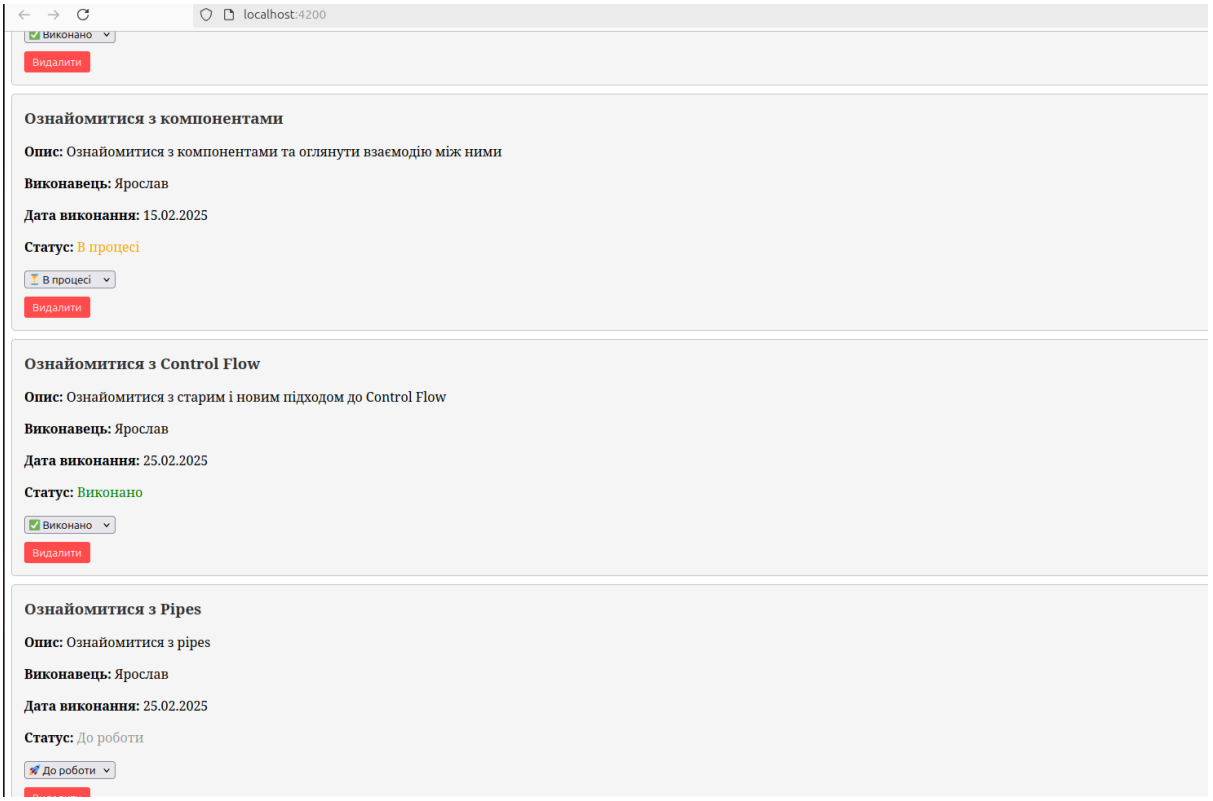
Перевіримо наше відображення



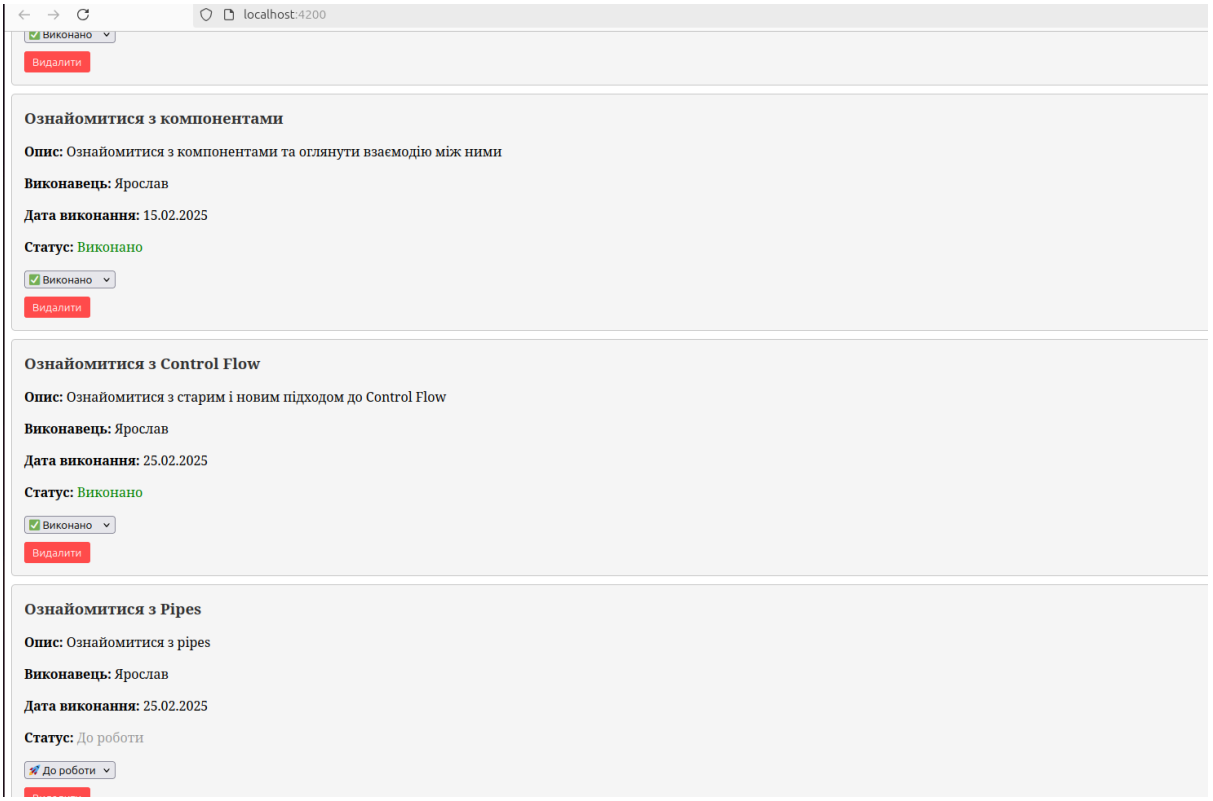
Можемо побачити, що з’явилося нове поле, яке дозволяє обрати статус, але за замовчуванням у кожно стоїть перший статус, а не його поточний. Для того, щоб це виправити можна використати [(ngModel)] з [ngValue]



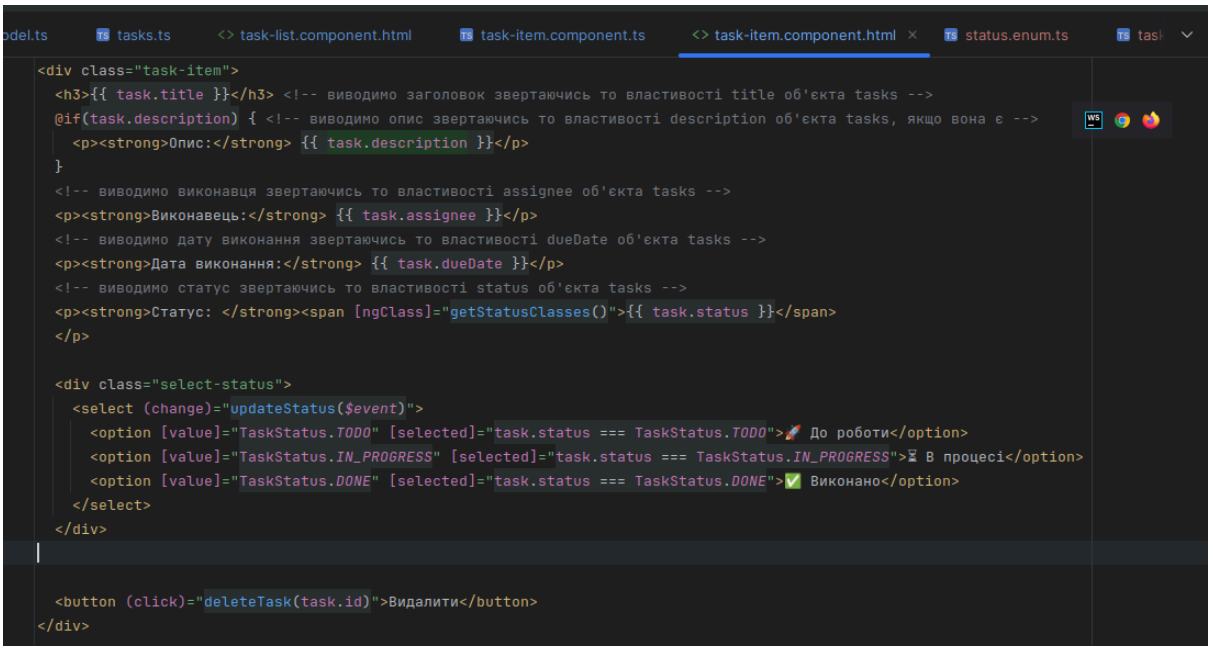
Перевіримо відображення



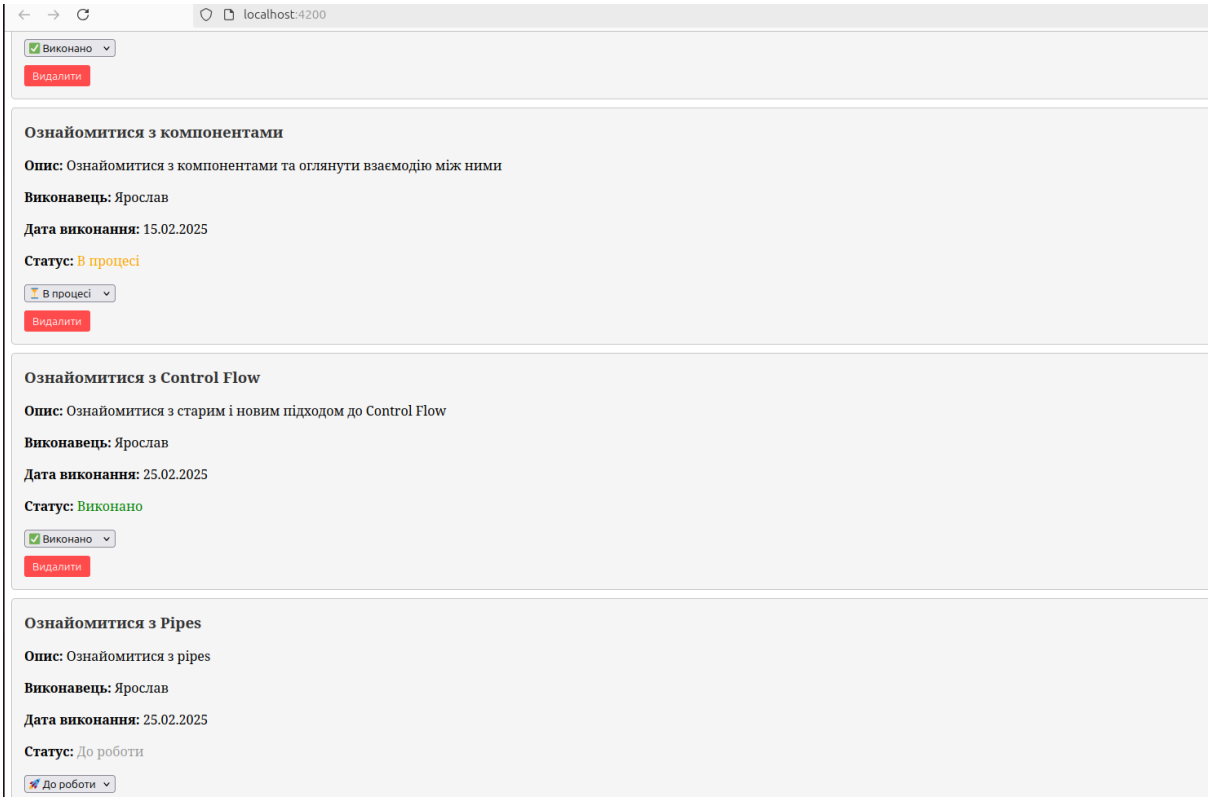
Як бачимо, тепер статуси відповідають статусу поточного завдання і якщо ми його змінимо, то за рахунок [(ngModel)] він запишеться у зміну task.status автоматично



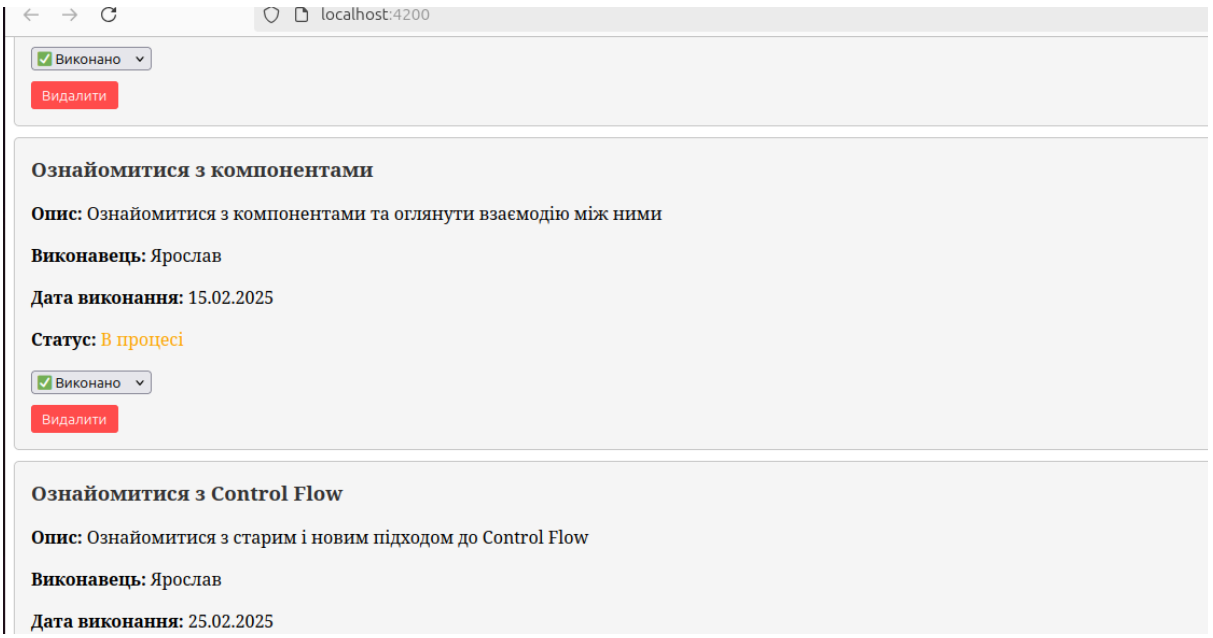
Але є і альтернативний підхід, можна звернутися до властивості selected кожного option



Перевіримо відображення та побачимо, що і з цим варіантом статуси відображаються коректно



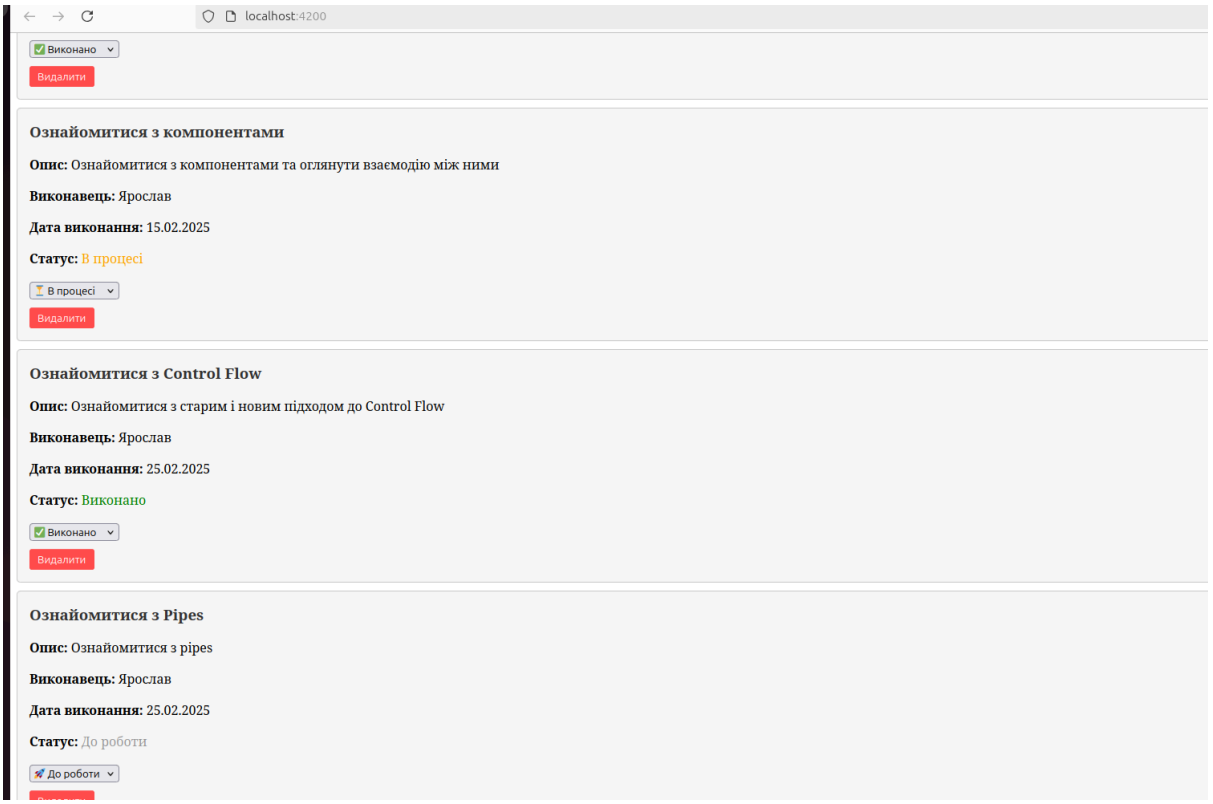
Але у цьому випадку, при зміні статусу, властивість автоматично не зміниться

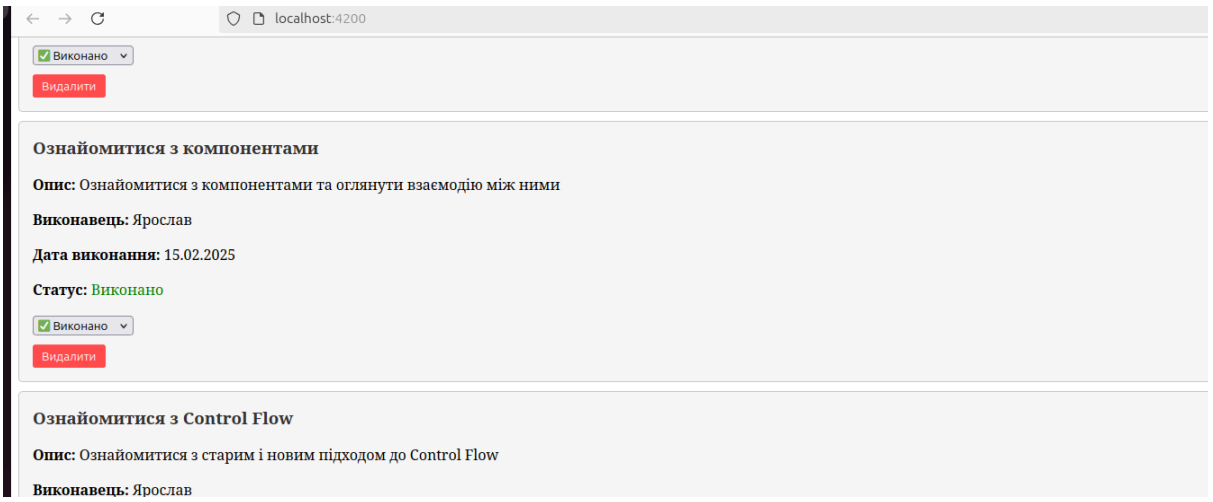


Ось тут нам і потрібен обробник події change, назначимо за його обробку функцію updateStatus та реалізуємо її

```typescript
export class TaskItemComponent {

  @Input() task!: Task; // отримуємо об'єкт завдання
  @Output() taskDeleted: EventEmitter<number>  = new EventEmitter<number>(); // подія для видалення завдання

  protected readonly TaskStatus : typeof TaskStatus  = TaskStatus;

  deleteTask(id: number): void {  Show usages  Kryvyi Yaroslav
    this.taskDeleted.emit(id); // викликаємо подію при натисканні кнопки "Видалити"
  }

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

  updateStatus(event: Event) : void  {  Show usages  new *
    const selectedValue : string  = (event.target as HTMLSelectElement).value;
    this.task.status = selectedValue as TaskStatus;
  }
}
```

Перевіримо відображення та роботу





Бачимо, що статус автоматично змінюється при зміні його через select. Також додамо стилі для нашого select

```css
select {
  padding: 5px;
  border: 1px solid #ccc;
  border-radius: 3px;
  cursor: pointer;
  background: white;
  font-size: 14px;
  transition: all 0.2s ease-in-out;
}

select:hover {
  border-color: #888;
}

select:focus {
  outline: none;
  border-color: #555;
}

option {
  padding: 5px;
}
```

3) Далі попрацюємо з pipes, як приклад створимо пайп фільтрації завдань за статусом. Для цього потрібно створити файл нашого пайпу вручну або за допомогою команди:

ng g p share/pipes/status-filter

```
svarog@svarog:~/projects/angular-course$ ng g p share/pipes/status-filter
CREATE src/app/share/pipes/status-filter.pipe.spec.ts (212 bytes)
CREATE src/app/share/pipes/status-filter.pipe.ts (250 bytes)
UPDATE src/app/app.module.ts (857 bytes)
svarog@svarog:~/projects/angular-course$
```

Наш пайп був автоматично зареєстрований у app.module.ts

```
import ...

@NgModule({  Show usages  Kryvyi Yaroslav
  declarations: [
    AppComponent,
    TaskListComponent,
    TaskItemComponent,
    TaskFormComponent,
    StatusFilterPipe
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Переходимо до нашого пайпу

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({  Show usages
  name: 'statusFilter',
  standalone: false
})
export class StatusFilterPipe implements PipeTransform {

  transform(value: unknown, ...args: unknown[]): unknown {  no usages
    return null;
  }

}
```

Ми бачимо у декораторі @Pipe назву нашого пайпу, саме її ми будемо використовувати при звернені до нього. Також інтерфейс

PipeTransform зобов'язує нас реалізувати функцію transform. Можемо побачити, що за замовчуванням функція приймає дані, які потрібно обробити та опціонально параметри. Змінимо базові значення на наші, ми будемо приймати масив завдань, тому перше значення у нас буде типу Task[], як параметр, ми будемо передавати статус через enum або значення “all” і повертати ми будемо також Task[]

```
import { Pipe, PipeTransform } from '@angular/core';
import { Task } from '../../core/models/task.model';
import {TaskStatus} from '../../core/models/status.enum';

@Pipe({  Show usages
  name: 'statusFilter',
  standalone: false
})
export class StatusFilterPipe implements PipeTransform {
  transform(tasks: Task[], status: TaskStatus | 'all'): Task[] {  no usages
    return tasks;
  }
}
```

Тепер реалізуємо логіку, потрібно повертати вхідний масив, якщо значення “all”, а також опціонально null або undefined, і фільтрований, якщо є статус.

```
import { Pipe, PipeTransform } from '@angular/core';
import { Task } from '../../core/models/task.model';
import {TaskStatus} from '../../core/models/status.enum';

@Pipe({  Show usages
  name: 'statusFilter',
  standalone: false
})
export class StatusFilterPipe implements PipeTransform {
  transform(tasks: Task[], status: TaskStatus | 'all'): Task[] {  no usages
    if (!tasks) {
      return [];
    }

    if (status === 'all' || status === null || status === undefined) {
      return tasks;
    }

    return tasks.filter(task : Task  => task.status === status);
  }
}
```

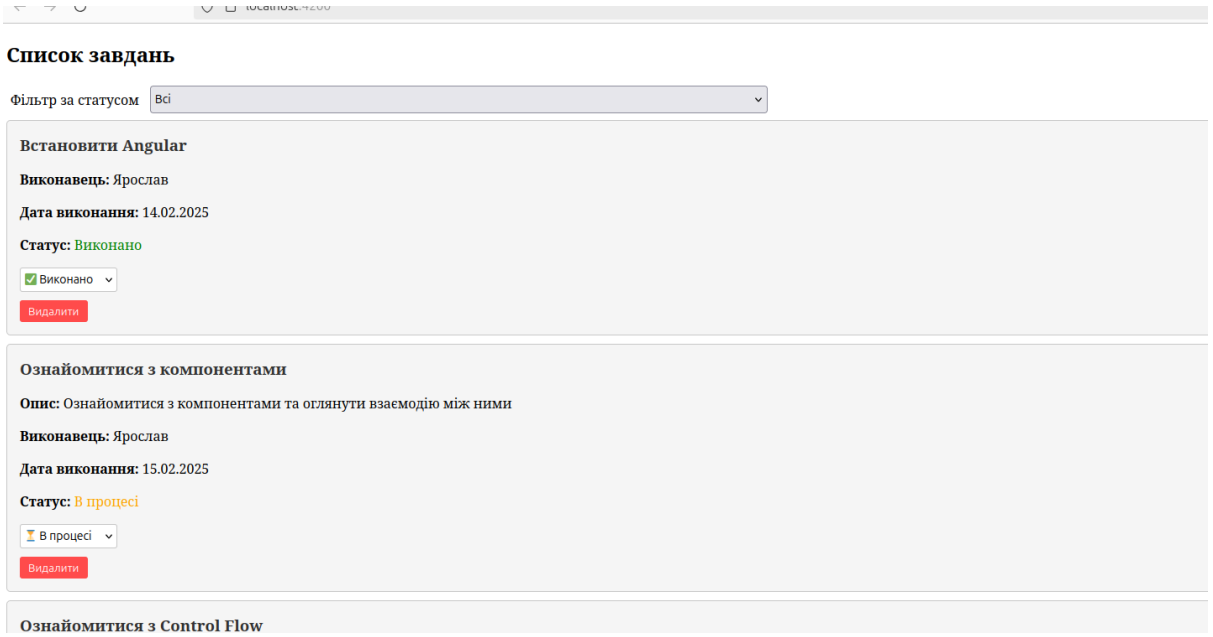
Тепер додамо до **task-list.component.html** select для вибору статусу

```
<h2>Список завдань</h2>
<div class="task-filter">
  <p>Фільтр за статусом</p>
  <select>
    <option [value]="'all'">Всі</option>
    <option [value]="TaskStatus.TODO">До роботи</option>
    <option [value]="TaskStatus.IN_PROGRESS">В процесі</option>
    <option [value]="TaskStatus.DONE">Виконано</option>
  </select>
</div>
@for(task of myTasks; track task.id; let index = $index) {
  <app-task-item
    [task]="task"
    (taskDeleted)="deleteTask($event)"
  >
  </app-task-item>
} @empty {
  <p>Курси відсутні</p>
}
```

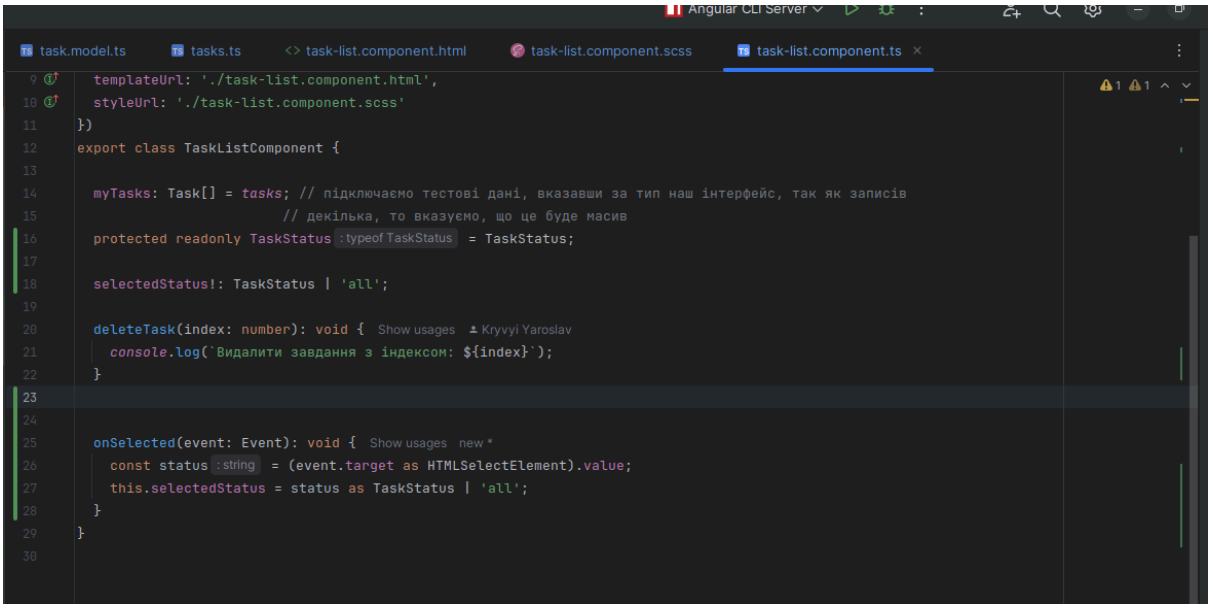
І трохи стилізуємо

```
nable File Watcher to compile SCSS to CSS?
.task-filter {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  gap: 0.5rem;
  margin-bottom: 0.5rem;

  p {
    margin: 5px;
  }

  select {
    width: 40%;
    cursor: pointer;
    font-size: 14px;
  }
}
```

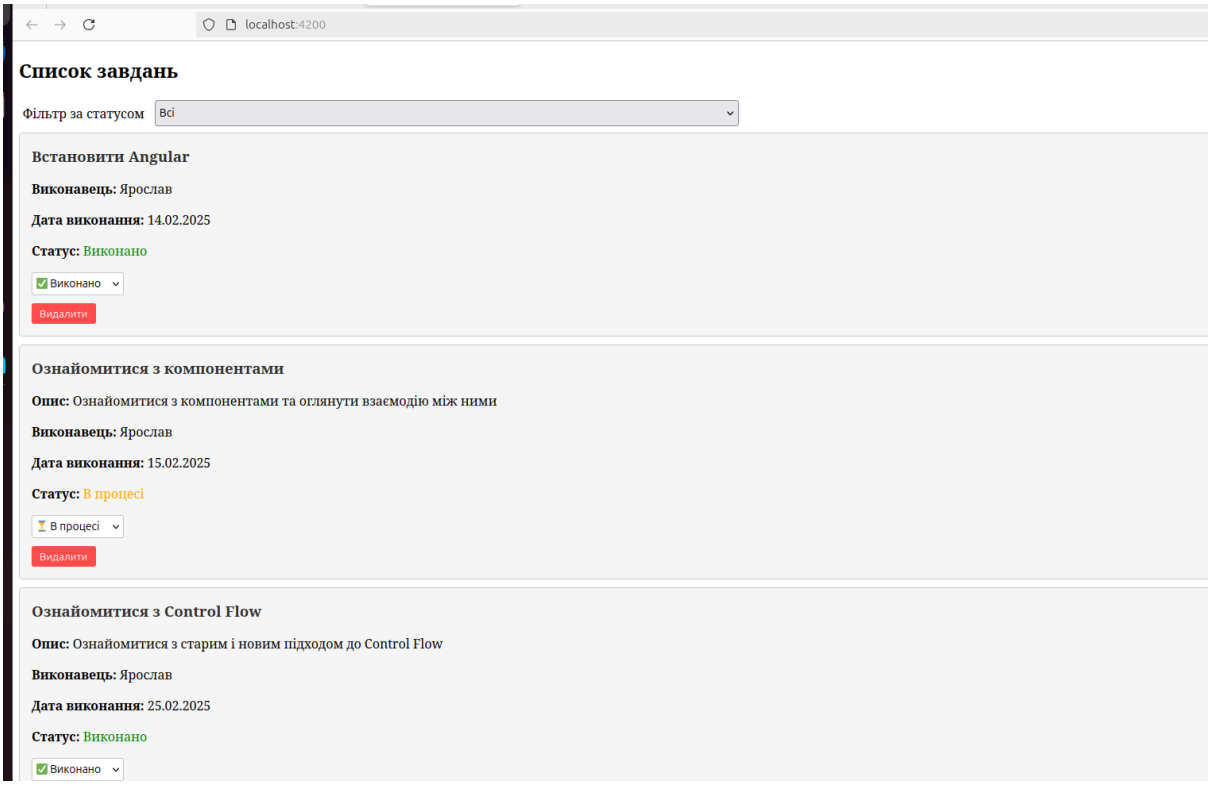
Отримаємо



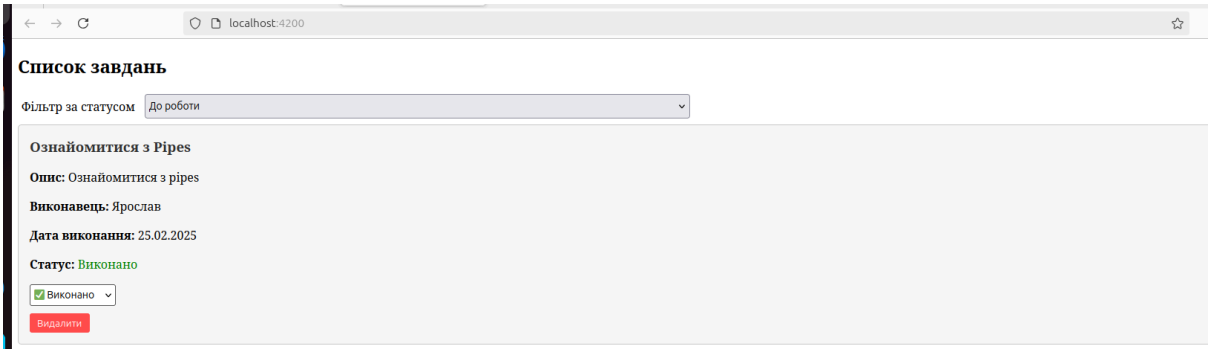
Тепер необхідно реалізувати логіку по отриманню параметра фільтрації та передачі його пайпу. Для цього у select пропишемо (change)="onSelected($event) і реалізуємо її.



Перевіряємо роботу



Але що буде, якщо під час фільтрації змінити статус?



Бачимо, що елемент не відфільтрувався. Чому так?

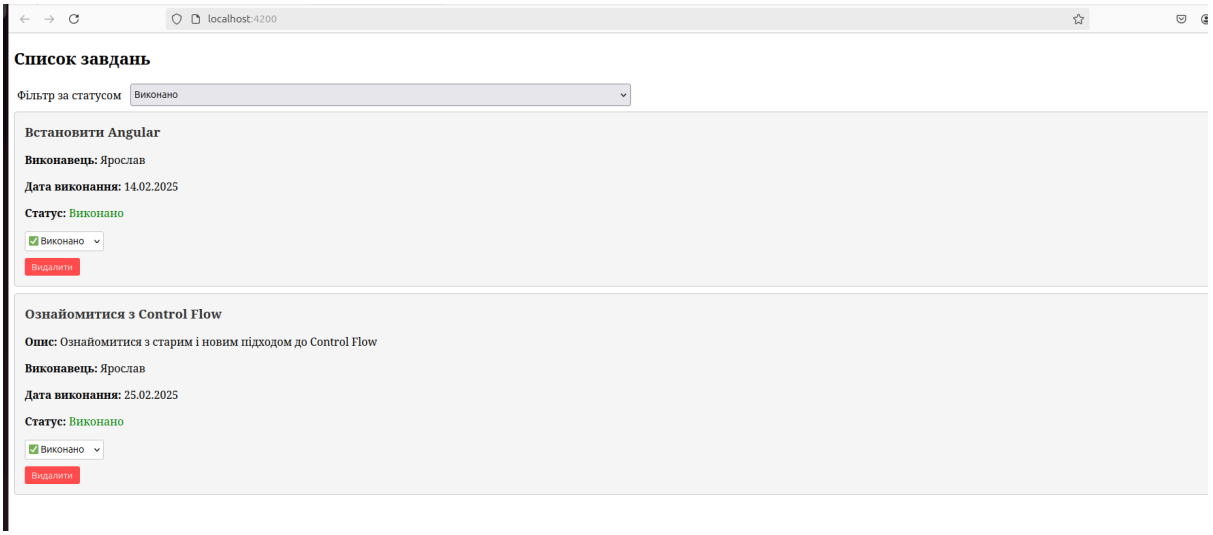
Вся справа у властивості декоратора @Pipe, а саме pure, яка за замовчуванням кешує значення і змінює його тільки якщо зміняться вхідні

дані. Щоб це обійти, потрібно встановити pure: false, і тоді пайп буде виконуватися при кожній зміні компонента

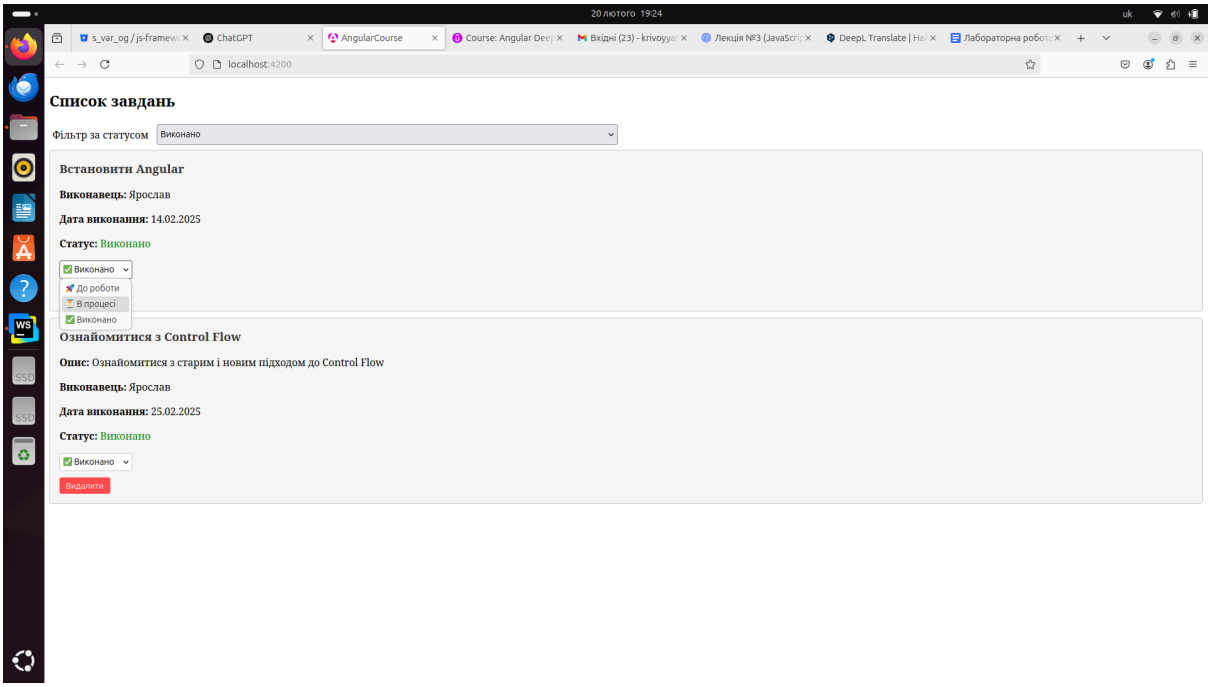
task.model.ts | tasks.ts | task-list.component.html | task-list.component.scss | task-list.component.ts | status-filter.pipe.ts

```ts
import { Pipe, PipeTransform } from '@angular/core';
import { Task } from '../../core/models/task.model';
import {TaskStatus} from '../../core/models/status.enum';

@Pipe({  Show usages
  name: 'statusFilter',
  pure: false,
  standalone: false
})
export class StatusFilterPipe implements PipeTransform {
  transform(tasks: Task[], status: TaskStatus | 'all'): Task[] {  Show usages
    if (!tasks) {
      return [];
    }

    if (status === 'all' || status === null || status === undefined) {
      return tasks;
    }

    return tasks.filter(task :Task  => task.status === status);
  }
}
```

Перевіряємо відображення



Змінимо статус

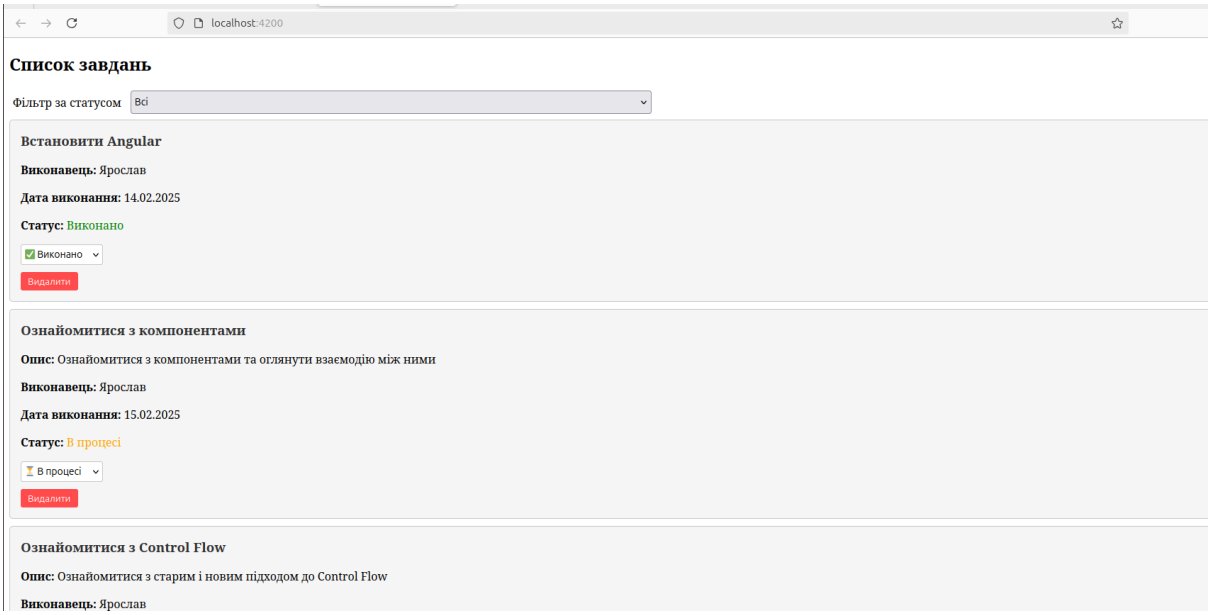


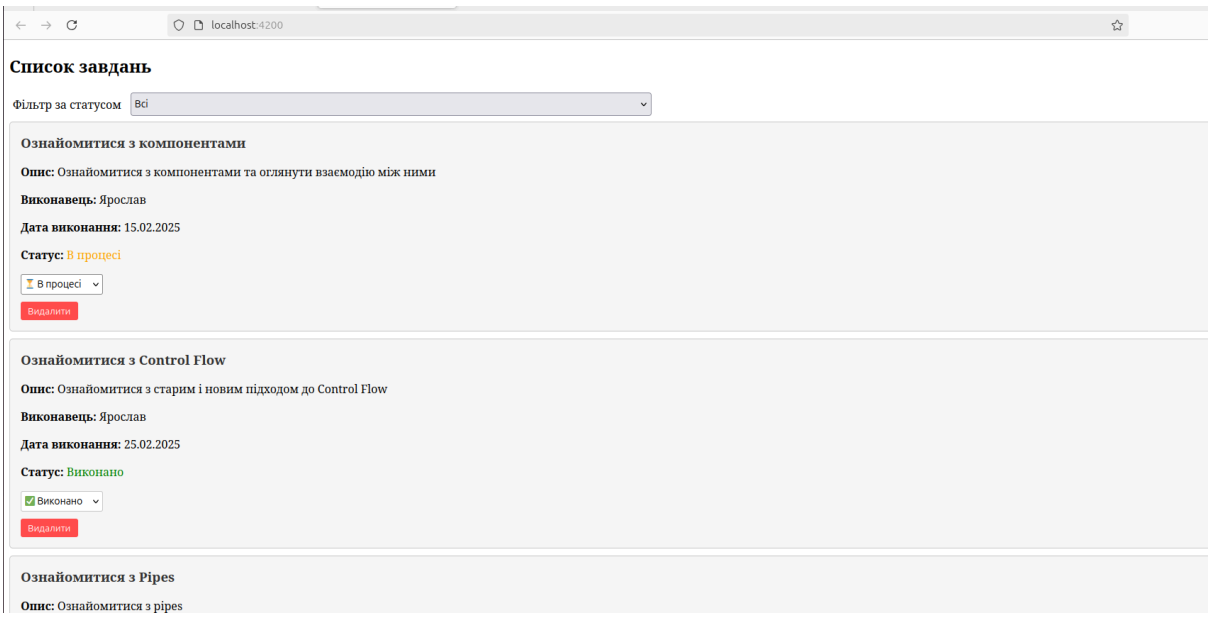
Перевіримо результат

Також за рахунок відображення завдань за допомогою @for можемо переробити функцію видалення завдань

```
  templateUrl: './task-list.component.html',
  styleUrl: './task-list.component.scss'
})
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';

  deleteTask(index: number): void {  Show usages  Kryvyi Yaroslav *
    this.myTasks = this.myTasks.filter(task :Task  => task.id !== index);
  }


  onSelected(event: Event): void {  Show usages  new *
    const status :string  = (event.target as HTMLSelectElement).value;
    this.selectedStatus = status as TaskStatus | 'all';
  }
}
```

Як приклад





Контрольні питання

1) Для чого потрібен track у @for? Чи є він обов’язковим?
2) Для чого використовується @Pipe?
3) На що впливає властивість pure у @Pipe?
4) За яким принципом @switch розуміє, який @case відобразити? Чи потрібно писати break?
5) Що може слугувати за ідентифікатор для track?
6) Чи можна за допомогою @ViewChild отримати доступ до дочірнього шаблону?