Лабораторна робота № 6. Вступ до Express.js та AsyncPipe Angular

Мета: ознайомитися та навчитися працювати з основними інструментами express щодо створення API-серверу. Навчитися обробляти потік даних за допомогою AsyncPipe в Angular.

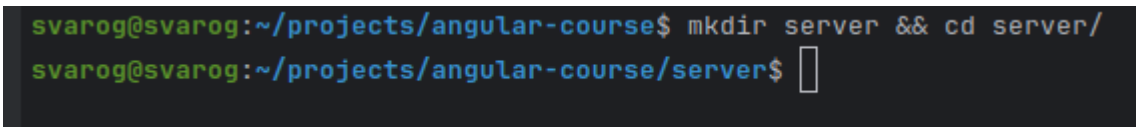
Завдання

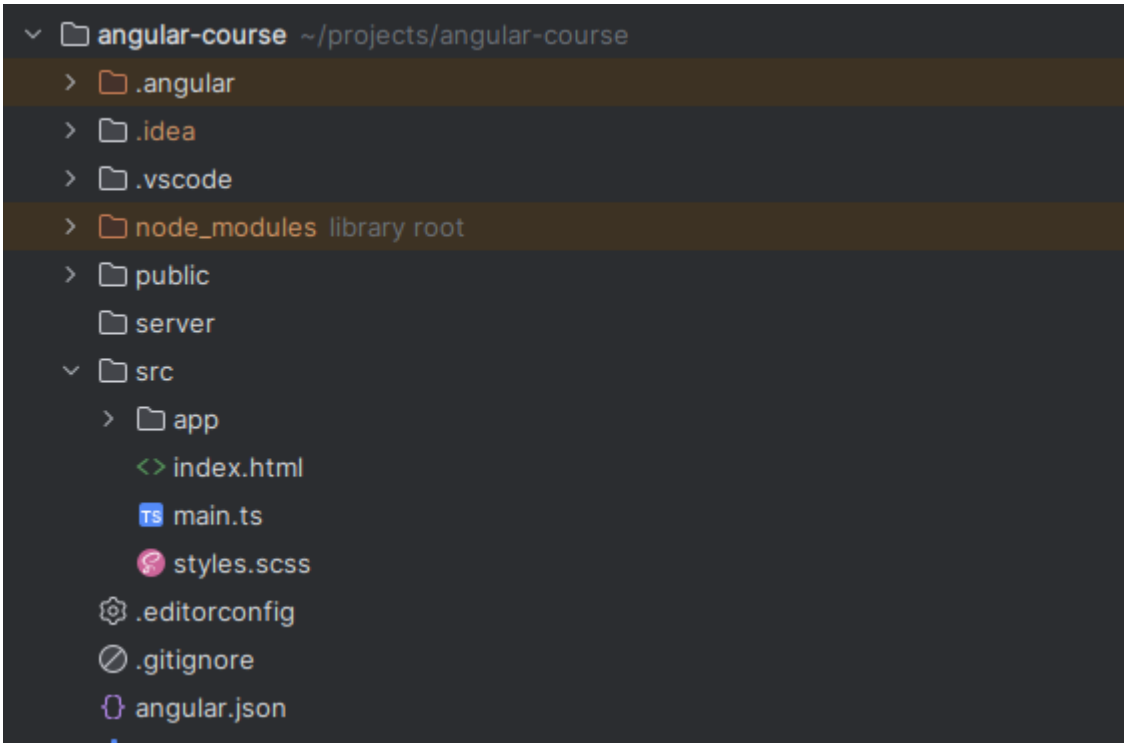
1) Ознайомитися з матеріалом лекції № 6;
2) Виконати пункти 1-7 ходу роботи;
3) У звіті відобразити кроки 1-7 ходу роботи;
4) Надати відповіді на контрольні питання.

Хід роботи

1) Для початку створимо сервер та ініціалізуємо package.json за допомогою команд у терміналі корня нашого Angular-проєкту

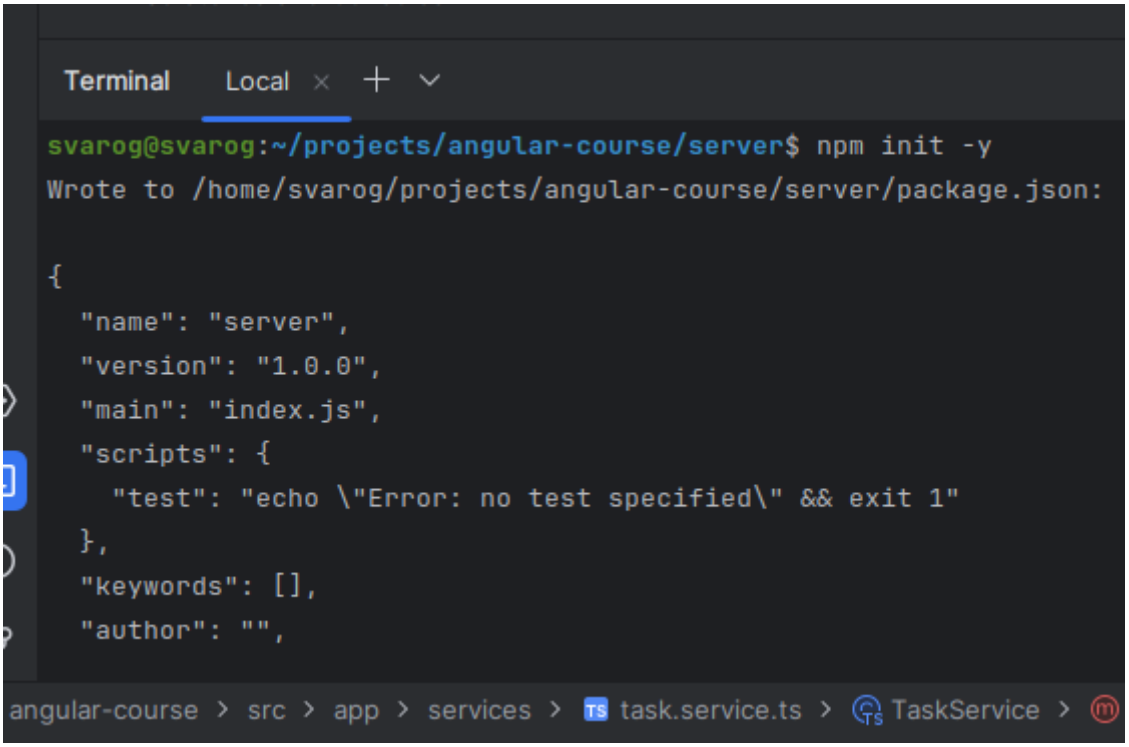
   mkdir server && cd server/





Ініціалізуємо базовий package.json за допомогою команди

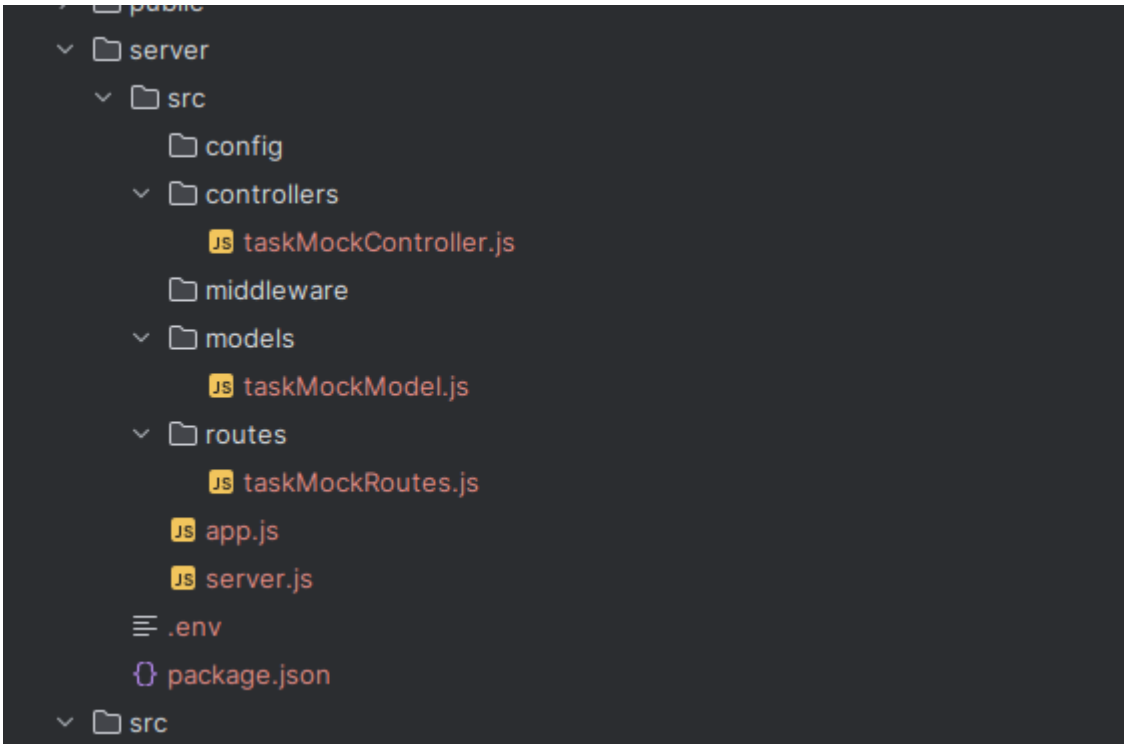
npm init -y



Відразу додамо скрипт запуску до **scripts**

```
{
  "name": "server",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node src/server.js"
  },
  "keywords": [],
```

2) Створимо базову структуру нашого REST API серверу



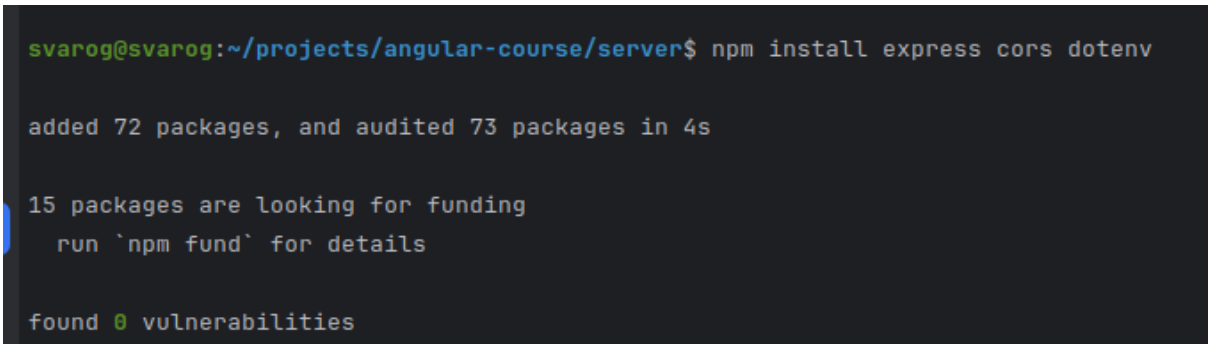
Розділяємо логіку наступним чином:

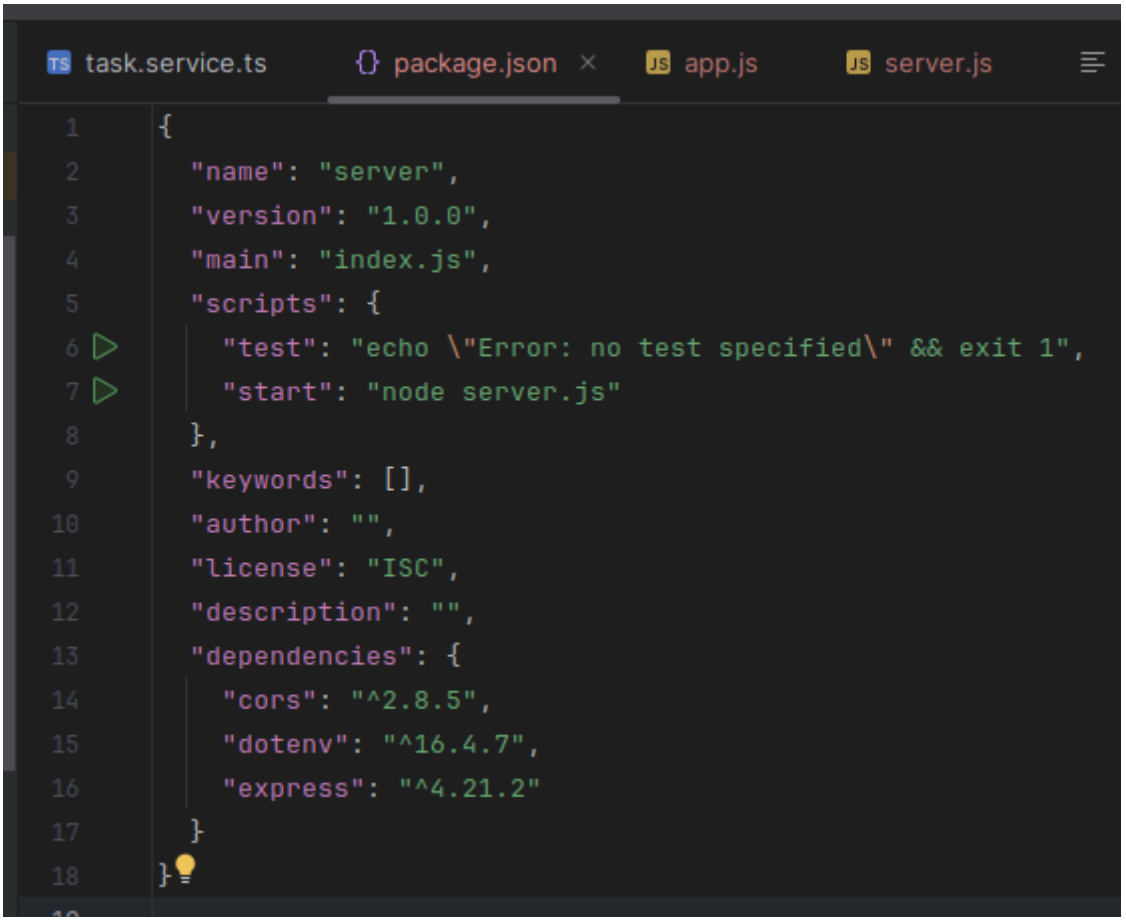
1. src/models/taskMockModel.js - аналог моделі, яка буде містити mock-дані;
2. src/controllers/taskMockController.js - контролер, який буде опрацьовувати звернення та повертати mock-дані;
3. src/routes/taskMockRoutes.js - прив’язка маршрутів до обробників HTTP-запитів у контролері;

4. src/app.js - файл, який містить необхідні налаштування для роботи сервера (підключення middleware, маршрутів тощо);
5. src/server.js - точка входу, файл який запускає сервер;
6. .env - містить конфіденційні змінні, які використовуються у проєкті.

Також для роботи з деякими наступними елементами (да і для серверу в цілому) потрібно встановити **express, dotenv** та відразу **cors**. Робимо це за допомогою команди

npm install express cors dotenv





3) Реалізуємо покроково модель, контролер та роутер:
    a) Почнемо з моделі

У файлі **taskMockModel.js** пишемо наступний код:

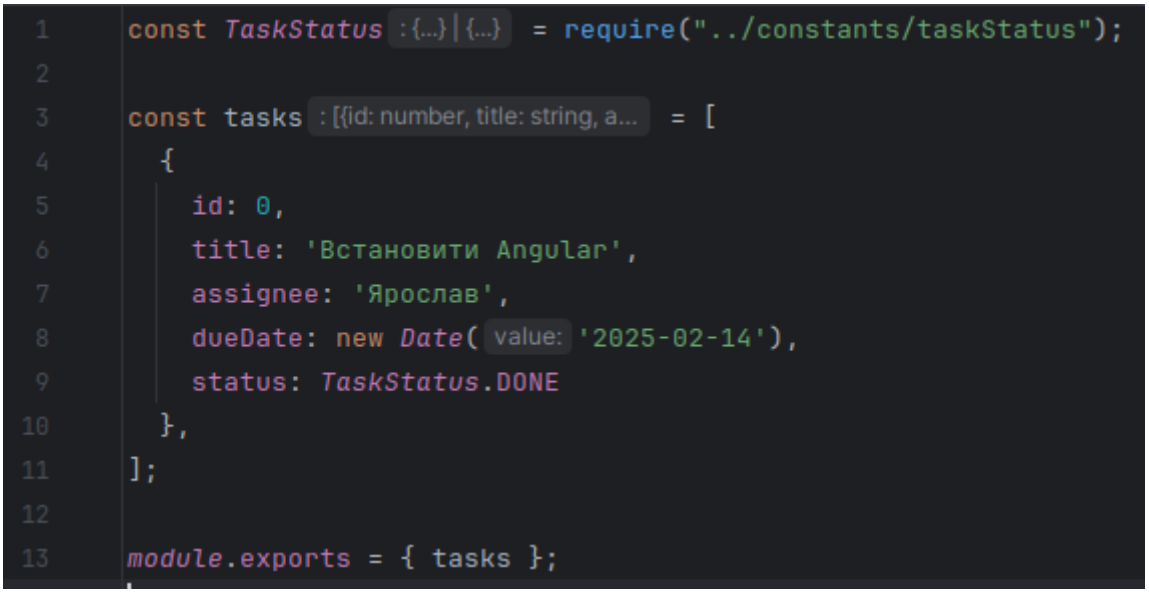
```
const TaskStatus :{…} = {
  TODO: 'TODO',
  IN_PROGRESS: 'IN_PROGRESS',
  DONE: 'DONE'
};

const tasks :[{id: number, title: string, a… = [
  {
    id: 0,
    title: 'Встановити Angular',
    assignee: 'Ярослав',
    dueDate: new Date( value: '2025-02-14'),
    status: TaskStatus.DONE
  },
];

module.exports = { tasks };
```

При тесті з Angular змінити id на 1

Так як TaskStatus представляє з себе набір констант і може бути необхідність використовувати його у інших частинах серверу, то краще винести у окремий файл (наприклад, src/constants/taskStatus.js)

```
const TaskStatus :{…} = {
  TODO: 'TODO',
  IN_PROGRESS: 'IN_PROGRESS',
  DONE: 'DONE'
};

module.exports = TaskStatus;
```

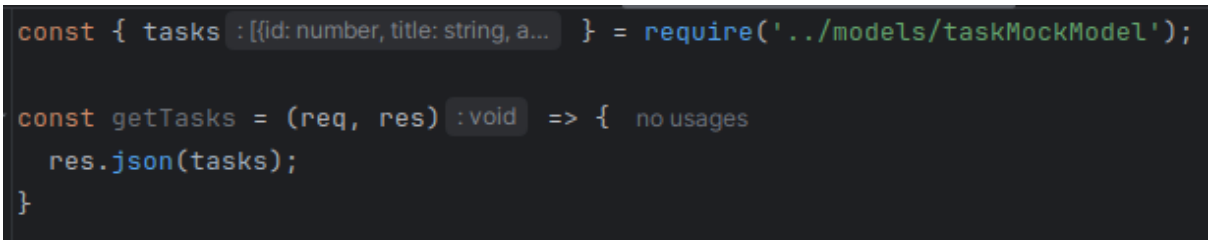
Підключаємо TaskStatus у файл **taskMockModel.ts**



b) Працюємо з контролером **taskMockController.js**

Для початку потрібно підключити до нього модель

Реалізовуємо метод **GET-запиту**



Базовий метод, який повертає всі наявні завдання. При невеликій кількості записів (наприклад у БД) така реалізація має право на існування. Якщо записів багато, то краще додати можливість фільтрації та пагінації.

Реалізовуємо метод **POST-запиту**

```
const createTask = (req, res) :void => {  Show usages
  const taskId :number = tasks.length ? Math.max(...tasks.map((task :{…} ) :number => task.id)) + 1 : 1;
  const newTask :any & {…} = {
    id: taskId,
    ...req.body,
    deuDate: new Date(req.body.deuDate),
  }

  tasks.push(newTask);

  res.status(201).json(newTask);
}
```

Базовий метод, який приймає нове завдання, назначає йому id та записує до mock-моделі. Має певні проблеми з безпекою, так як не перевіряє чи всі обов'язкові властивості були передані та чи не містить об'єкт неправильних властивостей. У подальшому реалізуємо ці перевірки.

Реалізовуємо метод **PUT-запиту**

```
const updateTask = (req, res) :void => {  Show usages
  const taskId :number = parseInt(req.params.id);
  const updatedTask :any & {…} = {...req.body, dueDate: new Date(req.body.dueDate),};

  const taskIndex :number = tasks.findIndex(task :{…} => task.id === taskId);
  if (taskIndex !== -1) {
    tasks[taskIndex] = { id: taskId, ...updatedTask };
    res.json(tasks[taskIndex]);
  } else {
    res.status(404).json({ message: 'Завдання не знайдено' });
  }
}
```

Базовий метод, що приймає id завдання через params та нове значення. Так як значення приходить, як string, то його потрібно конвертувати у int через функцію **parseInt**. Далі за допомогою **findIndex** шукаємо у нашій mock-моделі індекс завдання з вказаним id. Якщо знайдено, то повністю оновлюємо його, якщо ні, то надсилаємо статус 404. Має ті самі проблеми, що і POST.

Реалізовуємо метод **PATCH-запиту**

```
const patchTask = (req, res) :void => {  no usages
  const taskId :number = parseInt(req.params.id);
  const updates = req.body;

  const task :{…} = tasks.find(task :{…} => task.id === taskId);
  if (task) {
    Object.assign(task, updates);
    res.json(task);
  } else {
    res.status(404).json({ message: 'Завдання не знайдено' });
  }
}
```

Базовий метод, що приймає id завдання через params та значення, які необхідно оновити. Так як значення приходить, як string, то його потрібно конвертувати у int через функцію **parseInt**. Далі за допомогою **find** шукаємо у нашій mock-моделі завдання з вказаним id. Якщо знайдено, то оновлюємо передані властивості за допомогою **Object.assign(target, source)**, який копіює всі поля (значення) з **source** у **target**, а так як find повертає по суті посилання на знайдене завдання, тому Object.assign оновлює значення напряму у mock-моделі **tasks.** Якщо завдання не знайдено, то надсилаємо статус 404. Має ті самі проблеми, що і POST за виключенням обов’язкових параметрів.

Реалізовуємо метод **DELETE-запиту**

```
const deleteTask = (req, res) :void => {  no usages
  const taskId :number = parseInt(req.params.id);
  const taskIndex :number = tasks.findIndex(task :{…} => task.id === taskId);

  if (taskIndex !== -1) {
    tasks.splice(taskIndex, deleteCount: 1);
    res.json({ message: `Завдання ${taskId} видалене` });
  } else {
    res.status(404).json({ message: 'Завдання не знайдено' });
  }
}
```

Базовий метод, що приймає id завдання через params. Так як значення приходить, як string, то його потрібно конвертувати у int через функцію **parseInt**. Далі за допомогою **findIndex** шукаємо у нашій

mock-моделі індекс завдання з вказаним id. Якщо знайдено, то видаляємо його за допомогою функції **splice**, яка видаляє дані починаючи з вказаного індекса та необов'язкового параметра кількості елементів, які необхідно видалити, якщо ні, то надсилаємо статус 404.

У кінці файлу експортуємо наші обробники

```
    res.json({ message: `Завдання ${taskId} видалене` });
  } else {
    res.status(404).json({ message: 'Завдання не знайдено' });
  }
}

module.exports = { getTasks, createTask, updateTask, patchTask, deleteTask };
```

c) Переходимо до реалізації роутера **taskMockRouter.js**

Для початку підключаємо express.Router та файл контролера

```
const express : e | () => core.Express = require('express');
const router : Router = express.Router();
const taskMockController : {...} | {...} = require('../controllers/taskMockController');
```

Далі пов’язуємо маршрути та обробники

```
const express : e | () => core.Express = require('express');
const router : Router = express.Router();
const taskMockController : {...} | {...} = require('../controllers/taskMockController');

router.get( path: '/', taskMockController.getTasks);
router.post( path: '/', taskMockController.createTask);
router.put( path: '/:id', taskMockController.updateTask);
router.patch( path: '/:id', taskMockController.patchTask);
router.delete( path: '/:id', taskMockController.deleteTask);
```

Експортуємо наші маршрути

```
router.post( path: '/', taskMockController.createTask);
router.put( path: '/:id', taskMockController.updateTask);
router.patch( path: '/:id', taskMockController.patchTask);
router.delete( path: '/:id', taskMockController.deleteTask);

module.exports = router;
```

4) Далі необхідно підключити наші маршрути у **app.js**, одразу підключимо вбудований middleware для роботи з json.

```
const express : e | () => core.Express  = require('express');
const taskMockRoutes : Router | {...}  = require('./routes/taskMockRoutes');

const app : any | Express  = express();

app.use(express.json());
app.use('/api/v1/tasks', taskMockRoutes);

module.exports = app;
```

Тепер наші маршрути будуть доступні за **api/v1/tasks**

5) Налаштовуємо файл запуску сервера (**server.js**)

```
const app : ...  = require('./app');
require("dotenv").config();

const PORT : string | number  = process.env.PORT || 3000;

app.listen(PORT,  hostname: () : void  => {
  console.log(`Server running on port ${PORT}`);
})
```

Та прописуємо порт у **.env**

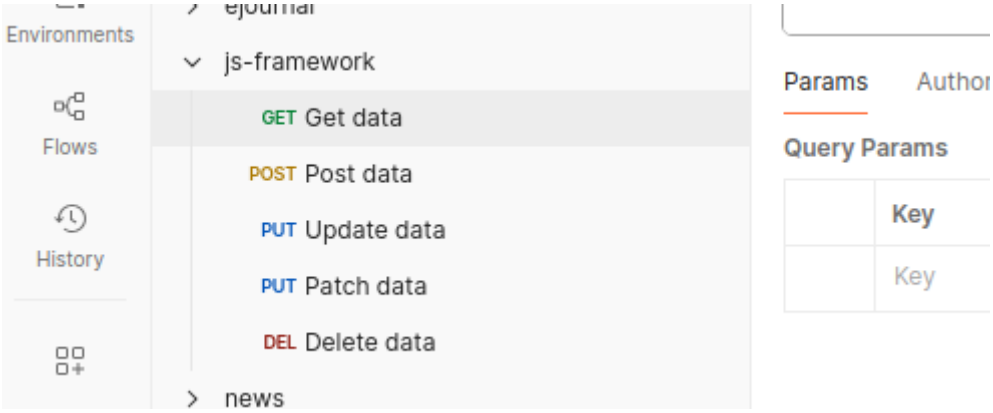
```
PORT = 3100
```

6) Запустимо наш сервер за допомогою **npm run start**

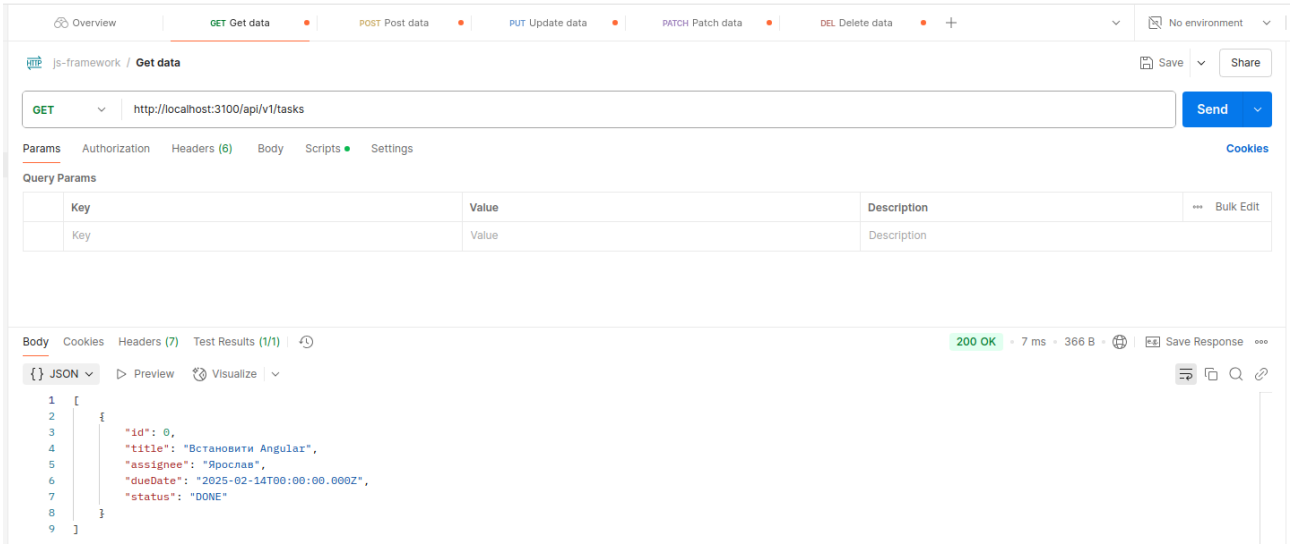
```
Terminal   Local
svarog@svarog:~/projects/angular-course/server$ npm run start

> server@1.0.0 start
> node src/server.js

Server running on port 3100
```

Перед тим, як перейти до зв’язку з Angular необхідно протестувати наші маршрути. Для цього використаємо Postman-desktop.

Створимо колекцію для REST API

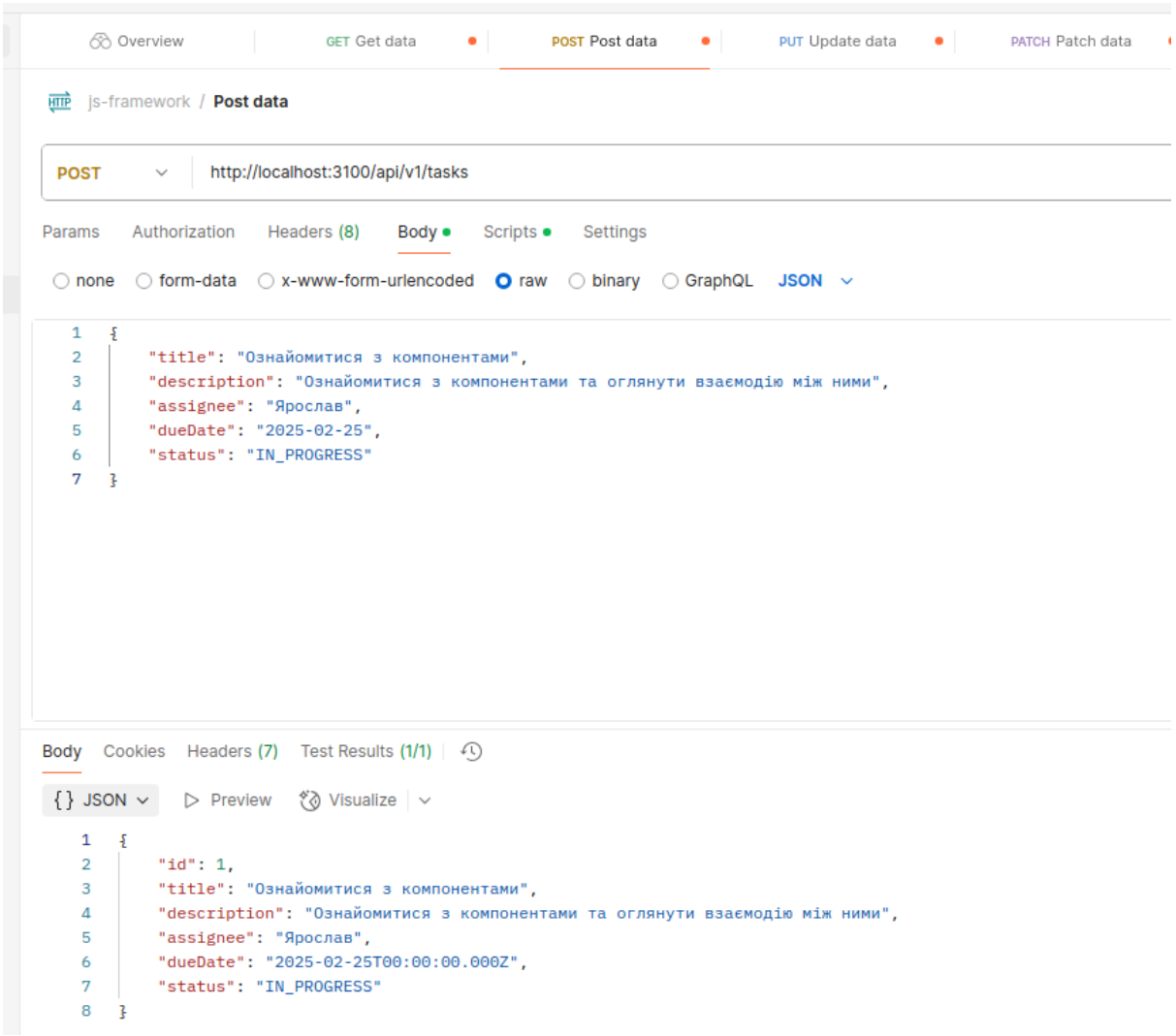


Тестуємо GET-запит за шляхом **http://localhost:3100/api/v1/tasks**

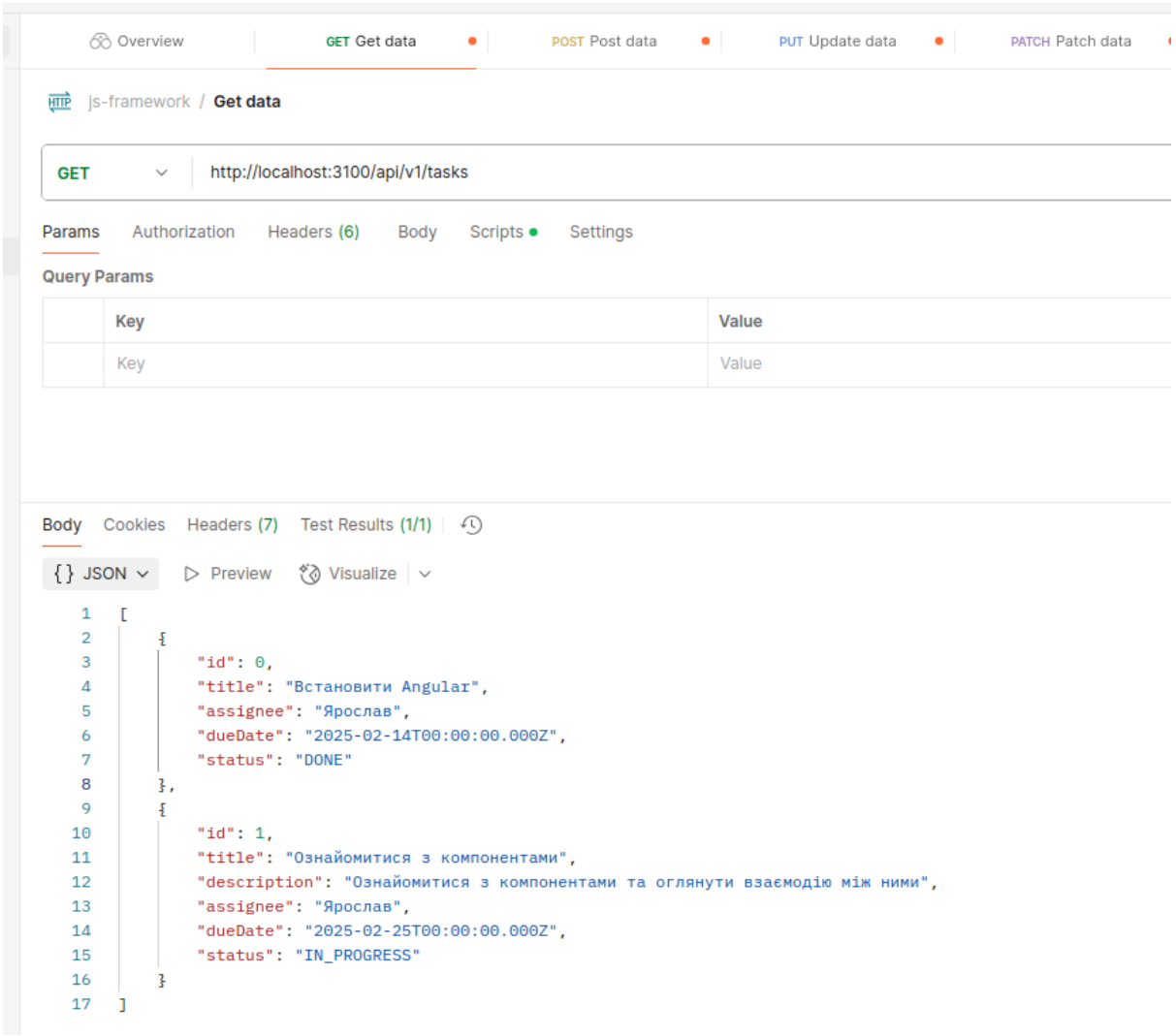


Тестуємо POST-запит за шляхом **http://localhost:3100/api/v1/tasks**

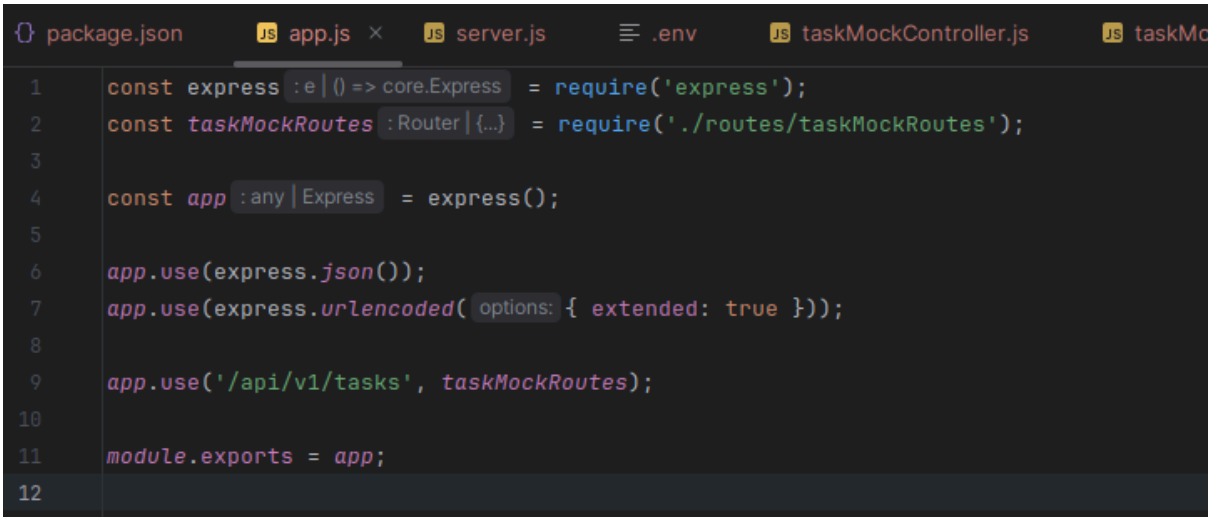
Якщо ми надсилаємо дані через **raw(JSON)**, як здебільше надсилає Angular, то ніякої додаткової обробки зі сторони express не потрібно, крім підключення **app.use(express.json())** у файлі **app.js**.



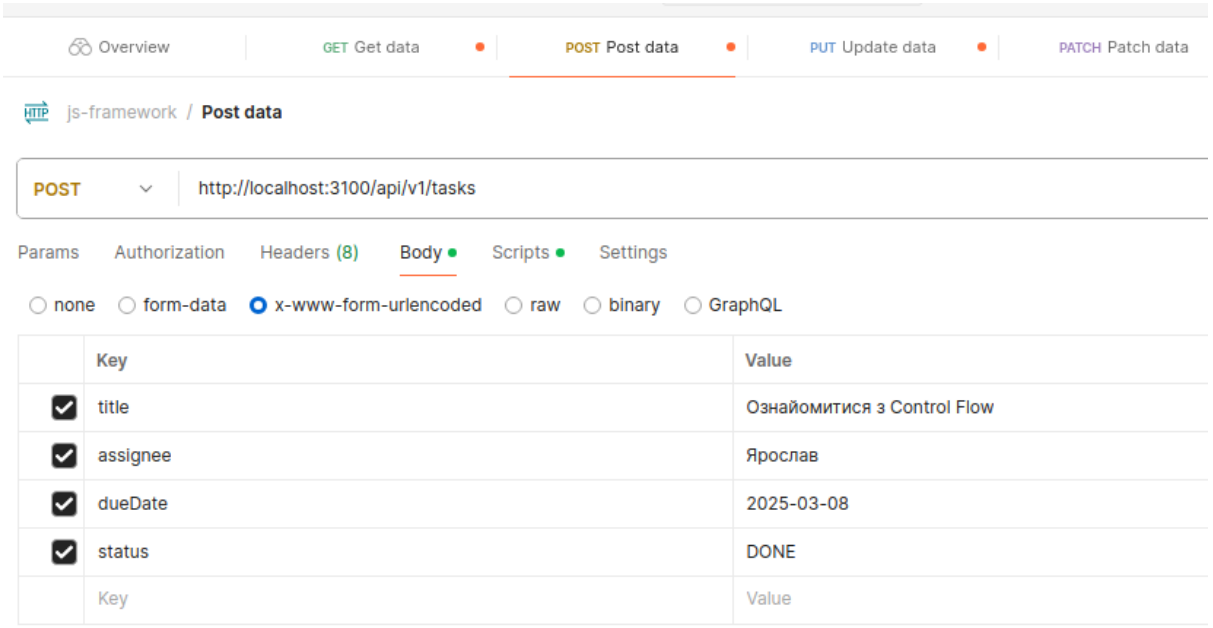
Перевіряємо список всіх завдань

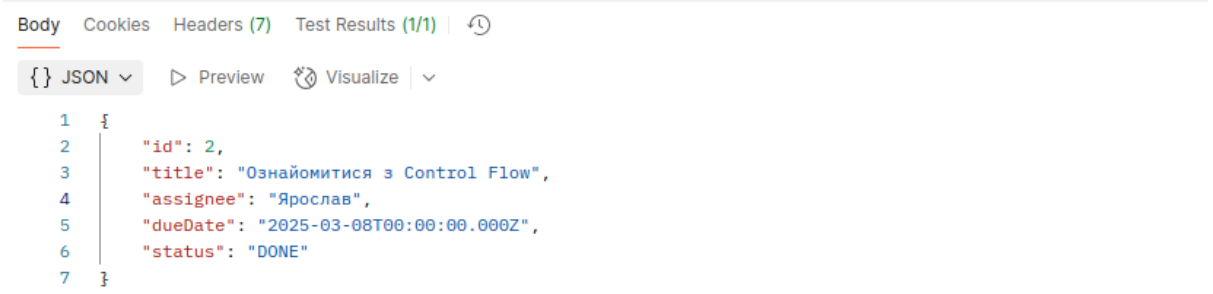


Якщо використовувати **x-www-form-urlencoded**, то необхідно у **app.js** додати **app.use(express.urlencoded({ extended: true }))**



Тоді запит через **x-www-form-urlencoded** потрапить до **req.body**



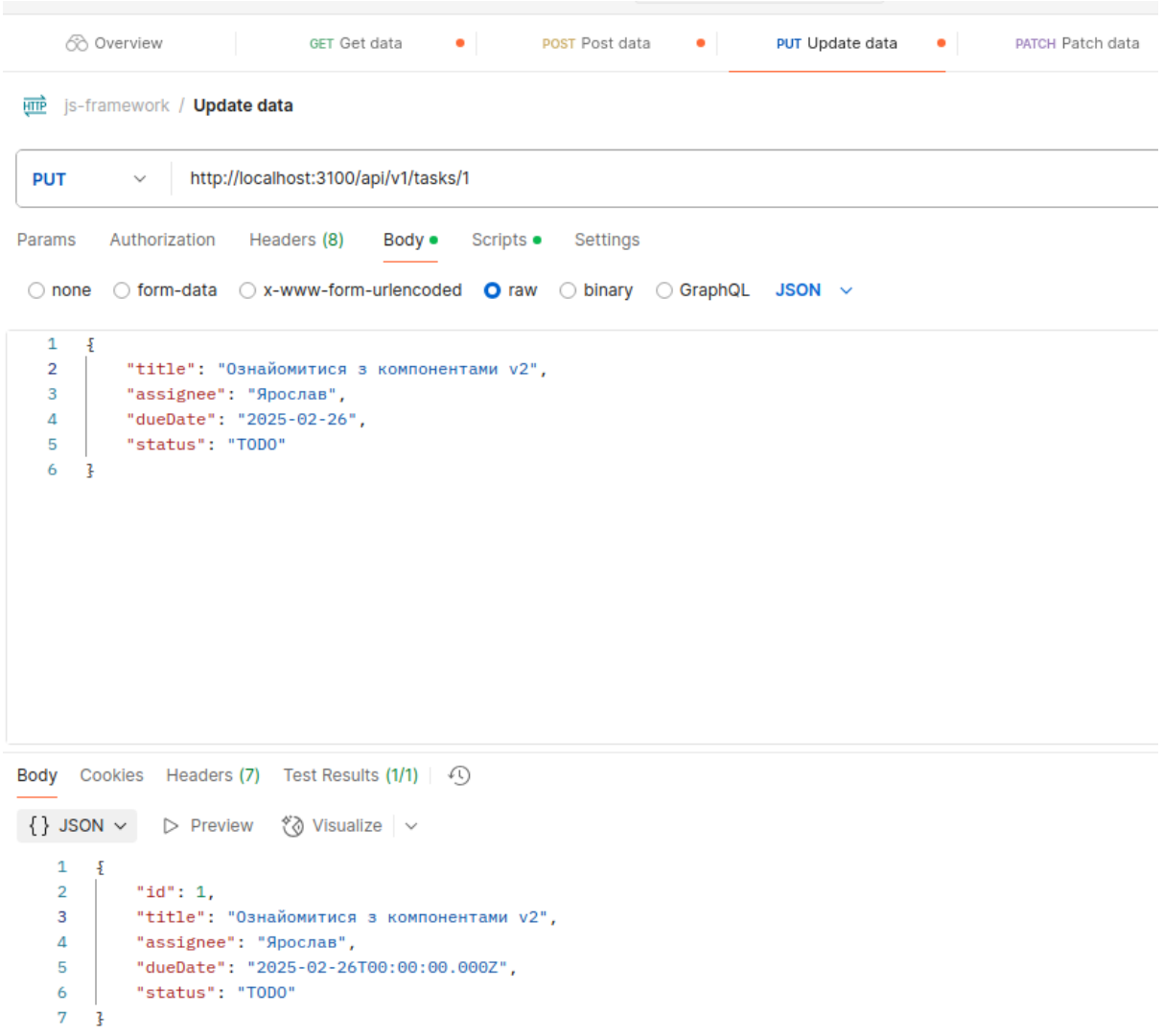


Для обробки form-data необхідно встановити пакет **multer** та підключити у **app.js**

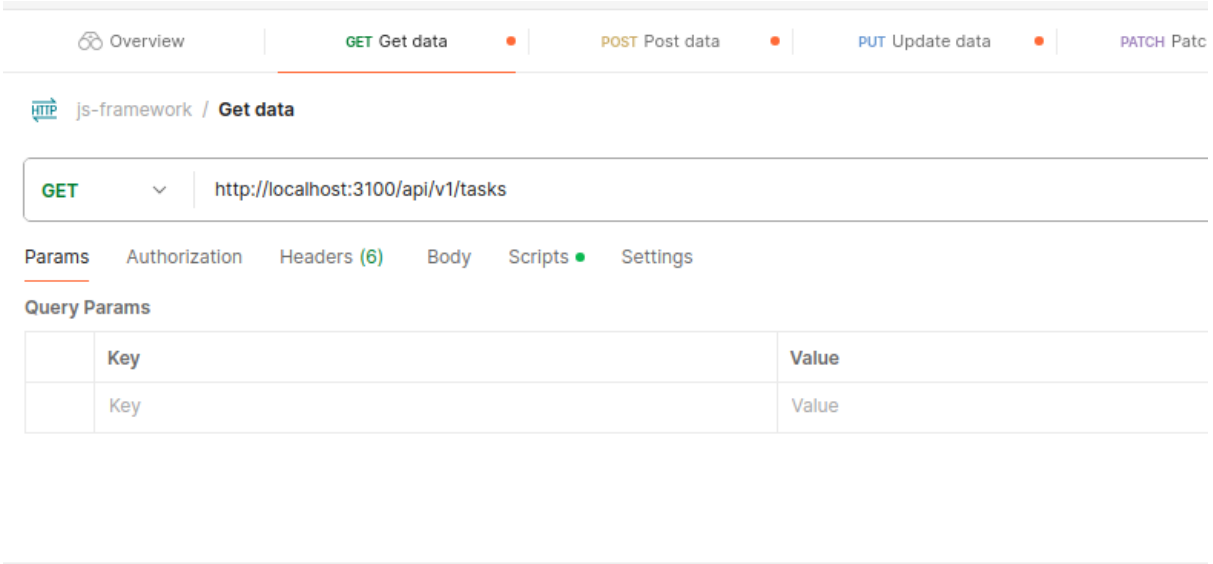
const multer = require('multer');

app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(upload.none());

Тестуємо PUT-запит за шляхом **http://localhost:3100/api/v1/tasks/1**

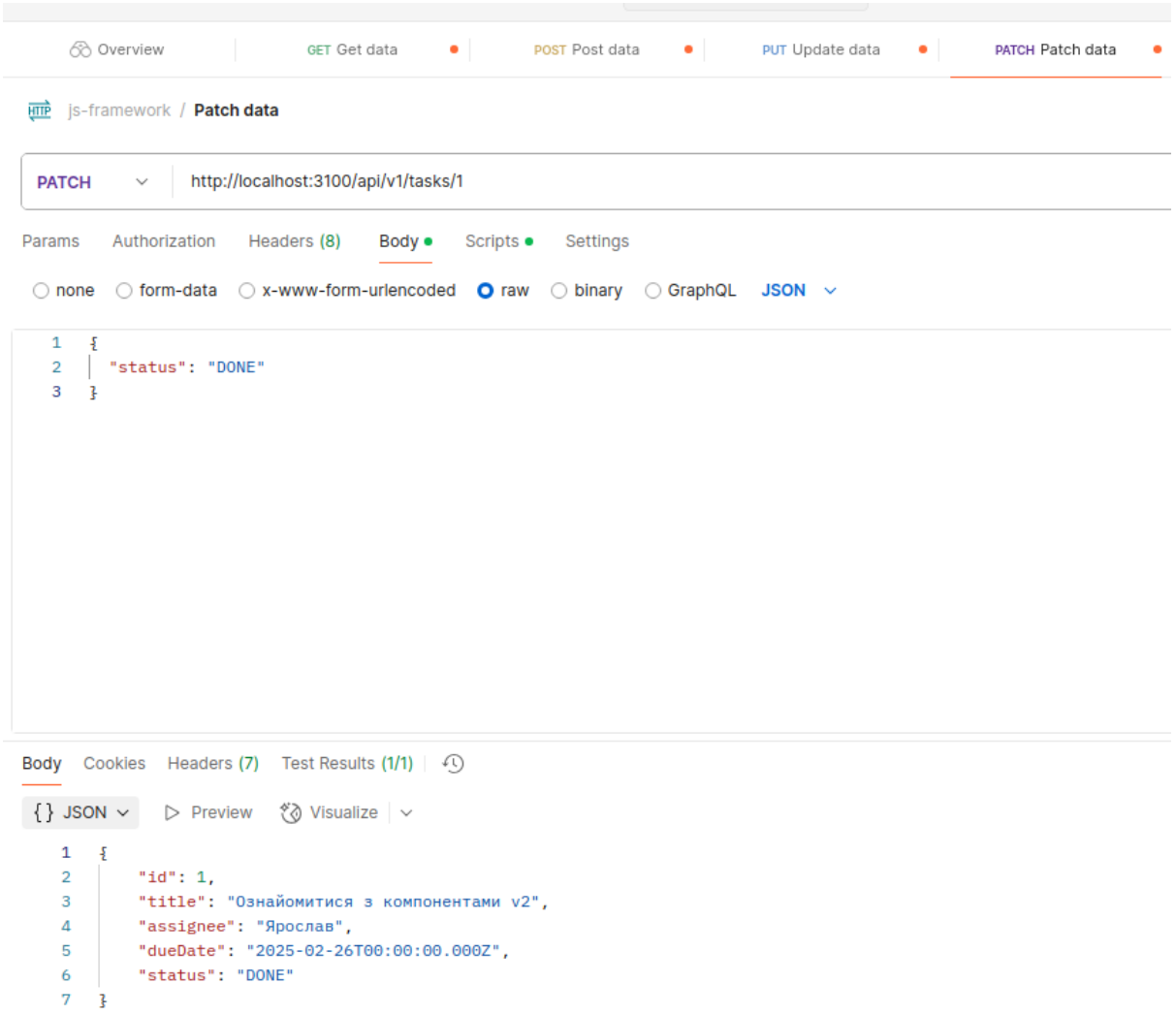


Перевіряємо відображення завдань

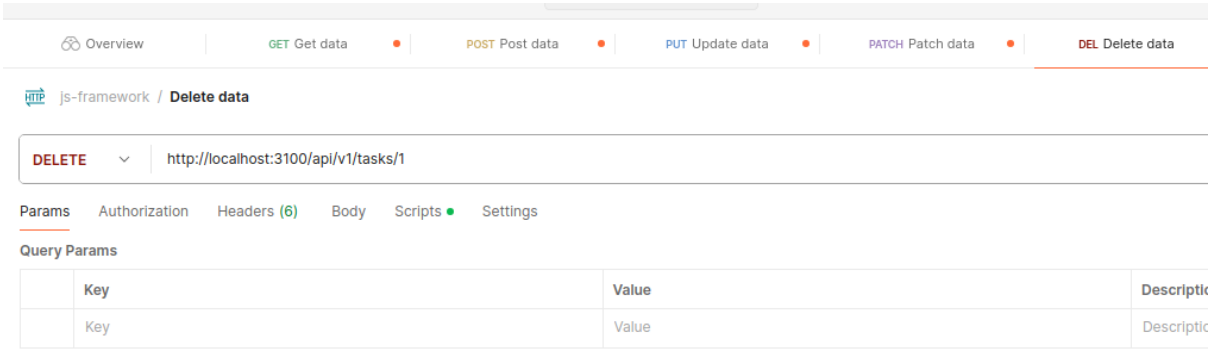


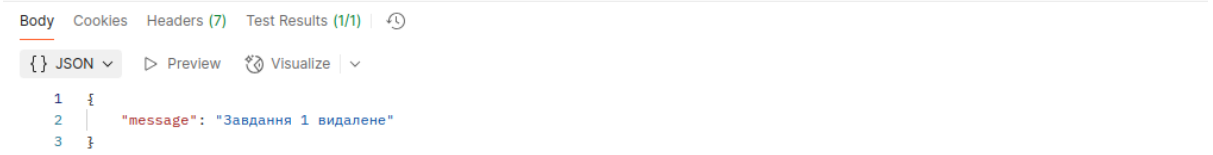
І дійсно бачимо, що ми повністю перезаписали об’єкт

Тестуємо PATCH-запит за шляхом **http://localhost:3100/api/v1/tasks/1**

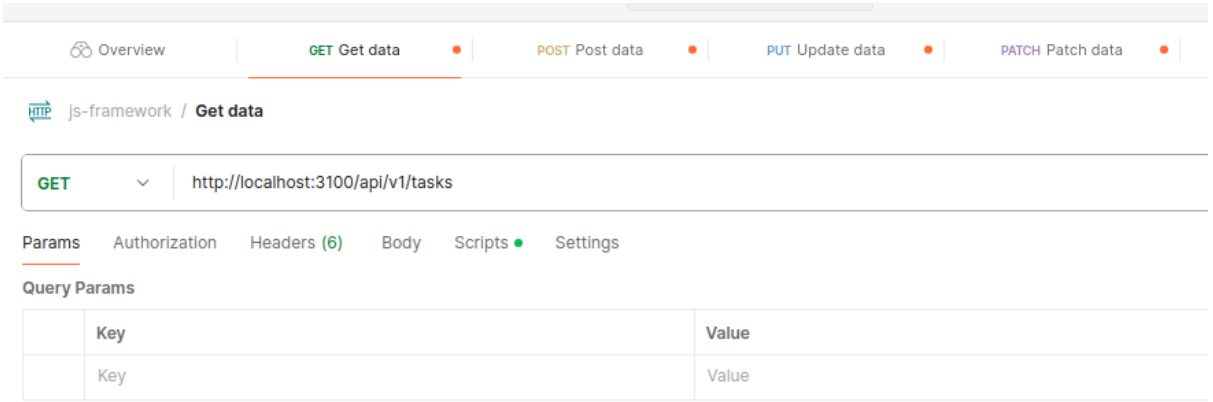


Тестуємо DELETE-запит за шляхом **http://localhost:3100/api/v1/tasks/1**

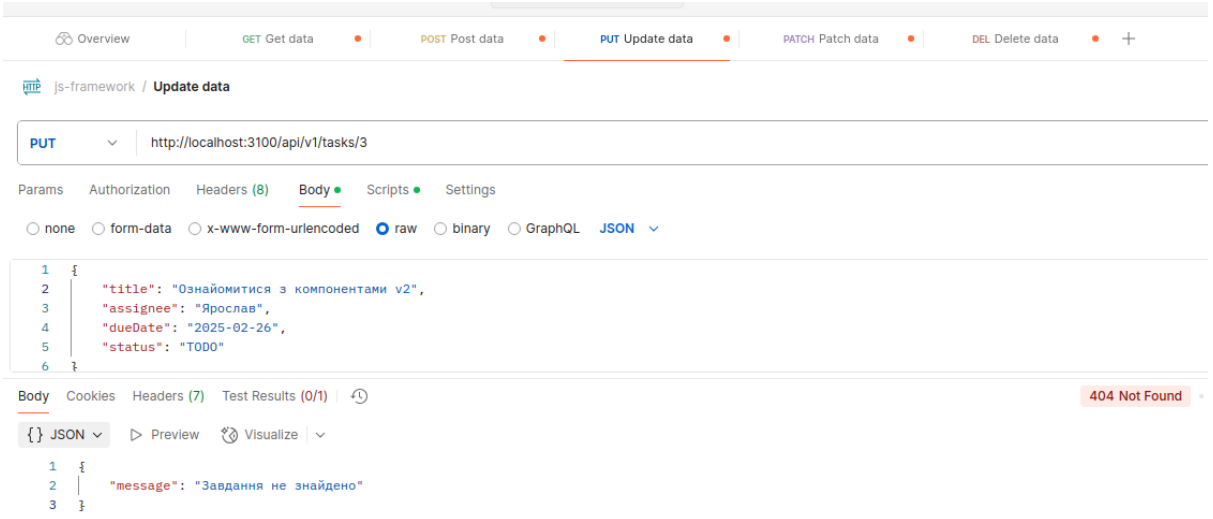


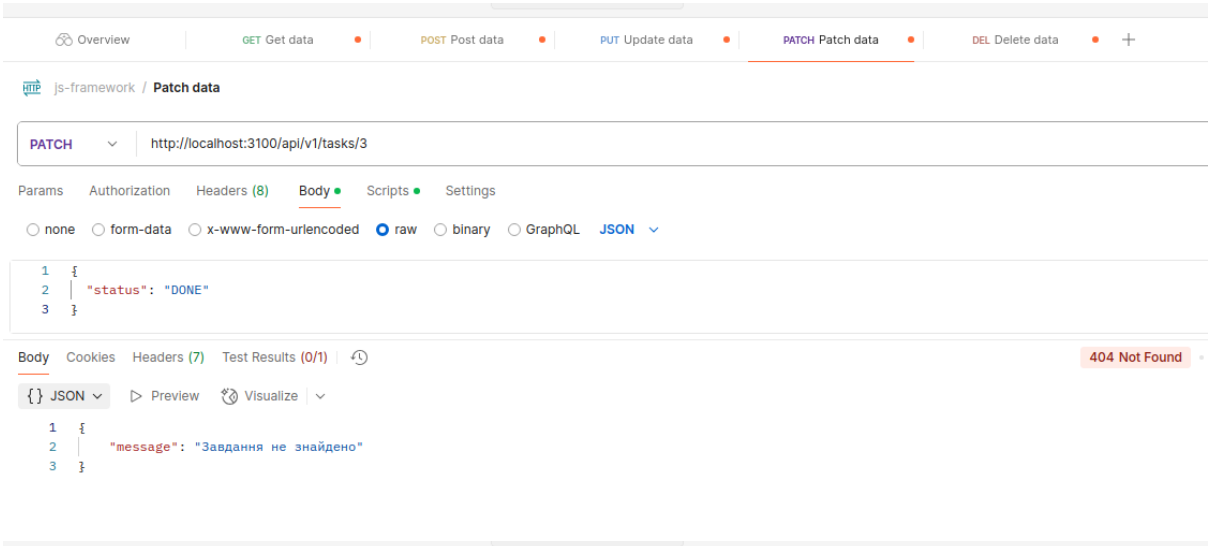


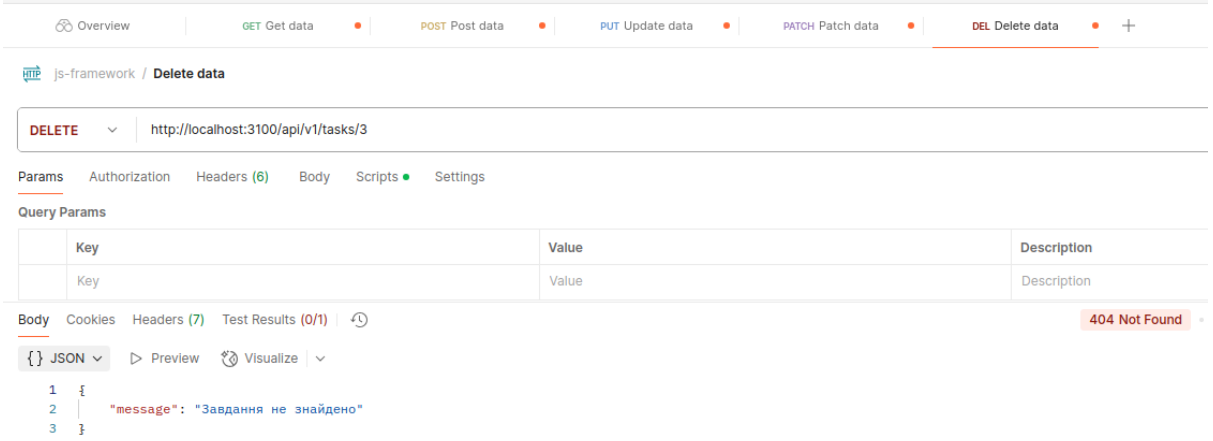
Перевіряємо і бачимо, що завдання видалено



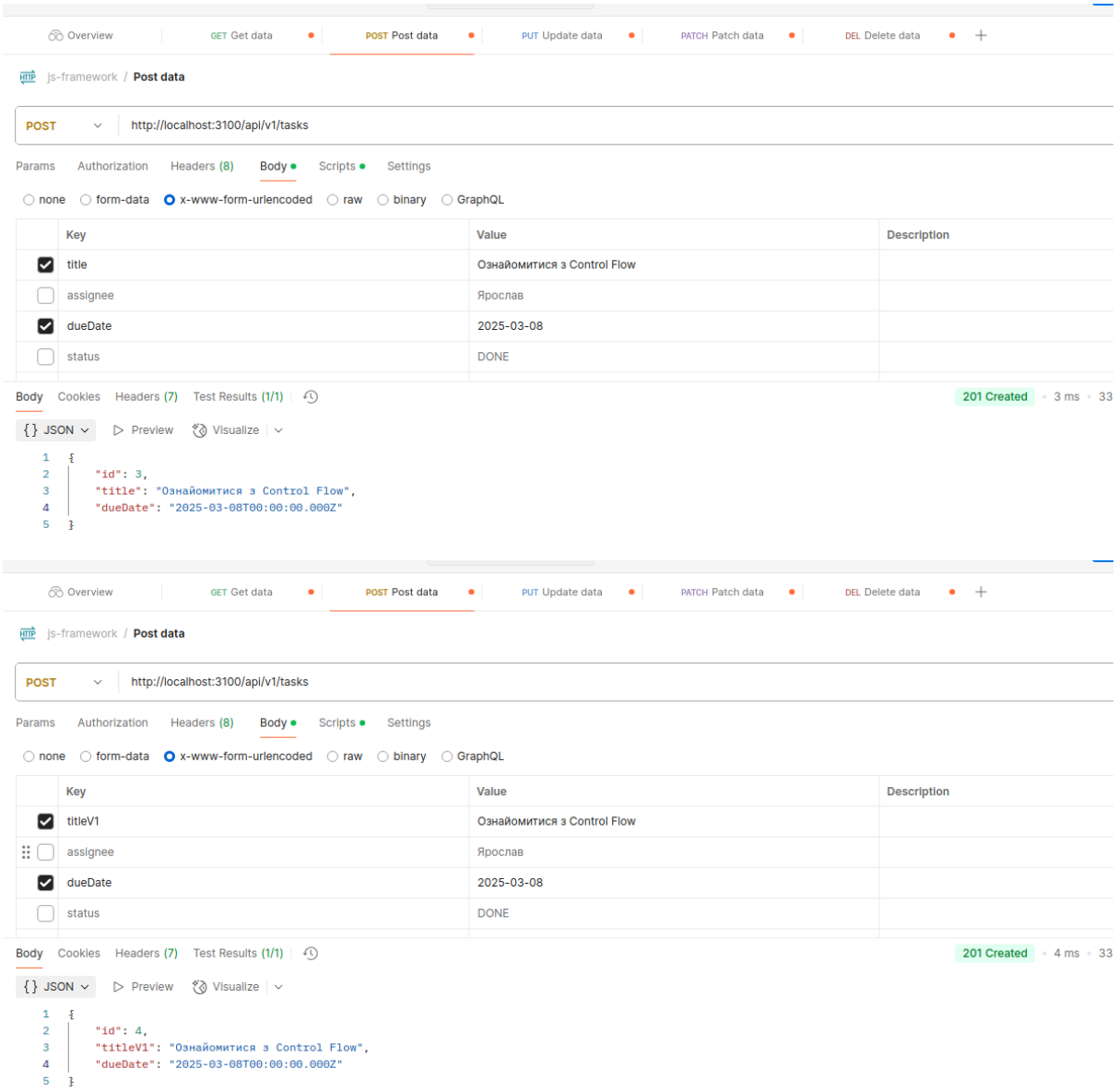
А що поверне сервер, якщо ми спробуємо оновити або видалити неіснуючий запис?







Як бачимо всі 3 запити відпрацювали коректно та повернули помилку 404, яку вже може обробити клієнт, а як себе поведе сервер у випадку передачі неіснуючих значень або якщо не передати всі обов’язкові поля? Перевіримо це за допомогою POST



Можемо побачити, що сервер некоректно обробляє ці запити, що призводить до запису у mock-модель неправильних даних, а при використанні БД викличе помилку. Для того щоб уникнути таких ситуацій можна використовувати готові рішення щодо валідації, а саме пакет **express-validator**.

Інсталюємо його за допомогою команди ***npm install express-validator***

Далі створюємо файл за шляхом src/validators/taskValidator.js та пишемо наступний код

```
const { check, validationResult } = require('express-validator');
const TaskStatus = require('../constants/taskStatus');

const ALLOWED_FIELDS = ['title', 'description', 'assignee', 'dueDate', 'status'];
```

```
// 🔹 Валідація `POST` та `PUT`
const validateTask = [
  check('title').notEmpty().withMessage('Поле title є обов’язковим'),
  check('assignee').notEmpty().withMessage('Поле assignee є обов’язковим'),
  check('dueDate').notEmpty().withMessage('Поле dueDate є обов’язковим').isISO8601().toDate(),
      check('status').isIn(Object.values(TaskStatus)).withMessage(`Статус має бути одним із: ${Object.values(TaskStatus).join(', ')}`),
  (req, res, next) => {
          const extraFields = Object.keys(req.body).filter(field => !ALLOWED_FIELDS.includes(field));
          if (extraFields.length) {
          return res.status(400).json({ error: `Зайві поля: ${extraFields.join(', ')}` });
          }
          next();
  }
];

// 🔹 Валідація `PATCH` (перевіряємо тільки статус)
const validateTaskPatch = [
   check('status').optional().isIn(Object.values(TaskStatus)).withMessage(`Статус має бути одним із: ${Object.values(TaskStatus).join(', ')}`),
  (req, res, next) => {
          const extraFields = Object.keys(req.body).filter(field => !ALLOWED_FIELDS.includes(field));
          if (extraFields.length) {
          return res.status(400).json({ error: `Зайві поля: ${extraFields.join(', ')}` });
          }
          next();
  }
];

// 🔹 Функція для перевірки помилок `express-validator`
const handleValidationErrors = (req, res, next) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
          return res.status(400).json({ errors: errors.array() });
  }
  next();
};

module.exports = { validateTask, validateTaskPatch, handleValidationErrors };
```

Основні методи перевірки значень

Метод	Опис	Приклад
.notEmpty()	Перевіряє, що значення **не порожнє**	check('title').notEmpty().withMessage('Поле title обов’язкове')
.isString()	Перевіряє, що значення є **рядком**	check('title').isString().withMessage('Має бути текстом')
.isNumeric()	Перевіряє, що значення є **числом**	check('price').isNumeric().withMessage('Має бути числом')
.isInt()	Перевіряє, що значення **ціле число**	check('age').isInt({ min: 18 }).withMessage('Має бути 18+')
.isBoolean()	Перевіряє, що значення є **булевим (true або false)**	check('isActive').isBoolean().withMessage('Має бути true або false')
.isEmail()	Перевіряє, що значення є **коректним email**	check('email').isEmail().withMessage('Некоректний email')
.isISO8601()	Перевіряє, що значення є **датою у форматі ISO**	check('dueDate').isISO8601().toDate().withMessage('Некоректна дата')
.isLength({ min, max })	Перевіряє **довжину рядка**	check('password').isLength({ min: 6 }).withMessage('Пароль має бути 6+ символів')

.matches(regex)	Перевіряє, чи значення відповідає **регулярному виразу**	check('username').matches(/^[a-zA-Z0-9]+$/).withMessage('Тільки літери та цифри')

Методи перевірки списків та специфічних значень

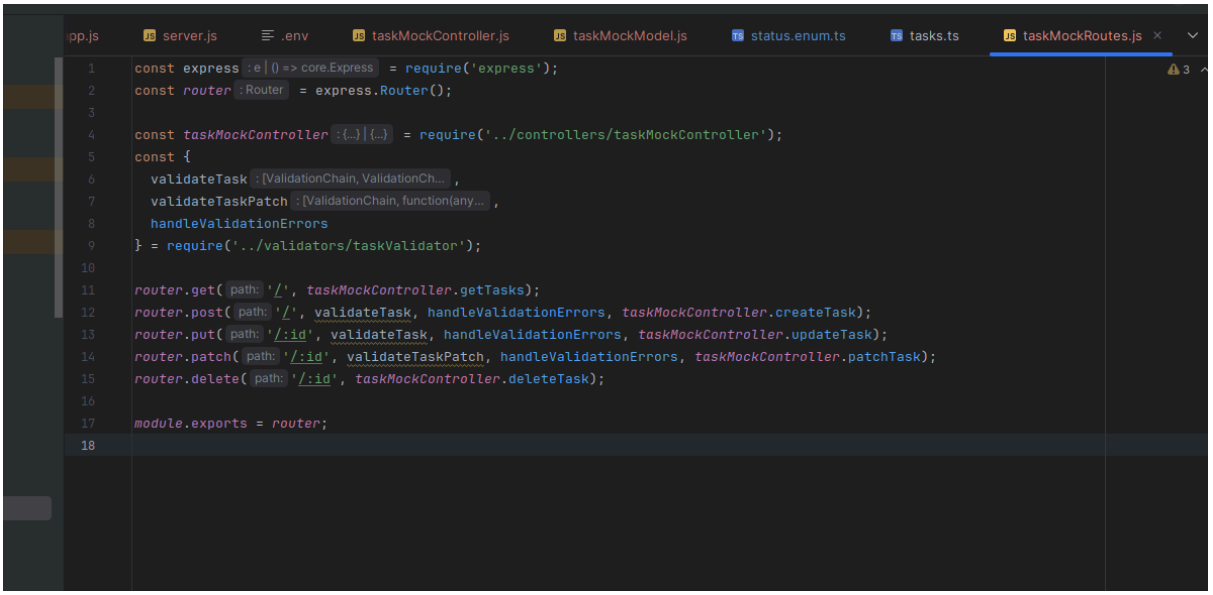
Метод	Опис	Приклад
`.isIn([...])`	Перевіряє, що значення є у **списку дозволених**	`check('status').isIn(['TODO', 'IN_PROGRESS', 'DONE']).withMessage('Недозволений статус')`
`.custom(value => ...)`	Використовує **кастомну функцію для перевірки**	`check('password').custom(value => value.includes('@')).withMessage('Має містити @')`
`.optional()`	Робить поле **необов'язковим**	`check('description').optional().isString()`

Обробка помилок

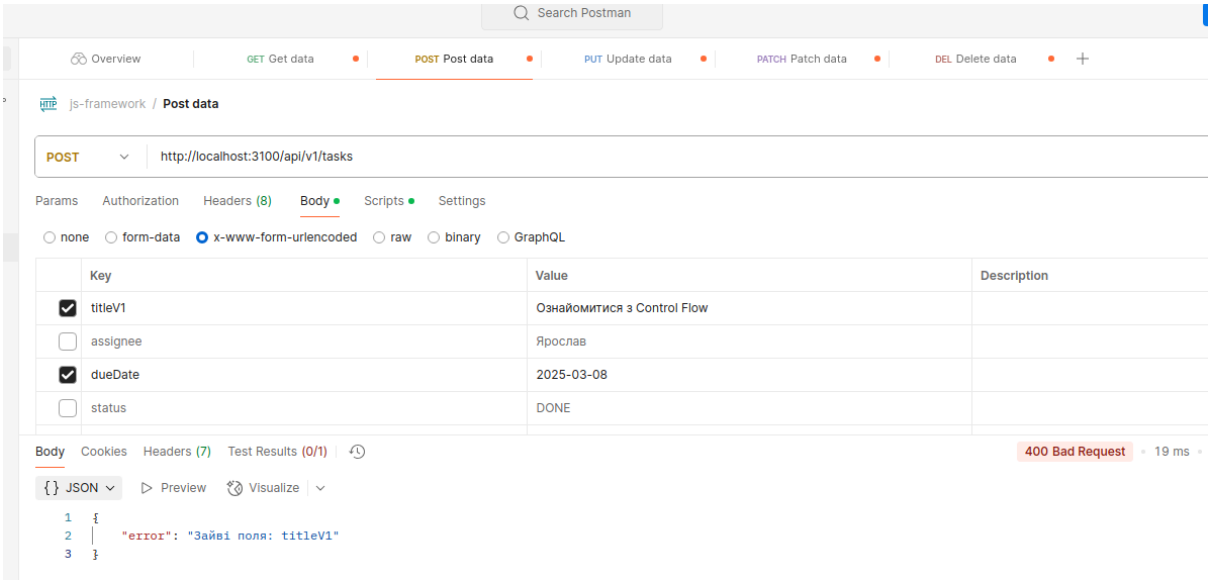
Завжди потрібно використовувати validationResult(), щоб отримати список помилок та передати їх у відповідь

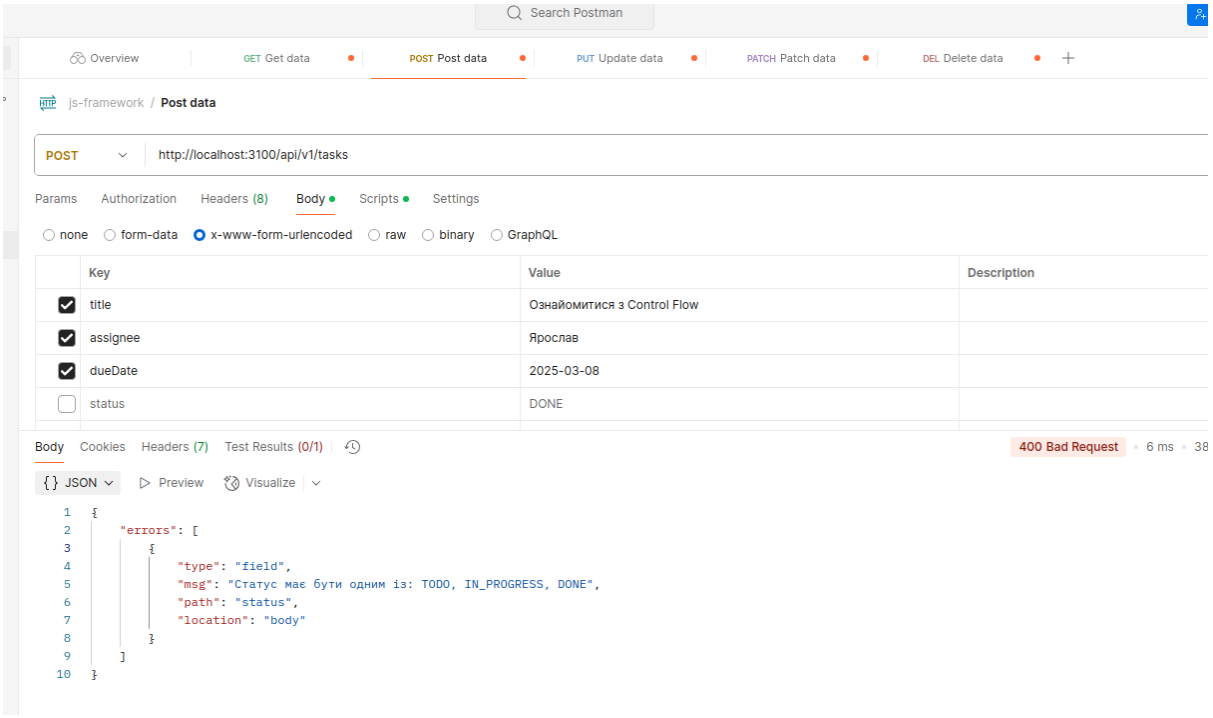
```
const { validationResult } = require('express-validator');

const handleValidationErrors = (req, res, next) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
            return res.status(400).json({ errors: errors.array() });
  }
  next();
};
```

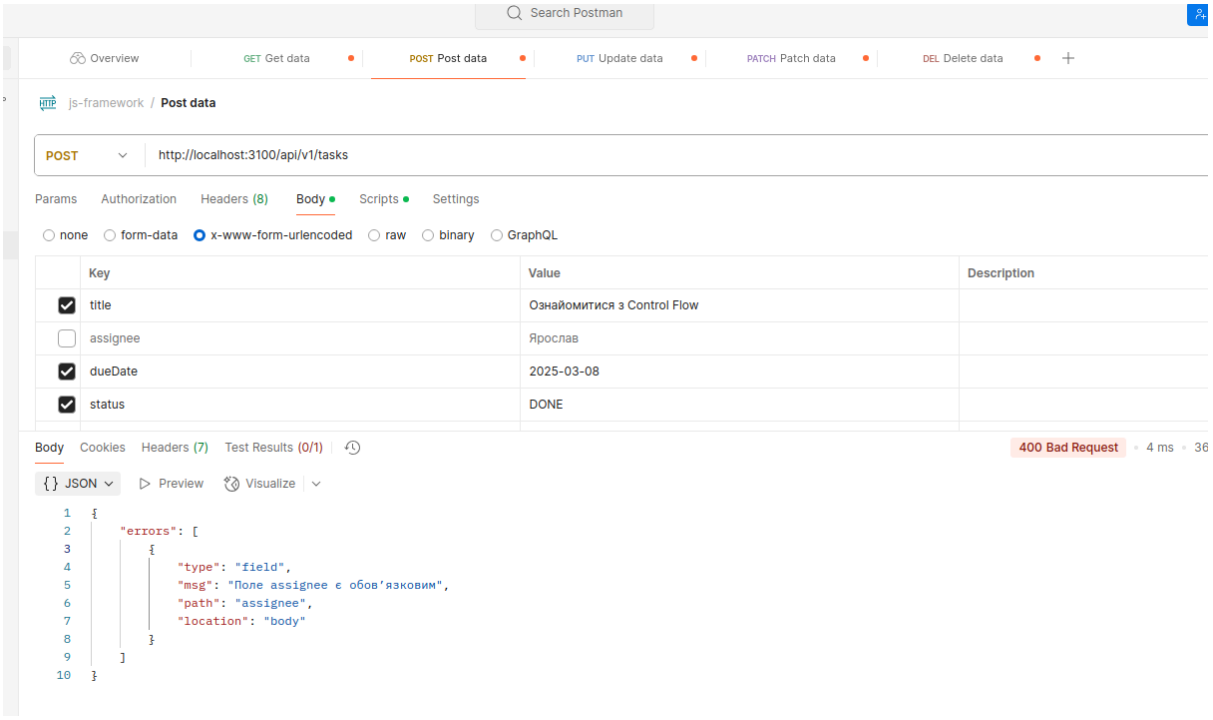
Після створення правил валідації підключаємо їх до файлу роутера та до потрібних маршрутів



Перевіряємо роботу







Тепер переходимо до вирішення наступної задачі. У попередніх роботах та у поточній ми конвертували дату між типами **string** та **Date**, як на стороні Angular, так і на стороні серверу. Для автоматизації цього процесу напишемо кастомний middleware.

Він буде мати назву **dateMiddleware.js**. Створюємо його у теці **middleware** та пишемо наступний код

```
const formatDate = (date) :null | string => {  Show usages
  if (!date) return null;
  const d :Date = new Date(date);
  return d.toISOString().split( separator: 'T')[0]; // Отримуємо 'YYYY-MM-DD'
};

const parseDateMiddleware = (req, res, next) :void => {  Show usages
  if (req.body.dueDate) {
    req.body.dueDate = new Date(req.body.dueDate); // Зберігаємо як `Date`
  }
  next();
};

const formatResponseDateMiddleware = (req, res, next) :void => {  Show usages
  const oldJson = res.json; // Зберігаємо оригінальний `res.json`
  res.json = function (data) :void {
    if (Array.isArray(data)) { // Якщо відповідь - масив об'єктів (наприклад, список завдань)
      data = data.map(task => ({
        ...task,
        dueDate: task.dueDate ? formatDate(task.dueDate) : null,
      }));
    } else if (typeof data === 'object' && data !== null) { // Якщо відповідь - один об'єкт (одне завдання)
      data.dueDate = data.dueDate ? formatDate(data.dueDate) : null;
    }
    oldJson.call(this, data); // Викликаємо оригінальний `res.json
  };
  next();
};
```

Давайте трохи розберемо, що за що відповідає:

1) **formatDate** - не є middleware, використовуємо як допоміжну функцію для **formatResponseDateMiddleware**. По суті призначена для того щоб конвертувати Date у формат 'YYYY-MM-DD';
2) **parseDateMiddleware** - це вже **middleware,** по суті перетворює отриману від Angular дату (YYYY-MM-DD) у об'єкт Date (наприклад, перед збереженням у mock-модель або БД);
3) **formatResponseDateMiddleware** - це також **middleware**, що конвертує дату у відповідях сервера (Date -> YYYY-MM-DD) перед відправкою Angular.

Тепер підключаємо ці middleware у файлі **app.js**

```
const express : e | () => core.Express  = require('express');

const taskMockRoutes : Router | {...}  = require('./routes/taskMockRoutes');

const {
  parseDateMiddleware : parseDateMiddleware | function(any, any, any): void ,
  formatResponseDateMiddleware : formatResponseDateMiddleware | function(any, any, any): void
} = require('./middleware/dateMiddleware');

const app : any | Express  = express();

app.use(express.json());
app.use(express.urlencoded( options: { extended: true }));

app.use(parseDateMiddleware);
app.use(formatResponseDateMiddleware);

app.use('/api/v1/tasks', taskMockRoutes);

module.exports = app;
```

Також можна прибрати взаємодію з датою у **createTask** та **updateTask**.

Наш невеликий сервер майже готов, єдине що залишилося, це підключити **cors** у файлі **app.js**

```
const express : e | () => core.Express  = require('express');
const cors : <T=e.CorsRequest extends e.Cor... | e  = require('cors');

const taskMockRoutes : Router | {...}  = require('./routes/taskMockRoutes');

const {
  parseDateMiddleware : parseDateMiddleware | function(any, any, any): void ,
  formatResponseDateMiddleware : formatResponseDateMiddleware | function(any, any, any): void
} = require('./middleware/dateMiddleware');

const app : any | Express  = express();

app.use(cors());

app.use(express.json());
app.use(express.urlencoded( options: { extended: true }));

app.use(parseDateMiddleware);
app.use(formatResponseDateMiddleware);

app.use('/api/v1/tasks', taskMockRoutes);

module.exports = app;
```

7) Переходимо до підключення нашого API до Angular

Для початку необхідно внести певні зміни у існуючі файли, враховуючи логіку сервера, а саме:

1) Змінимо значення кирилицею у файлі **status.enum.ts** на їх аналоги латиницею (зробимо, як на сервері)

```
export enum TaskStatus {  Show usages  Kryvyi Yaroslav *
  TODO = 'TODO',
  IN_PROGRESS = 'IN_PROGRESS',
  DONE = 'DONE'
}
```

2) Приберемо з Angular логіку, що пов’язана з форматуванням дати

Для цього у файлі **task.model.ts** змінуємо тип властивості dueDate на string та образу робимо id необов’язковим параметром (не найкраща практика, приміняємо тільки у контексті навчання, можливі інші варіанти, коли id обов’язковий: окремий інтерфейс для POST, використання Partial (максимально незручно, якщо об’єкт містить багато властивостей), використання id = null (врахуйте зміни в інтерфейсі, а також додаткові перевірки))

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

export interface Task { // експортуємо інтерфейс для використання у інших частинах коду
  id?: number;
  title: string; // заголовок завдання
  description?: string; // опис завдання (? - необов'язковий параметр)
  assignee: string; // виконавець
  dueDate: string; // дата дедлайну
  status: TaskStatus; // статус завдання
}

```

Модернізуємо **task-form.component.ts**

```
ngOnChanges(changes: SimpleChanges): void {  no usages  Kryvyi Yaroslav *
  if(changes['editTask'] && this.editTask) {
    this.taskForm.patchValue(this.editTask);
  }
}

addTask(): void {  Show usages  Kryvyi Yaroslav *
  if (this.taskForm.valid) {
    let taskData : { id: number | undefined; title?: string ... } = {
      ...this.taskForm.value,
      id: this.editTask ? this.editTask.id : undefined,
    };
    this.taskAdd.emit(taskData as Task);
    this.taskForm.reset();
  }
}
```

Так як id став необов'язковим параметром, то трохи модернізуємо функцію deleteTask у **task-item.component.ts**

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

deleteTask(id: number | undefined): void {  Show usages  new *
  if (!id) return; // Якщо id відсутній - виходимо
  this.taskDeleted.emit(id); // Викликаємо подію при натисканні кнопки "Видалити"
}
```

Тепер переходимо до адаптації нашого сервісу. Для початку запишемо url нашого REST API-сервера до змінної у config.ts

```
import {InjectionToken} from '@angular/core';

export interface AppConfig {  Show usages
  apiUrl: string;
}

export const APP_CONFIG = {  Show usages
  apiUrl: 'http://localhost:3100/api'
}

export const CONFIG_TOKEN = new InjectionToken<AppConfig>('CONFIG_TOKEN', {  no usages
  providedIn: 'root',
  factory: () : { apiUrl: string } => APP_CONFIG
});
```

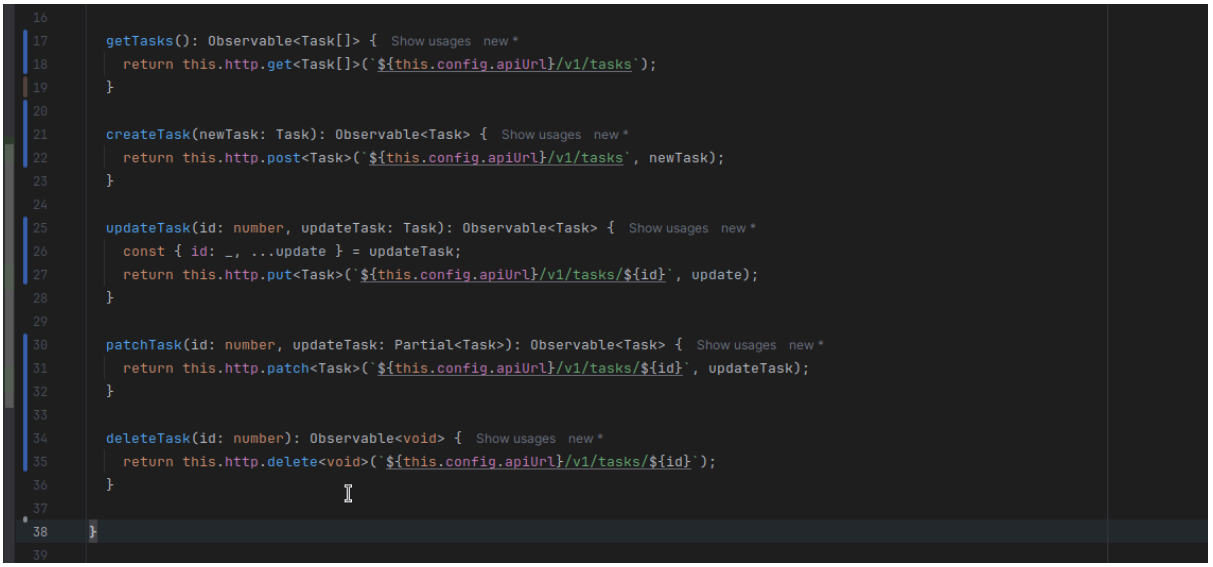
Підключимо HttpClient у **app.module.ts**

```ts
import { TaskListComponent } from './components/task-list/task-list.component';
import { TaskItemComponent } from './components/task-item/task-item.component';
import { StatusFilterPipe } from './share/pipes/status-filter.pipe';
import { TaskFormComponent } from './components/task-form/task-form.component';
import {provideHttpClient, withInterceptorsFromDi} from '@angular/common/http';

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

Модифікуємо TaskService, для початку інжектимо необхідні залежності (HttpClient, CONFIG_TOKEN)

```ts
import {Inject, Injectable} from '@angular/core';
import {Task} from '../core/models/task.model';
import {HttpClient} from '@angular/common/http';
import {AppConfig, CONFIG_TOKEN} from '../share/config/config';

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

  constructor(  no usages  new *
    private http: HttpClient,
    @Inject(CONFIG_TOKEN) private config: AppConfig
  ) { }
```

Реалізовуємо методи



```
  getTasks(): Observable<Task[]> {  Show usages  new *
    return this.http.get<Task[]>(`${this.config.apiUrl}/v1/tasks`);
  }

  createTask(newTask: Task): Observable<Task> {  Show usages  new *
    return this.http.post<Task>(`${this.config.apiUrl}/v1/tasks`, newTask);
  }

  updateTask(id: number, updateTask: Task): Observable<Task> {  Show usages  new *
    const { id: _, ...update } = updateTask;
    return this.http.put<Task>(`${this.config.apiUrl}/v1/tasks/${id}`, update);
  }

  patchTask(id: number, updateTask: Partial<Task>): Observable<Task> {  Show usages  new *
    return this.http.patch<Task>(`${this.config.apiUrl}/v1/tasks/${id}`, updateTask);
  }

  deleteTask(id: number): Observable<void> {  Show usages  new *
    return this.http.delete<void>(`${this.config.apiUrl}/v1/tasks/${id}`);
  }
}
```

Модифікуємо **task-list.component.ts**

```
export class TaskListComponent implements OnInit {

 myTasks$!: Observable<Task[]>;

 selectedStatus!: TaskStatus | 'all';

 editingTask: Task | null = null;

 constructor(private taskService: TaskService) {
 }

 ngOnInit(): void {
  this.myTasks$ = this.taskService.getTasks();
 }

 loadTasks(): void {
        this.myTasks$ = this.taskService.getTasks();
 }

 addTask(task: Task): void {
        if (this.editingTask) {
        if (!task.id) return;
        this.taskService.updateTask(task.id, task).subscribe({
```

```
        next: () => this.loadTasks(),
        error: error => console.log(error),
        })
        this.editingTask = null;
        } else {
        this.taskService.createTask(task).subscribe({
        next: () => this.loadTasks(),
        error: error => console.log(error),
        });
        }
  }

  editTask(task: Task): void {
        this.editingTask = {...task};
  }

  deleteTask(id: number): void {
        this.taskService.deleteTask(id).subscribe({
        next: () =>  this.loadTasks(),
        error: error => console.log(error),
        });
  }

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

  protected readonly TaskStatus = TaskStatus;

}
```

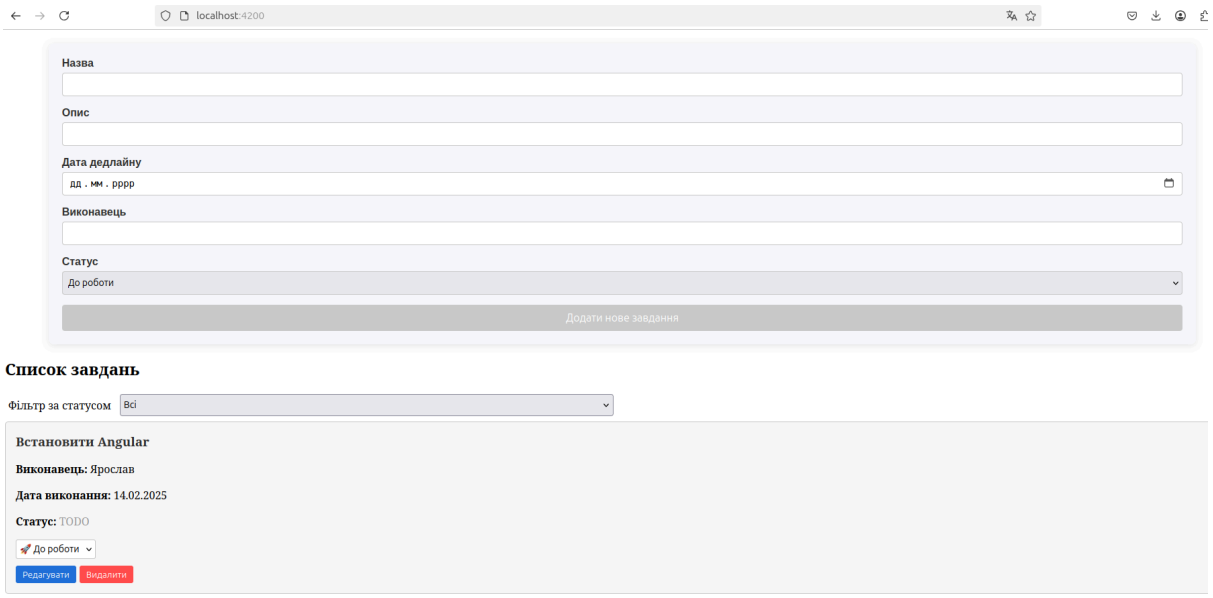
Також оновимо метод **updateStatus** для демонстрації роботи patch у **task-item.component.ts**

```
updateStatus(event: Event) :void {  Show usages  Kryvyi Yaroslav *
  const selectedValue :string  = (event.target as HTMLSelectElement).value;
  if (!this.task.id) return;

  this.taskService.patchTask(this.task.id, { status: selectedValue as TaskStatus }).subscribe({
    next: updatedTask :Task  => this.task.status = updatedTask.status,
    error: error :any  => console.error('Помилка оновлення статусу', error),
  });
}
```

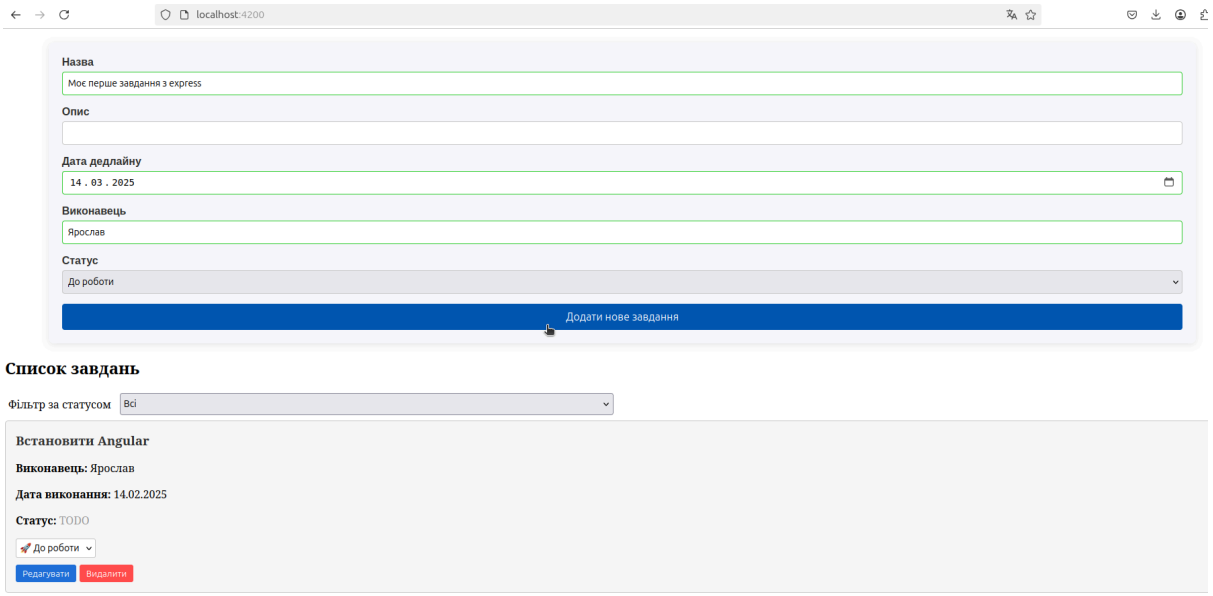
Оновлюємо **task-list.component.ts** додаючи до циклу @for async pipe

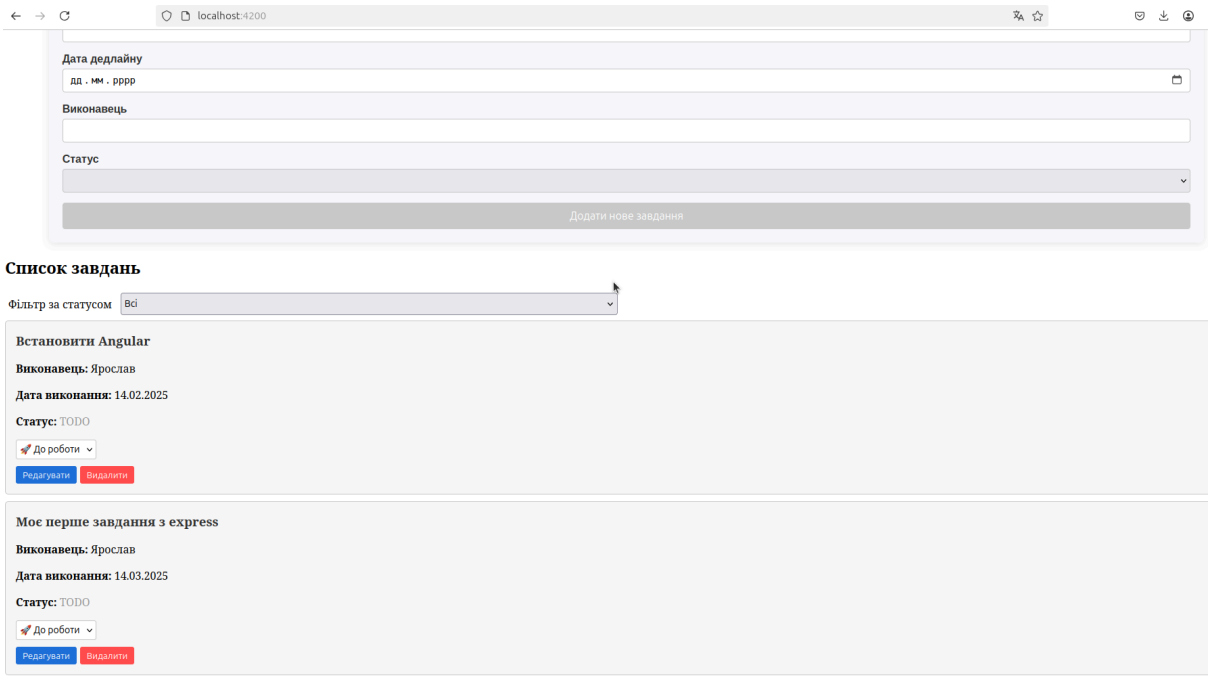
```
<app-task-form
  [editTask]="editingTask"
  (taskAdd)="addTask($event)"
></app-task-form>
<h2>Список завдань</h2>
<div class="task-filter">
  <p>Фільтр за статусом</p>
  <select (change)="onSelected($event)">
    <option [value]="'all'">Всі</option>
    <option [value]="TaskStatus.TODO">До роботи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

Перевіряємо роботу додатка

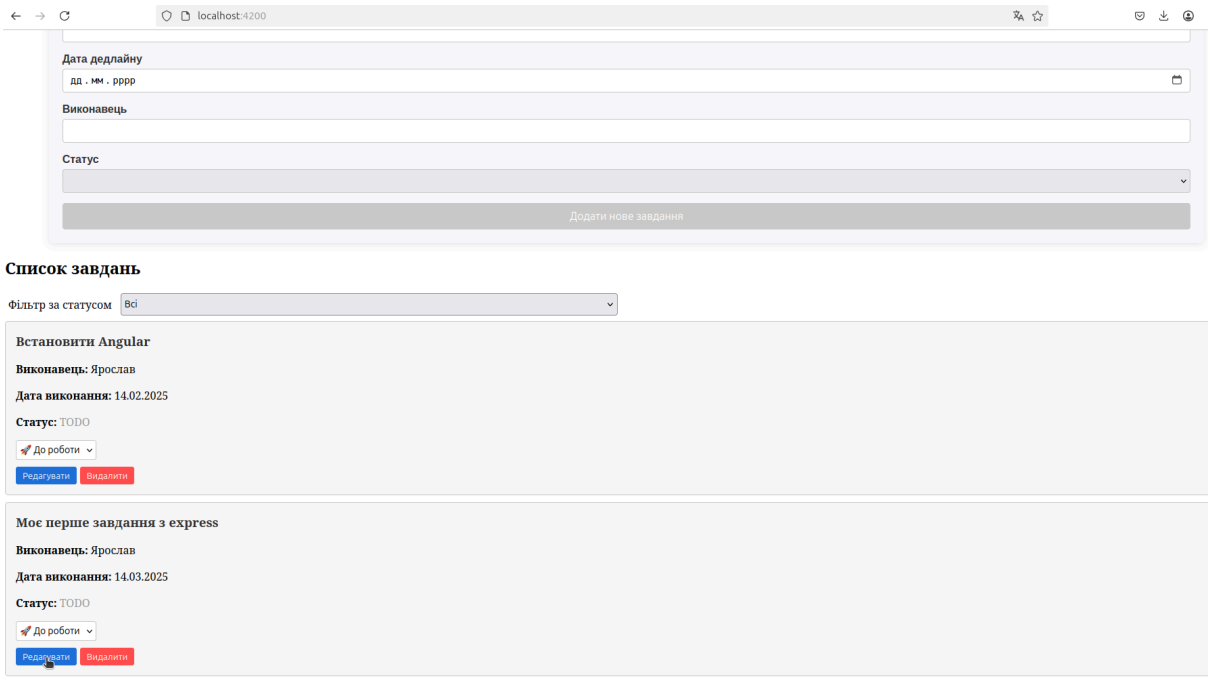


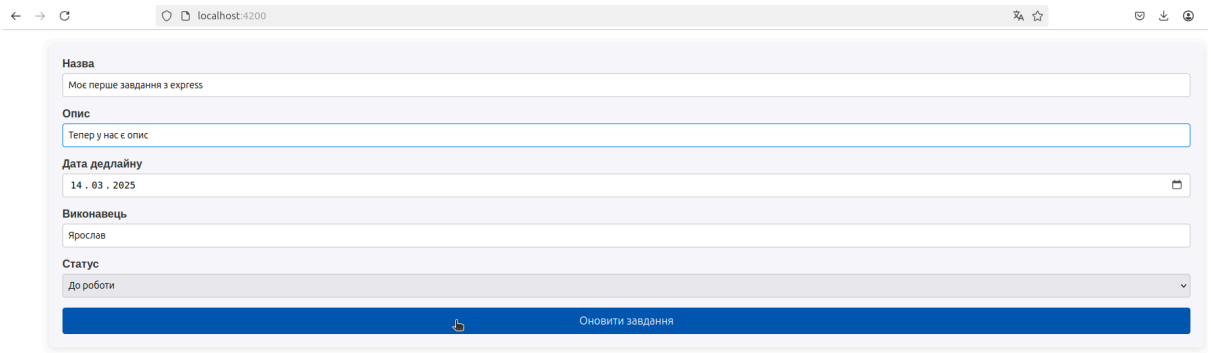
Додаємо нове завдання



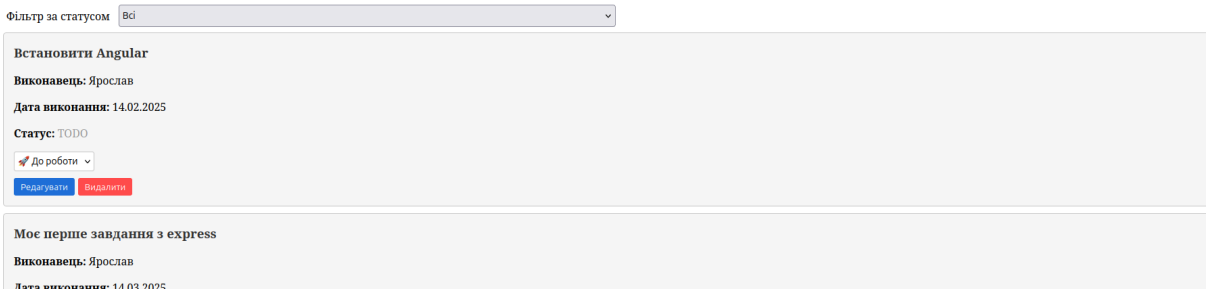


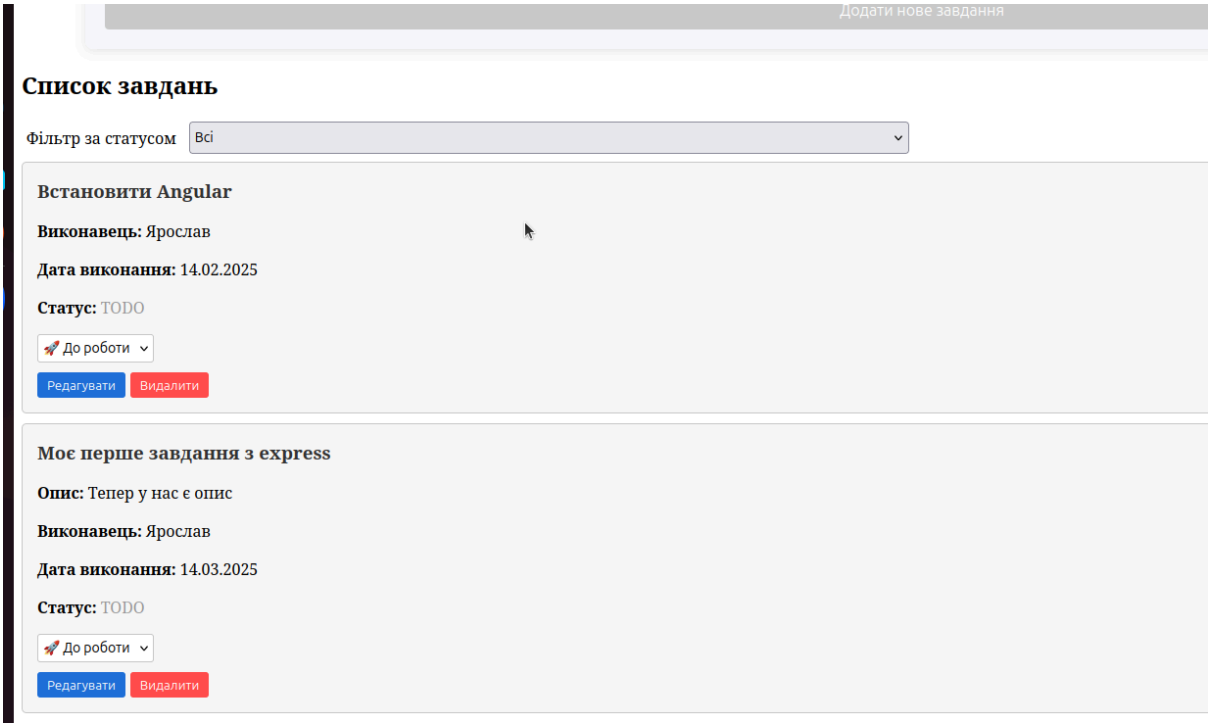
Оновлюємо завдання



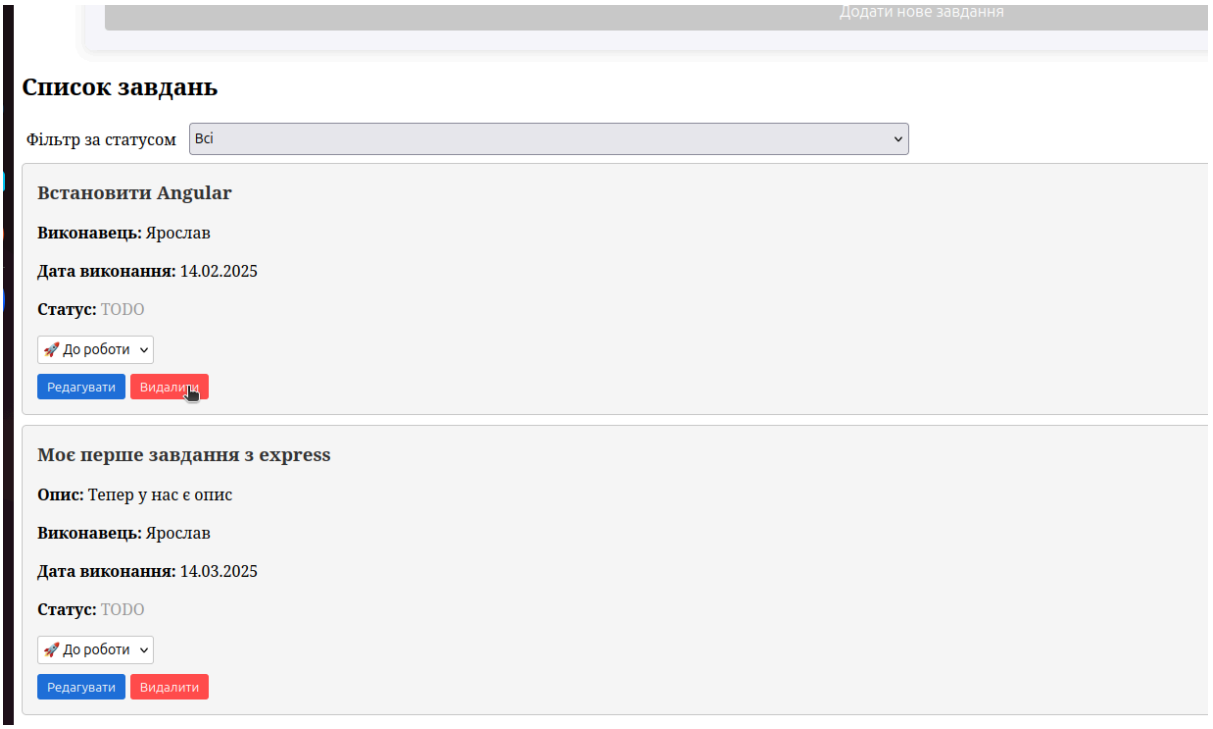


Список завдань





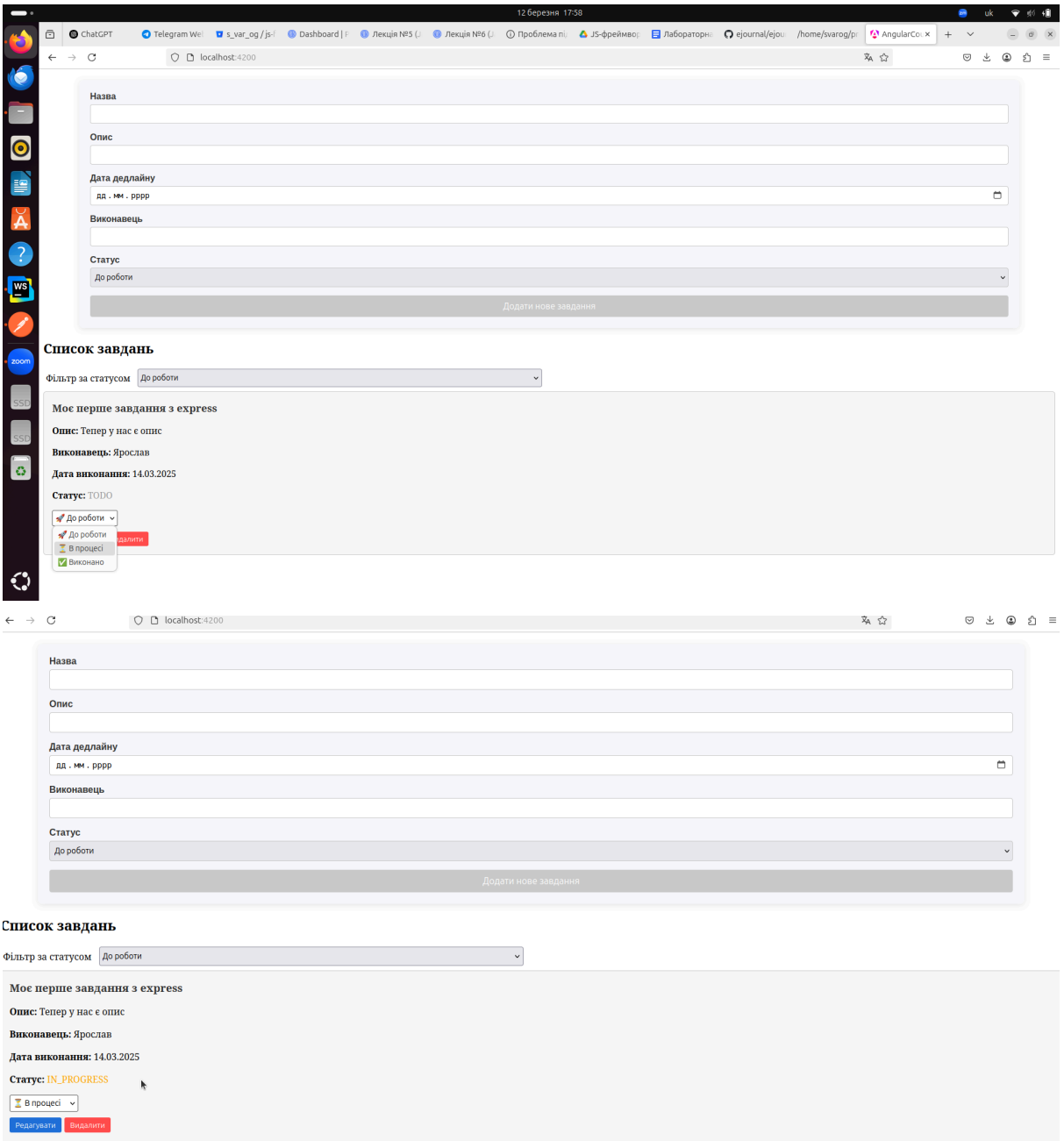
Видалення


Додати нове завдання
Список завдань
Фільтр за статусом
Всі
Встановити Angular
Виконавець: Ярослав
Дата виконання: 14.02.2025
Статус: TODO
До роботи
Редагувати
Видалити
Моє перше завдання з express
Опис: Тепер у нас є опис
Виконавець: Ярослав
Дата виконання: 14.03.2025
Статус: TODO
До роботи
Редагувати
Видалити


localhost:4200
Назва
Опис
Дата дедлайну
дд . мм . рррр
Виконавець
Статус
До роботи
Додати нове завдання


Список завдань
Фільтр за статусом
Всі
Моє перше завдання з express
Опис: Тепер у нас є опис
Виконавець: Ярослав
Дата виконання: 14.03.2025
Статус: TODO
До роботи
Редагувати
Видалити

Часткове оновлення



Вирішимо ще дві проблеми. Перша заключається у тому, що статуси тепер виводяться на латиниці та більше не працює фільтр за статусом (бо прибрали з @for наш кастомний pipe). Переклад вирішимо за допомогою іншого кастомного пайпу, можна звісно використати @switch, але якщо потрібно буде використовувати переклад у різних частинах коду, то це буде надлишково. Створюємо наш кастомний пайп у тій же теці, що і попередній:

```
svarog@svarog:~/projects/angular-course$ ng g p share/pipes/task-status
CREATE src/app/share/pipes/task-status.pipe.spec.ts (204 bytes)
CREATE src/app/share/pipes/task-status.pipe.ts (246 bytes)
UPDATE src/app/app.module.ts (1118 bytes)
```

Реалізуємо пайп, він повинен приймати TaskStatus

```
@Pipe({  Show usages
  name: 'taskStatus',
  standalone: false
})
export class TaskStatusPipe implements PipeTransform {

  transform(status: TaskStatus): string {  Show usages
    const statusMap: { [key in TaskStatus]: string } = {
      [TaskStatus.TODO]: 'До роботи',
      [TaskStatus.IN_PROGRESS]: 'В процесі',
      [TaskStatus.DONE]: 'Виконано'
    };
    return statusMap[status] || 'Невідомий статус';
  }

}
```

Далі реалізуємо фільтрацію, для цього спочатку модернізуємо get-запити на стороні сервера та Angular

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

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

  return this.http.get<Task[]>(`${this.config.apiUrl}/v1/tasks`, {params: params});
}
```

Тепер метод приймає необов’язковий параметр статусу та перевіряє, якщо він є, то записує його як параметр та передаємо серверу.

```
const getTasks = (req, res) :void  => {  Show usages
  let filteredTasks :({...})[]  = [...tasks];

  if (req.query.status) {
    filteredTasks = filteredTasks.filter(task :{...}  => task.status === req.query.status);
  }

  res.json(filteredTasks);
}
```

На сервері також робимо перевірку, якщо статус наявний у запиті, то фільтруємо та повертаємо тільки завдання з вказаним статусом, якщо ні, то всі завдання.

Далі необхідно внести деякі зміни у **task-list.component.ts**

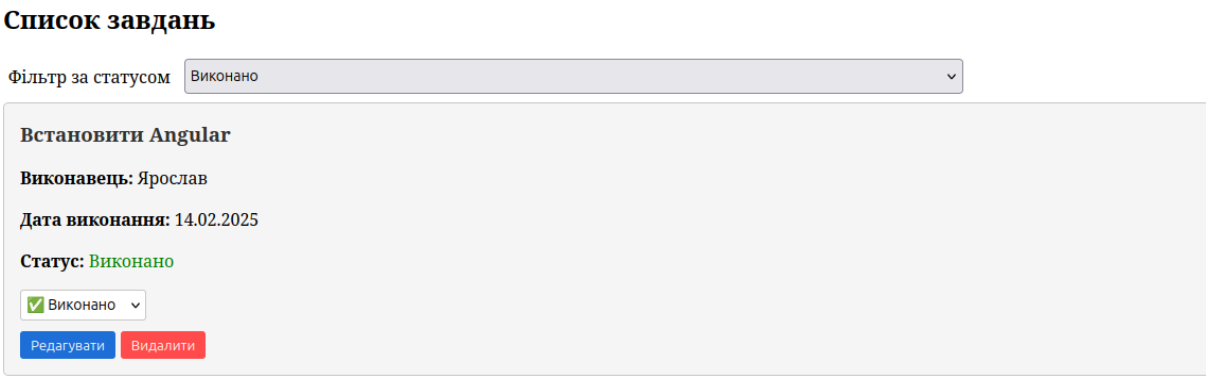
```
loadTasks(status?: string): void {  Show usages  new *
  this.myTasks$ = this.taskService.getTasks(status);
}
```

Для **loadTasks** також вказуємо необов’язковий параметр статусу

```
selectedStatus: TaskStatus | '' = '';
```

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

Тепер при зміні статусу буде автоматично викликатися функція **loadTasks**, якій автоматично буде передаватися поточний статус.



Якщо є потреба при додаванні, оновлені або видалені залишати поточну фільтрацію, то необхідно передавати змінну **selectedStatus** при виклику методу **loadTasks** відповідних методів додавання, оновлення або видалення.

Контрольні питання

1) Яка роль та зона застосування middleware у Express.js? Який метод дозволяє їх підключати?
2) Яке призначення req у Express.js?
3) Яке призначення res у Express.js?
4) Який метод групує маршрути за одним шляхом?
5) Для чого використовується AsyncPipe в Angular?