Лабораторна робота №3

Mongoose. Rest API

Мета: ознайомитись з Mongoose та Rest API

Завдання

Розробити RESTful API-додаток з обробкою таких запитів для даних users і tasks:

- GET (/users, /tasks)
- GET (/users/:id, /tasks/:id)
- POST (/users, /tasks)
- PATCH (/users/:id, /tasks/:id)
- DELETE (/users/:id, /tasks/:id)
- DELETE (/users, /tasks)

Встановити

Сервер БД MongoDB або хмарний варіант

Postman – створення і виконання запитів

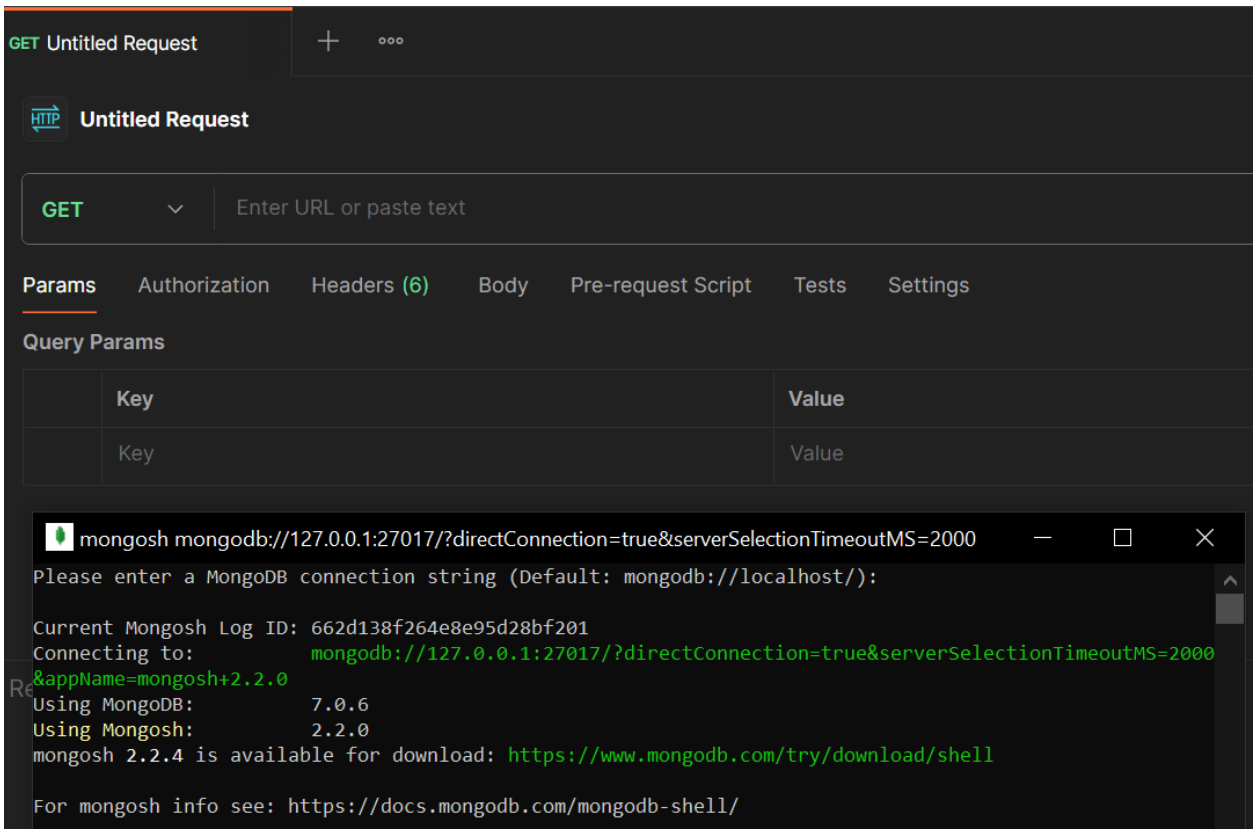
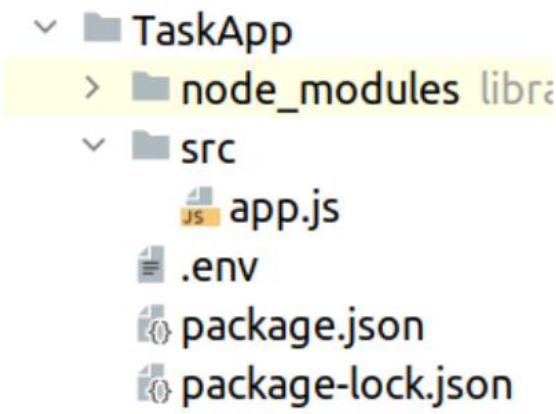


Рис. 1. Результат встановлення

Завдання 1. Створіть проект TaskApp

- Встановіть модулі *mongoose*, *validator*, *express*, *dotenv*
- Створіть папку *src* та виконавчий файл *src/app.js*



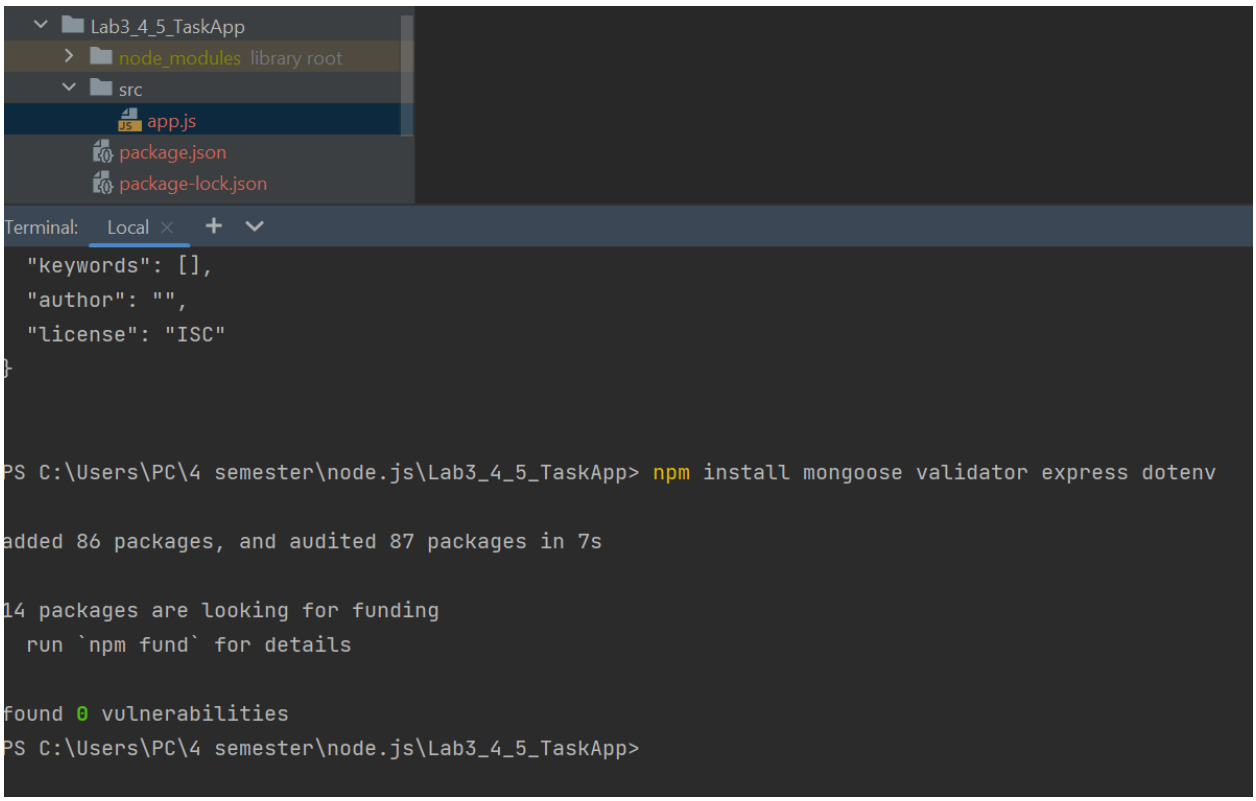


Рис. 2. Результат

Завдання 2. Підключимось до бази даних та створимо модель User

- Рядок з'єднання винесіть у файл .env в змінну MONGO_URL

```
//Створюємо модель
const User = mongoose.model('User', {
    name: {type: String},
    age: {type: Number}
});
```

Створимо екземпляр моделі User

```
//Створюємо екземпляр моделі
const user = new User({name: 'Alex', age: 34});

//Зберігаємо екземпляр в базу даних та виводимо повідомлення
user.save().then(() => {
    console.log(user);
}).catch((error) => {
    console.log(error);
});
```

Протестуйте додаток. Перевірте валідацію полів при введенні значень невірних типів

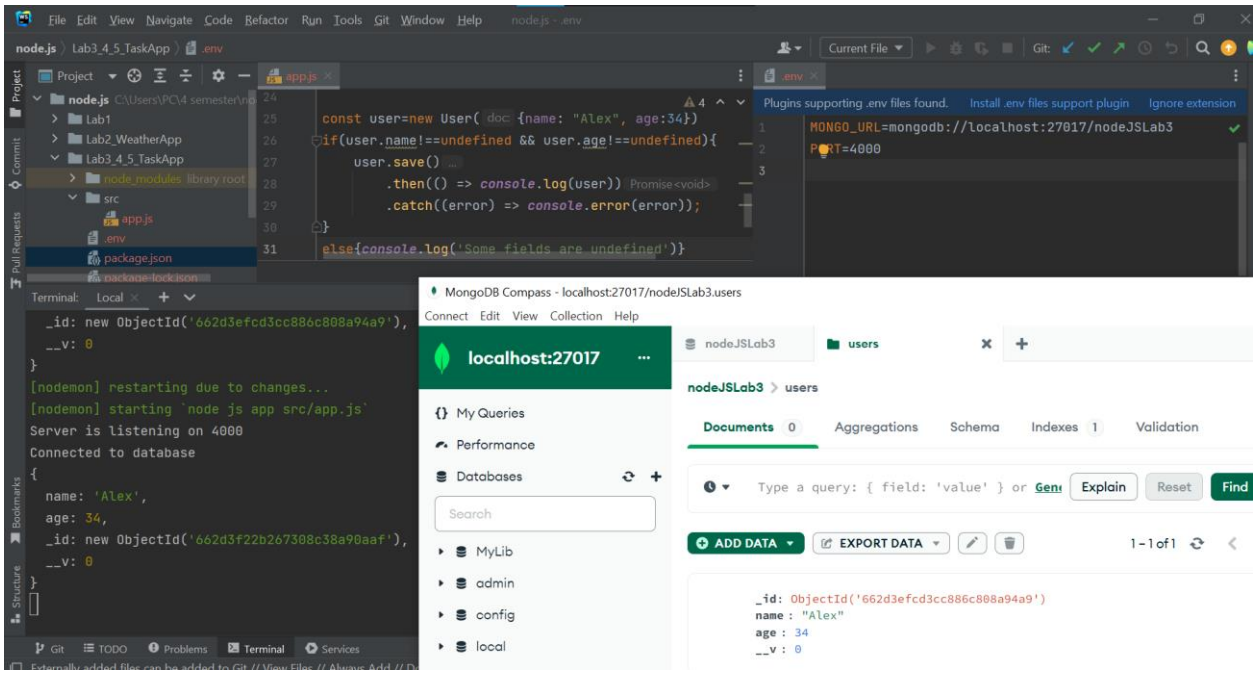


Рис. 3. Результат

Валідація даних

- Для поля *name* моделі *User* встановіть обов'язкову наявність непорожнього значення:

```
name: {
    type: String,
    required: true
},
```

- Перевірте роботу валідатора даних при створенні екземпляра з порожнім полем *name*

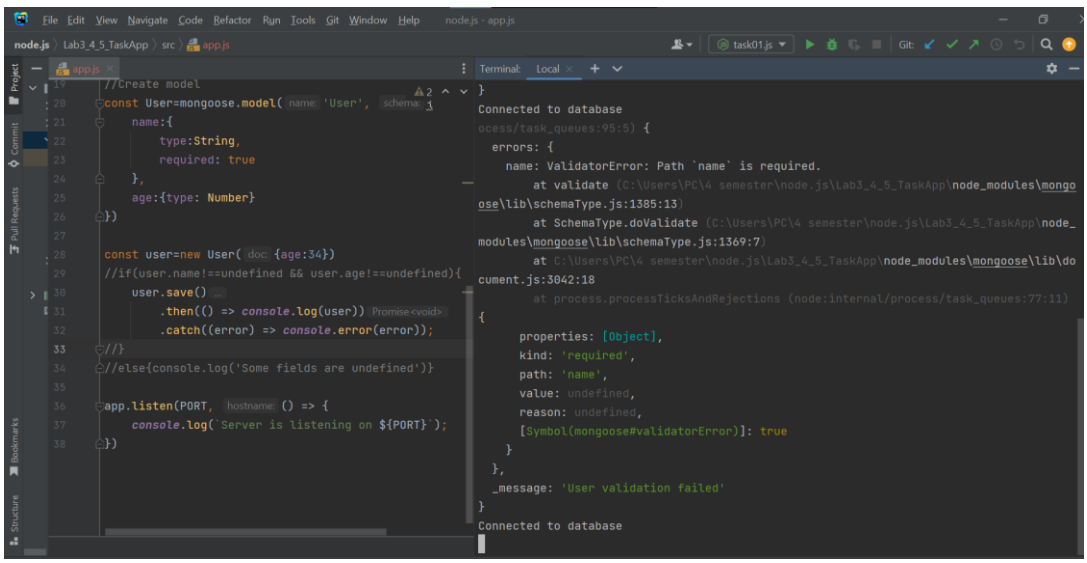


Рис. 4. Результат

Валідація даних:

- Для поля *age* моделі *User* створіть метод *validate(), в* якому забезпечте виключення від'ємних значень:

```
age: {
    type: Number,
    validate(value) {
        if (value < 0) {
            throw new Error( message: "Age must be a positive number");
        }
    }
}
```

```
//Create model
const User = mongoose.model( name: 'User', schema: {
    name: {
        type: String,
        required: true
    },
    age: {
        type: Number,
    validate(value){
            if(value<0){
                throw new Error('Age must be a positive number')
            }
    }}
})

const user = new User( doc: {name: 'Alex', age: -76})
user.save()
    .then(() => console.log(user)) Promise<void>
    .catch((error) => console.error(error));
```

```
ongoose\lib\document.js:3042:18
        at process.processTicksAndRejections (node:internal/process/task_
queues:77:11) {
      properties: [Object],
      kind: 'user defined',
      path: 'age',
      value: -76,
      reason: Error: Age must be a positive number
          at model.validate (C:\Users\PC\4 semester\node.js\Lab3_4_5_Task
App\src\app.js:29:23)
          at SchemaType.doValidate (C:\Users\PC\4 semester\node.js\Lab3_4
_5_TaskApp\node_modules\mongoose\lib\schemaType.js:1349:24)
          at C:\Users\PC\4 semester\node.js\Lab3_4_5_TaskApp\node_modules
\mongoose\lib\document.js:3042:18
          at process.processTicksAndRejections (node:internal/process/tas
k_queues:77:11),
      [Symbol(mongoose#validatorError)]: true
    }
  },
  _message: 'User validation failed'
}
Connected to database
```

Рис. 5. Результат

Validator

- Для більш складніших перевірок (emails, passwords, phone numbers) можна використовувати модуль npm-validator
- Встановіть та підключіть даний модуль в проект
- Для моделі User створіть поле *email* та зробіть перевірку на його коректність
- При неправильному заданні email, повинно виводитись повідомлення:

```
Error: Email is invalid
```

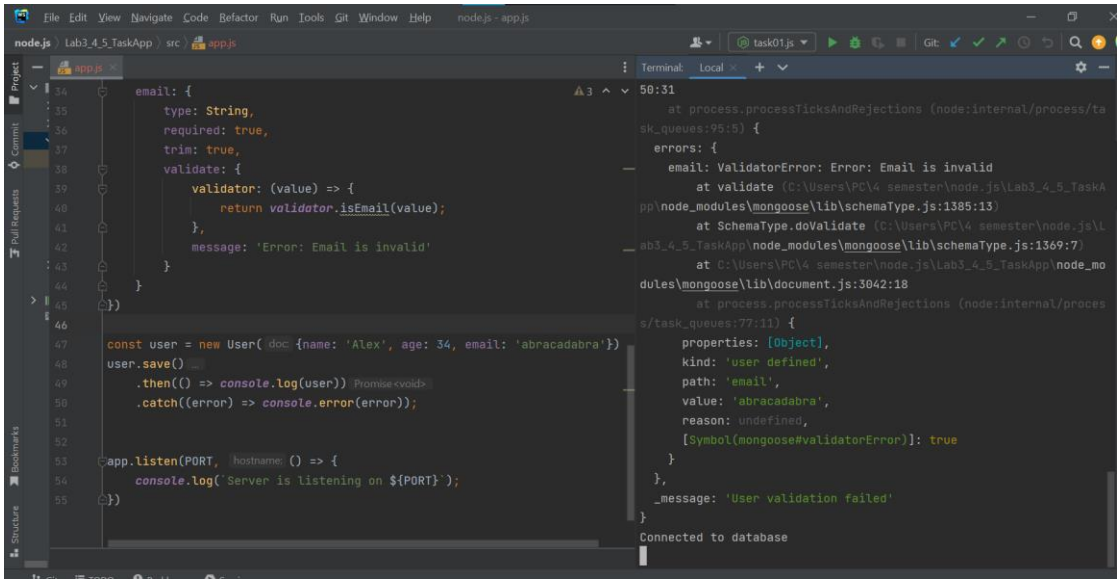


Рис. 6. Результат

Фільтрація даних (Data Sanitization)

- Додайте до поля name моделі User обрізку крайніх пробілів:

```
name: {
    type: String,
    required: true,
    trim: true
},
```

- Додайте до поля *email* моделі User конвертацію в нижній регістр
- Для поля *age* задайте значення по замовчуванню *0*

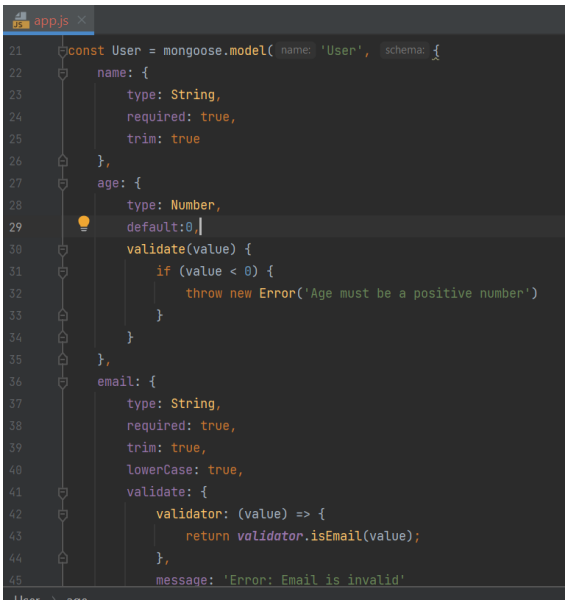


Рис. 7. Результат

Завдання для моделі User

1. Створіть поле *password* і зробіть його обов'язковим
2. Довжина поля не повинна бути меншою ніж 7
3. Обріжте крайні пробіли в полі password
4. Пароль не повинен містити слово *"password"*
5. Для поля *email* встановіть його унікальність (відсутність однакових значень)
6. **Винесіть модель User в файл */models/user.js* та підключіть його в додаток як модуль**
7. Протестуйте роботу

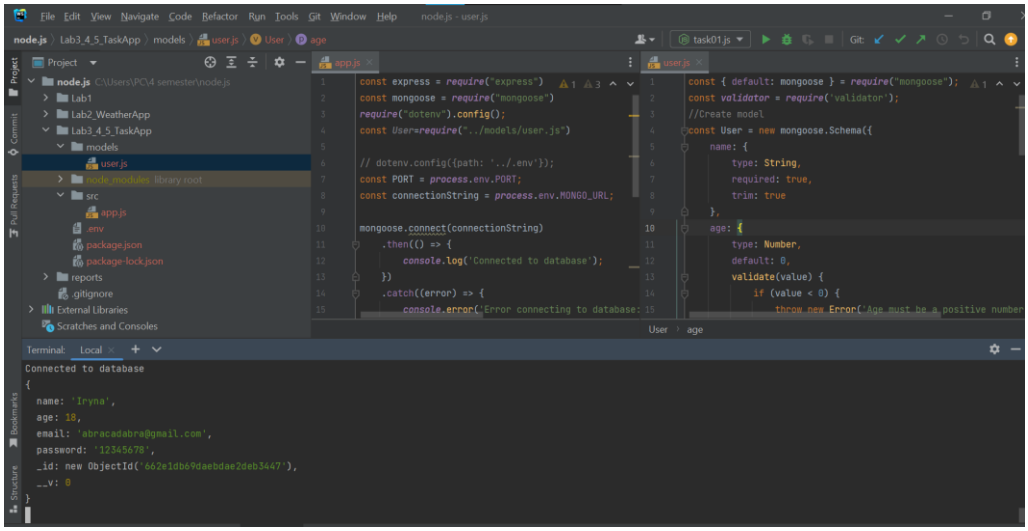
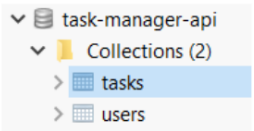


Рис. 8. Результат

Завдання для моделі Task

1. В файлі */models/task.js* створіть модель *Task* з полями *title (String)*, *description (String)* та *completed (Boolean)*
2. Зробіть поле *title, description* обов'язковим та з обрізкою крайніх пробілів
3. Make *completed* optional and default it to false
4. Test your work with and without errors
5. Створіть новий екземпляр моделі
6. Збережіть об'єкт в базі даних
7. Перевірте наявність БД *task-manager-api* та колекцій *tasks* та *users*



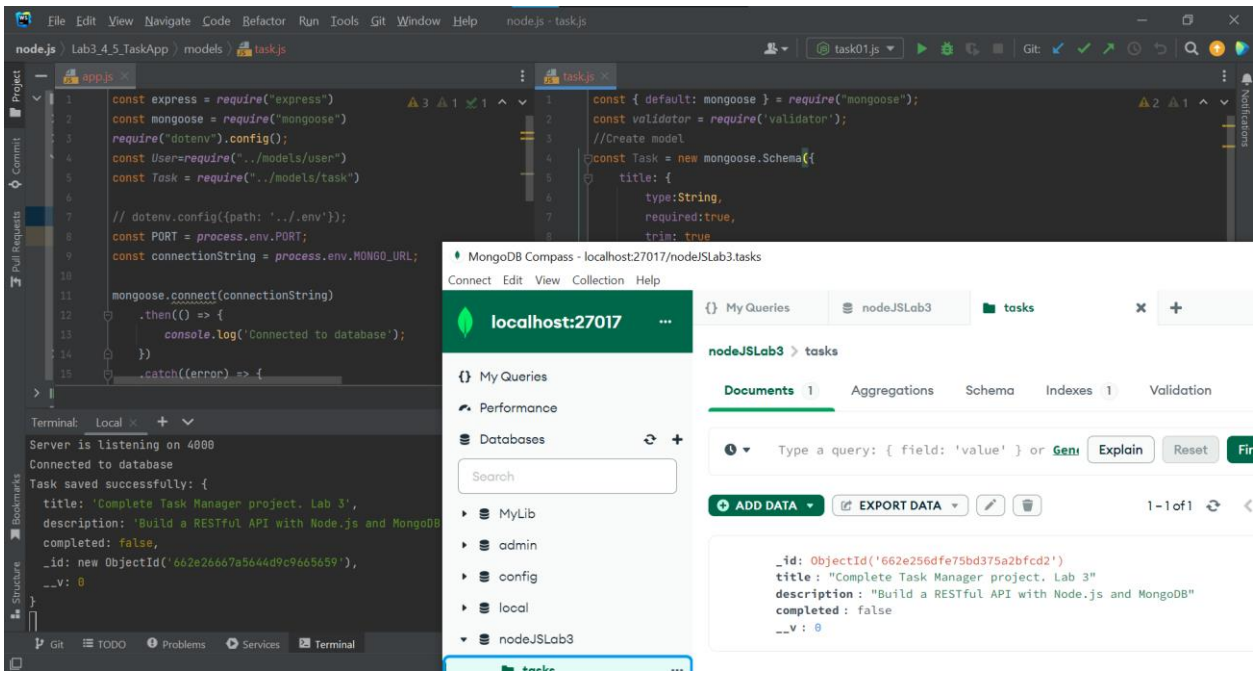


Рис. 9. Результат, перевірка, коли нема помилки

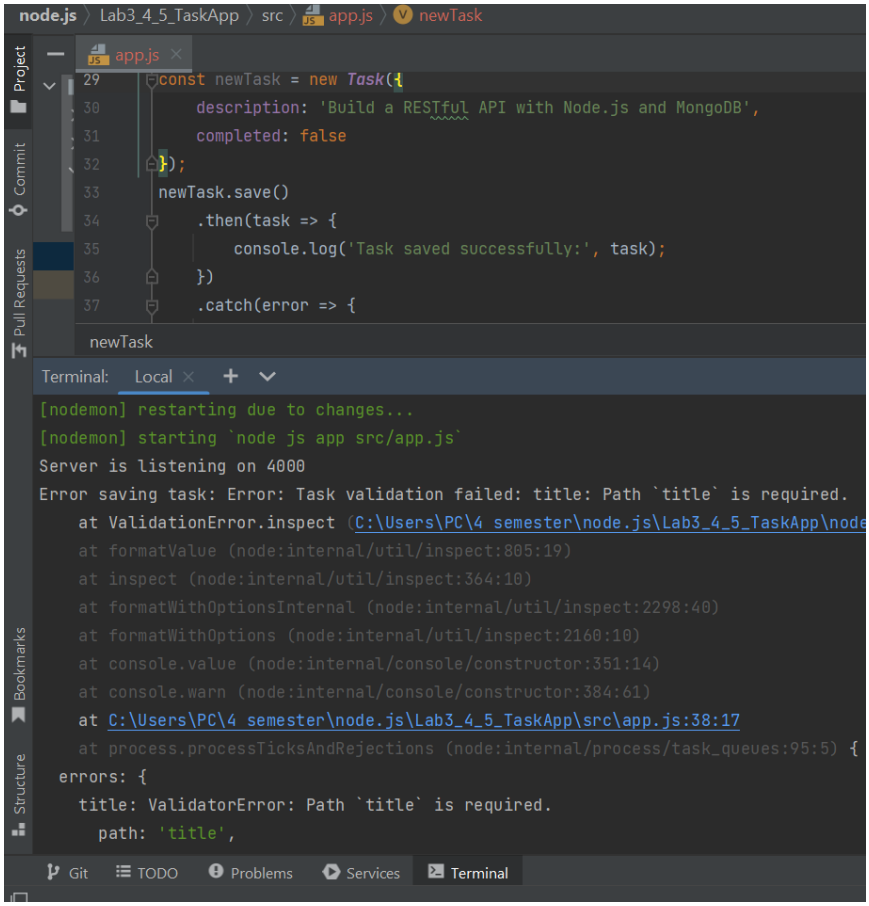


Рис. 10. Результат, перевірка, коли є помилки

Структуруйте проект

- Винесіть моделі User і Task у відповідні файли
- Підключення до БД також винесіть в окремий файл
- Забезпечте підключення даних файлів як модулів в додатку index.js:

```
1  const express = require( id: "express");
2  require( id: './db/mongoose');
3  const User = require( id: './models/user');
4  const Task = require( id: './models/task');
```

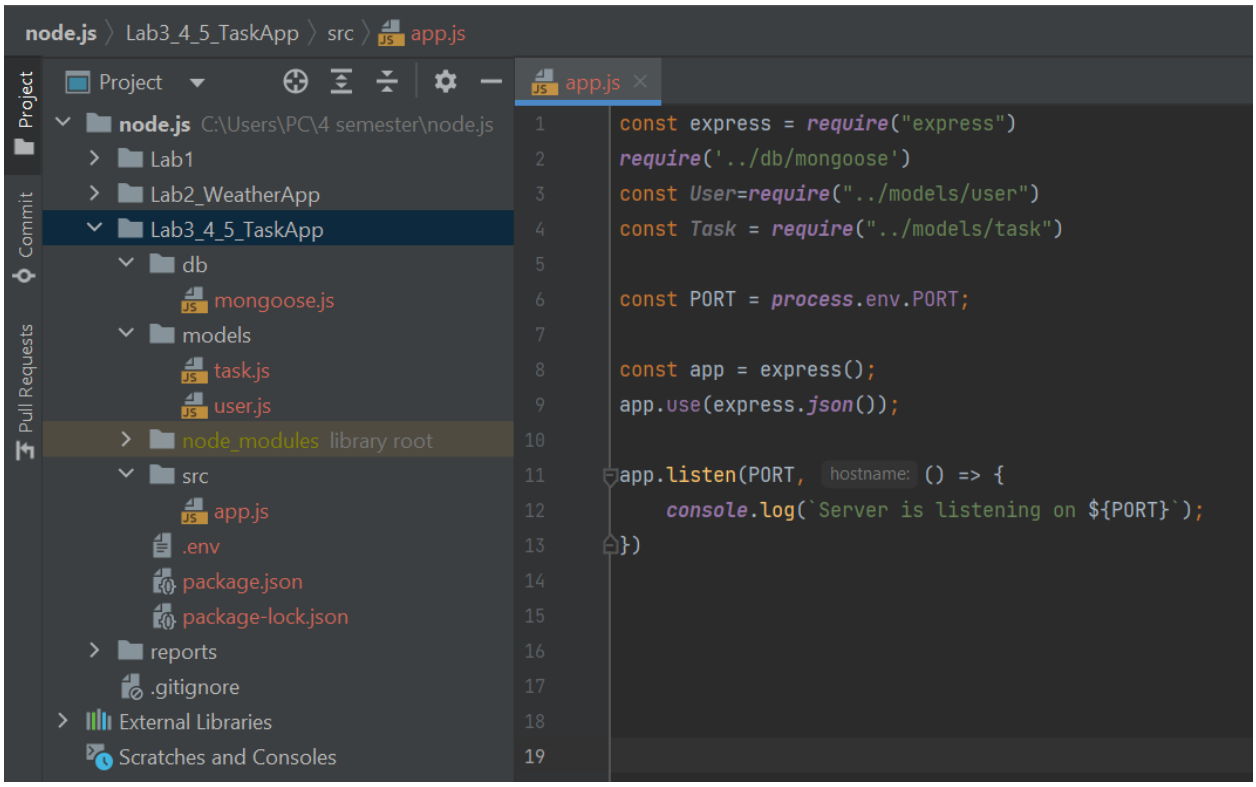


Рис. 11. Результат

Обробка GET-запиту

- В додатку створіть обробник запиту на отримання всіх користувачів з БД:

```
20  app.get("/users", (req, res) => {
21      User.find({}).then((users) => {
22          res.status(200).send(users);
23      }).catch((error) => {
24          res.status(500).send();
25      });
26  });
```

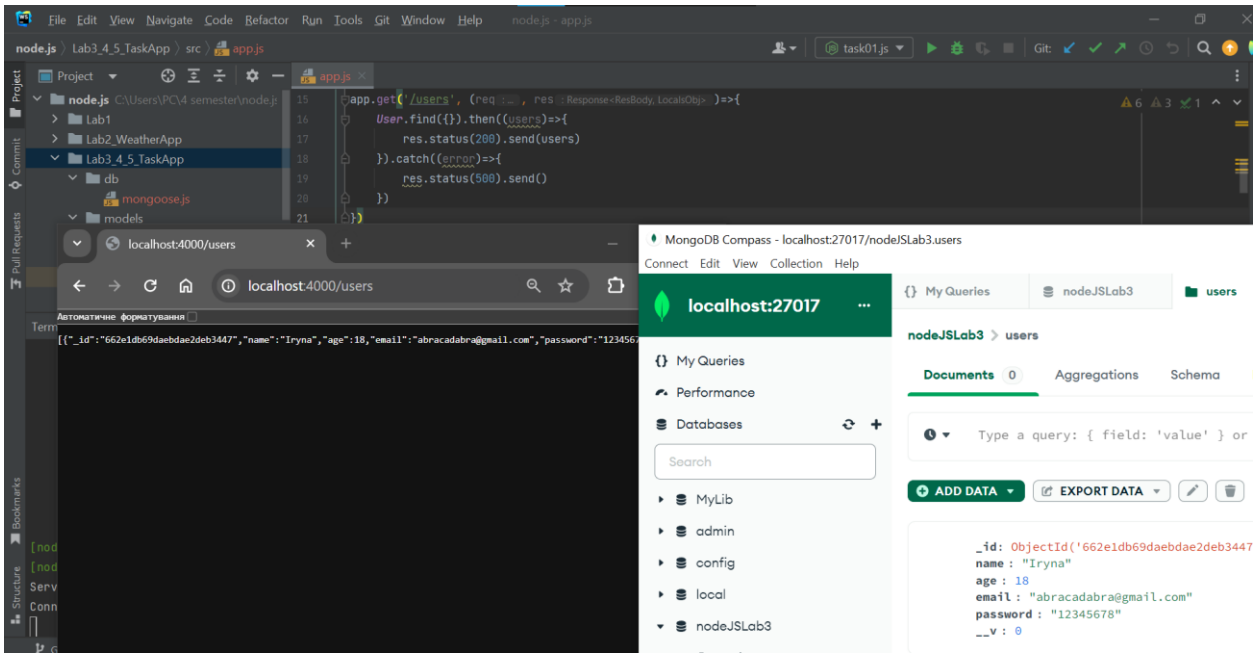
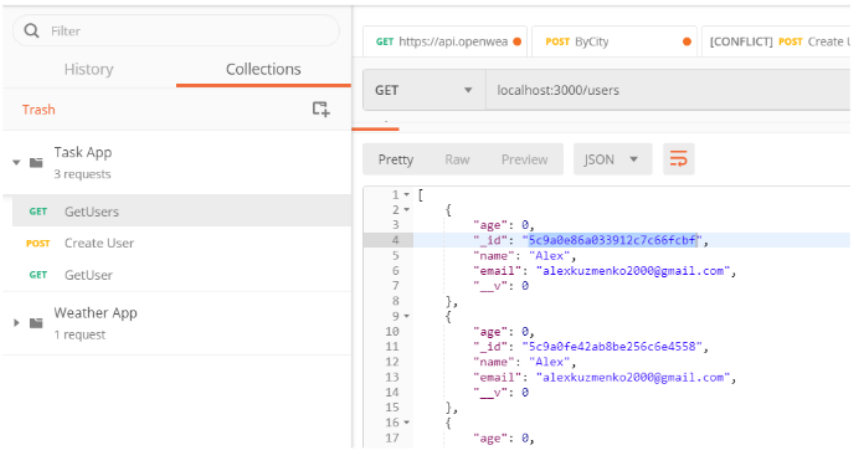


Рис. 12. Результат

З допомогою програми POSTMAN протестуйте виконання даного запиту:

- Створіть категорію Task App
- В цій категорії створіть та протестуйте запит GetUsers



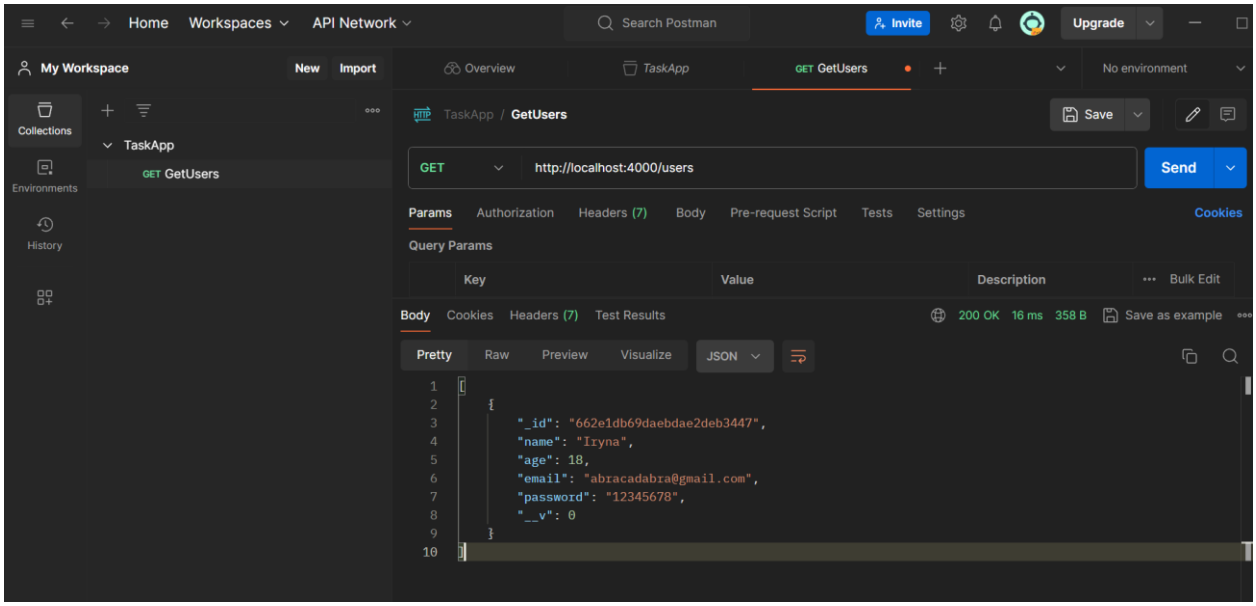


Рис. 13. Результат

Завдання. Створіть обробку запитів

- отримання користувача з БД по id
- додавання нового користувача в БД
- Створіть відповідний набір операцій REST API для моделі Task

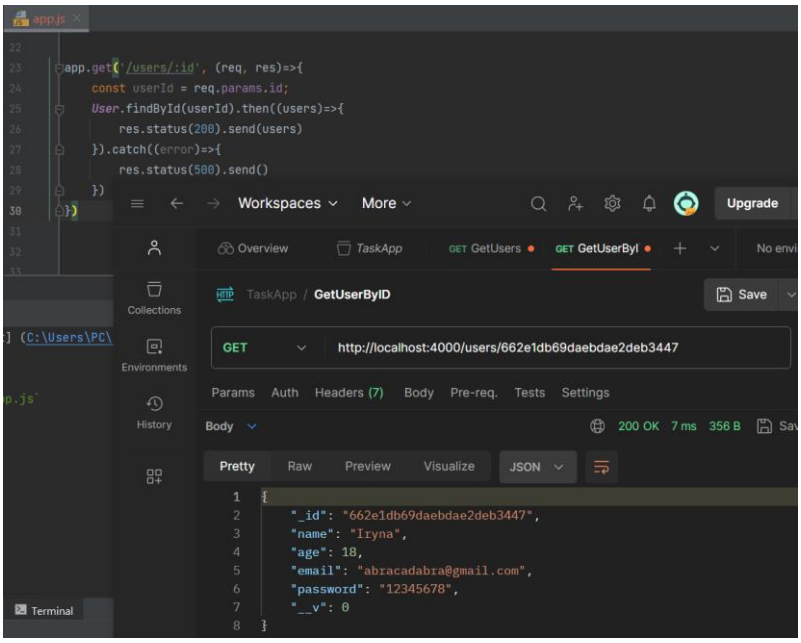


Рис. 14. Результат, отримання користувача за ID

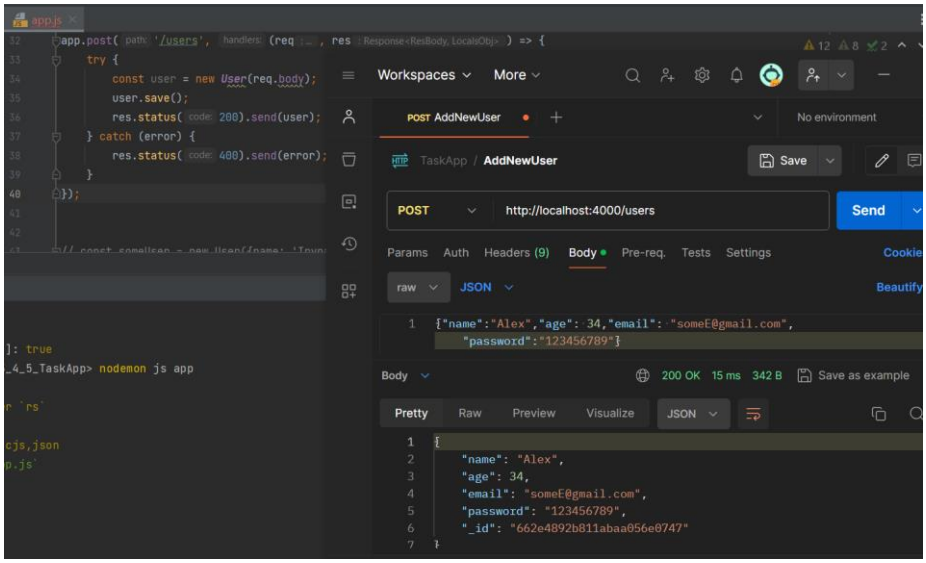


Рис. 15. Результат, додавання нового користувача

Рис. 16. Результат в БД

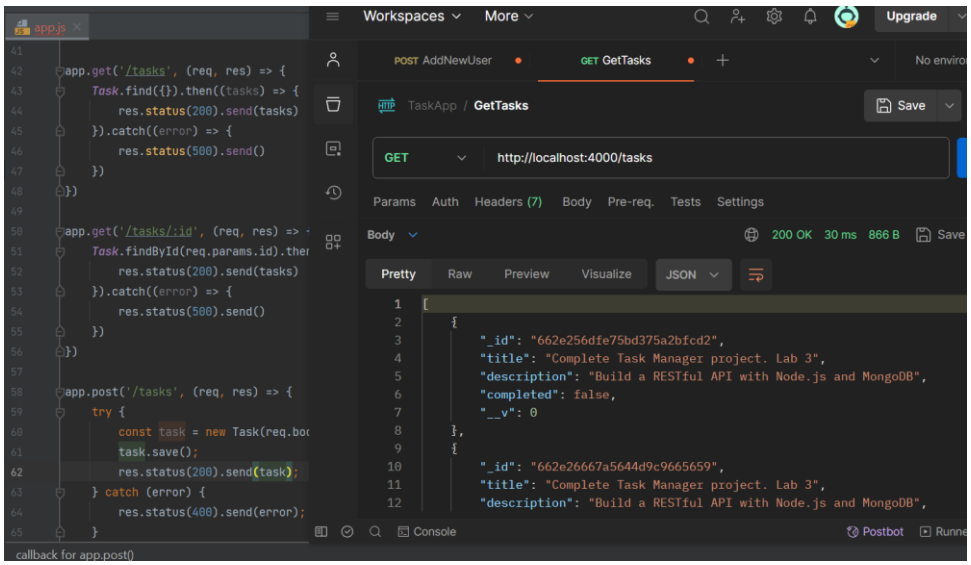


Рис. 17. Результат, всі task

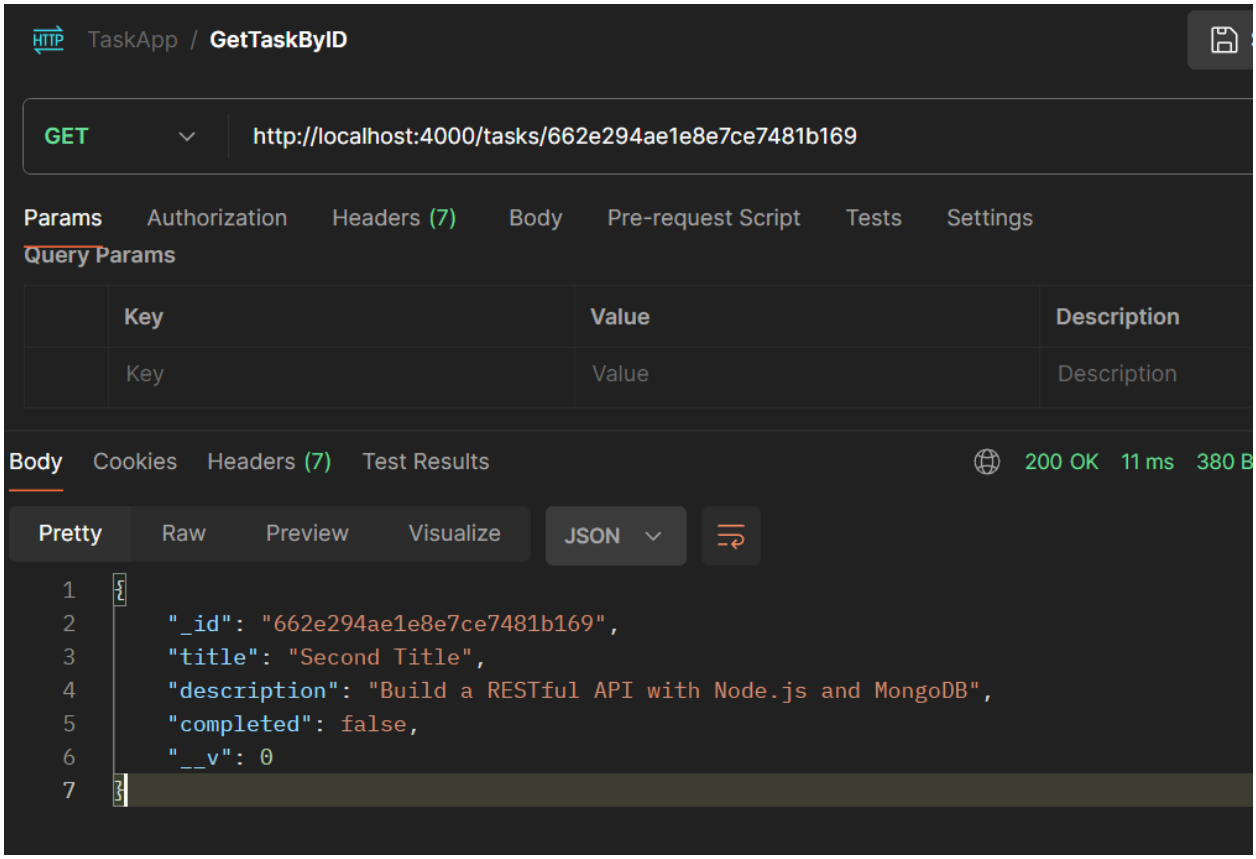


Рис. 18. Результат отримання task за ID

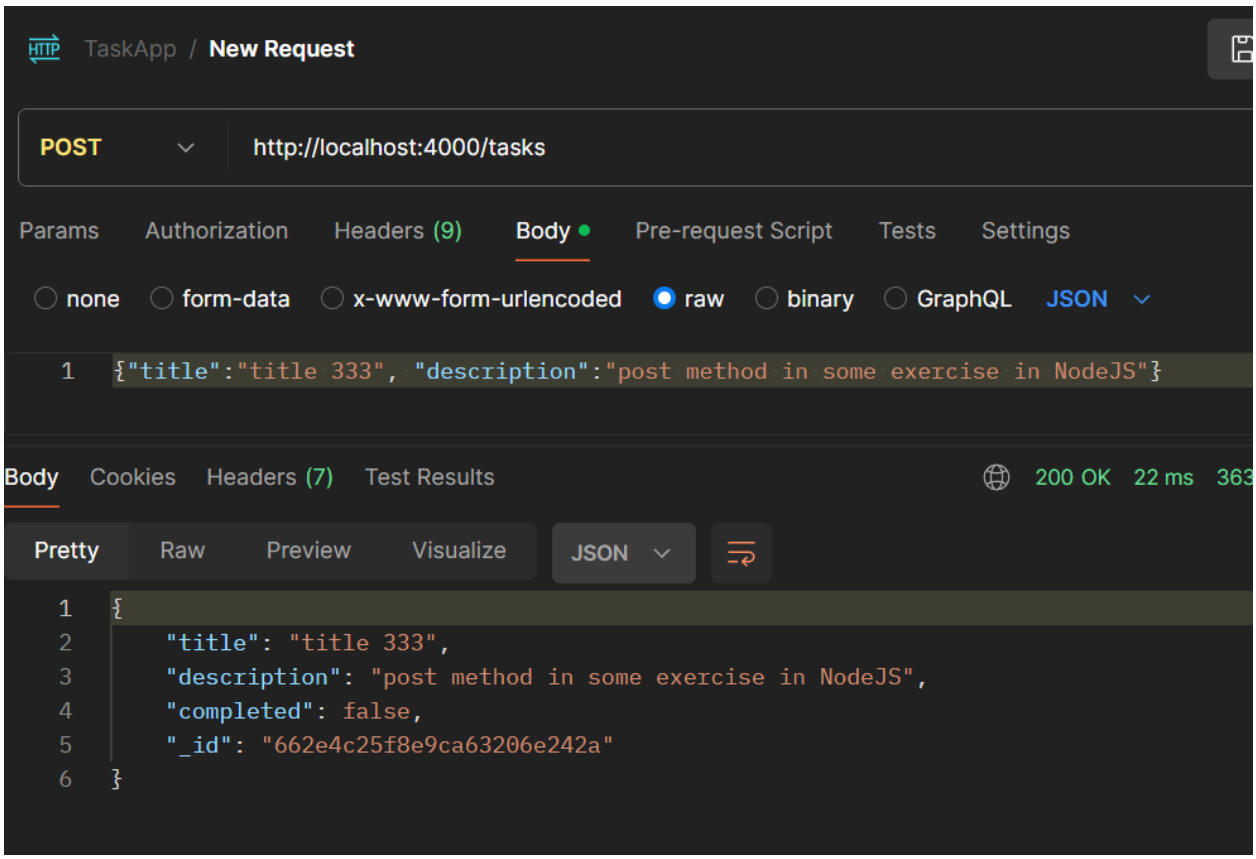


Рис. 19. Результат додавання запису в tasks

Здійсніть рефакторинг коду в додатку Task Application з використанням *async/await*

Приклад

```
app.post("/places", async(req, res) => {
    let place = new Place(req.body);
    try {
        await place.save();
        res.status(201).send(place);
    } catch (e) {
        res.status(500).send();
    }
})
```

```
31  app.post('/users', async (req : … , res : Response<ResBody, LocalsObj> ) => {
32      try {
33          const user = new User(req.body);
34          await user.save();
35          res.status(200).send(user);
36      } catch (error) {
37          res.status(400).send(error);
38      }
39  });
40
41
42
43  app.get('/tasks', async (req : … , res : Response<ResBody, LocalsObj> ) => {
44      try {
45          const tasks = await Task.find();
46          res.send(tasks);
47      } catch (error) {
48          res.status(500).send(error);
49      }
50  })
```

Рис. 20. Результат, приклад рефакторингу

Створіть обробку запиту видалення по id для моделей User і Task

- Шлях для видалення */user/:id*
- Операція повинна включати блоки *try/catch*
- Обробник повинен включати *async/await*
- Результат видалення повинен бути одним із:
  - Success (200)
  - Not Found (404)
  - Error (400)

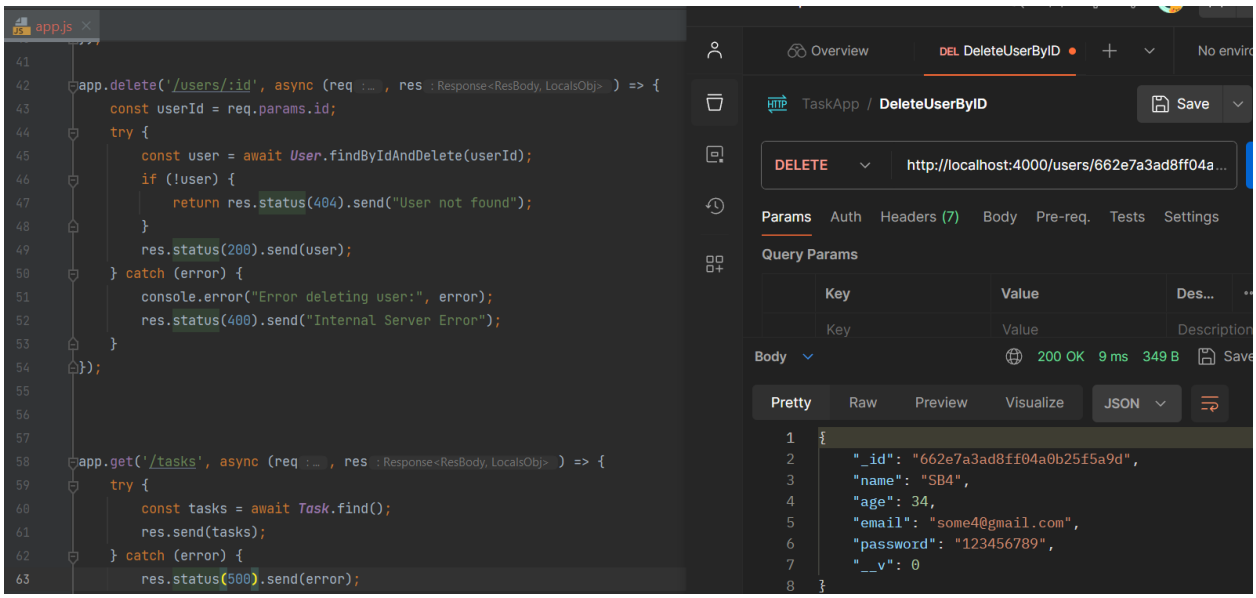


Рис. 21. Результат видалення user за id

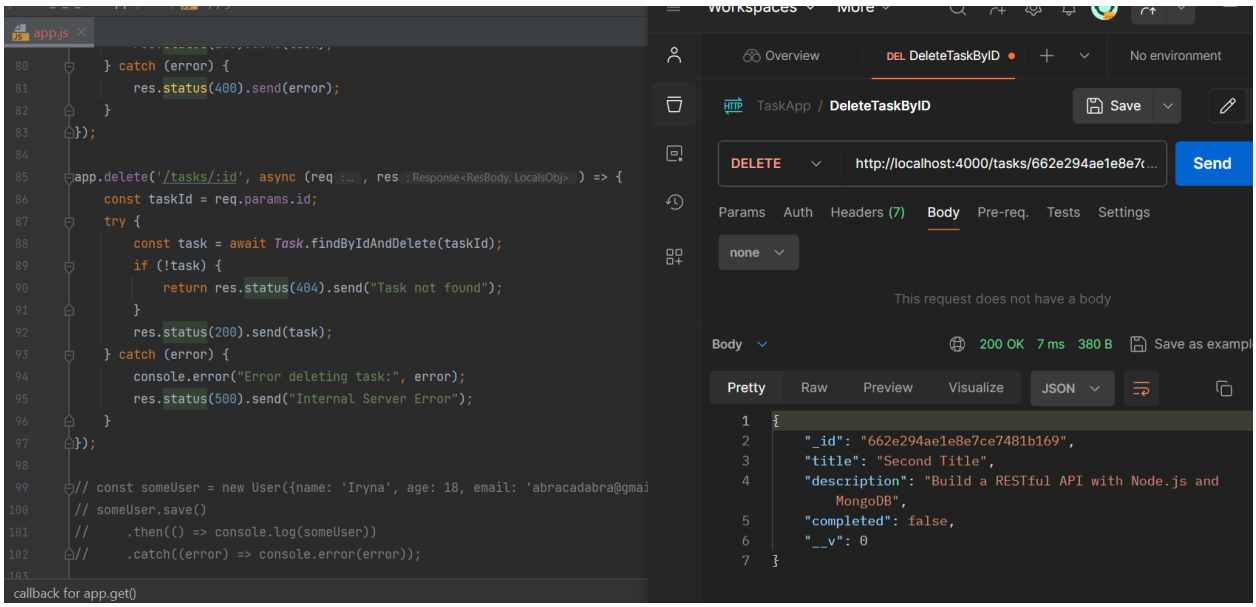


Рис. 22. Результат видалення task за id

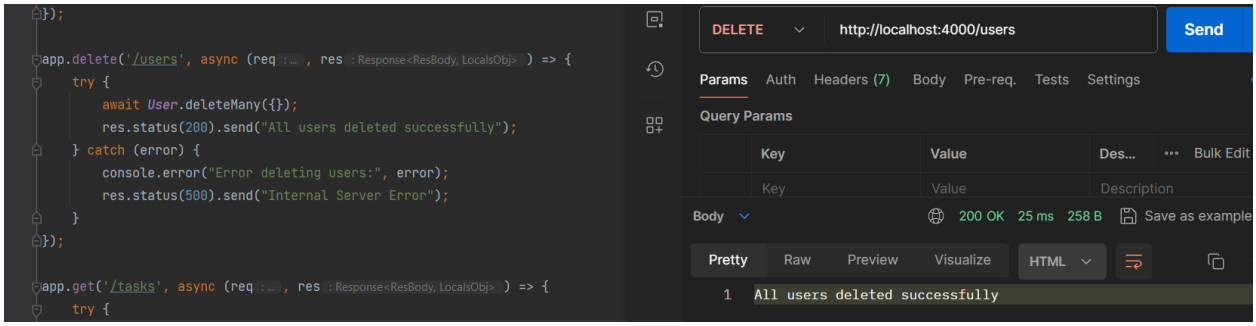


Рис. 23. Видалення всіх users

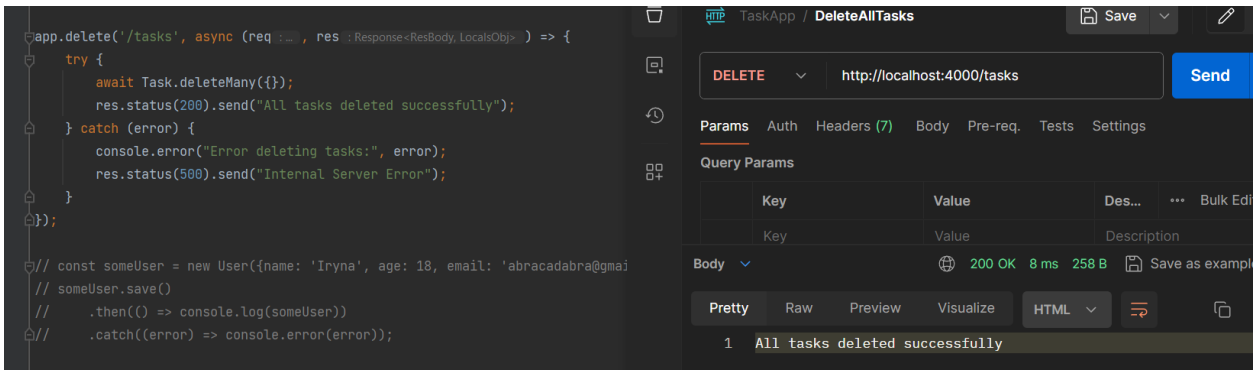


Рис. 24. Видалення всіх tasks

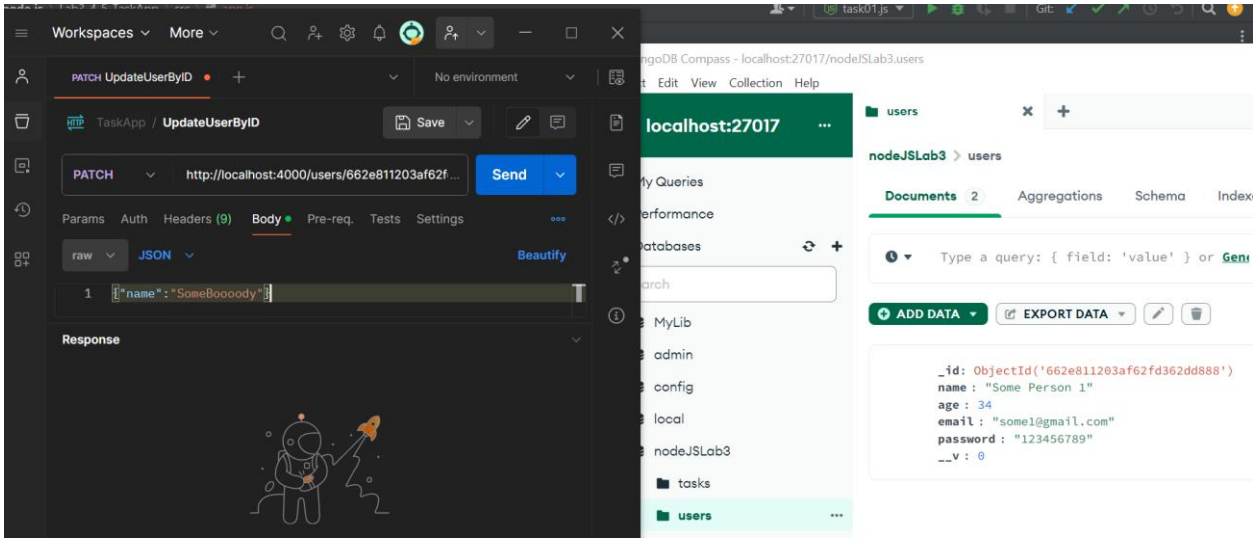


Рис. 25. До оновлення імені user

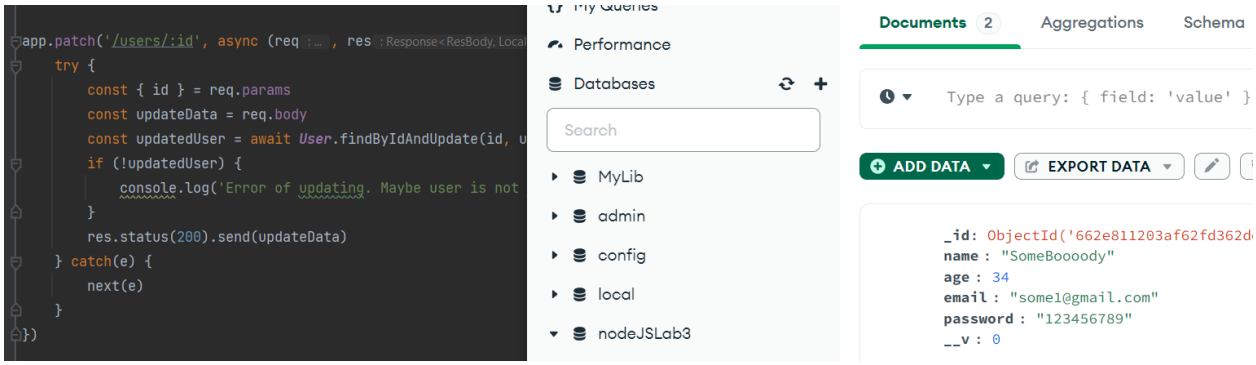


Рис. 26. Після оновлення user

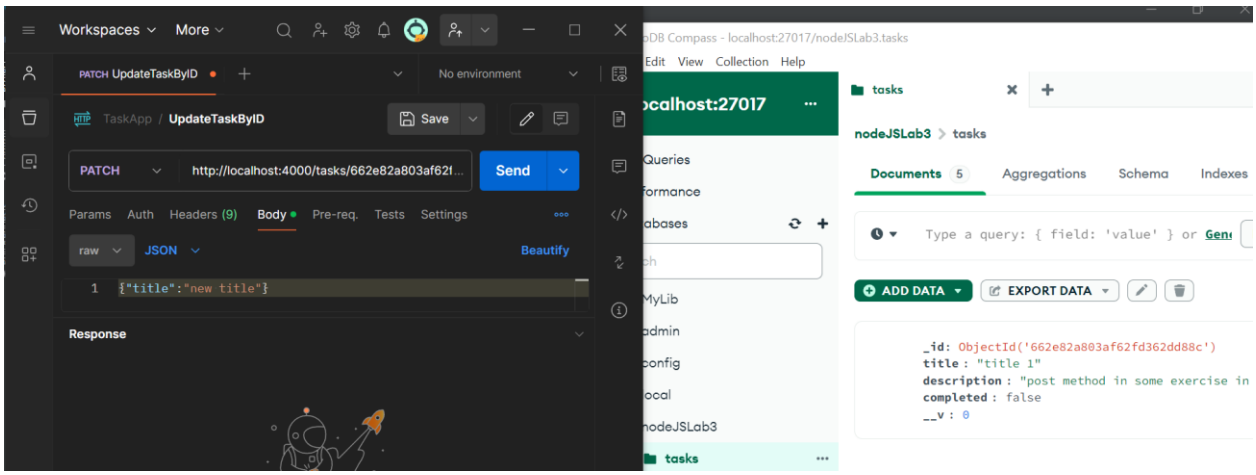


Рис. 27. До оновлення task

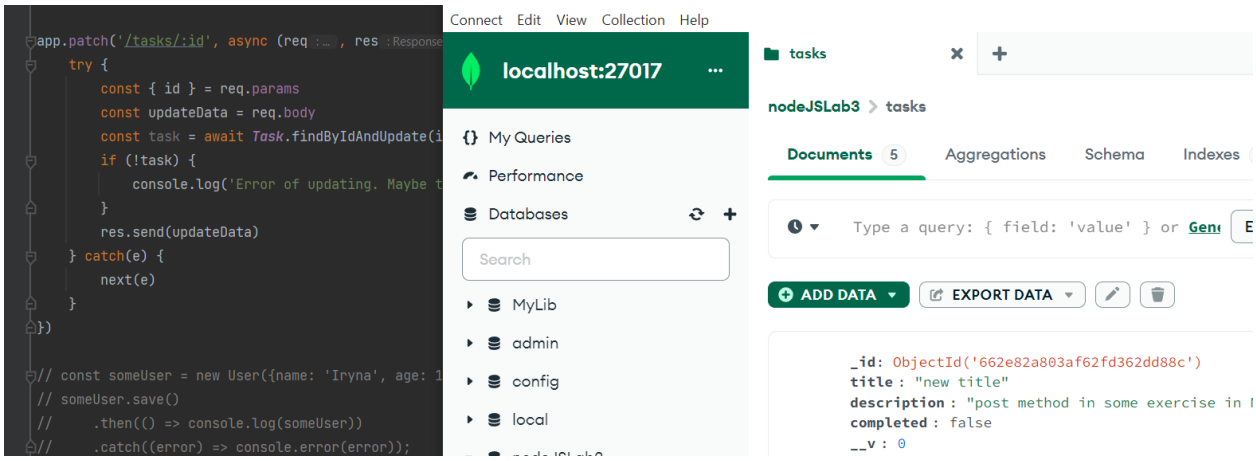
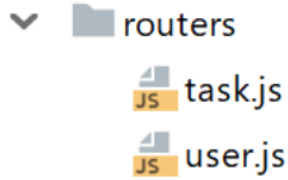


Рис. 28. Після оновлення task

Separate Route Files (маршрутизатори)

- Розділіть всі обробники маршрутів для користувачів і задач на два окремі файли, що міститимуться src/routers:



- Дані файли називаються маршрутизаторами. В кожному маршрутизаторі повинні міститись по 6 обробників маршрутів

Приклад створення роутера

- В файлі-маршрутизаторі routers/user.js створимо роутер:

```
const express = require("express");
const router = new express.Router();

router.get('/test', (req, res) => {
    res.send("From a new File");
})

module.exports = router;
```

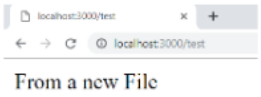
- В index.js підключимо роутер:

```
const userRouter = require('./routers/user');
```

- Додамо його в middleware додатку:

```
app.use(userRouter);
```

- Перевіримо виконання:



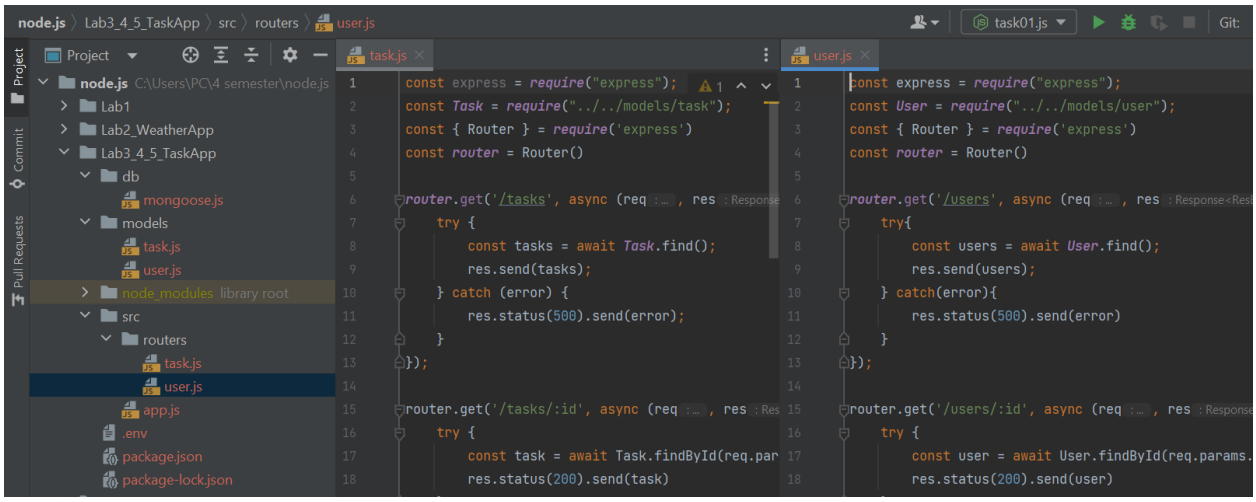


Рис. 29. Результат

Висновок: на лабораторному занятті ми ознайомились з Mongoose та Rest API.