

Розділ 2. BACK-END ПРОЕКТУВАННЯ WEB-ДОДАТКУ

2.1.2. Практичний модуль

У практичній частині продовжуємо проектну розробку web-додатку Tours City (екскурсії містом) у якому можна рекламувати подорожі будь-яким містом у галузі культурного міського туризму.

Етап 2.1. Побудова сервера з архітектурою MVC

Дія 2.1.1. Будуємо серверну архітектуру web-додатку. В корні проекту (якщо виконали Етап 1.4 то для повернення в корень проекту - `cd ..`) створюємо папку з ім'ям `server`:

```
mkdir server
```

- переходимо до цієї папки:

```
cd server
```

- у папці `server` будуємо архітектуру MVC, створюючи папки `controllers`, `models`, `view`:

```
mkdir models
```

```
mkdir views
```

```
mkdir controllers
```

- для повернення в корень проекту

```
cd ..
```

отримуємо вигляд архітектури сервера проекту (рис.2.2)

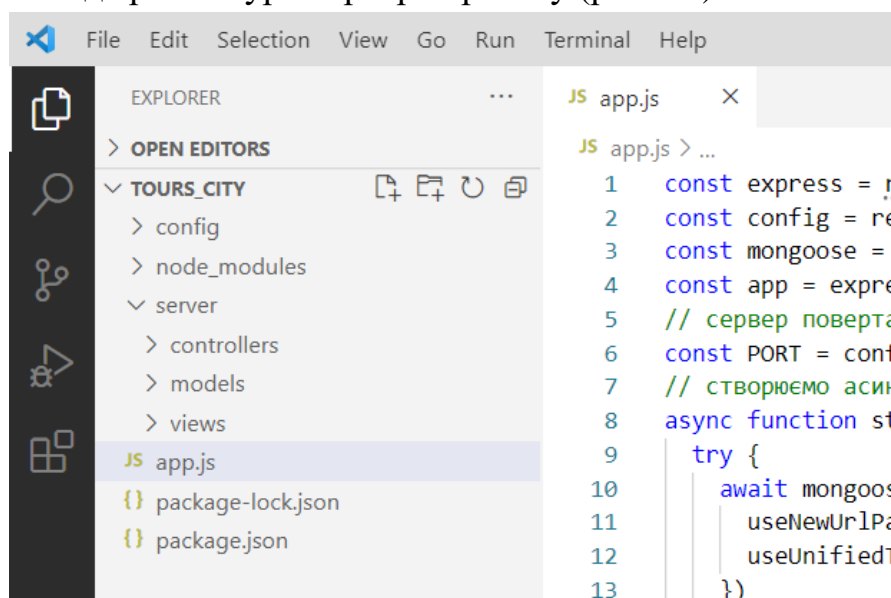


Рис. 2.2. Перегляд архітектури MVC у Visual Studio Code

Дія 2.1.2. Відповідно до технічного завдання проекту (Етап 1.1) передбачається авторизація для управління доступом до back-end web-додатку, після чого авторизована особа (Менеджер контенту) може здійснювати управління контентом. Для цього відповідно буде створюватись колекція бази даних Auth (від англ. Authentic).

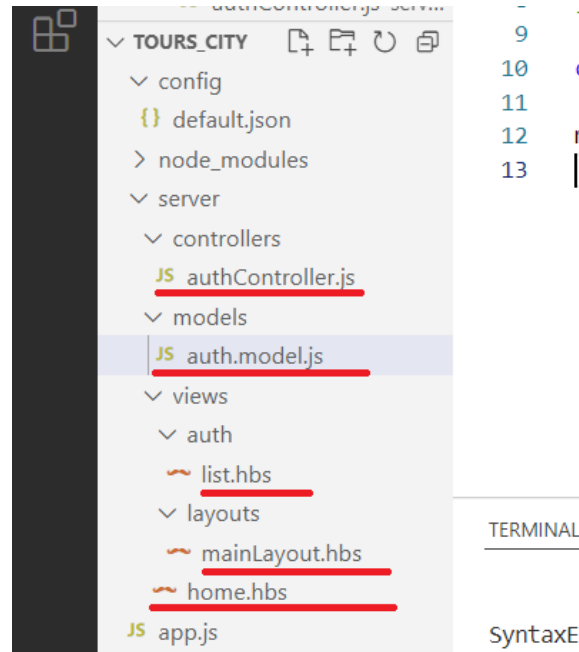


Рис. 2.3. Файли архітектури MVC

Архітектура MVC передбачає (дивись рис. 2.3) взаємодію головної сторінки web-додатку (mainLayout.hbs) зі сторінками із змінним контекстом (home.hbs - контент стартовий та list.hbs – контент перегляду даних). Вони розробляються на основі рушія-шаблонізатору для Node/JavaScript Handlebars, який забезпечує динамічну зміну контенту. Тому, потрібно встановити рушій-шаблонізатор Handlebars для відображення представлень на сторінках сервера. У командному рядку:

npm i express-handlebars

Зміною сторінок-відображень керуватиме контролер authController.js. За передавання даних контенту відповідатиме модель auth.model.js.

Дія 2.1.3. Засобами Visual Studio Code в папці /server/controllers створюємо файл authController.js.

У редакторі виконуємо такі дії:

- виділяємо папку controllers (рис. 2.4 зона 1)
- активізуємо команду New File (рис. 2.4 зона 2)
- вводим назву файлу authController.js (рис. 2.4 зона 3)
- натискаємо Enter

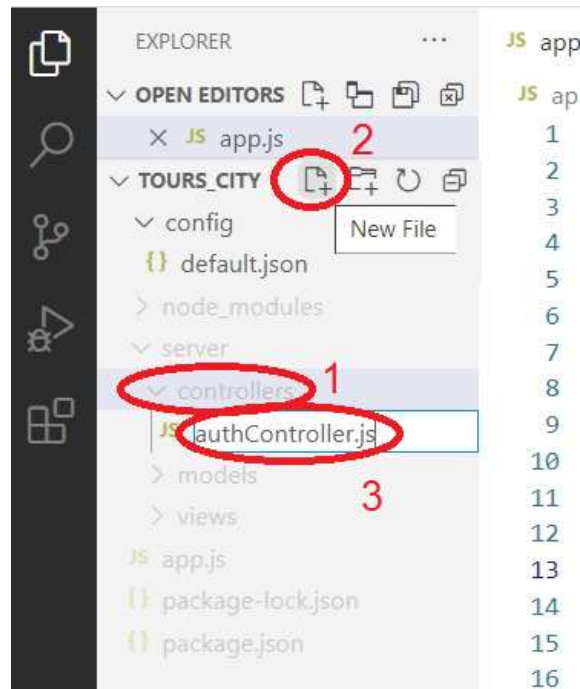


Рис. 2.4. Створення нового файлу в Visual Studio Code

Дія 2.1.4. Відкриваємо файл `authController.js` та пишемо у середині файлу код:

```
const express = require('express');
const { Router } = require('express')
const router = Router ()
// зв'язки (routes) для архітектури MVC для виведення сторінки list.hbs
router.get('/list', (req, res) => {
  res.render ("auth/list", {
    list: "architecture Model-View-Controller"
  });
});
module.exports = router;
```

Дія 2.1.5. У Visual Studio Code в папці `/server/models` аналогічно Дії 2.1.2 створюємо файл `auth.model.js`.

Дія 2.1.6. Відкриваємо файл `auth.model.js` та пишемо в середині файлу код:

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const authSchema = new Schema ({
// Залишаємо пусте місце
// Схема колекції буде прописана на наступному етапі
});
const Auth = mongoose.model('Auth', authSchema);
module.exports = Auth;
```

Дія 2.1.7. У Visual Studio Code в папці /server/views аналогічно Дії 2.1.5, активувавши команду New Folder, створюємо папку layouts, а в ній головний файл mainLayout.hbs шаблону вигляду. Відкриваємо файл mainLayout.hbs та пишемо в середині файлу код як на лістингу 2.1:

Лістинг 2.1 : Вміст файлу mainLayout.hbs

```
<!DOCTYPE html>
<html>
<head>
  <title>TourSity server</title>
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/
4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXw
EOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
crossorigin="anonymous">
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/font-
awesome/4.7.0/css/font-awesome.min.css">
</head>
<body class="bg-info">
  <div class="row" >
    <div class="col-md-6 offset-md-3" style="background-color: #fff;margin-top:
25px;padding:20px;">
```

Продовження лістингу 2.1

```
      {{{body}}}
    </div>
  </div>
</body>
</html>
```

кінець лістингу 2.1

Дія 2.1.8. У Visual Studio Code в папці /server/views аналогічно Дії 2.1.5, створюємо файл home.hbs.

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Architecture Model-View-Controller</a>
</nav>
<div style="margin-top: 30px">
```

```
<a class="btn btn-primary btn-lg active" href="/auth/list">Auth</a>
</div>
```

Дія 2.1.9. У Visual Studio Code в папці `/server/views` аналогічно Дії 2.1.5, створюємо файл `list.hbs`.

```
<h3> Auth List</h3>
<p> {{list}} </p>
```

Дія 2.1.10. Встановлюємо зв'язки (routes) для архітектури MVC між моделями, виглядами та контролерами.

- файл `authController.js` реалізує контролер управління
- файл `auth.model.js` визначатиме конфігурацію даних моделі. Данні будуть вибиратись відповідно з колекції бази даних `Auth` (від англ. `Authentic`), яка буде створена на наступному Етапі
- виклик основного вигляду `back-end web-додатку` формується кодом `mainLayout.hbs`
- динамічно змінюємий контент у `mainLayout.hbs` формується записом `{{body}}`
- параметр `body` відповідно від команд може вміщувати теги `home.hbs` (запуск з головного файлу `web-додатку app.js`) або теги `list.hbs`, що будуть виводити данні для авторизації осіб (запуск з відповідного контролера `authController.js`).

Відкриваємо файл `app.js` та дописуємо в середині файлу код. Код файлу `app.js` повинен виглядати як на лістингу 2.2.

Лістинг 2.2 : Вміст файлу `app.js`

```
const express = require('express')
const config = require('config')
const mongoose = require('mongoose')
const path = require('path')
const exphbs = require('express-handlebars')

// зв'язуємо app з моделями
require('./server/models/auth.model')
const Auth = mongoose.model('Auth')

// зв'язуємо app з контролерами
const authController = require('./server/controllers/authController')
const app = express() // створюємо сервер під ім'ям app
const PORT = config.get('port') || 5000
app.use('/auth', authController)

// зв'язуємо app з виглядами
```

```

app.set('views', path.join(__dirname, 'server', '/views/'))
app.engine('hbs', exphbs.engine ({extname: 'hbs', defaultLayout: 'mainLayout',
    runtimeOptions: {allowProtoPropertiesByDefault: true,
allowProtoMethodsByDefault: true} })))
app.set('view engine', 'hbs')
app.get('/', (req, res) => { res.render('home'); })
// створюємо асинхронну функцію
async function start() {
  try {
    await mongoose.connect(config.get('mongoUri'), {
      useUrlParser: true,
      useUnifiedTopology: true
    })
    app.listen(PORT, () => console.log(`Server App has been started on port
    ${PORT}...`))
  } catch (e) {
    console.log('Server Error', e.message)
    process.exit(1)
  }
}
start()

```

кінець лістингу 2.2

Дія 2.1.11. Виконуємо тестовий запуск сервера проекту

- у консолі вводимо:

```
npm run server
```

- у консолі отримуємо відповідь:

...

```
[nodemon] starting `node app.js`
```

```
Server App has been started on port 5000...
```

- у браузері вводимо:

```
localhost:5000
```

- відкривається стартова сторінка-вигляд back-end



Рис. 2.5. Вигляд стартової сторінки-вигляду beck-end

- виконуємо клік на кнопці Auth для переходу на показ контенту з вигляду list.hbs :

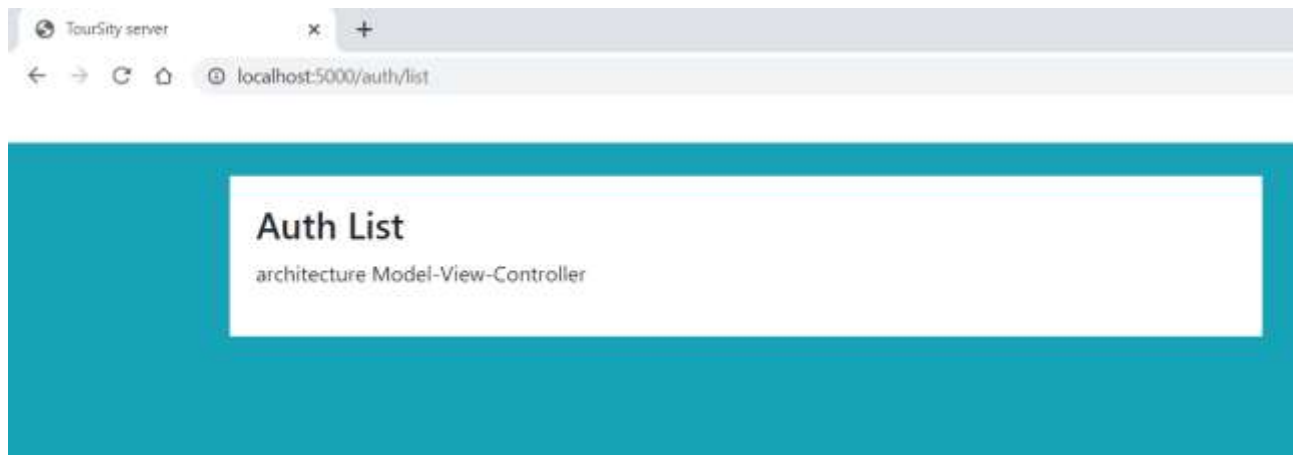


Рис. 2.6. Кконтенту з вигляду list.hbs

- зупиняємо проект:

Ctrl + C.

2.2. АРХІТЕКТУРА КОДУ MVC. ПОБУДОВА CRUD

2.2.2. Практичний модуль

У практичній частині продовжуємо проектну розробку web-додатку Tours City (екскурсії містом) у якому можна рекламувати подорожі будь-яким містом у галузі культурного міського туризму.

Етап 2.2. Побудова CRUD для роботи з даними на сервері

Дія 2.2.1. Встановлюємо проміжне програмне забезпечення body-parser для аналізу вмісту body у Node.js, який стає доступними під властивістю req.body. Властивість req.body дозволить мати доступ до введених даних якими керує користувач web-додатку. У консолі вводимо:

```
npm install body-parser
```

Дія 2.2.2. Встановлюємо зв'язок з body-parser для доступності req.body. Відкриваємо файл app.js та прописуємо зв'язок з модулем:

```
const bodyparser = require('body-parser')
//----- після запуску сервера const app = express() -----
app.use(bodyparser.urlencoded({
  extended: true
}))
app.use(bodyparser.json())
```

Дія 2.2.3. Відповідно до архітектури MVC, модель відповідає за зв'язок сервера з даними бази даних. Колекція бази даних auths сервера web-додатку вміщує данні авторизації на сервері. Це данні реєстрації користувача: Прізвище (lastName), Ім'я (firstName), адреса електронної пошти (email) та пароль входу (password). Для опису схеми колекції з визначенням типу даних відкриваємо файл auth.model.js та кодуємо схему моделі. Код файлу auth.model.js повинен виглядати як на лістингу 2.3.

Лістинг 2.3 : Вміст файлу auth.model.js

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
```



```
const authSchema = new Schema ({
  firstName: {
    type: String,
    required: 'This field is required.'
  },
  lastName: {
    type: String
  },
  email: {
    type: String
  },
  password: {
    type: String
  }
});
// Custom validation for email
authSchema.path('email').validate((val) => {
  emailRegex = /^(([^<>()\\[\]\\.,;:~\s@"]+\\.([^\s@"]+)|"(.+)")@((\[[0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\]|((\[[a-zA-Z0-9]+\]|([a-zA-Z0-9]{2,}))$)/;
  return emailRegex.test(val);
}, 'Invalid e-mail. ');
const Auth = mongoose.model('Auth', authSchema);
module.exports = Auth;
```

кінець лістингу 2.3

Дія 2.2.4. Контролер у архітектурі MVC виконує функції передавання, валідації, вибірки та управління даними на рівні серверу у взаємодії з базою даних. Відкриваємо файл `authController.js` та дописуємо в середині файлу код з функціями CRUD. Код файлу `authController.js` повинен виглядати як на лістингу 2.4.

```
const express = require('express');
const {Router} = require('express')
const router = Router ()
const mongoose = require('mongoose');
const Auth = mongoose.model('Auth');
const crypto = require('crypto');
    // функція генерації хеш паролю
const getHashedPassword = (password) => {
    const sha256 = crypto.createHash('sha256');
    const hash = sha256.update(password).digest('base64');
    return hash;
}
router.get('/', (req, res) => {          // виведення сторінки
    res.render("auth/register", {
        viewTitle: "Insert Auth"
    });
});
router.post('/', (req, res) => {
    if (req.body.password === req.body.confirmPasswordInput) {
        if (req.body._id !== "") updateRecord(req, res);
        Auth.find((err, docs) => {
            if (!err) {
                if(docs.find(docs => docs.email === req.body.email)) {
                    res.render('auth/register', {
                        message: 'This email already exists',
                        messageClass: 'alert-danger'  });
                    return;  }
            }
            else { console.log('Error in retrieving Auth list :' + err);
                }
        }
    }
});
```

```
        if (req.body._id == "") insertRecord(req, res);
    });
}
else
{
    res.render('auth/register', {
        message: 'Invalid password',
        messageClass: 'alert-danger'
    });
}
});

function insertRecord(req, res) {
    const hashedPassword = getHashedPassword(req.body.password);
    var auth = new Auth();
    auth.firstName = req.body.firstName;
    auth.lastName = req.body.lastName;
    auth.email = req.body.email;
    auth.password = hashedPassword;
    auth.save((err, doc) => {
        if (!err)
            res.redirect('auth/list');
        else {
            if (err.name == 'ValidationError') {
// Обробка помилок валідації. Валідація реалізована інструментами Mongoose
                handleValidationError(err, req.body);
                res.render("auth/register", {
                    viewTitle: "Insert Auth",
                    auth: req.body
                });
            }
            else
```

```
        console.log('Error during record insertion : ' + err);
    }
});
}

function updateRecord(req, res) {
    const hashedPassword = getHashedPassword(req.body.password);
    req.body.password = hashedPassword;
    Auth.findOneAndUpdate({ _id: req.body._id }, req.body, { new: true }, (err, doc)
=> {
        if (!err) {
            res.redirect('auth/list'); }
        else {
            if (err.name === 'ValidationError') {
// Обробка помилок валідації. Валідація реалізована інструментами Mongoose
                handleValidationError(err, req.body);
                res.render("auth/register", {
                    viewTitle: 'Update Auth',
                    auth: req.body
                });
            }
            else
                console.log('Error during record update : ' + err);
        }
    });
}

router.get('/list', (req, res) => {
    Auth.find((err, docs) => {
        if (!err) {
            res.render ("auth/list", {
                list: docs
            });
        }
    })
})
```

```
    else {
      console.log('Error in retrieving Auth list :' + err);
    }
  });
});

function handleValidationError(err, body) {
  for (field in err.errors) {
    switch (err.errors[field].path) {
      case 'firstName':
        body['firstNameError'] = err.errors[field].message;
        break;
      case 'email':
        body['emailError'] = err.errors[field].message;
        break;
      default:
        break;
    }
  }
}

router.get('/:id', (req, res) => {
  Auth.findById(req.params.id, (err, doc) => {
    if (!err) {
      res.render("auth/register", {
        viewTitle: "Update Auth",
        auth: doc
      });
    }
  });
});

router.get('/delete/:id', (req, res) => {
  Auth.findByIdAndRemove(req.params.id, (err, doc) => {
```

```
    if (!err) {  
        res.redirect('/auth/list');  
    }  
    else { console.log('Error in Auth delete :' + err); }  
});  
});  
module.exports = router;
```

кінець лістингу 2.4

Дія 2.2.5. Вигляд у архітектурі MVC виконує функції відображення та виведення даних на рівні сервера. Відкриваємо файл `home.hbs` та дописуємо в середині файлу код. Код файлу `home.hbs` повинен виглядати як на лістингу 2.5.

Лістинг 2.5 : Вміст файлу `home.hbs`

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">  
  <a class="navbar-brand" href="#">Authentication App</a>  
</nav>  
  
<div style="margin-top: 30px">  
  <a class="btn btn-primary btn-lg active" href="/auth/list">Register</a>  
</div>
```

кінець лістингу 2.5

Дія 2.2.6. Відкриваємо файл `list.hbs` та дописуємо в середині файлу код. Код файлу `list.hbs` повинен виглядати як на лістингу 2.6.

```
<h3><a class="btn btn-secondary" href="/auth"><i class="fa fa-plus"></i> Create  
New</a> Auth Data</h3>
```

```
<table class="table table-striped">  
  <thead>  
    <tr style="font-size:14px" >  
      <th>First Name</th>  
      <th>Last Name</th>  
      <th>Email</th>  
      <th>Password</th>  
      <th></th>  
    </tr>  
  </thead>  
  <tbody>  
    {{#each list}}  
    <tr style="font-size:14px">  
      <td>{{ this.firstName }}</td>  
      <td>{{ this.lastName }}</td>  
      <td>{{ this.email }}</td>  
      <td>{{ this.password }}</td>  
      <td>  
        <a href="/auth/{{ this._id }}"><i class="fa fa-pencil fa-lg" aria-  
hidden="true"></i></a>  
        <a href="/auth/delete/{{ this._id }}" onclick="return confirm('Are you sure  
to delete this record ?');"><i class="fa fa-trash fa-lg" aria-hidden="true"></i></a>  
      </td>  
    </tr>  
    {{/each}}  
  </tbody>  
</table>
```

Дія 2.2.7. Для реєстрації користувачів потрібна окрема сторінка web-додатку. Створюємо файл register.hbs та дописуємо в середині файлу код. Код файлу register.hbs повинен виглядати як на лістингу 2.7.

Лістинг 2.7 : Вміст файлу register.hbs

```
<h3>{{viewTitle}}</h3>
{{#if message}}
  <div class="alert {{messageClass}}" role="alert">
    {{message}}
  </div>
{{/if}}
<form name="register" action="/auth" method="POST" autocomplete="off">
  <input type="hidden" name="_id" value="{{auth._id}}">
  <div class="form-group">
    <label>First Name</label>
    <input type="text" class="form-control" style="font-size:14px"
name="firstName" id="firstNameInput" placeholder="First Name"
value="{{auth.firstName}}">
    <div class="text-danger">
      {{auth.firstNameError}}</div>
    </div>
    <div class="form-group">
      <label>Last Name</label>
      <input type="text" class="form-control" style="font-size:14px"
name="lastName" id="lastNameInput" placeholder="Last Name"
value="{{auth.lastName}}">
    </div>
    <div class="form-group">
      <label>Email</label>
      <input type="text" class="form-control" style="font-size:14px" name="email"
id="emailInput" placeholder="Email" value="{{auth.email}}">
      <div class="text-danger">
        {{auth.emailError}}</div>
    </div>
    <div class="form-group">
      <label>Password</label>
```



```
<input type="password" class="form-control" style="font-size:14px"
name="password" id="passwordInput" placeholder="Password"
value="{{ auth.password }}">
</div>
<div class="form-group">
  <label>confirmPasswordInput</label>
  <input type="password" class="form-control" style="font-size:14px"
name="confirmPasswordInput" id="confirmPasswordInput"
  placeholder="Re-enter your password here">
</div>
<div class="form-group">
  <button type="submit" class="btn btn-info"><i class="fa fa-database"></i>
Submit</button>
  <a class="btn btn-secondary" href="/auth/list"><i class="fa fa-list-alt"></i>
View All</a>
</div>
</form>
```

кінець лістингу 2.7

Дія 2.2.8. Виконуємо тестовий запуск сервера проекту

- у консолі вводимо:

```
npm run server
```

- у браузері вводимо:

```
localhost:5000
```

- Виконуємо послідовність дій з операціями CRUD на сторінках-виглядах
beck-end (рис. 2.7 – 2.10).

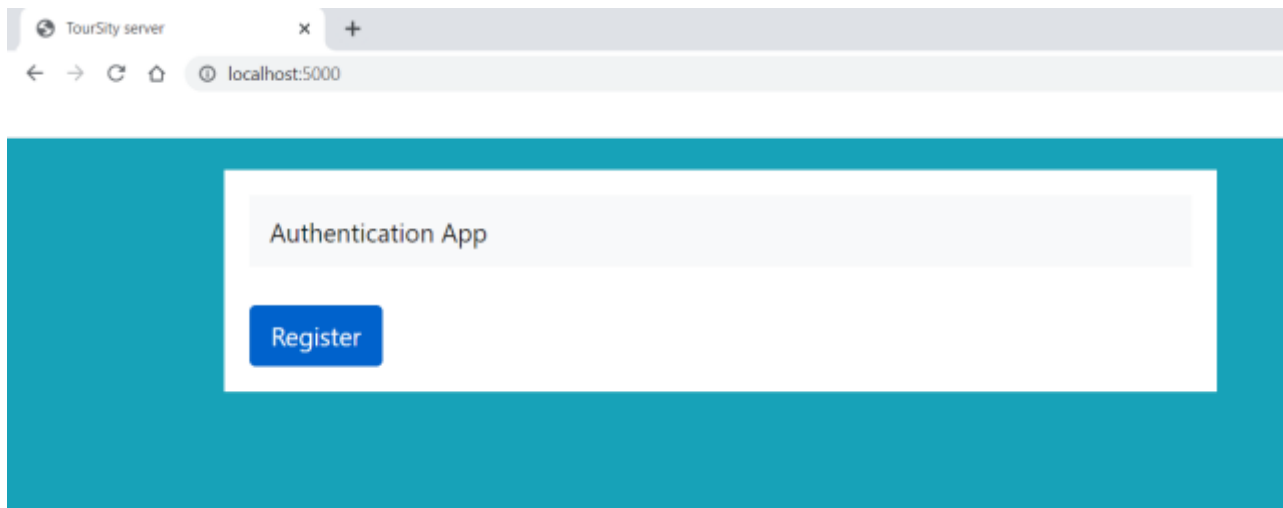


Рис. 2.7. Вигляд home.hbs

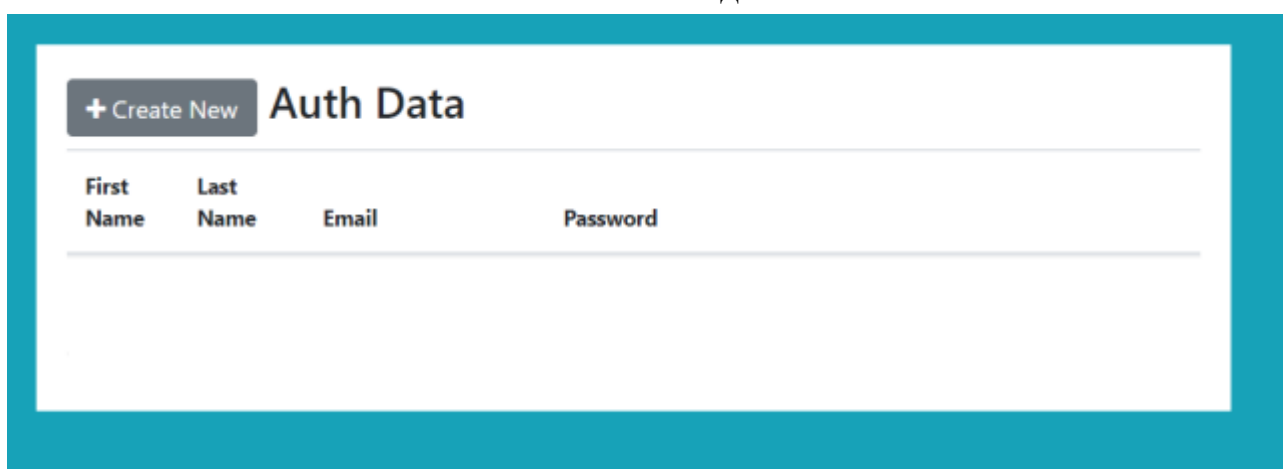


Рис. 2.8. Контент з вигляду list.hbs до введення даних

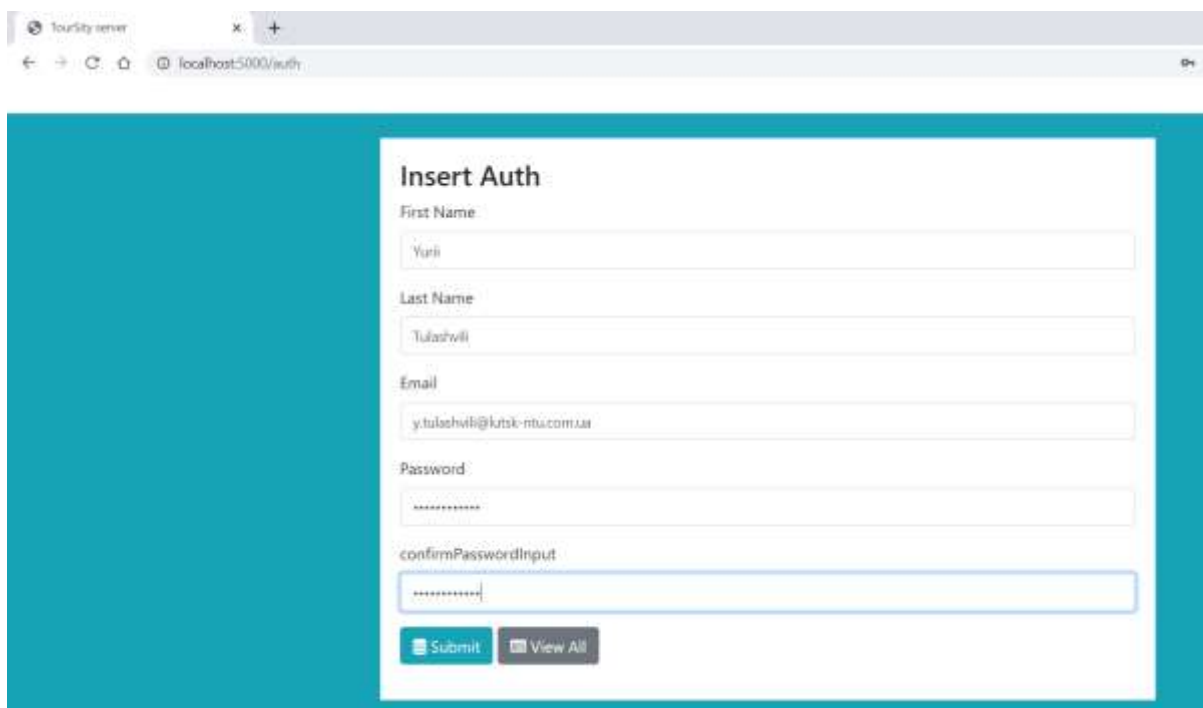


Рис. 2.9. Вигляд register.hbs для введення даних

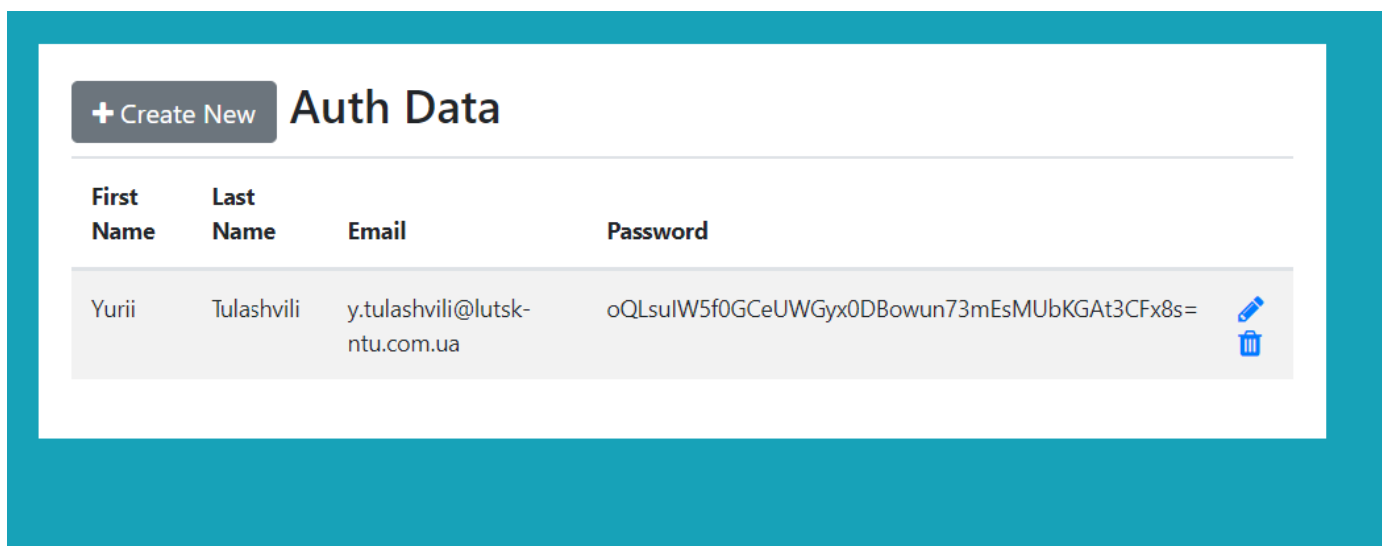


Рис. 2.10. Контент з вигляду list.hbs після введення даних

- зупиняємо проект:

Ctrl + C.

2.3. АВТОРИЗАЦІЯ НА СЕРВЕРІ. ЗАХИСТ ІНФОРМАЦІЇ

2.3.2. Практичний модуль

У практичній частині продовжуємо проектну розробку web-додатку Tours City (екскурсії містом) у якому можна рекламувати подорожі будь-яким містом у галузі культурного міського туризму.

Етап 2.3. Реалізація бізнес-логіки для авторизації на сервері

Дія 2.3.1. На сервері web-додатку, відповідно до бізнес-логіки, передбачено адміністрування допуску авторизованих користувачів до даних протягом заданого інтервалу часу. Тому, для входження на сервер потрібна авторизація, яка передбачає введення даних збережених у колекції бази даних Auth (email та password). Розпізнавання авторизованого користувача використовуємо Cookie-файли, що дозволяють зберігати на стороні клієнта деякі дані протягом заданого інтервалу часу. В нашому випадку це дозволить при роботі з сервером web-додатку ідентифікувати користувача. Node.js для цього використовує програмний модуль cookie-parser.

У консолі вводимо:

npm install cookie-parser

Дія 2.3.2. Встановлюємо зв'язок з cookie-parser для створення кукен-токену. Відкриваємо файл app.js та прописуємо зв'язок з модулем:

```
const cookieParser = require('cookie-parser')
а нижче код застосування ресурсів модуля для хешування паролів та
генерації токену аутентифікації:
const crypto = require('crypto')
та створюємо об'єкт:
const authTokens = {}
// для експорту ролі в контролер authController
const role = { key1: 'admin'};
// функція генерації хеш паролю
const getHashedPassword = (password) => {
  const sha256 = crypto.createHash('sha256');
  const hash = sha256.update(password).digest('base64');
  return hash;
}
// генерування токену
const generateAuthToken = () => {
  return crypto.randomBytes(30).toString('hex');
}
```

Дія 2.3.3. Для логізації користувача потрібна окрема сторінка web-додатку. Створюємо файл login.hbs та дописуємо в середині файлу код. Код повинен виглядати як на лістингу 2.8.

Лістинг 2.8 : Вміст файлу login.hbs

```
<div class="row justify-content-md-center" style="margin-top: 100px">
  <div class="col-md-6">
    //
    {{#if message}}
      <div class="alert {{messageClass}}" role="alert">
        {{message}}
      </div>
    {{/if}}
    <form method="POST" action="/login">
      <div class="form-group">
        <label for="exampleInputEmail1">Email address</label>
```

```
        <input      name="email"      type="email"      class="form-control"
id="exampleInputEmail1" placeholder="Enter email">
    </div>
    <div class="form-group">
        <label for="exampleInputPassword1">Password</label>
        <input  name="password"  type="password"  class="form-control"
id="exampleInputPassword1" placeholder="Password">
    </div>
    <button type="submit" class="btn btn-primary">Login</button>
</form>
</div>
</div>
```

кінець лістингу 2.8

Дія 2.3.4. Для переходу до колекцій бази даних на сервері створюємо сторінку Protected Page web-додатку. Створюємо файл protected.hbs та дописуємо в середині файлу код. Код повинен виглядати як на лістингу 2.9.

Лістинг 2.9 : Вміст файлу protected.hbs

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Protected Page</a>
</nav>

<div style="margin-top: 30px">
  <a class="btn btn-primary btn-lg active" href="/auth/list">Auth</a>
</div>
```

кінець лістингу 2.9

Дія 2.3.5. Для передавання даних та куки до контролерау authController у файл app.js прописуємо:

```
//----- після запуску сервера const app = express() -----
app.use(cookieParser())
app.use((req, res, next) => {
  const authToken = req.cookies['AuthToken'];
  req.users = authTokens[authToken];
  next();
});
//----- експорт до контролерау authController
app.get('/auth/', function(req, res){
  authController (req, res, role);
});
```

Дія 2.3.6. Для опрацювання логінізації користувача відкриваємо файл app.js та прописуємо опрацювання форми логінізації:

```
app.get('/login', (req, res) => {
  res.render('login');
});

app.post('/login', (req, res) => {
  const hashedPassword = getHashedPassword(req.body.password);
  Auth.find((err, docs) => {
    if (!err) {
```

```

    if(docs.find(docs => (docs.email === req.body.email && docs.password ===
hashedPassword ))) {
        const authToken = generateAuthToken();
        authTokens[authToken] = req.body.email;
// створюємо КУКИ AuthToken в App
        res.cookie('AuthToken', authToken);
        res.redirect('protected');
        return authToken;
    }
    else {
        res.render('login', {
            message: 'Invalid username or password',
            messageClass: 'alert-danger'
        });
    }
});
});
app.get('/protected', (req, res) => {
    const took = req.cookies.AuthToken; // Получаем куки AuthToken, что
созданный в app
    // console.log(took);
    if (took) {
        res.render('protected');
    } else
    {
        res.render('login', {
            message: 'Please login to continue',
            messageClass: 'alert-danger'
        });
    }
});
});

```

Дія 2.3.7. У методах `router.get` контролера `authController` для переходу на відповідні сторінки прописуємо перевірку токєну та при його відсутності переведення на сторінку логізації:

```

// отримуємо куки AuthToken, що були створено в app.js
const took = req.cookies.AuthToken;

if (took ) {
    res.render(" ... ", {
        viewTitle: " ... "
    });
}
else {

```

```

    res.render('login', {
      message: 'Please login to continue',
      messageClass: 'alert-danger'
    });
  }
});

```

Дія 2.2.8. Виконуємо тестовий запуск сервера проекту

- у консолі вводимо:

```
npm run server
```

- у браузері вводимо:

```
localhost:5000
```

- Виконуємо послідовність дій ідентифікації користувача на сторінках-виглядах back-end (рис. 2.13 – 2.16).

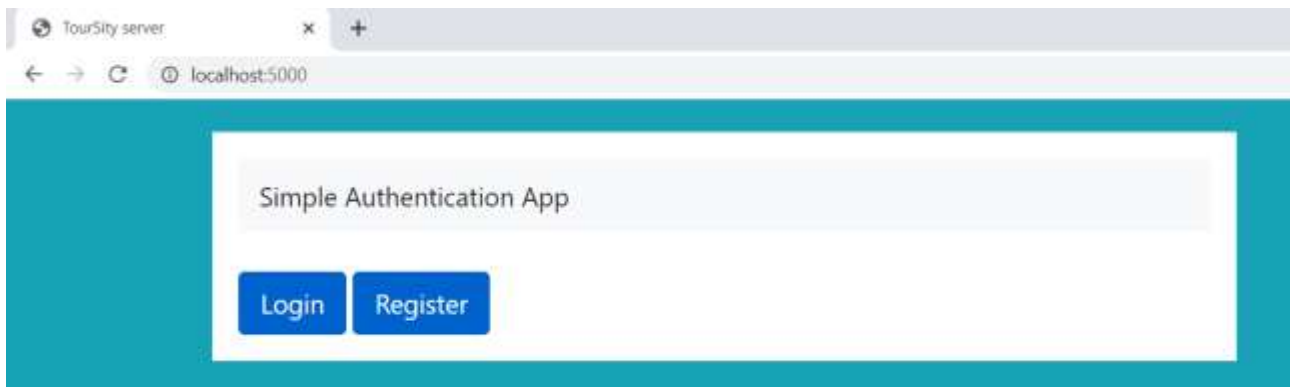


Рис. 2.13. Вигляд home.hbs

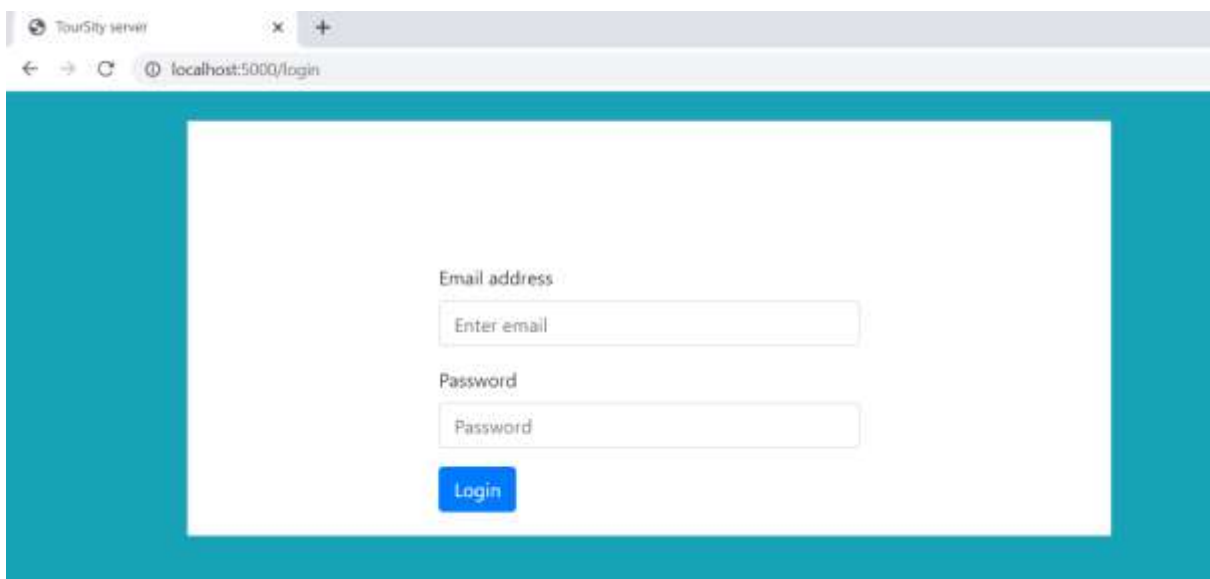


Рис. 2.14. Сторінка login

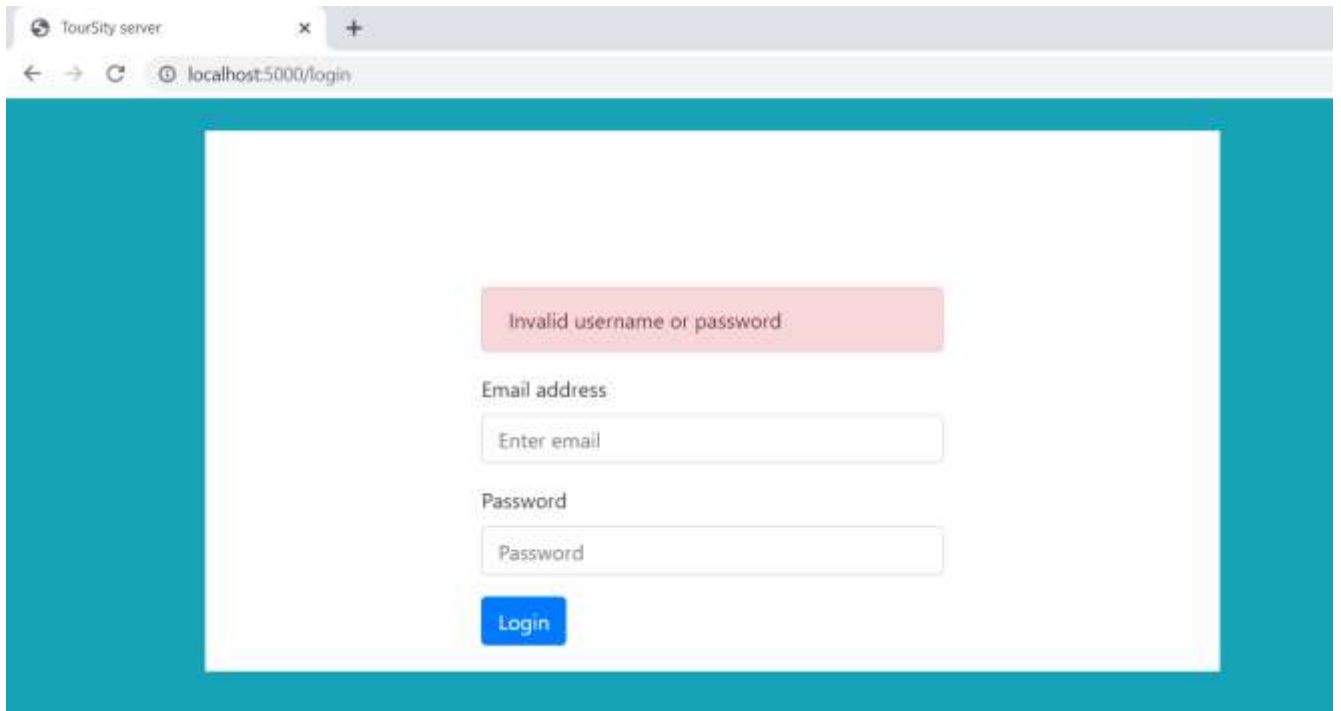


Рис. 2.15. Виведення повідомлення щодо помилки на сторінці login

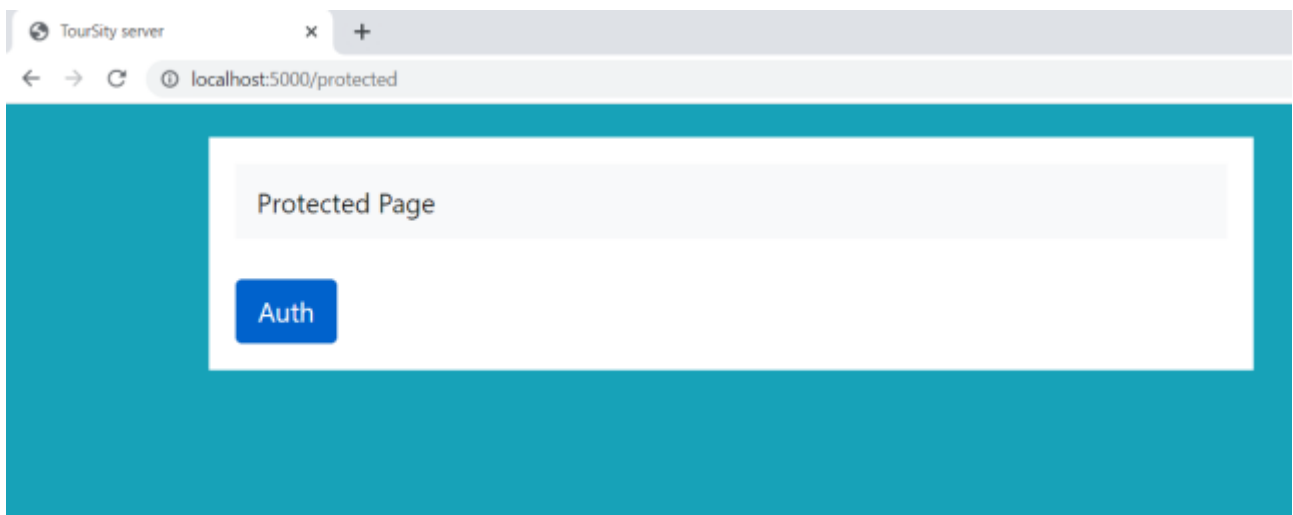


Рис. 2.16. Перехід на сторінку опрацювання даних Auth

- зупиняємо проект:

Ctrl + C.

2.4. РОЗРОБКА АРХІТЕКТУРИ КОНТЕНТУ WEB-ДОДАТКУ

2.4.2. Практичний модуль

У практичній частині продовжуємо проектну розробку web-додатку Tours City (екскурсії містом) у якому можна рекламувати подорожі будь-яким містом у галузі культурного міського туризму.

Етап 2.4. Реалізація бізнес-логіки CRUD контенту Web-додатка на сервері

Дія 2.4.1. Розглянувши технічне завдання на розробку Web-додатка (Етап 1. Дія 1.1.1) виконаємо моделювання потоків даних контенту з використанням діаграми DFD (рис. 2.19).

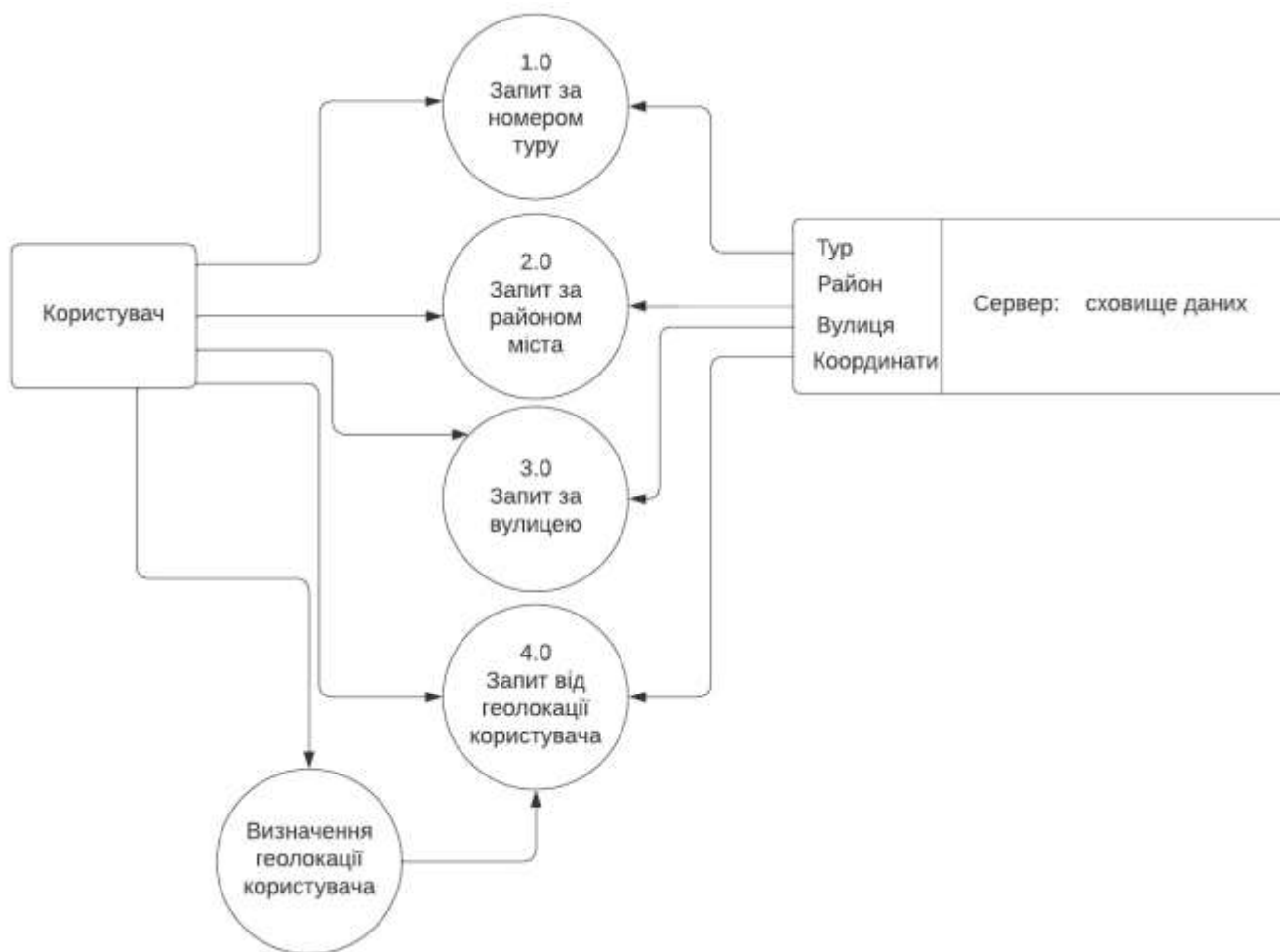


Рис. 2.19. DFD діаграма 2-го рівня

Дія 2.4.2. Розглянувши вимоги до контенту web-додатка на рівні front-end (Етап 1. Дія 1.1.2) формуємо склад колекції Destination у базі даних. Для цього потрібно створити у БД колекцію контенту, якій відповідатиме модель

destination.model.js у архітектурі MVC, що буде розміщена в папці models. Створюємо файл destination.model.js та дописуємо в середині файлу код. Код повинен виглядати як на лістингу 2.10.

Лістинг 2.10 : Вміст файлу destination.model.js

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;
const destinationSchema = new Schema({
    // назва дестинації
    destName: String,
    // район в якому розміщена дестинація
    regionName: String,
    // вулиця на який розміщена дестинація
    streetName: String,
    // географічна координата широта
    destLat: String,
    // географічна координата довгота
    destLon: String,
    // лічильник кількості переглядів користувачами
    destShow: Number,
    // номер туру до якого входить дестинація
    turNumber: Number,
    // опис дестинації ( short - скорочено, long – розлого,
    // path - як дістатись)
    destShort: String,
    destLong: String,
    destPath: String,
    // фото дестинації
    image: String
});
const Destination = mongoose.model('Destination', destinationSchema);
module.exports = Destination;
```

кінець лістингу 2.10

Дія 2.4.3. Для управління функціями CRUD контенту створюємо файл destinationController.js в папці controllers відповідно до архітектури MVC. Основна відмінність від попередньо створеного контролеру authController.js

полягає у тому, що за бізнес-логікою CRUD контенту в контролері `destinationController.js` потрібно завантажувати на сервер додатковий ресурс у вигляді графічних файлів та при необхідності змінювати завантажені файли з їх видаленням з сервера. Для завантаження файлів на сервер будемо використовувати модуль Node JS `multer`, а для видалення модуль `fs`.

У консолі вводимо:

```
npm install multer fs
```

В середині файлу прописуємо код з функціями CRUD аналогічно до дії 2.2.4 за прикладом лістингу 2.4 з врхуванням ключів колекції моделі `destination.model.js`.

Дія 2.4.4. Для завантаження на сервер графічних файлів та їх видалення з серверу в `destinationController.js` у методі `router.post` прописуємо такий код:

```
// для використання модуля multer
const upload = multer({ dest: 'public/files/' });
// метод post
router.post('/', upload.single('inputFile'), function (req, res, next) {
  // Новий запис до БД
  if (req.body._id == "")
  {
    var destination = new Destination();
    destination.destName = req.body.destName;
    // ... завантаження даних за ключами моделі destination.model.js
    destination.destPath = req.body.destPath;
    destination.image = req.file.filename;
    destination.save((err, doc) => {
      if (!err)
        res.redirect('destination/list');

      else {
        if (err.name == 'ValidationError') {
          handleValidationError(err, req.body);
          res.render("destination/addOrEdit", {
            viewTitle: "Insert Destination",
            destination: req.body
          });
        }
        else
          console.log('Error during record insertion : ' + err);
      }
    });
  }
});
```

```

    });

    if (!req.file) {
        return res.send('Please select an image to upload');
    }
}
else // Оновлення існуючого запису по _id в БД
{
    if (req.body.flag=="on") { // Перевірка прапорця заміни файлу
// Перезапис img та текстових даних по _id в БД
    Destination.findOneAndUpdate({ _id: req.body._id }, req.body, { new: true },
(err, doc) => {});
// Пошук старого імені та видалення файлу з сервера
    Destination.findOne({ _id: req.body._id }, (err, doc) => {
        console.log(doc.image);
// видалення файлу з використанням модуля fs
        fs.unlink('./public/files/'+ doc.image, function(err){
            if(err) return console.log(err);
            console.log('file deleted successfully');
        });
        doc.image=' ';
    });
// Заміна імя файлу та завантаження
    Destination.findOneAndUpdate({ _id: req.body._id }, { $set: { image:
req.file.filename } }, { new: true }, (err, doc) => {
        doc.image=' ';
        if (!err) {
            res.redirect('destination/list'); }
        else {
            if (err.name == 'ValidationError') {
                handleValidationError(err, req.body);
                res.render("destination/addOrEdit", {
                    viewTitle: 'Update Destination',
                    destination: req.body
                });
            }
            else
                console.log('Error during record update : ' + err);
        }
    });
});

```

```

    }
    // Перезапис тільки текстових даних по _id в БД
    else Destination.findOneAndUpdate({ _id: req.body._id }, req.body, { new:
true }, (err, doc) => {
        // res.json (doc.image);
        if (!err) {
            res.redirect('destination/list'); }
        else {
            if (err.name == 'ValidationError') {
                handleValidationError(err, req.body);
                res.render("destination/addOrEdit", {
                    viewTitle: 'Update Destination',
                    destination: req.body
                });
            }
            else
                console.log('Error during record update : ' + err);
        }
    });
}
});

```

Дія 2.4.5. Для управління доступом до CRUD контенту для переходу на відповідні сторінки в методах `router.get` прописуємо перевірку токена логізації аналогічно до дії 2.3.7.

Дія 2.4.6. Для завантаження графічних файлів у корні проекту web-додатка створюємо папку `public`. Відкриваємо файл `app.js` та прописуємо шлях до цієї папки з використанням методу `express.static`:

```
app.use('/static', express.static(path.join(__dirname, 'public')))
```

Дія 2.4.7. Створюємо у папці `views` папку `destination`, в ній файл `list.hbs`. Відкриваємо файл та дописуємо в середині файлу код. Код файлу `list.hbs` повинен виглядати як на лістингу 2.11.

Лістинг 2.11 : Вміст файлу `list.hbs`

```

<h3><a class="btn btn-secondary" href="/destination"><i class="fa fa-plus"></i>
Create New</a> Destination Data</h3>
<table class="table table-striped">

```

```

<thead>
  <tr style="font-size:12px" >
    <th>Destination Name</th>
    <th>Street Name</th>

```

Продовження лістингу 2.11

```

    <th>Tur Number</th>
    <th>Short Descript</th>
    <th>Image</th>
    <th></th>
  </tr>
</thead>
<tbody>
  {{#each list}}
    <tr style="font-size:14px" >
      <td>{{this.destName}}</td>
      <td>{{this.streetName}}</td>
      <td>{{this.turNumber}}</td>
      <td>{{this.destShort}}</td>
      <td>
        </td>
      <td>
        <a href="/destination/{{this._id}}"><i class="fa fa-pencil fa-lg" aria-
hidden="true"></i></a>
        <a href="/destination/delete/{{this._id}}"
onclick="return confirm('Are you sure to delete this record ?');">
<i class="fa fa-trash fa-lg" aria-hidden="true"></i></a>
      </td>
    </tr>
  {{/each}}
</tbody>
</table>

```

кінець лістингу 2.11

Дія 2.4.8. Відкриваємо файл `protected.hbs` та дописуємо в середині файлу код. Код файлу `protected.hbs` повинен виглядати як на лістингу 2.12.

Лістинг 2.12 : Вміст файлу `protected.hbs`

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Protected Page</a>
</nav>
<div style="margin-top: 30px">
  <a class="btn btn-primary btn-lg active" href="/auth/list">Auth</a>
  <a class="btn btn-primary btn-lg active" href="/destination/list">Destination</a>
</div>
```

кінець лістингу 2.12

Дія 2.4.9. Для введення даних контенту потрібна окрема сторінка веб-додатку. Створюємо файл `addOrEdit.hbs` та дописуємо в середині файлу код. Код файлу `addOrEdit.hbs` повинен виглядати як на лістингу 2.13.

Лістинг 2.13 : Вміст файлу `addOrEdit.hbs`

```
<h3>{{ viewTitle }}</h3>
<form action="/destination" method="POST" enctype="multipart/form-data"
autocomplete="off">
  <input type="hidden" name="_id" value="{{ destination._id }}">
<div class="form-group">
  <label>Destination Name</label>
  <input type="text" class="form-control" name="destName"
value="{{ destination.destName }}">
</div>
<div class="form-group">
  <label>Street Name</label>
  <input type="text" class="form-control" name="streetName"
value="{{ destination.streetName }}">
</div>
<div class="form-group">
  <label>Destination Lat</label>
```



```

        <input type="text" class="form-control" name="destLat"
value="{{ destination.destLat }}">
    </div>

```

Продовження лістингу 2.13

```

<div class="form-group">
    <label>Destination Lon</label>
    <input type="text" class="form-control" name="destLon"
value="{{ destination.destLon }}">
</div>
<div class="form-group">
    <label>Tur Show</label>
    <input type="text" class="form-control" name="destShow"
value="{{ destination.destShow }}">
</div>
<div class="form-group">
    <label>Tur Number</label>
    <input type="text" class="form-control" name="turNumber"
value="{{ destination.turNumber }}">
</div>
<div class="form-group">
    <label>Short Description</label>
    <textarea class="form-control" style="font-size:14px" name="destShort" rows="4">
        {{ destination.destShort }}</textarea>
</div>
<div class="form-group">
    <label>Long Description</label>
    <textarea class="form-control" style="font-size:14px" name="destLong" rows="4">
        {{ destination.destLong }}</textarea>
</div>
<div class="form-group">
    <label>Destination Path</label>
    <textarea class="form-control" style="font-size:14px" name="destPath" rows="4">
        {{ destination.destPath }}</textarea>
</div>

```

```
<div class="mb-3">
  <div>
    
    <input type="checkbox" name="flag"> Відмітьте для заміни файлу </div>
  <label for="image" class="form-label" >Select Image</label>
    <input class="form-control form-control-lg" type="file" name="inputFile"
value="{{ destination.image }}" />
  </div>
<div class="form-group">
  <button type="submit" class="btn btn-info"><i class="fa fa-database"></i>
Submit</button>
  <a class="btn btn-secondary" href="/destination/list"><i class="fa fa-list-
alt"></i> View All</a>
</div>
</form>
```

кінець лістингу 2.13

Дія 2.4.10. Виконуємо тестовий запуск сервера проекту

- у консолі вводимо:

npm run server

- у браузері вводимо:

localhost:5000

- пройдіть авторизацію користувача (за попередньо створеними логіном та паролем).

- виконуємо послідовність дій з операціями CRUD контенту на сторінках-виглядах back-end. Переходимо на сторінку колекцій БД (рис. 2.20).

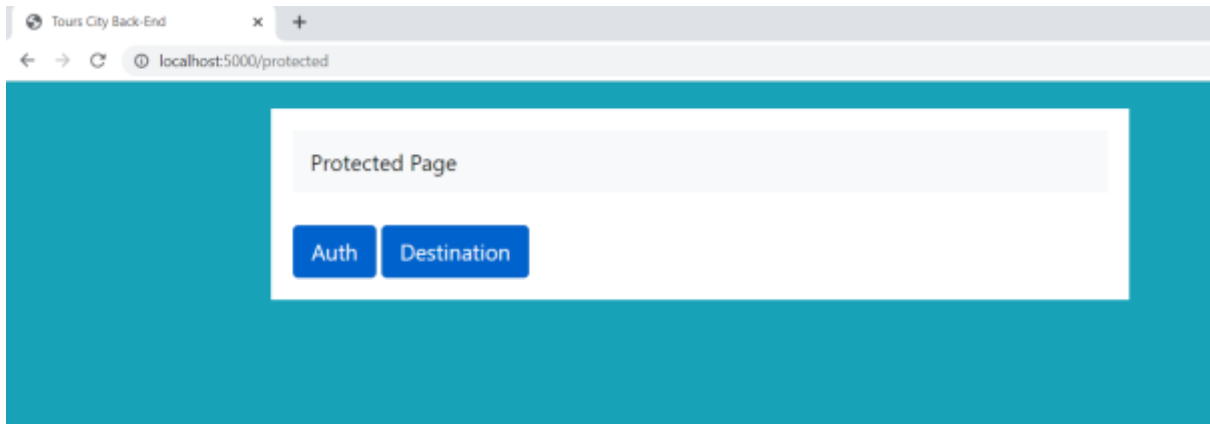


Рис. 2.20. Вигляд protected.hbs

- на сторінці вигляду list.hbs (рис. 2.22) активізувавши Create New переходимо на сторінку введення даних (рис. 2.21).

Tours City Back-End

localhost:5000/destination/f62dda351d62934064da6c02b

Update Destination

Destination Name

Замок / Любарта

Street Name

Кафедральна, 1А

Destination Lat

50.7390205855821

Destination Lon

25.323295440317835

Tur Show

0

Tur Number

1

Short Description


Верхній замок Луцька, один із двох збережених замків міста, пам'ятка архітектури та історії національного значення.

Long Description

Один з найбільших, найдавніших і найкраще збережених в Україні замків. Головний об'єкт історико-культурного заповідника «Старий Луцьк», культурний осередок та найстаріша споруда Луцька. Луцький замок не з'явився на порожньому місці. Його передвісником заведено вважати невелике поселення, що

Destination Path

Старе місто.

 Відмітьте для заміни файлу

Select Image

Вибрати файл Файл не вибрано

Submit View All

Рис. 2.21. Вигляд addOrEdit.hbs з введеними даними

- На сторінці вигляду addOrEdit.hbs натиснувши клавiшу Submit переходимо на сторiнку збережених даних (рис. 2.22).

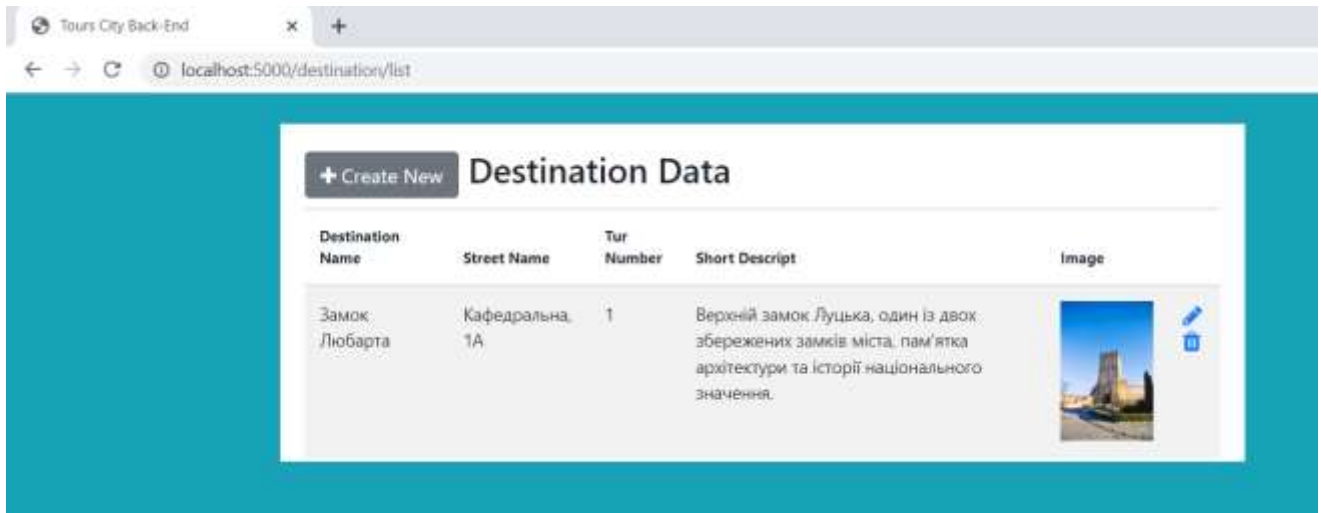


Рис. 2.22. Контент з вигляду list.hbs до введення даних

- зупиняємо проект:
Ctrl + C.

2.5. REST API. ПЕРЕДАВАННЯ ДАНИХ ІЗ WEB-СЕРВЕРА НА БІК КЛІЄНТА

2.5.2. Практичний модуль

У практичній частині продовжуємо проектну розробку web-додатку Tours City (екскурсії містом) у якому можна рекламувати подорожі будь-яким містом у галузі культурного міського туризму.

Етап 2.5. Побудова REST API сервера

Дія 2.5.1. Відкриваємо файл `app.js` та прописуємо зв'язок із запитамі API для майбутнього front-end:

```
app.use("/api", require("./api"))
```

та вказуємо на керуючий контролер

```
app.use('/api/destination/', destinationController)
```

Дія 2.5.2. У Visual Studio Code в корні проекту, аналогічно Дії 2.1.5, створюємо файл `api.js`. Після чого вписуємо код:

```
const express = require("express");
const router = express.Router();
const Destination = require("../server/models/destination.model")
router.get("/destinations", (req, res)=>{
  Destination.find({ })
    .then(destination => {
```

```

    res.send(destination);
  });
});
module.exports = router;

```

Дія 2.5.3. Відкриваємо файл `\server\views\protected.hbs` та прописуємо зв'язок з колекцією із запитом API:

```

<div style="margin-top: 30px">
  <a class="btn btn-primary btn-lg active" href="/auth/list">Auth</a>
  <a class="btn btn-primary btn-lg active"
    href="/api/destination/list">Destination</a>
</div>

```

Дія 2.5.4 Виконуємо тестовий запуск сервера проекту

- у консолі вводимо:

```
npm run server
```

- у браузері вводимо:

```
localhost:5000
```

- впевнюємось, що сервер запущено.

Дія 2.5.5. Виконуємо тестування REST API

- відкриваємо командний рядок, натиснувши Win + R

- вводимо:

```
cmd
```

- у вікні командного рядка вводимо:

```
curl -X GET "http://localhost:5000/api/destinations/"
```

- отримуємо відповідь у форматі JSON (рис. 2.23):

```

[{"_id":"62dda351d62934064da6c02b","destName":"Бул. Шевченка",
  "streetName":"Бул. Шевченка", "destLat":"50.7390205855821",
  "destLon":"25.323295440317835","destShow":0,"turNumber":1,"destShort":
  "..."}]

```

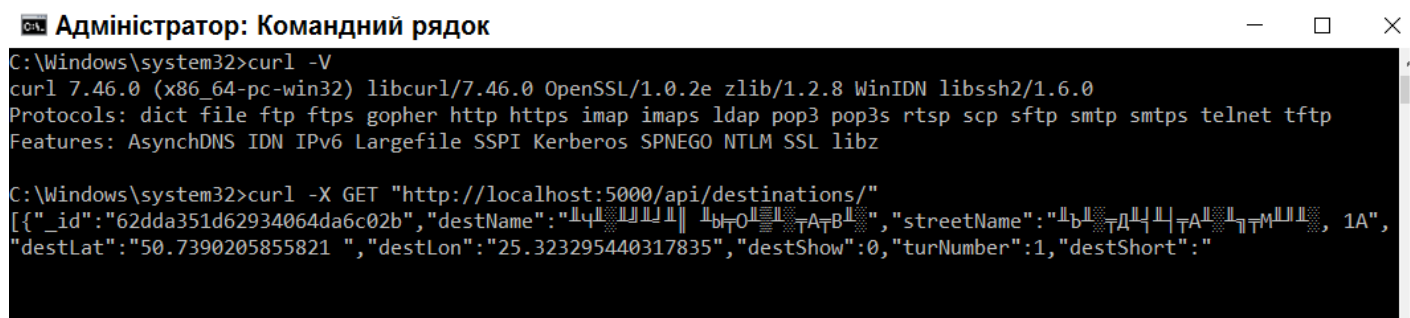


Рис. 2.23. Приблизна відповідь на тестування REST API