Лабораторна робота № 7. Підключення на взаємодія з MongoDB

Мета: ознайомитися та навчитися створювати підключення та виконувати CRUD-операції до MongoDB.

Завдання

1) Ознайомитися з матеріалом лекції № 7;
2) Виконати пункти 1-9 ходу роботи;
3) У звіті відобразити кроки 1-9 ходу роботи;
4) Надати відповіді на контрольні питання.

Хід роботи

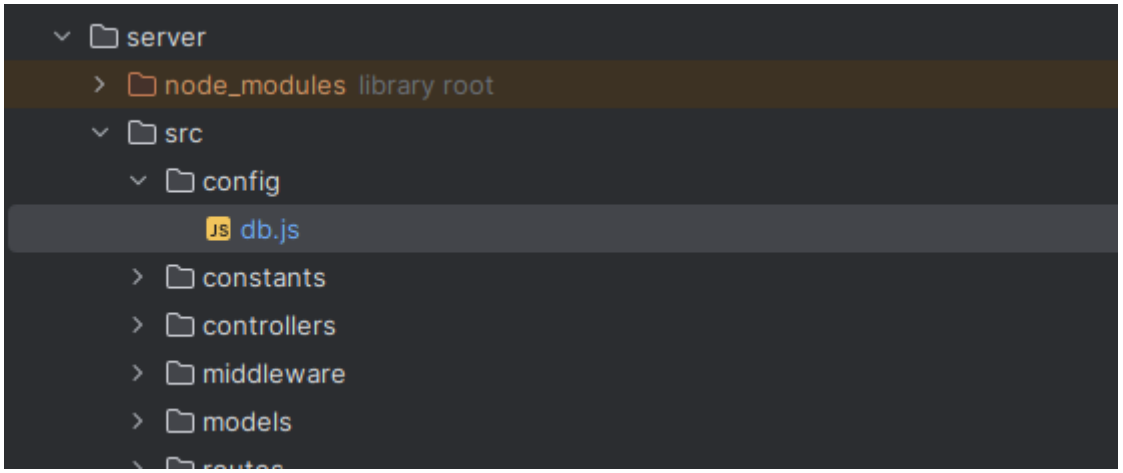
1) Встановимо пакет **mongoose** до залежностей нашого сервера

Для цього використаємо команду ***npm install mongoose***

Після виконання команди перевіряємо, що пакет додано до package.json

```
{
  "name": "server",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node src/server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "cors": "^2.8.5",
    "dotenv": "^16.4.7",
    "express": "^4.21.2",
    "express-validator": "^7.2.1",
    "mongoose": "^8.12.1"
  }
}
```

2) У теці **server/src/config** створимо файл **db.js**, він буде відповідати за підключення до БД, а також обробку стану підключення



3) Пропишемо наступний код у файлі **db.js**

Спочатку підключимо до файлу пакет **mongoose** та змінні оточення. А також зробимо перевірку, що у файлі **.env** існує URI підключення до MongoDB. Якщо його немає, то повідомляємо про це та аварійно завершуємо роботу.

```
require('dotenv').config();
const mongoose :... = require('mongoose');

const MONGO_URI :string = process.env.MONGO_URI || '';

if (!MONGO_URI) {
  console.error('Відсутній MONGO_URI у .env файлі');
  process.exit( code: 1);
}
```

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

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

Відсутній MONGO_URI у .env файлі
```

Далі створюємо функцію, яка буде відповідати за підключення до MongoDB. Так як **mongoose.connect** повертає **Promise**, то обгортаємо підключення у **try…catch**, а також робимо функцію асинхронною. також задаємо параметри, скільки чекати з'єднання (connectTimeoutMS) та скільки шукати доступний сервер (serverSelectionTimeoutMS).

Якщо виникає помилка підключення, то через 5 секунд намагаємося підключитися знов.

```
const connectDB = async () :Promise<void> => {  Show usages  new *
  try {
    await mongoose.connect(MONGO_URI, options: {
      connectTimeoutMS: 10000,
      serverSelectionTimeoutMS: 5000,
    });
    console.log('Підключено до MongoDB');
  } catch (error) {
    console.error('Помилка підключення до MongoDB:', error.message);
    console.log(' Повторна спроба через 5 секунд...');
    setTimeout(connectDB, timeout: 5000);
  }
};
```

Також можна додати обробники подій підключення, в них можна альтернативно вказати дії при різному стані підключення. Тобто можна не писати **setTimeout(connectDB, 5000)** у catch функції connectDB.

```
mongoose.connection.on( eventName: 'connected',  listener: () : any  => console.log('Mongoose підключено'));  new *

mongoose.connection.on( eventName: 'error',  listener: (err) : void  => {  new *
  console.error('Помилка з'єднання Mongoose:', err.message);
  console.log(' Повторна спроба через 5 секунд...');
  setTimeout(connectDB,  timeout: 5000);
});

mongoose.connection.on( eventName: 'disconnected',  listener: () : void  => {  new *
  console.warn( message: 'Mongoose відключено. Повторне підключення...');
  setTimeout(connectDB,  timeout: 5000);
});

mongoose.connection.on( eventName: 'reconnected',  listener: () : any  => console.log('Mongoose успішно перепідключено'));  new *
```

Також непогано обробити додатково сигнали SIGINT та SIGTERM:

- SIGINT - сигнал переривання (наприклад, CTRL+C у консолі);
- SIGTERM - сигнал завершення, викликається командою kill<PID>. Стандартний сигнал для коректного завершення процесу.

Ці сигнало варто обробляти, щоб дати можливість коректно завершити інші процеси у разі переривання або завершення процесу. У нашому випадку це закриття з'єднання з БД через **mongoose.connection.close(false)** або **mongoose.disconnect()**.

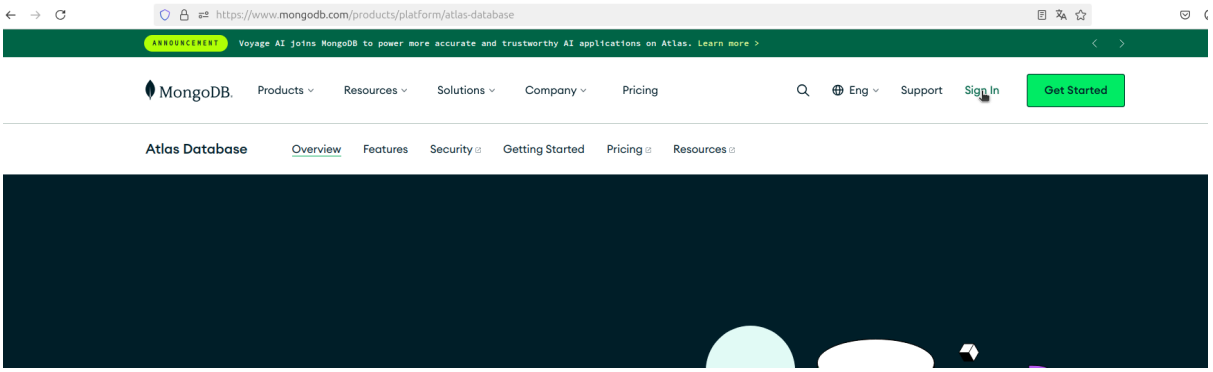
Також експортуємо нашу функцію.

```
const gracefulExit = async (signal) : Promise<void>  => {  Show usages  new *
  console.log(`Отримано сигнал ${signal}. Завершуємо роботу...`);
  try {
    await mongoose.connection.close( force: false);
    console.log('Відключено від MongoDB');
    process.exit( code: 0);
  } catch (error) {
    console.error('Помилка при відключенні:', error);
  }
};

// Обробка завершення процесу
process.on( eventName: 'SIGINT', gracefulExit);
process.on( eventName: 'SIGTERM', gracefulExit);

module.exports = connectDB;
```

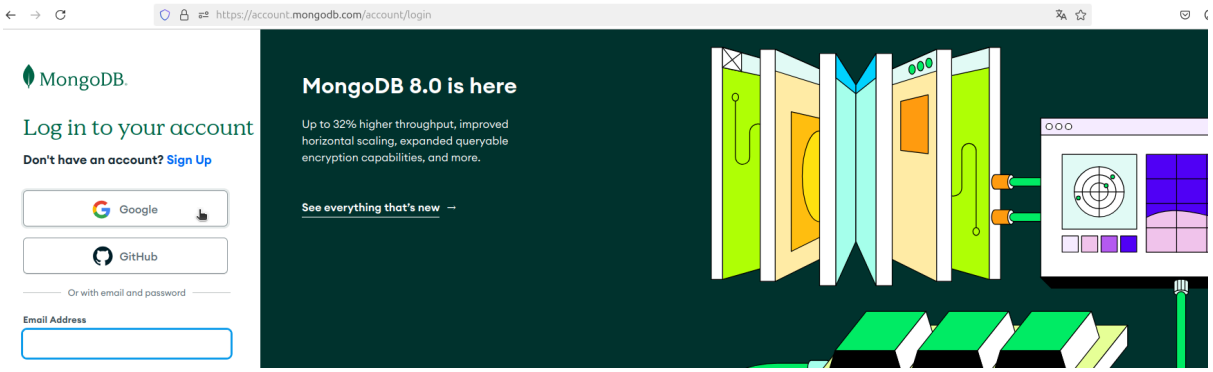
4) Наступним кроком нам необхідно отримати URI для підключення до MongoDB. Це можна зробити двома шляхами: розгорнути локально

та отримати локальний URI (просто на скучно) або відкрити безкоштовний кластер на MongoDB Atlas (також просто, але трохи цікавіше). Також відразу для роботи можна встановити MongoDB Compass (https://www.mongodb.com/docs/compass/current/install/)

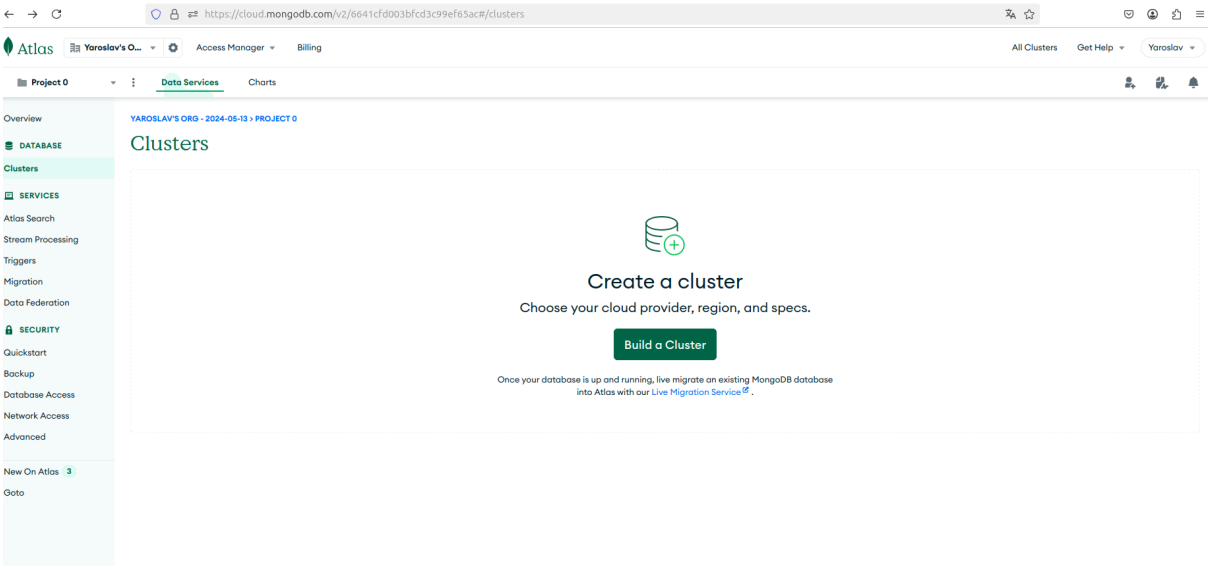
Розберемо процес створення кластера. Для цього переходимо на сторінку https://www.mongodb.com/products/platform/atlas-database



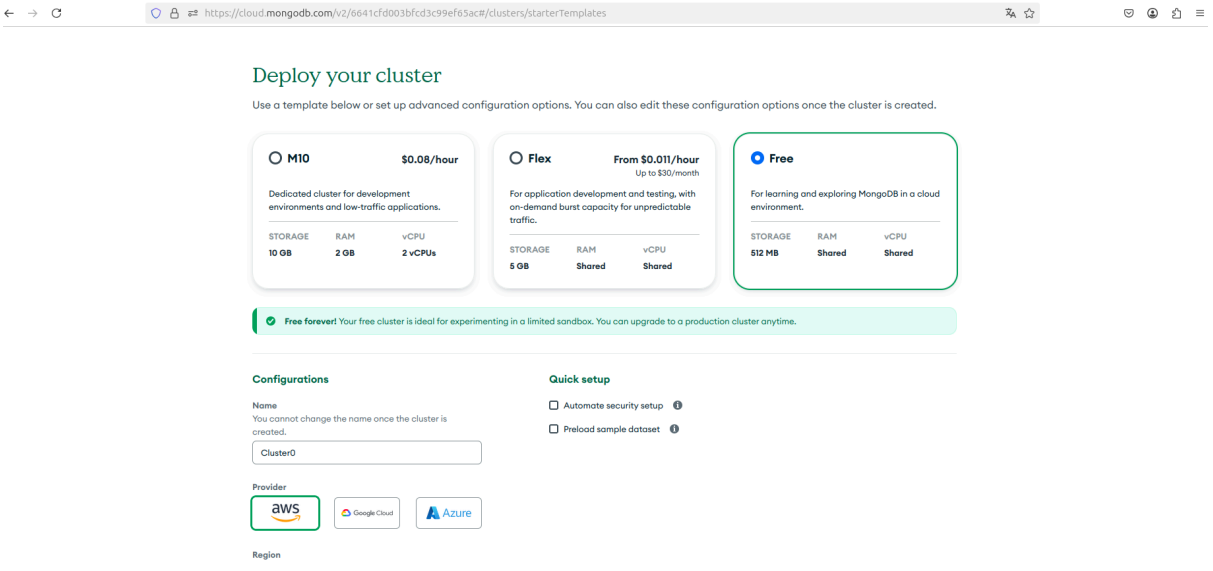
Реєструємося та авторизуємося

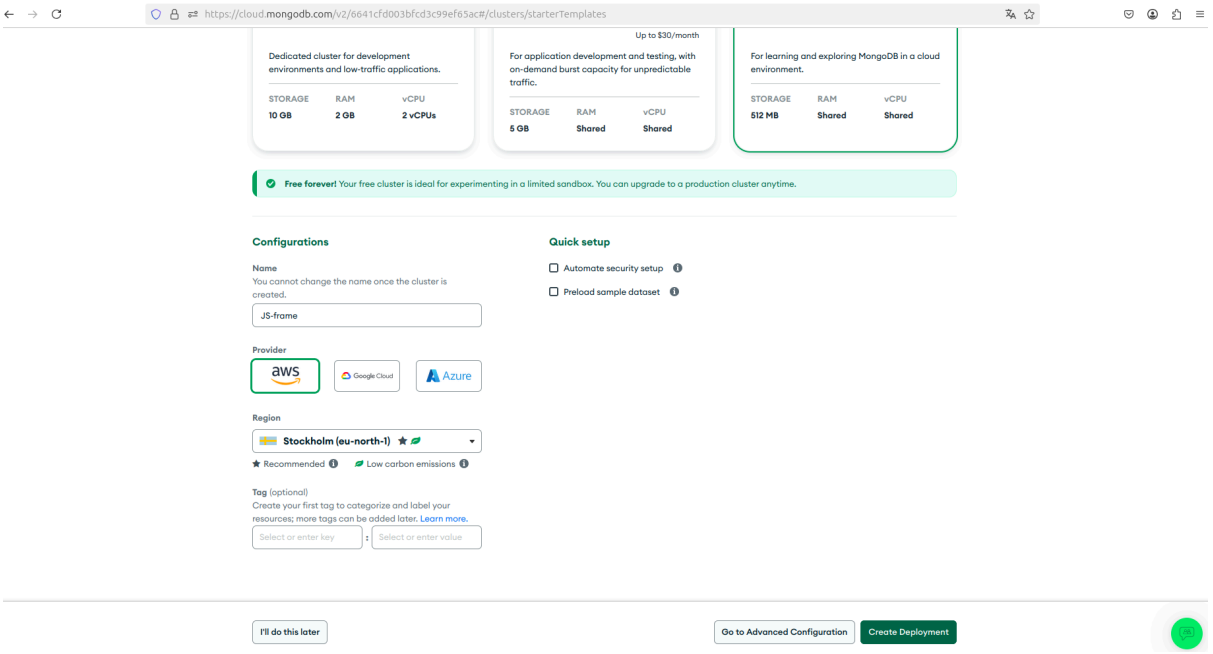


На сторінці Clusters натискаємо на кнопку “Build a Cluster”



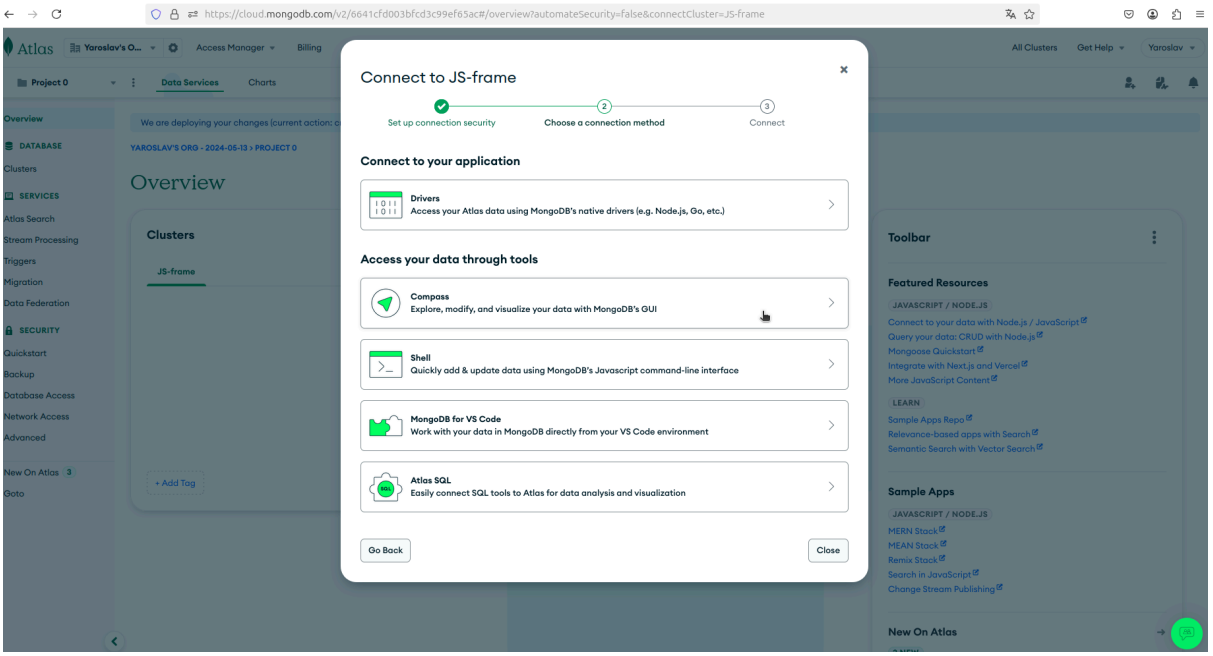
Далі обираємо безкоштовний план та налаштовуємо параметри конфігурації



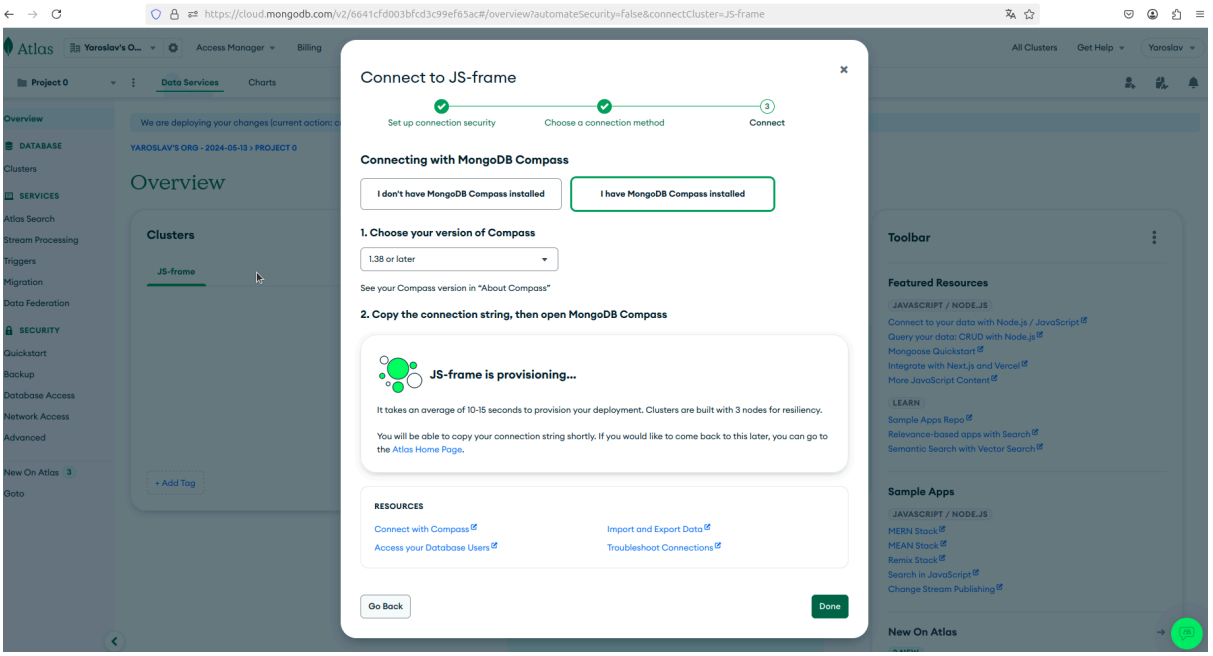


Після цього натискаємо на кнопку “Create Deployment” (зелена кнопка у правому нижньому кутку).

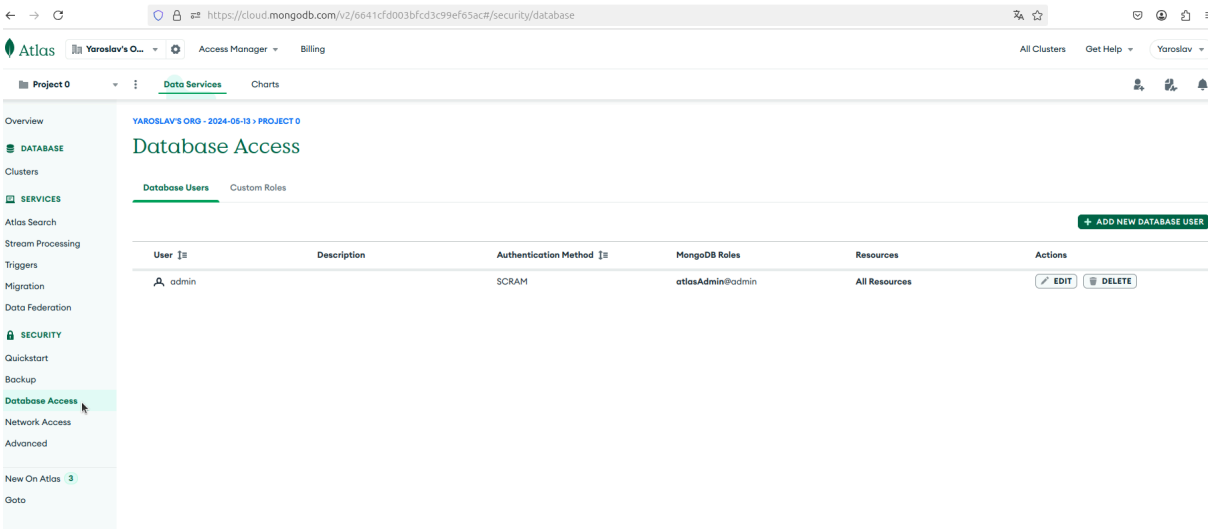
Далі налаштовуємо параметри підключення (якщо окно не з’явилося, то натисніть на кнопку “Connect”). Обираємо Compass (можна використовувати і інші шляхи).



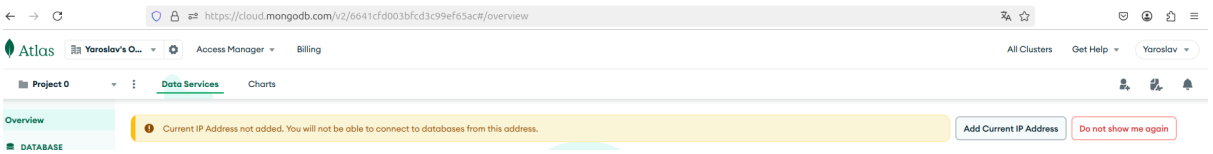
Обираємо, що вже встановлено Compass (якщо ви його встановили) та обираємо версію. Після цього натискаємо на кнопку “Done”.



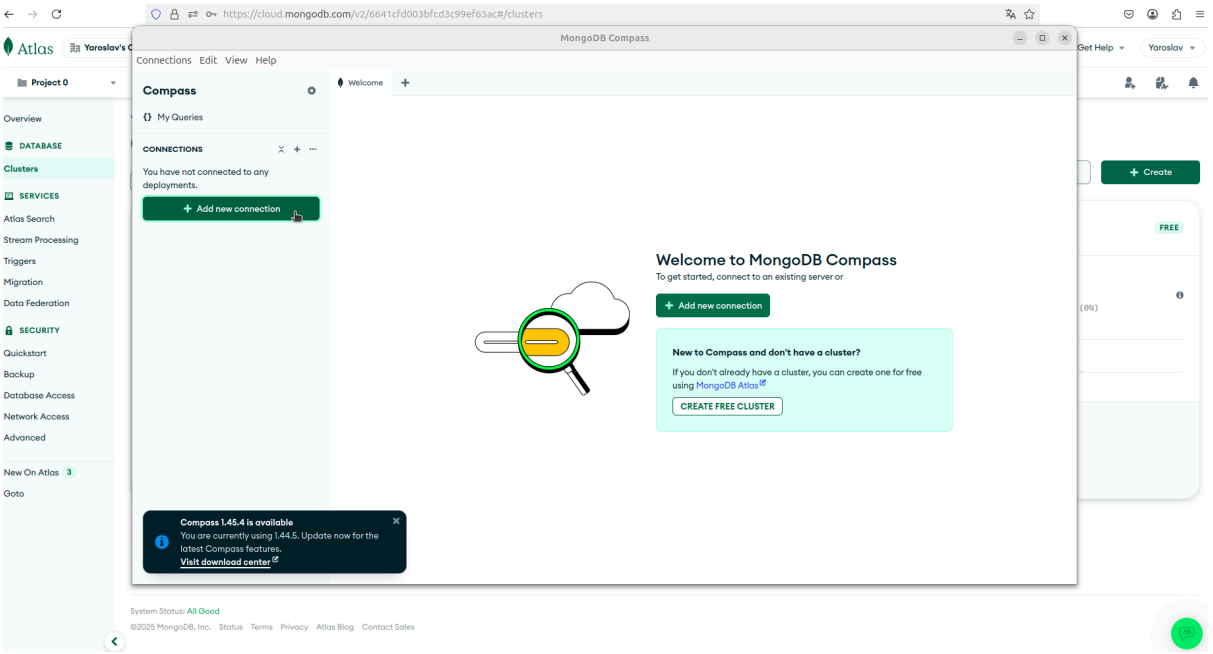
2 пункті повинна зв’явитися строка підключення, її потрібно буде скопіювати. Якщо нічого не відбувається, то натисніть Done та повторіть connect-кроки. Користувач за замовчуванням має логін та пароль admin. Якщо користувач не відображається, то його необхідно створити на сторінці Database Access



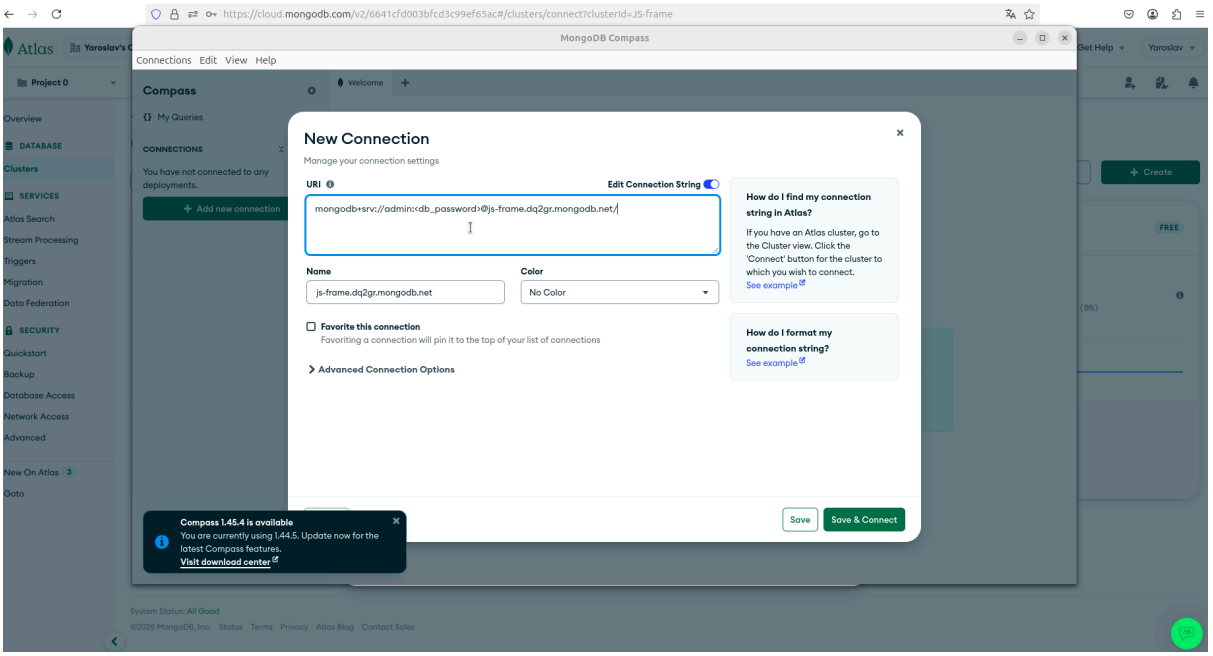
Також при вході (при першому точно) ви побачите, що ваш поточний ip не додано до дозволених. Необхідно натиснути на кнопку “Add Current IP Address”. Це потрібно буде робити після кожної зміни вашого ip.



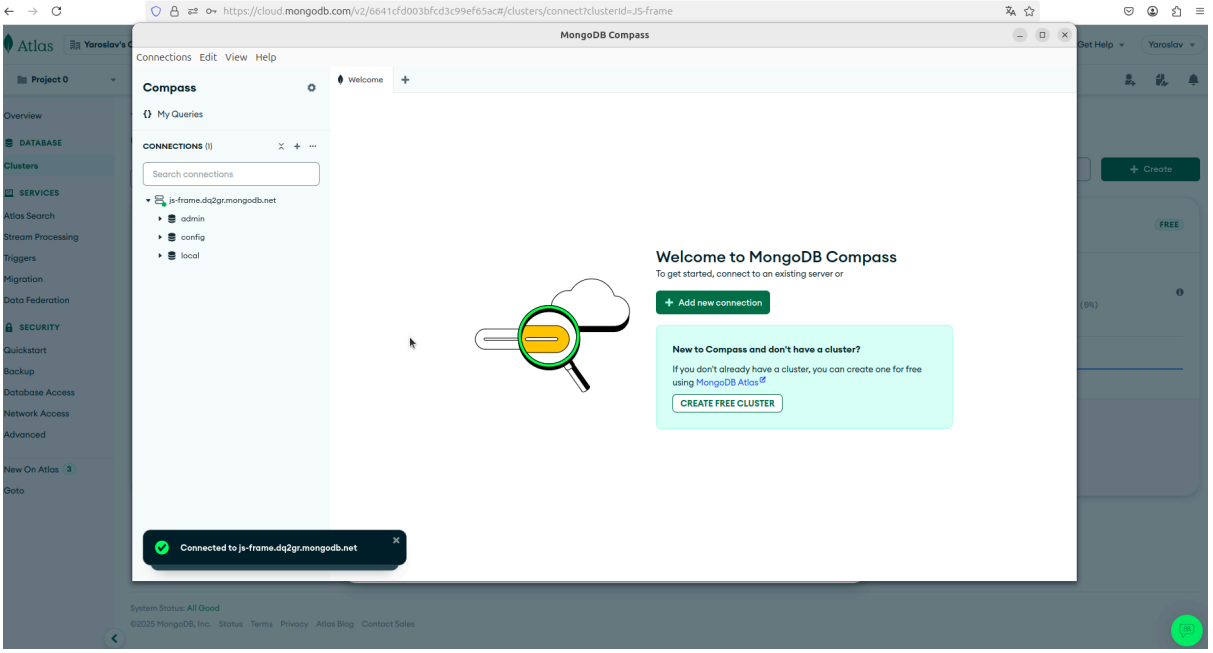
Далі створюємо нове підключення у Compass та створюємо БД за прикладом нижче



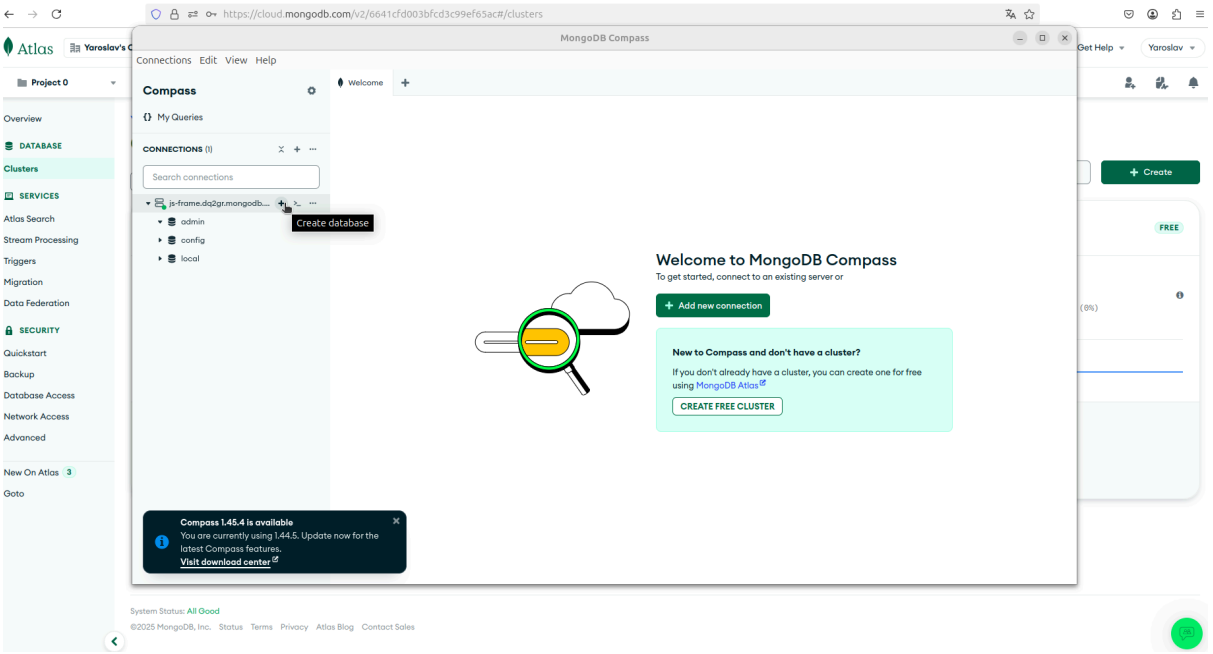
Ось тут знадобить строка підключення

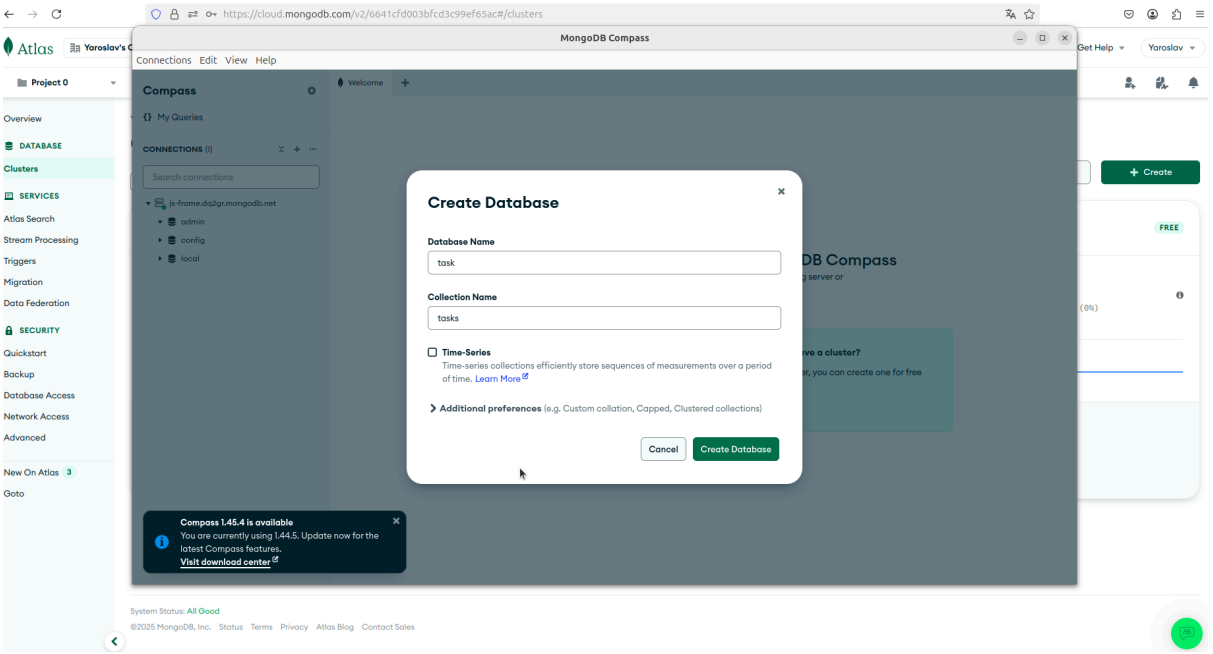


Підключаємося (можливо потрібно буде почекати декілька хвилин)



Створюємо нову БД



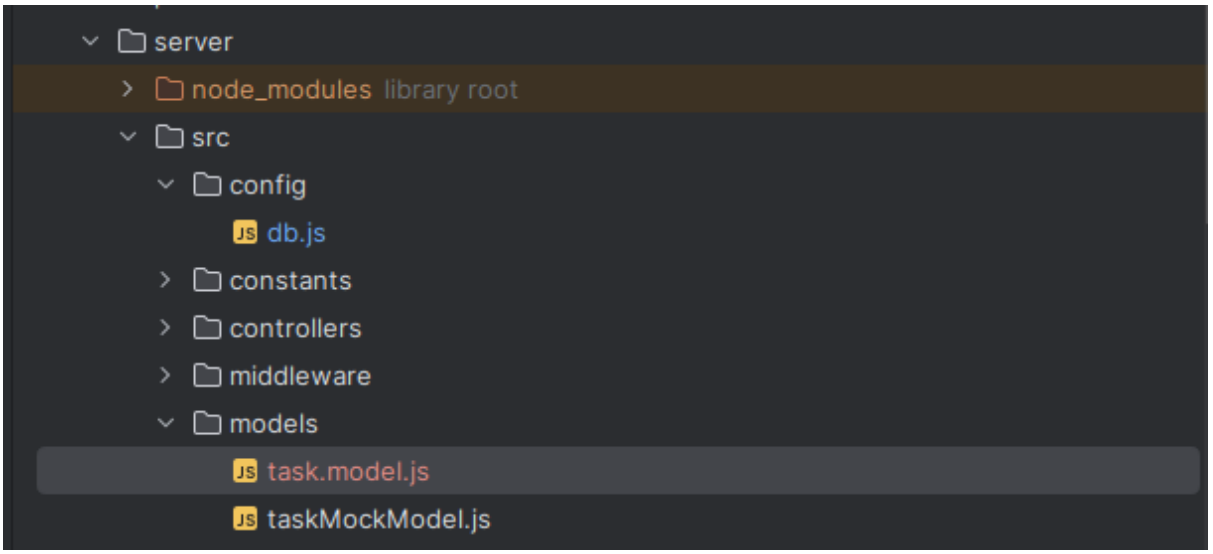


Також додаємо строку підключення до файлу оточення

```
PORT = 3100
MONGO_URI = 'Строка підключення/назва БД'
```

Не забуваємо вказати БД

5) Переходимо до створення моделі, для цього у теці server/src/models створимо файл task.model.js

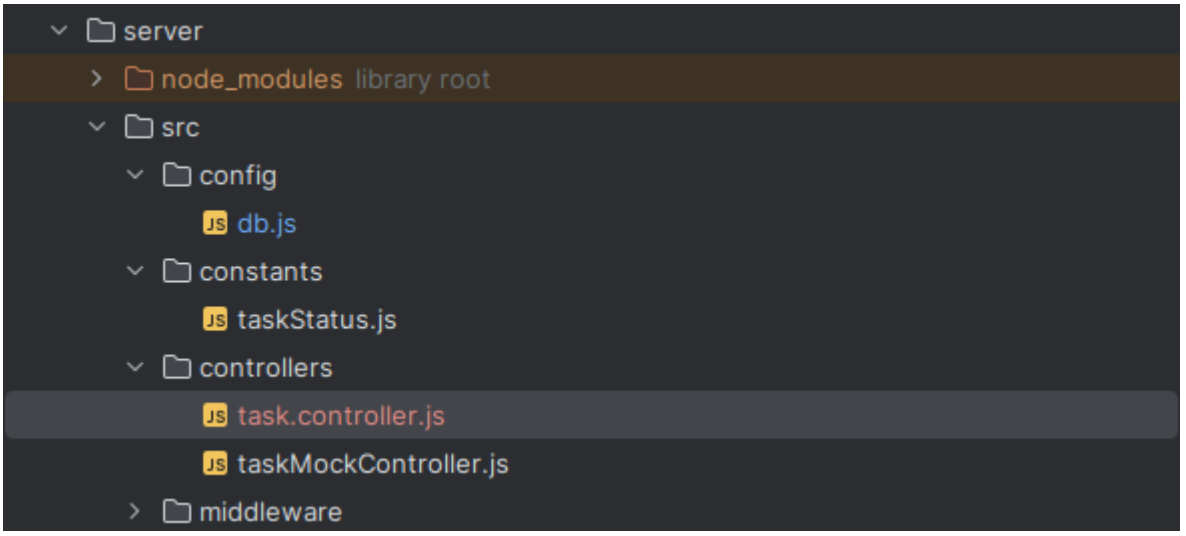


Пропишемо у файлі наступний код

```
const mongoose : … = require('mongoose');
const TaskStatus : {…} | {…} = require("../constants/taskStatus");

const TaskSchema : Schema<any, Model<…>, {…}, {…}, {…}, {…}, {…}> = new mongoose.Schema( definition: {
  title: { type: String, trim: true, required: true, minlength: 3, maxlength: 100 },
  description: { type: String, trim: true, default: "" },
  assignee: { type: String, required: true },
  dueDate: { type: Date, required: true },
  status: { type: String, enum: Object.values(TaskStatus), default: TaskStatus.TODO }
}, options: { timestamps: true });

const TaskModel : Model<…> = mongoose.model( name: "Task", TaskSchema);
module.exports = TaskModel;
```

Ми створюємо модель, яка буде пов’язана з колекцією tasks. Також налаштовуємо її схему, прописуючи правила валідації.

6) Наступним кроком створимо контролер, який буде обробляти наші HTTP-запити та взаємодіяти з моделлю. Для цього у теці server/src/controllers створимо файл task.controller.js



Пропишемо у ньому наступний код.

Підключаємо модель

Реалізуємо GET

```
const getTasks = async (req, res) : Promise<void> => {  Show usages
  try {
    let query : {} = {};

    if (req.query.status) {
      query.status = req.query.status;
    }

    let tasksQuery : QueryWithHelpers<GetLeanResultType<...> = TaskModel.find(query);

    const tasks = await tasksQuery;
    res.json(tasks);
  } catch (error) {
    res.status(500).json({ message: "Error retrieving tasks", error });
  }
};
```

Реалізуємо POST

```
const createTask = async (req, res) :Promise<...> => {  no usages
  try {
    const newTask :HydratedDocument<InferSchemaType<...>> = new TaskModel(req.body);
    await newTask.save();
    res.status(201).json(newTask);
  } catch (error) {
    if (error.name === "ValidationError") {
      const errors :any[] = Object.values(error.errors).map(err => err.message);
      return res.status(400).json({ message: "Validation error", errors });
    }
    if (error.code === 11000) {
      return res.status(400).json({ message: "Task already exists" });
    }
    res.status(400).json({ message: "Error creating task", error });
  }
};
```

Реалізуємо PUT

```
const updateTask = async (req, res) :Promise<...> => {  no usages
  try {
    const updatedTask :Query<...> = await TaskModel.findOneAndReplace(
      filter: {'_id': req.params.id},
      req.body,
      options: { new: true, runValidators: true }
    );

    if (!updatedTask) {
      return res.status(404).json({ message: "Task not found" });
    }

    res.json(updatedTask);
  } catch (error) {
    if (error.name === "ValidationError") {
      const errors :any[] = Object.values(error.errors).map(err => err.message);
      return res.status(400).json({ message: "Validation error", errors });
    }
    if (error.code === 11000) {
      return res.status(400).json({ message: "Task already exists" });
    }
    res.status(400).json({ message: "Error updating task", error });
  }
};
```

Реалізуємо PATCH

```
const patchTask = async (req, res) :Promise<...> => {  no usages
  try {
    const updatedTask :Query<...> = await TaskModel.findByIdAndUpdate(
      req.params.id,
      req.body,
      options: { new: true, runValidators: true }
    );

    if (!updatedTask) return res.status(404).json({ message: "Task not found" });

    res.json(updatedTask);
  } catch (error) {
    if (error.name === "ValidationError") {
      const errors :any[] = Object.values(error.errors).map(err => err.message);
      return res.status(400).json({ message: "Validation error", errors });
    }
    if (error.code === 11000) {
      return res.status(400).json({ message: "Task already exists" });
    }
    res.status(400).json({ message: "Error partially updating task", error });
  }
};
```

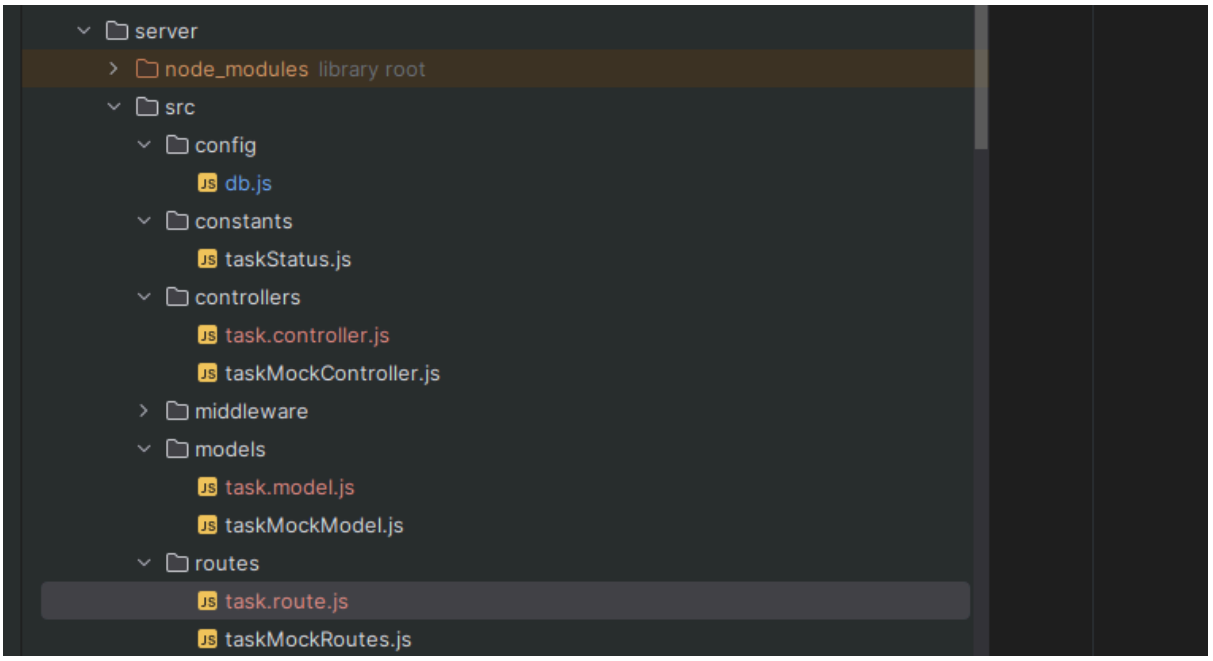
Реалізуємо DELETE

```
const deleteTask = async (req, res) :Promise<...> => {  no usages
  try {
    const deletedTask :Query<...> = await TaskModel.findByIdAndDelete(req.params.id);
    if (!deletedTask) return res.status(404).json({ message: "Task not found" });

    res.json({ message: "Task deleted" });
  } catch (error) {
    res.status(500).json({ message: "Error deleting task", error });
  }
};
```

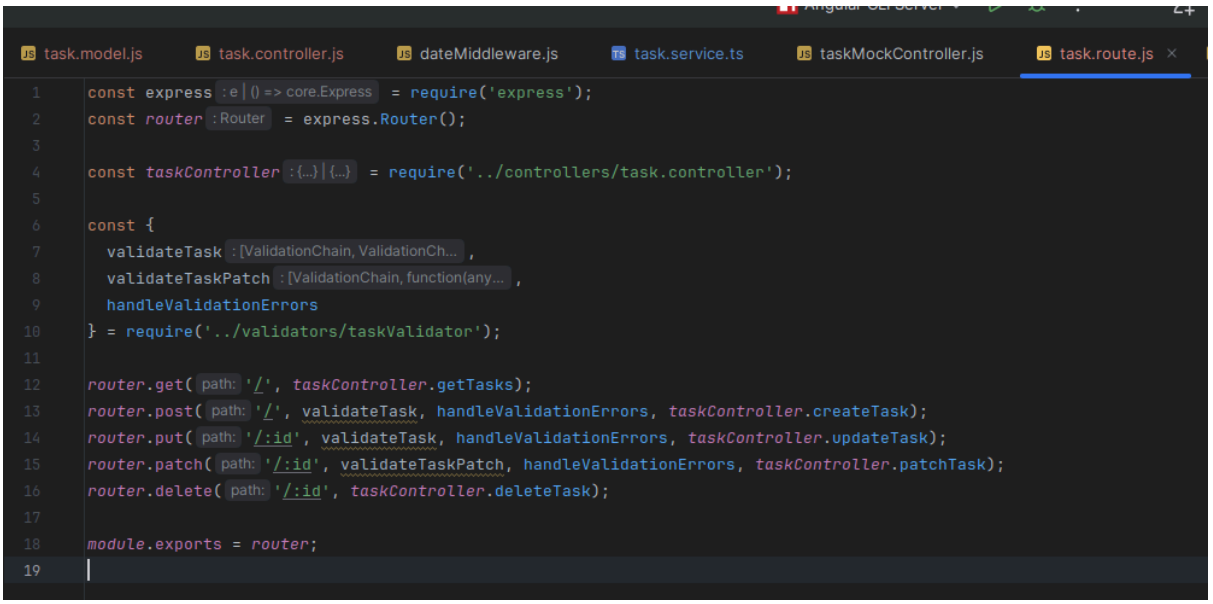
Експортуємо

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

7) Далі створюємо файл роутера, який пов’яже HTTP-запити та обробники. Для цього у теці server/src/routes створимо файл task.route.js



Підключимо до нього контролер та пропишемо обробку шляхів, а також валідацію з попередньої лабораторної.



8) Далі підключимо функцію з’єднання з БД до файлу app.js, а також прив’яжемо роутер до шляху api/v2/tasks

task.controller.js | taskMockController.js | task.route.js | taskValidator.js | taskMockRoutes.js | taskStatus.js

```
const express : e | () => core.Express = require('express');
const cors : <T=e.CorsRequest extends e.Cor... | e = require('cors');
const connectDB : function(): Promise<void> | {...} = require("./config/db");

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

connectDB();

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);
app.use('/api/v2/tasks', taskRoutes);

module.exports = app;
```

Запустимо сервер та перевіримо з’єднання

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

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

Server running on port 3100
Mongoose підключено
Підключено до MongoDB
```

Перед тим як перейти до тестування в Angular, необхідно внести певні зміни:

1) Angular очікує id, а MongoDB буде повертати _id. Вирішимо цю проблему, для цього нас знадобиться у моделі прописати set для toJson, для того щоб щоразу, коли викликаємо res.json(task) або JSON.stringify(task), зміни застосовувалися автоматично. Також у ньому можна прописати правила

конвертації дати, що надасть нам можливість відключити кастомні middleware, які ми писали у попередній лабораторній. Так як mongoose-схема при вказанні типу поля Date автоматично конвертую значення до цього типу (якщо це можливо, а в нашому випадку ми надсилаємо у загальноприйнятому форматі yyyy-mm-dd), то писати pre-хуки не потрібно

```
TaskSchema.set( key: "toJSON",  value: {
  transform: function (doc : HydratedDocument<DocType, TInstanceMethods, QueryHelpers> , ret : Record<string, any> ) : Record<string, any>  {
    ret.id = ret._id.toString();
    delete ret._id;

    if (ret.dueDate) {
      ret.dueDate = ret.dueDate.toISOString().split( separator: 'T')[0];
    }

    return ret;
  }
});
```

Тепер при зверненні до цих функцій, ми будемо отримувати id замість _id та коректну для клієнта дату.

2) Відключити непотрібні middleware у app.js

```
8   connectDB();
9
10  const {
11    parseDateMiddleware : parseDateMiddleware | function(any, any, any): void ,
12    formatResponseDateMiddleware : formatResponseDateMiddleware | function(any, any, any): void
13  } = require('./middleware/dateMiddleware');
14
15  const app : any | Express  = express();
16
17  app.use(cors());
18
19  app.use(express.json());
20  app.use(express.urlencoded( options: { extended: true }));
21
22  app.use(parseDateMiddleware);
23  app.use(formatResponseDateMiddleware);
```

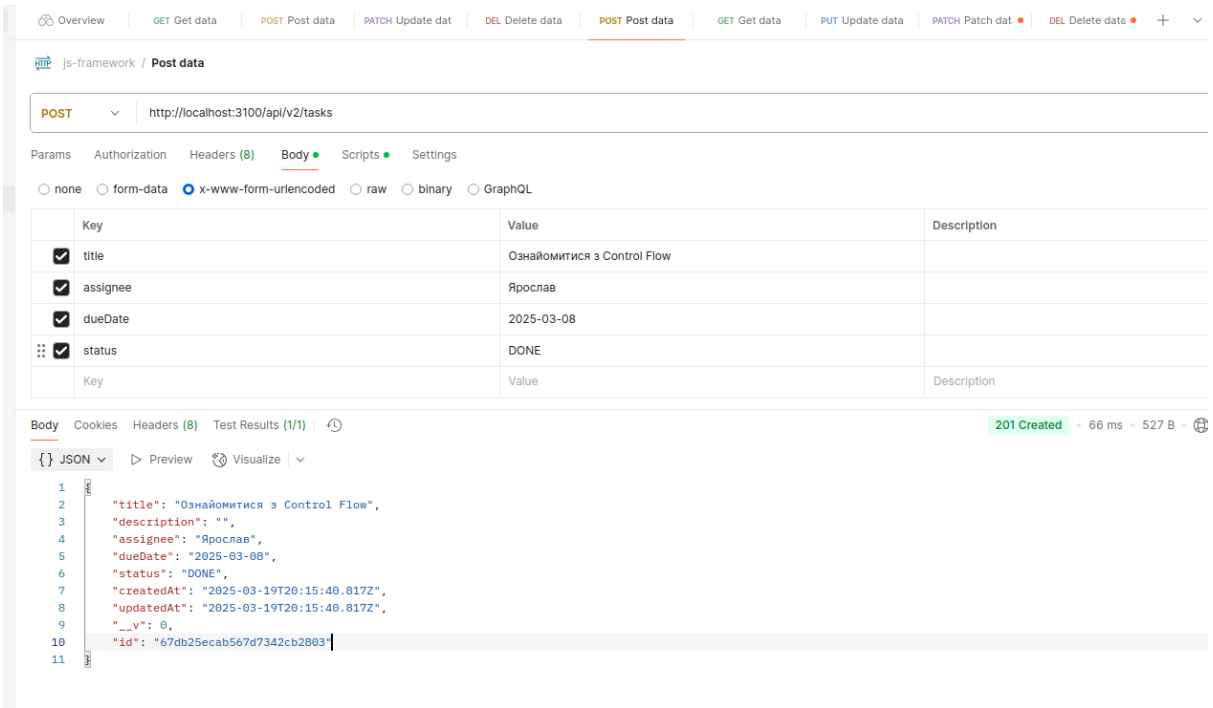
Можемо закоментувати або видалити

task.controller.js | taskMockController.js | task.route.js | taskValidator.js | taskMockRoutes.js | taskStatus.js | ap

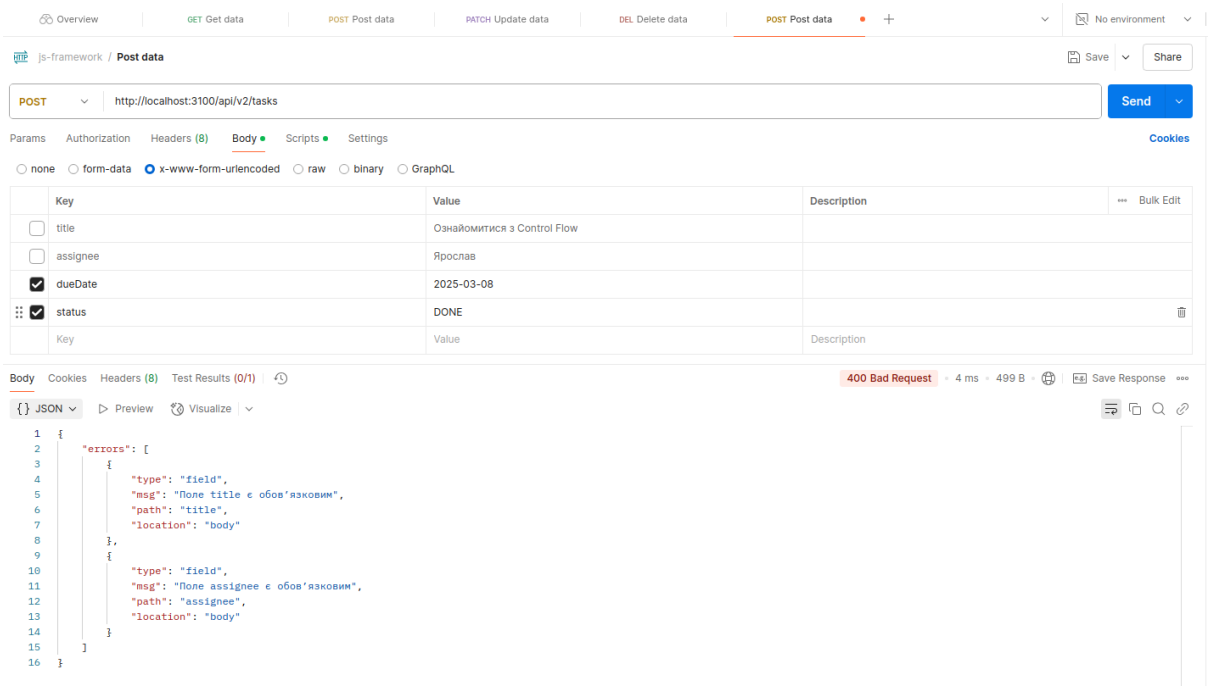
```
const express : e | () => core.Express  = require('express');
const cors : <T=e.CorsRequest extends e.Cor... | e  = require('cors');
const connectDB : function(): Promise<void> | {...}  = require("./config/db");

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

connectDB();

const app : any | Express  = express();

app.use(cors());

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


app.use('/api/v1/tasks', taskMockRoutes);
app.use('/api/v2/tasks', taskRoutes);

module.exports = app;
```

Далі необхідно перевірити, що наш сервер працює коректно та у відповідь надсилає саме ті дані, які ми очікуємо. Для цього використаємо Postman

Тестуємо створення нового завдання

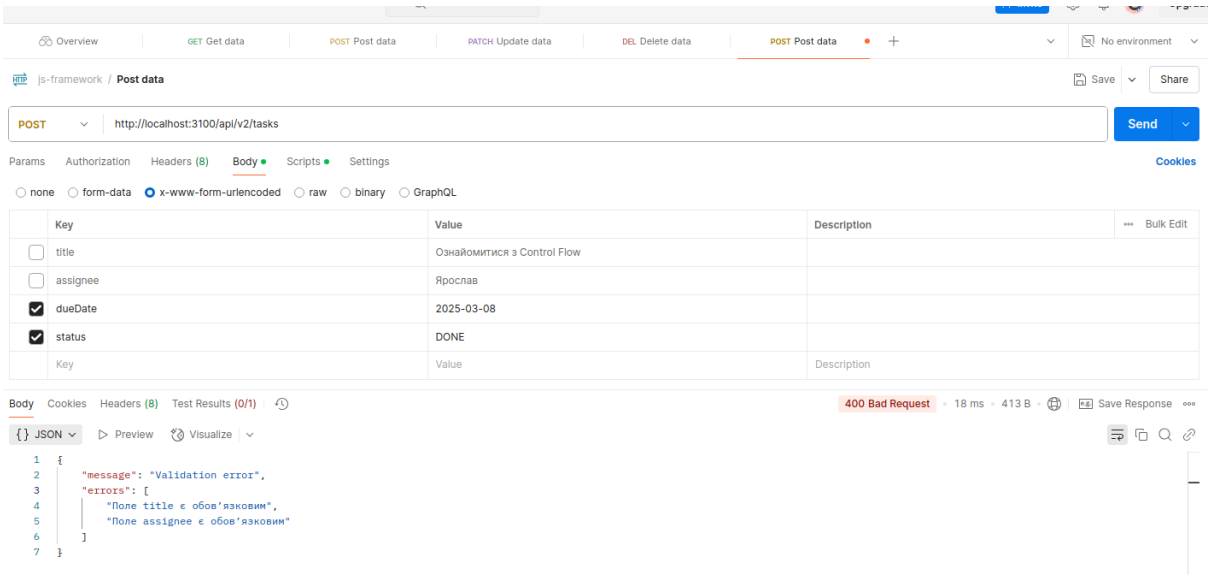


Перевіряємо валідацію



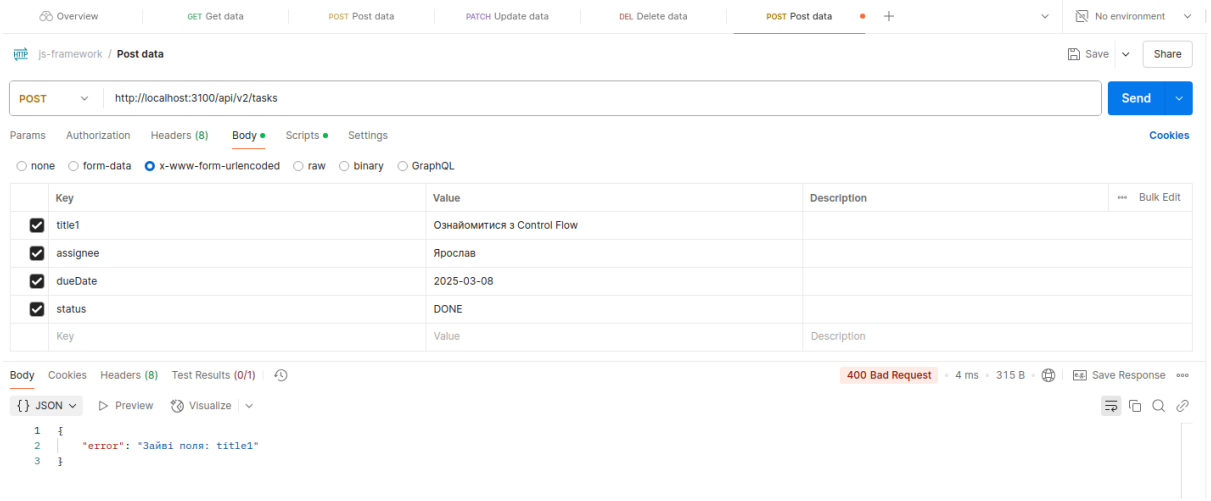
Можемо побачити, що відповідь помилки відрізняється від тієї, що ми вказували у контролері. Для цього поправимо обробник помилок валідації

```
// • Функція для перевірки помилок `express-validator`
const handleValidationErrors = (req, res, next) : any | undefined => {  Show usages  Kryvyi Yaroslav *
  const errors : Result<...> = validationResult(req);
  if (!errors.isEmpty()) {
    const valErrors : any[] = Object.values(errors.errors).map(err => err.msg);
    return res.status(400).json({ message: "Validation error", errors: valErrors });
  }
  next();
};
```

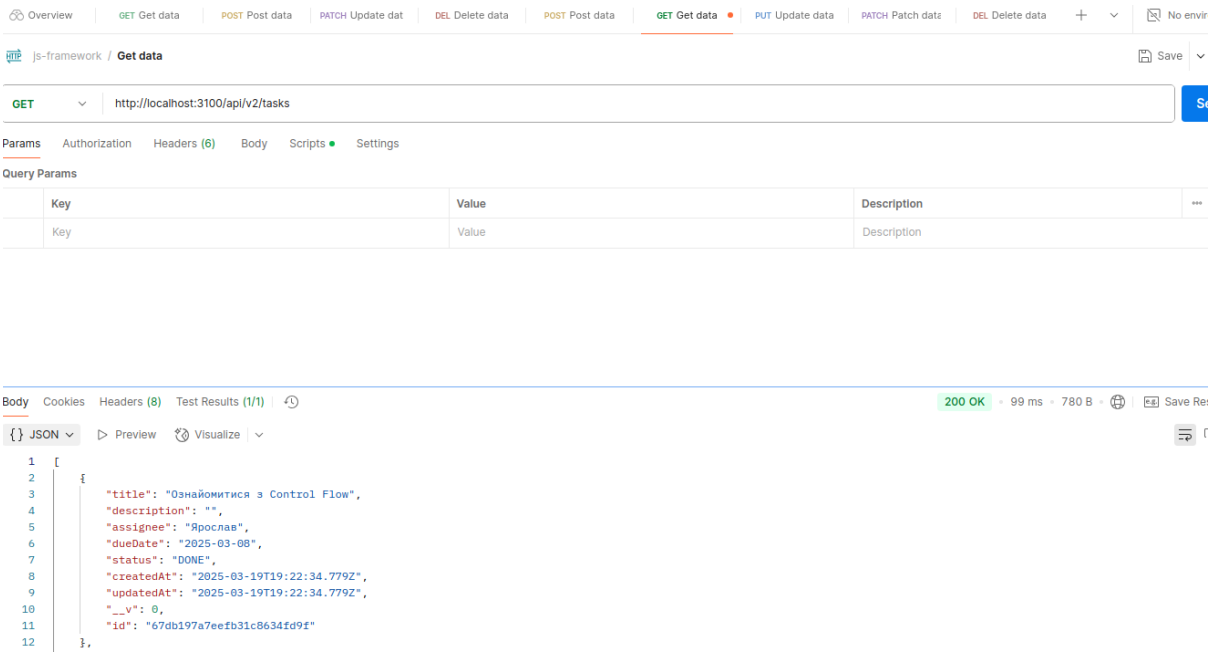


Після перевірки бачимо, що тепер у нас єдиний формат, що дозволить краще обробляти помилки клієнту

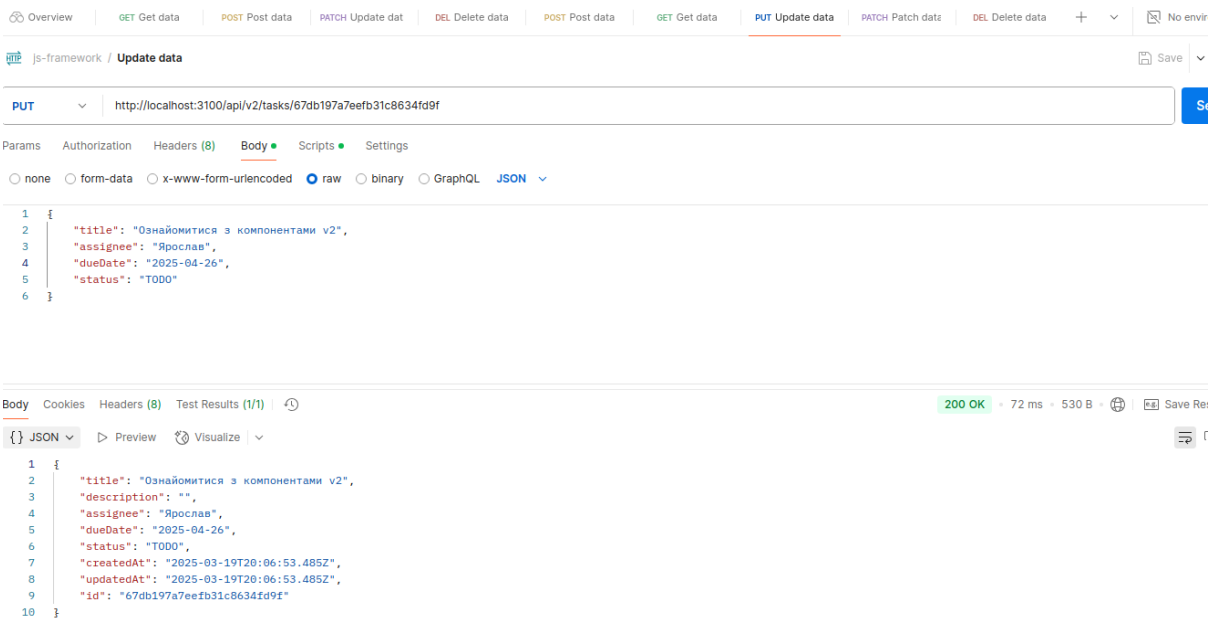
Також перевіряємо відповідь на зайве поле



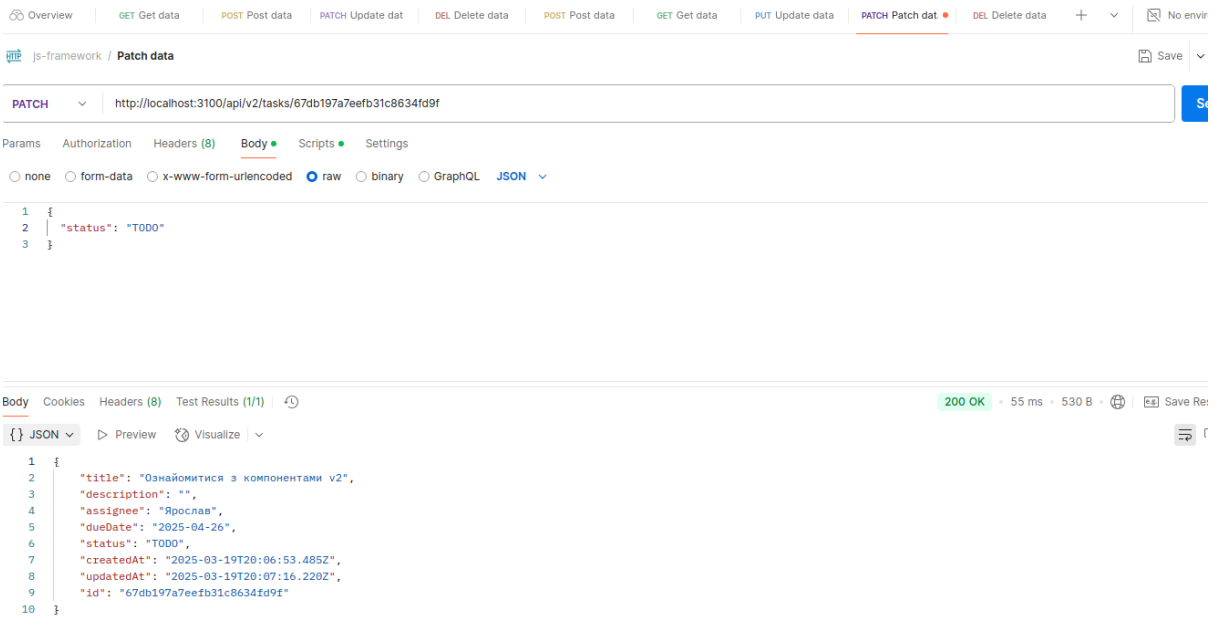
Тестуємо отримання всіх завдань



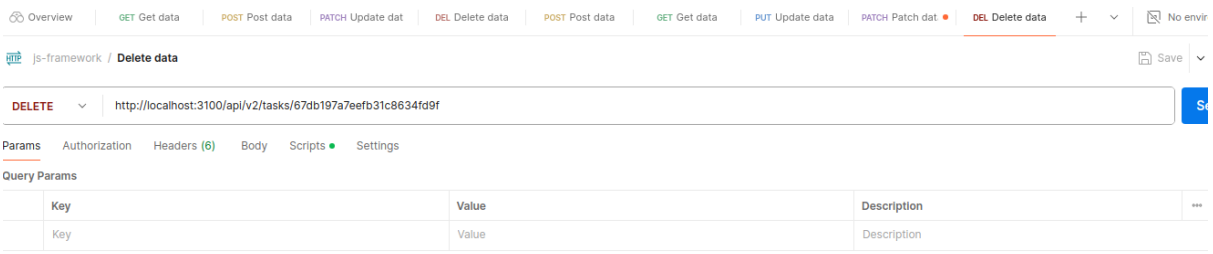
Тестуємо повне оновлення завдання

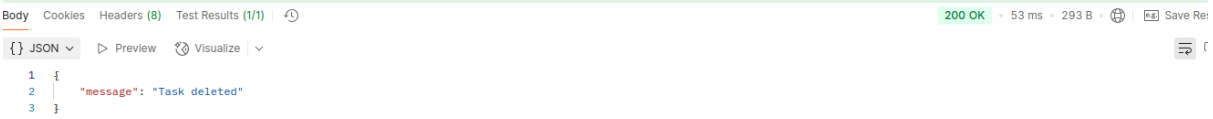


Тестуємо часткове оновлення



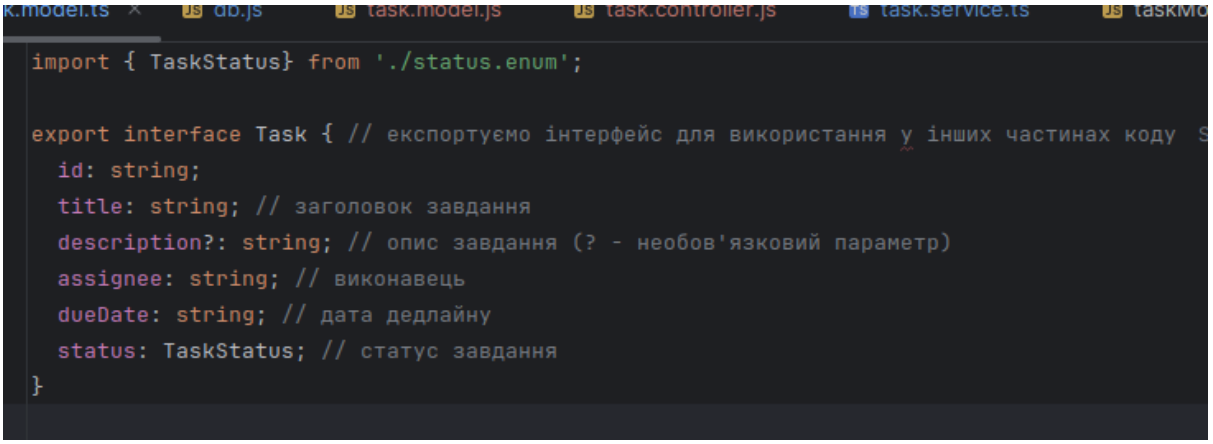
Тестуємо видалення





9) Переходимо до Angular. Перед тестуванням також є ряд проблем, які потрібно вирішити, а саме:
    a) Зробити id обов’язковим, перевести його у тип string та обробити його відсутність при створенні завдання

Оновлюємо інтерфейс



Змінюємо тип number на string у всіх функціях та емітерах, де ми його використовували

```
48      editTask(task: Task): void {  Show usages  Kryvyi Yaroslav
49        this.editingTask = {...task};
50      }
51
52      deleteTask(id: string): void {  Show usages  Kryvyi Yaroslav *
53        this.taskService.deleteTask(id).subscribe({
54          next: () :void  =>  this.loadTasks(this.selectedStatus),
55          error: error :any  => console.log(error),
56        });
57      }
58
59      onSelected(event: Event): void {  Show usages  Kryvyi Yaroslav
```

```
13
14      @Input() task!: Task; // отримуємо об'єкт завдання
15      @Output() taskDeleted: EventEmitter<string>  = new EventEmitter<string>(); // поді
16      @Output() taskEdited: EventEmitter<Task>  = new EventEmitter<Task>(); // подія для
17
18      constructor(private taskService: TaskService) {  no usages  Kryvyi Yaroslav
19      }
20
21      deleteTask(id: string): void {  Show usages  Kryvyi Yaroslav *
22        if (!id) return; // Якщо id відсутній - виходимо
23        this.taskDeleted.emit(id); // Викликаємо подію при натисканні кнопки "Видалити"
24      }
```

```
19
20        if (status) params = params.set('status', status);
21
22        return this.http.get<Task[]>(`${this.config.apiUrl}/v2/tasks`, {params: params});
23      }
24
25      createTask(newTask: Omit<Task, "id">): Observable<Task> {  Show usages  new *
26        return this.http.post<Task>(`${this.config.apiUrl}/v2/tasks`, newTask);
27      }
28
29      updateTask(id: string, updateTask: Task): Observable<Task> {  Show usages  Kryvyi Yaroslav *
30        const { id: _, ...update } = updateTask;
31        return this.http.put<Task>(`${this.config.apiUrl}/v2/tasks/${id}`, update);
32      }
33
34      patchTask(id: string, updateTask: Partial<Task>): Observable<Task> {  Show usages  new *
35        return this.http.patch<Task>(`${this.config.apiUrl}/v2/tasks/${id}`, updateTask);
36      }
37
38      deleteTask(id: string): Observable<void> {  Show usages  new *
39        return this.http.delete<void>(`${this.config.apiUrl}/v2/tasks/${id}`);
40      }
41
42    }
```

У функції createTask замість використання newTask: Task, використаємо newTask: Omit<Task, "id">. Це дозволить створити новий інтерфейс на основі існуючого але без id. Так як зараз у нас один обробник

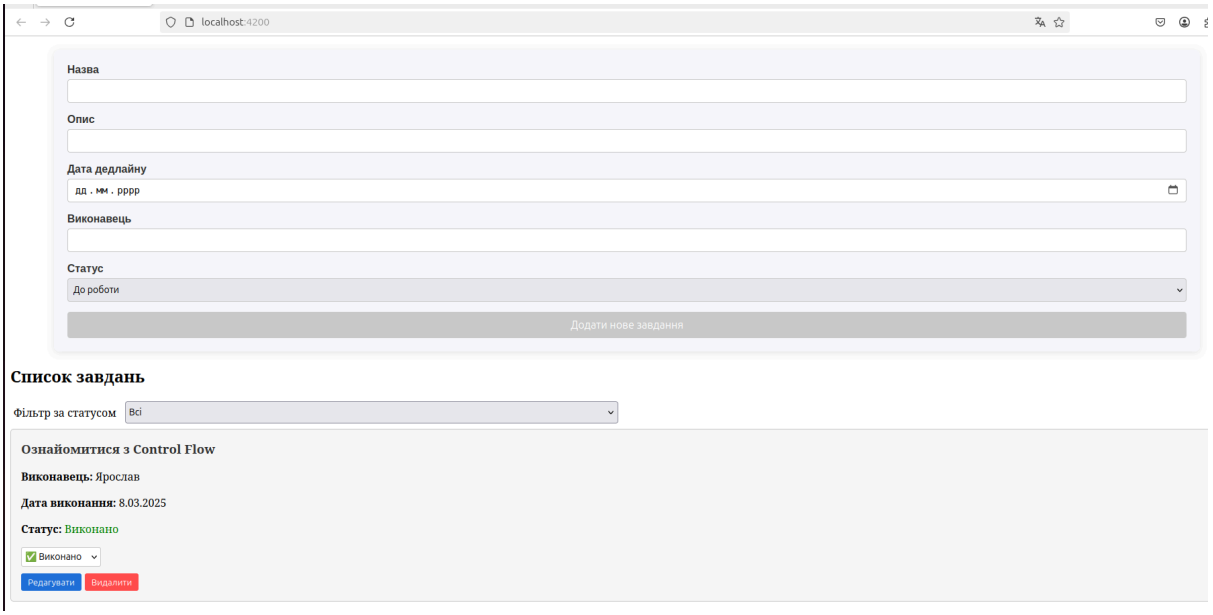
оновлення та додавання завдання, то проблем навіть без Omit у нас не буде. Але при розділені або зміні логіки - застрахує від помилки.

```
createTask(newTask: Omit<Task, "id">): Observable<Task> {  Show usages  new *
  return this.http.post<Task>(`${this.config.apiUrl}/v2/tasks`, newTask);
}
```

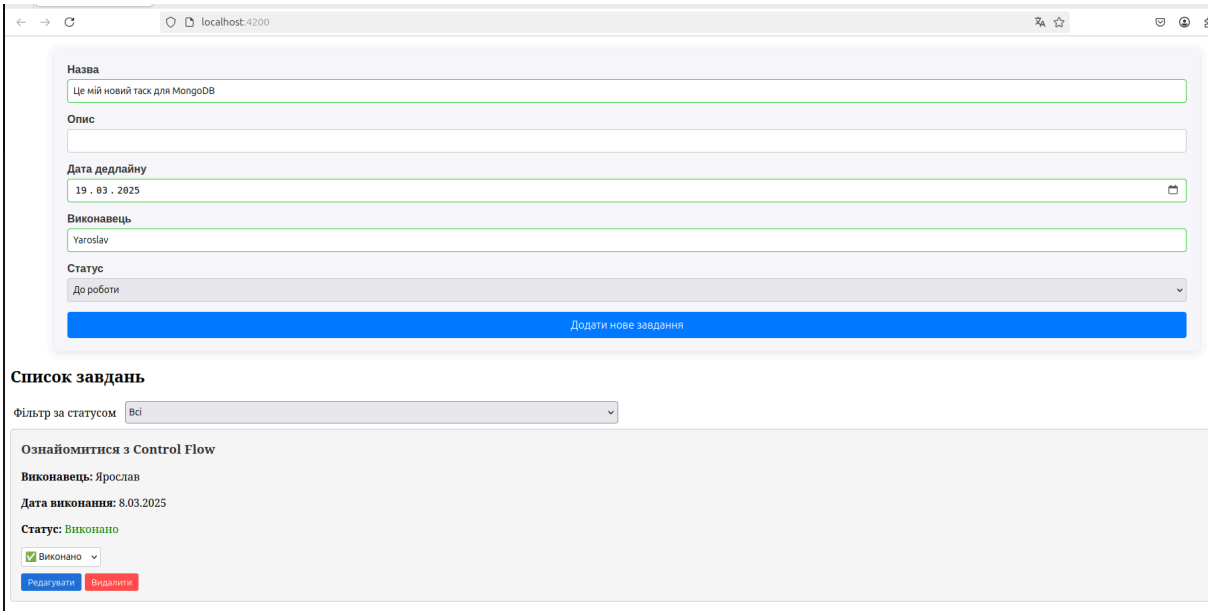
Також не забуваємо змінити версію роутів на v2.

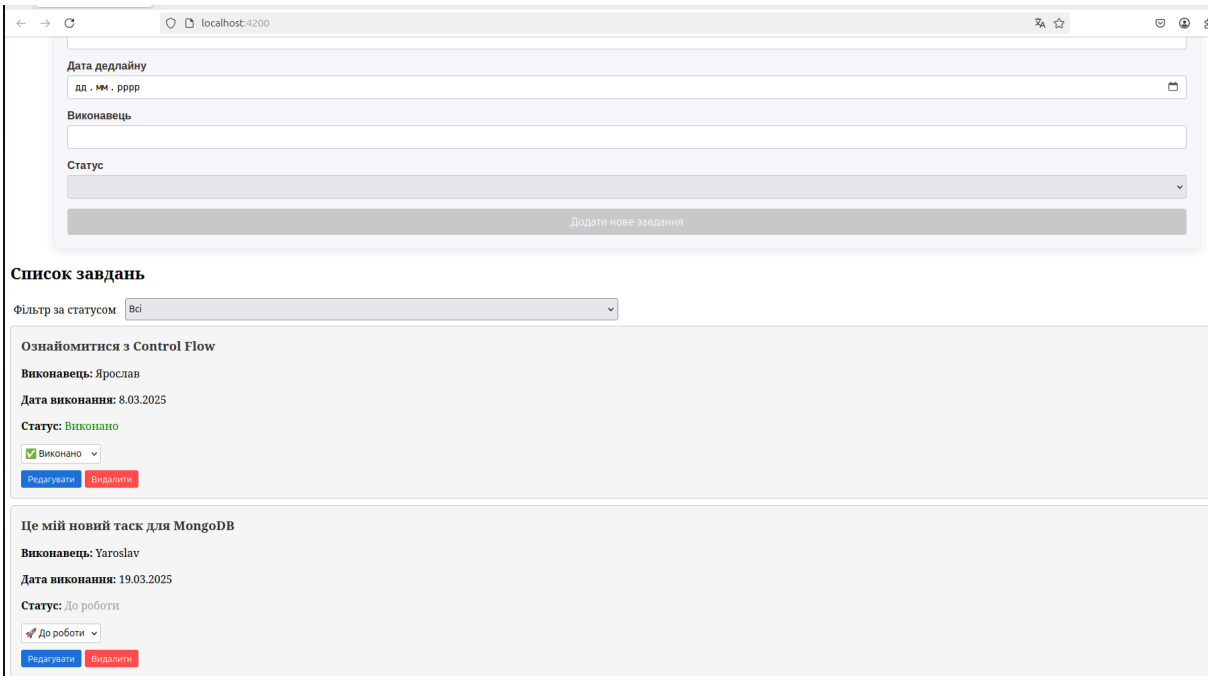
Перевіряємо роботу

Відображення

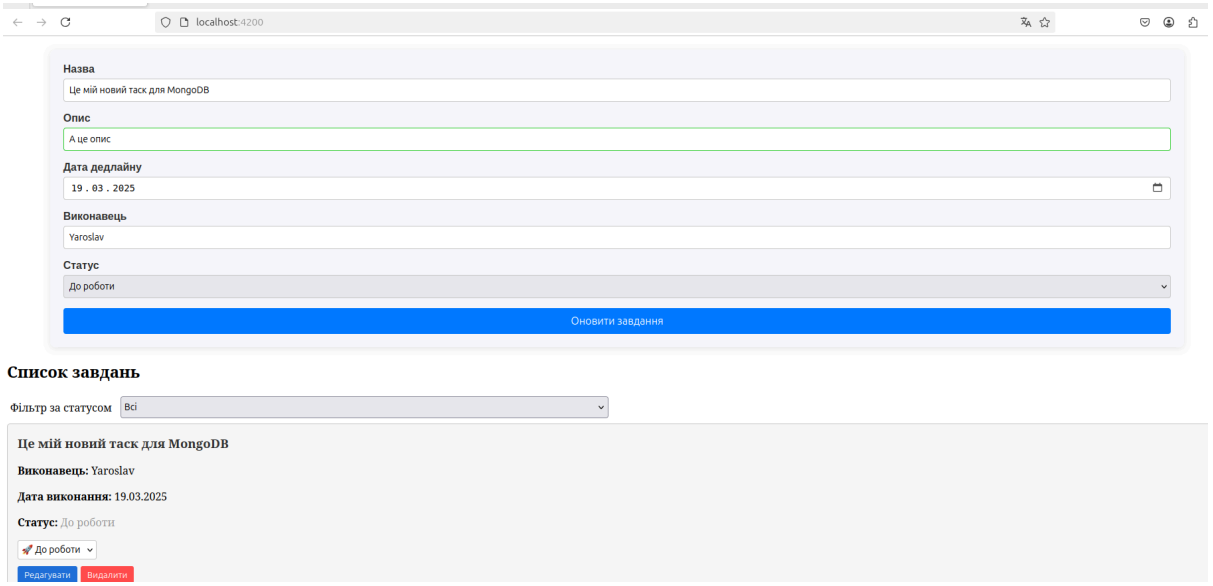


Створення

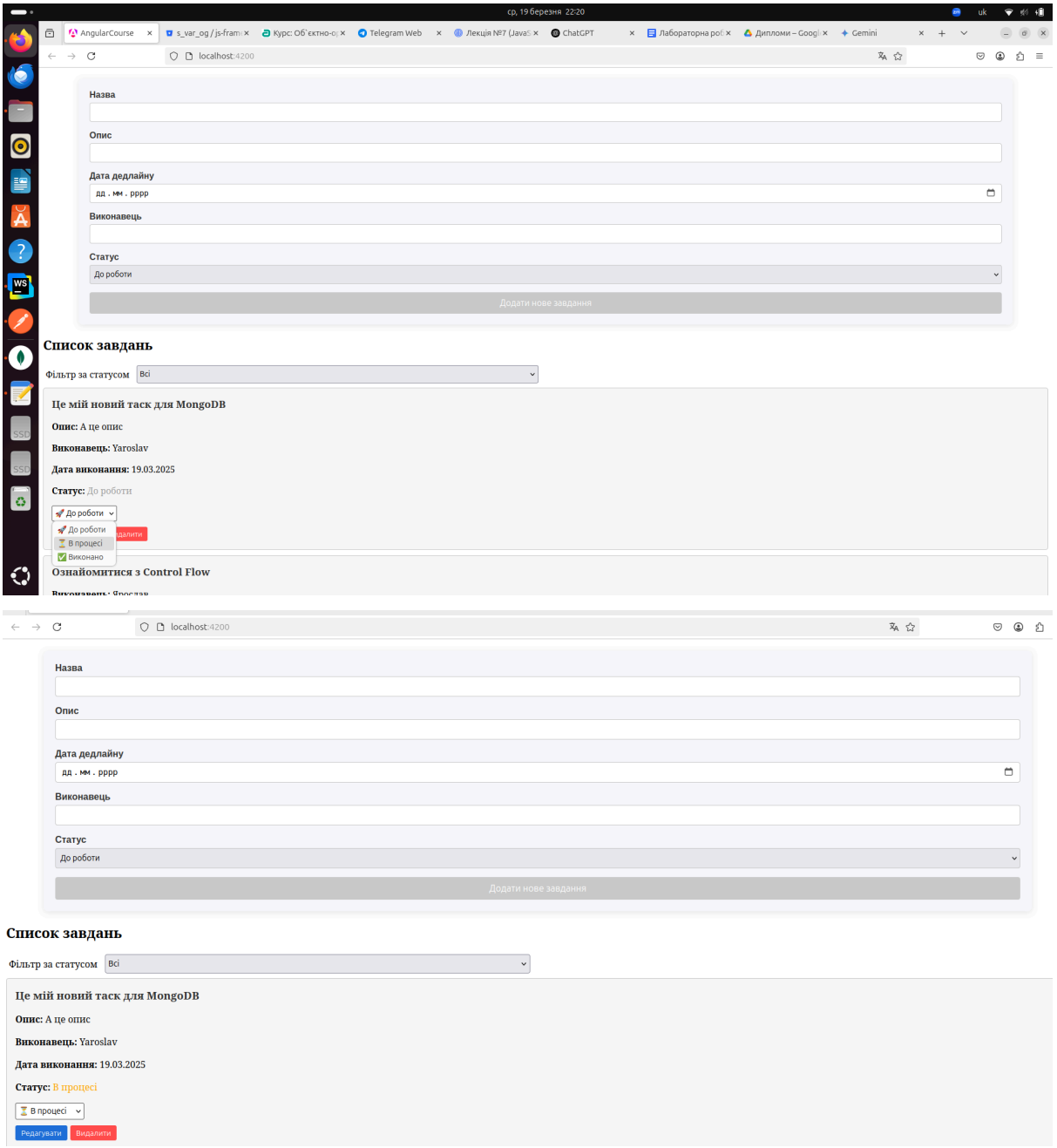

localhost:4200
Назва
Це мій новий таск для MongoDB
Опис
Дата дедлайну
19 . 03 . 2025
Виконавець
Yaroslav
Статус
До роботи
Додати нове завдання
Список завдань
Фільтр за статусом
Всі
Ознайомитися з Control Flow
Виконавець: Ярослав
Дата виконання: 8.03.2025
Статус: Виконано
✅ Виконано
Редагувати
Видалити


localhost:4200
Дата дедлайну
дд . мм . рррр
Виконавець
Статус
Додати нове завдання
Список завдань
Фільтр за статусом
Всі
Ознайомитися з Control Flow
Виконавець: Ярослав
Дата виконання: 8.03.2025
Статус: Виконано
✅ Виконано
Редагувати
Видалити
Це мій новий таск для MongoDB
Виконавець: Yaroslav
Дата виконання: 19.03.2025
Статус: До роботи
🚀 До роботи
Редагувати
Видалити

Оновлення

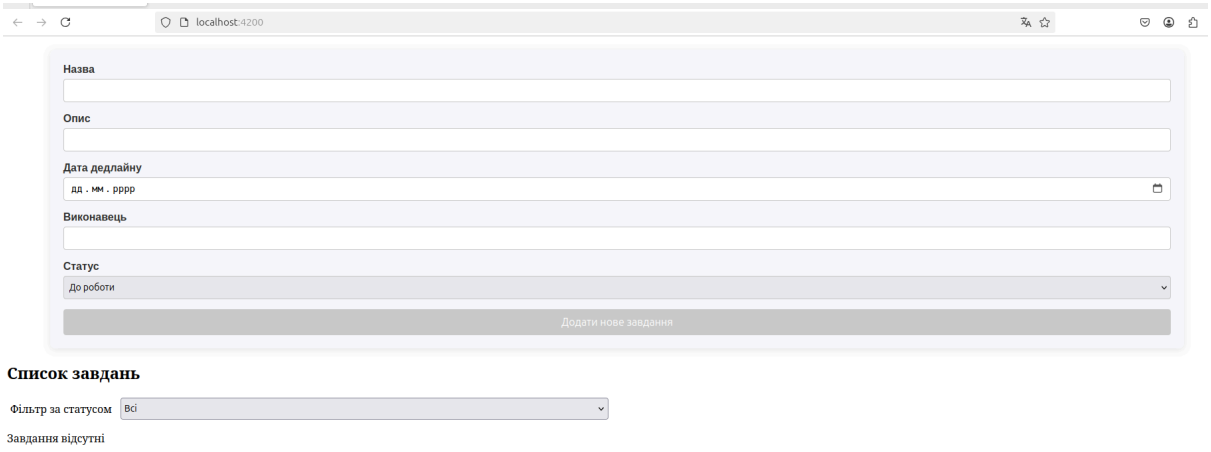


Часткове оновлення


ср, 19 березня 22:20
localhost:4200
Назва
Опис
Дата дедлайну
дд . мм . рррр
Виконавець
Статус
До роботи
Додати нове завдання
Список завдань
Фільтр за статусом
Всі
Це мій новий таск для MongoDB
Опис: А це опис
Виконавець: Yaroslav
Дата виконання: 19.03.2025
Статус: До роботи
До роботи
В процесі
Виконано
Ознайомитися з Control Flow

localhost:4200
Назва
Опис
Дата дедлайну
дд . мм . рррр
Виконавець
Статус
До роботи
Додати нове завдання
Список завдань
Фільтр за статусом
Всі
Це мій новий таск для MongoDB
Опис: А це опис
Виконавець: Yaroslav
Дата виконання: 19.03.2025
Статус: В процесі
В процесі
Редагувати
Видалити

Видалення



Контрольні питання

1) Які middleware представлені у mongoose?
2) Які обробники подій підключення до БД існують в mongoose?
3) Для чого використовується схема при створенні mongoose-моделі?
4) Як формується назва колекції?
5) Чим **.save()** відрізняється від **.create()**
6) Як створити кастомний валідатор у mongoose-схемі? Наведіть типовий синтаксис. Чи можна додати декілька для одного поля?
7) Який error.name повертається, якщо валідація не пройдена?