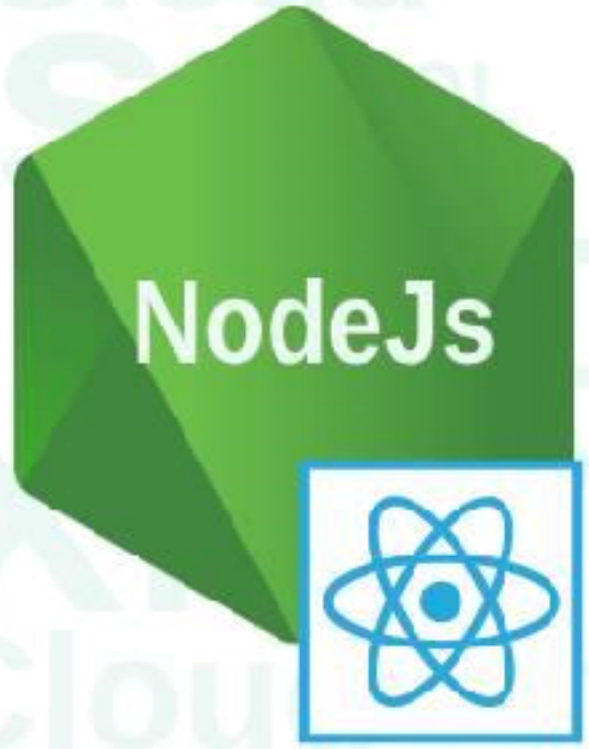
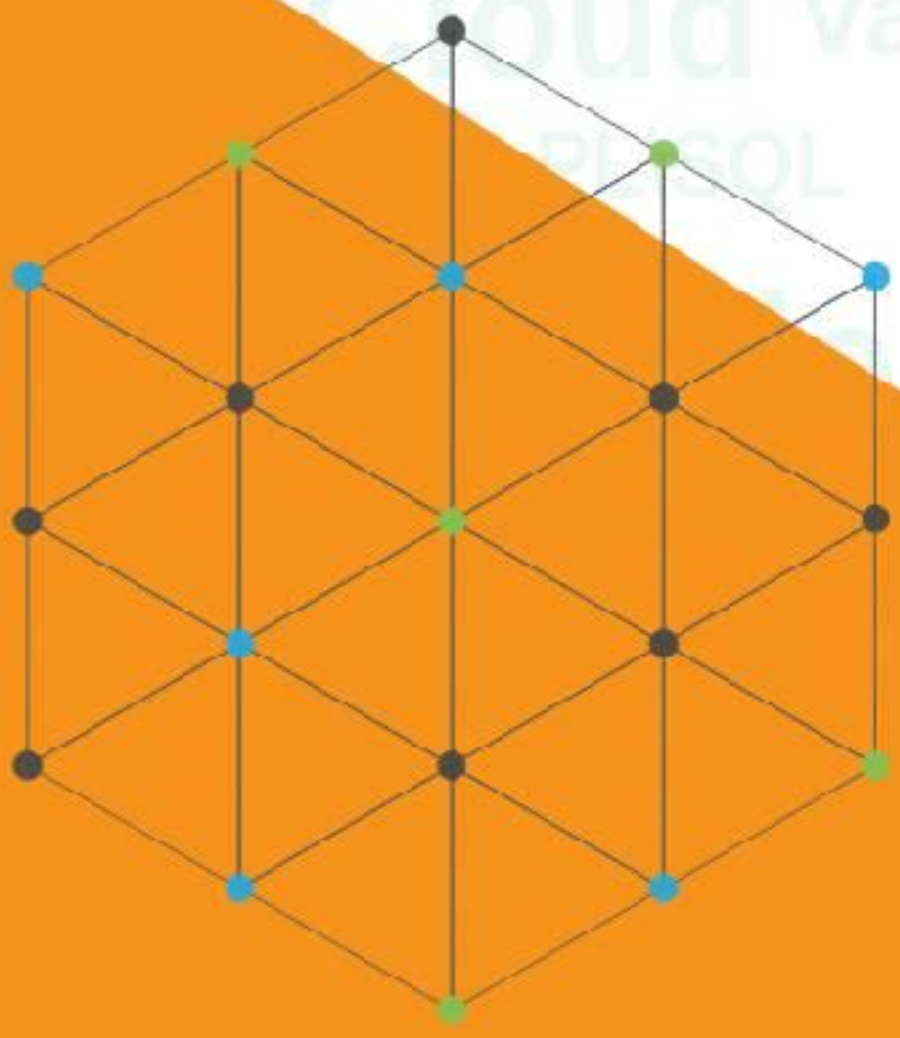
Юрій Тулашвілі

WEB ПРОГРАМУВАННЯ

MERN fullstack development

Навчальний посібник





Міністерство освіти і науки України
Луцький національний технічний університет

Юрій Тулашвілі

WEB-ПРОГРАМУВАННЯ

MERN fullstack development

Навчальний посібник

Рекомендовано Луцьким національним технічним університетом

РЕДАКЦІЙНО-ВИДАВНИЧИЙ ВІДДІЛ
ЛУЦЬКОГО НАЦІОНАЛЬНОГО ТЕХНІЧНОГО УНІВЕРСИТЕТУ

2023

УДК 004.5+378.1
ББК 74.58
Т 47

Тулашвілі Юрій. Web-програмування. MERN fullstack development: навчальний посібник. Луцьк: РВВ ЛНТУ, 2023. 242 с.

У навчальному посібнику подано основні теоретичні засади та практичні прийоми web-програмування з використанням платформи Node.js. Розкрито підходи до організації бізнес-логіки та формування системних вимог на створення web-додатків. Окреслено засади побудови сайтів за допомогою html, css. Розглянуто основні прийоми програмування на мові JavaScript на рівні full stack development.

Розкрито принципи структурної побудови web-додатків на рівні full stack development. Для цього окремо подано засади розробки на серверному рівні - back-end з використанням фреймворка Express.js за архітектурою MVC та реалізацією операцій опрацювання даних CRUD. Приклад реалізації web-додатку на стороні клієнта - front-end виконаний із застосуванням бібліотеки React.js для розробки інтерфейсів користувача. Розкрито підходи до авторизації та захисту інформації на сервері, підключення REST API.

Призначено для студентів та викладачів IT-спеціальностей, фахівців інженерного напрямку з IT. Матеріал посібника також може бути корисним для самостійного набуття навичок з web-програмування.

Рецензенти:

Ю.В. Турбал, доктор технічних наук, професор, завідувач кафедри комп'ютерних наук та прикладної математики Національного університету водного господарства та природокористування

Р.М. Горбатюк, доктор педагогічних наук, професор, завідувач кафедри машинознавства та комп'ютерної інженерії Тернопільського національного педагогічного університету ім. В.Гнатюка

Ю.П. Фірс, керівник міжнародної компанії з комп'ютерного програмування ТОВ "ВЕТЕЛО"

Рекомендовано рішенням вченої ради Луцького національного технічного університету (протокол № 12 від 30 червня 2023 року)

ЗМІСТ

ПЕРЕДМОВА

Цифрова трансформація бізнес-моделей діяльності людства безпосередньо пов’язана з перенесенням бізнес-процесів та відповідних бізнес-правил на рівень цифрових технологій. Цей процес - діджиталізація (digitalization – англійською оцифровування) визначається як сучасна найбільш значуща технологічна тенденція, що змінює як сучасний бізнес, так і суспільство безпосередньо.

В цьому випадку діджиталізація – є результатом діяльності IT-фахівців, що у процесі оцифровування з використання різних мов програмування створюють програмні додатки для бізнес-процесів. Програмування давно вже стало галуззю, яка об'єднує під однією назвою безліч напрямків діяльності IT-фахівців.

Web-програмування - галузь розробки web-додатків, у завдання якої входить проектування дизайну користувальницьких інтерфейсів для web-сайтів або web-додатків, розробка засобів передавання, обміну та накопичення інформації у сховищах даних на серверах.

Web-програмування безпосередньо поділяється два види: клієнтське (Front-end) та серверне (Back-end). За допомогою Front-end програмування створюється інтерактивний інтерфейс: меню сайту, керуючи кнопки, поля для введення інформації, місця для виведення малюнків, динамічні слайдери, таблиці, списки тощо - тобто весь зовнішній вигляд web-сайту або web-додатку. Back-end програмування реалізує створення програмно-адміністративної частини проекту - формує архітектуру, реалізує бізнес-логіку, забезпечує зв’язок зі сховищами даних на серверах шляхом написання програмного код на стороні сервера.

Навчальний посібник “Web-програмування: MERN fullstack development” присвячений висвітленню аспектів розробки web-додатків з використанням платформи Node.js, що є розділом навчальної дисципліни “Web-програмування (Cloud Computing)”, яка викладається студентам спеціальності “Комп’ютерні науки” та є основною з формування компетентностей з Web-програмування. Розкрито технології розробки з використанням ресурсів стеку MERN. Подано засади розробки на серверному рівні - back-end з використанням фреймворка Express.js за архітектурою MVC та реалізацією операцій опрацювання даних CRUD. Приклад реалізації web-додатку на стороні клієнта - front-end виконаний із застосуванням бібліотеки React.js для розробки інтерфейсів користувача.

Розкрито підходи до авторизації та захисту інформації на сервері, підключення REST API.

Посібник містить чотири розділи, які повністю розкривають основні засади програмування web-додатків з використанням Node.js. У навчальному посібнику наведені теоретичні основи програмування web-додатків в обсязі, що є достатнім для оволодіння основними методами web-програмування, такими як: програмування на рівнях backend та frontend, застосування сучасних систем управління базами даних (MongoDB), програмування web-ресурсів у вигляді односторінкових SPA та багатосторінкових MPA web-додатків.

Посібник містить, поряд із поданням теоретичних викладів конкретні приклади виконання практичних задач на рівнях: проектування бізнес аналітики, програмування backend та frontend, які розкривають послідовність дій web-програмування з використанням платформи Node.js.

Кожен розділ навчального посібника поряд з теоретичним та практичним змістовими модулями містить тестові завдання для самоконтролю рівня вивчення матеріалу, перелік самостійних завдань, а також список використаної літератури. Матеріал практичних модулів дозволяє студенту отримати достатньо вичерпну інформацію щодо технології створення web-додатків.

Матеріали посібника ґрунтуються на довголітніх результатах та особистому досвіді автора, набутому при викладанні освітньої компоненти „Web-програмування". Структура посібника побудована таким чином, що кожний розділ дозволяє отримати знання з основних напрямків діяльності у web-програмуванні з використанням платформи Node.js та набути уміння створення web-додатків. Поряд із викладенням теоретичного матеріалу, поданого як теоретичні модулі, у посібнику наведений наскрізний приклад виконання практичних дій, в яких розкривається послідовність web-програмування у стеці MERN. Наприкінці кожного розділу подані тестові завдання для самоконтролю, за допомогою яких можна перевірити повноту засвоєння матеріалу.

Посібник призначений для студентів спеціальностей галузі знань 12 - Інформаційні технології. У ньому застосована сучасна форма розкриття особливостей навчального матеріалу, що забезпечить його успішне використання в навчальному процесі підготовки майбутніх IT-фахівців. За результатами ознайомлення та вивчення матеріалу посібника студенти зможуть під керівництвом викладача або самостійно навчитись створювати сучасні web-додатки. Матеріал посібника також може бути корисним тим фахівцям, які бажають самостійно набути навичок проектування web-додатків з використанням платформи Node.js.

ПОДЯКИ

Основний період написання цього навчального посібника припав на моє стажування у міжнародній компанії з комп'ютерного проектування ТОВ "ВЕТЕЛО". Колектив компанії - це конгломерат фахівців з web-програмування, які володіють знаннями передових технологій і принципів розробки, професіоналізмом і високими навичками, а також здоровим глуздом, діловою етикою і почуттям гумору. Ось як самі фахівці компанії оцінюють себе та свій потенціал: “Wetelo - Web, Technolgy, Logic. На перший погляд, може здатися, що ми як і інші компанії застрягли у всесвітній павутині та пропагуємо лише логічне мислення та останні тех тренди. Стоп, насправді, це не так! SaaS, Web3, Open AI та інші штуки ще не затьмарили наше бажання ділитися з бізнесовим світом та й не тільки, інноваційними та творчими ідеями та рішеннями, приправленими шаленими амбіціями, креативністю та безкінечним потенціалом нашої команди. Ми допомагаємо компаніям відходити від шаблонів і традиційних бізнесових підходів та надихаємо творити та змінювати цей світ на краще. Якщо ми створюємо чатбот на основі GPT-3, то він буде чудовою альтернативою людині-коучу і зміцнюватиме моральний дух працівників. Якщо ми пропонуємо крипто платформи на основі DeFi, то ми піклуємося про фінансову безпеку в цілому, яка можлива в умовах децентралізації. Ми дбаємо про навколишнє середовище, створюючи додатки для сортування сміття. Ми не боїмося викликів, тому викладаємося по максимуму, щоб залишатися унікальними. Wetelo - це смілива і натхненна команда, яка творить, надихає, волонтерить, воює та не зупиняється на досягнутому”.

Безперечно, я розумію, що в компанії працюють не тільки ті, кого я зараз згадаю персонально, кожен з вас своїми порадами допомагали у створенні моєї книги, дякую вам! Ось ті, з ким я особисто мав справу. Рушійною силою книги були мої кураторі стажування у компанії: її керівник Юрій Фірс та провідний фахівець Сергій Войтович, технічний директор Микитюк Віталій та Роман Вакульчук - senior розробник, який є випускником ЛНТУ. Дякую їм за чудові ідеї та позитивні відгуки.

Я вдячний колективу компанії ТОВ "ВЕТЕЛО"!

Розділ 1. ОСНОВИ ПРОЕКТНОЇ ДІЯЛЬНОСТІ WEB ПРОГРАМУВАННЯ

1.1. ФОРМУВАННЯ ВИМОГ НА СТВОРЕННЯ WEB-ДОДАТКУ. ПОБУДОВА ШАБЛОНУ ЗА ДОПОМОГОЮ HTML, CSS, JAVASCRIPT

1.1.1. Теоретичний модуль

Проектування web-додатків

Створення web-додатку завжди починається з визначення його призначення. Основна задача будь-якого успішного проектування полягає в тому, що під час створення web-додатку потрібно реалізувати такі основні проектні завдання:

- детально ***визначити предметну галузь*** (англ. application domain), тобто сукупність об'єктів, понять, зв’язків, відношень і способів перетворення та взаємодії цих об'єктів під час розв'язування задач, що відносяться до певної сфери людської діяльності [1];

- максимально ***розкрити функціональне наповнення***, що відбиває специфіку предметної галузі, як сукупність конструктивних елементів (модулів), з яких складається web-додаток [2]. Функціональне призначення компонентів формує багаторівневе представлення завдань web-додатку ***як реалізацію бізнес-логіки***, що відображає інтерфейси, функції транслювання, зв’язки із базою даних та маршрутизацію подання інформації. Функціональне наповнення розкривається у формі моделі предметної галузі;

- повністю ***усвідомити ключові поняття***, що розкриваються термінами (словами), як засіб інтерфейсної взаємодії (спілкування) користувача з web-додатком. ***Мова завдань*** подається у вигляді репозітарію як опису завдань, що реалізуються інформаційною системою у формі web-додатку. Саме через розкриття термінології мови завдань користувач сприймає й оцінює, які є послуги й наскільки зручні вони у використанні;

- раціонально ***обрати технологію розробки*** проекту, виконати кодування з подальшим тестуванням;

- ***здійснити верифікацію***, тобто перевірку відповідності результатів проектування та розробки до визначених вимог у технічному завданні проекту.

Технічне завдання (ТЗ) - сукупність документів, які визначають технологічний процес виготовлення виробу [3]. У нашому випадку вміщують опис предметної галузі, функціонального наповнення, мови завдань та системних вимог (технології, мови програмування тощо).

Моделювання предметної галузі та ***функціонального наповнення*** здійснюється з використанням уніфікованої нотації, заснованій на застосуванні Уніфікованої Мови Моделювання (Unified Modeling Language, UML).

Для розкриття процесу проектування на етапі визначення функціонального наповнення інформаційних систем Мартін Фаулер окреслив ***три шаблони з організації бізнес-логіки*** [4]:

- сценарій транзакцій

Бізнес-логіка додатку може сприйматися як послідовність транзакцій. Кожен акт взаємодії клієнта з сервером описується певним фрагментом логіки. Сценарій транзакцій полягає в тому, що в більшості випадків доводиться мати справу з одним сценарієм для кожної транзакції рівня системи бази даних. Одна транзакція здатна модифікувати дані, інша - сприймати їх в структурованому вигляді тощо. Наприклад, якщо користувачеві необхідно замовити номер у готелі, процедура, яка обслуговуватиме транзакцію повинна передбачати дії з:

- перевірки наявності відповідного номера;
- обчислення суми оплати;
- фіксації замовлення в базі даних.

Тобто, ***завдання транзакції*** - отримати вхідну інформацію, опитати базу даних, зробити висновки і зберегти результати. Сценарій транзакцій організовує логіку обчислювального процесу переважно у вигляді єдиної процедури. Розміщувати код сценарію транзакцій є доцільним у різних підпрограмах, а враховуючи принципи об'єктно-орієнтованого проектування - в класах.

- модуль таблиці

Охоплює логіку обробки всіх записів, що зберігаються, або розміщуються у віртуальній таблиці бази даних. Описується мовою моделювання даних Integration DEFinition for information modeling (IDEF1X) (рис. 1.1).

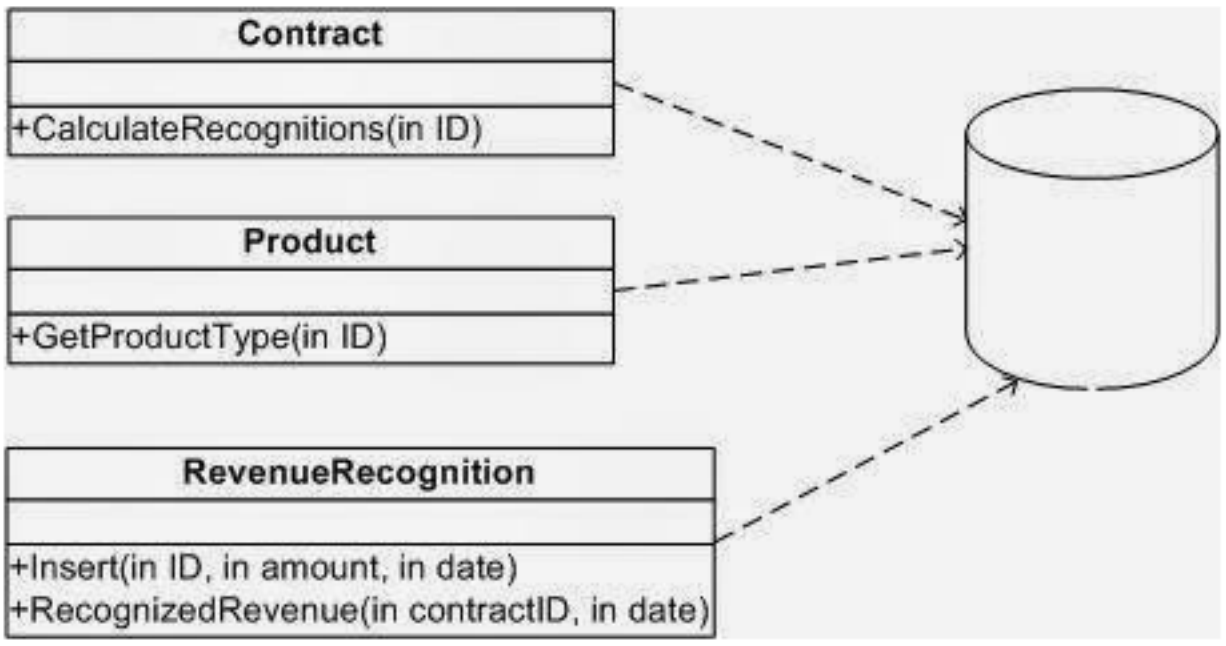


Рис. 1.1. Ілюстрація логіки обробки записів, що зберігаються у таблицях бази даних модулями таблиць [5]

Типове рішення шаблону модуля таблиці передбачає створення по одному класу на кожну таблицю бази даних. Тоді єдиний екземпляр класу містить всю логіку обробки даних таблиці.

- модель предметної галузі

Це об'єктна модель домену (рис. 1.2), що охоплює поведінку (функції) і властивості (дані).

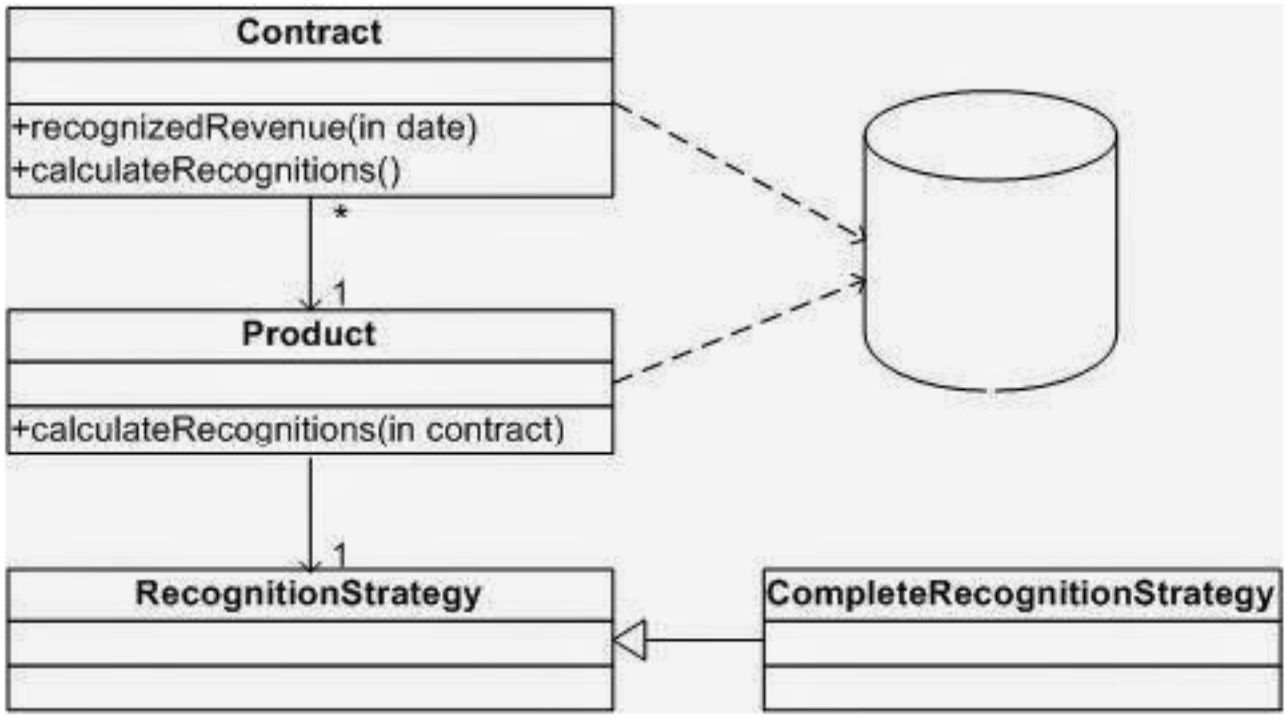


Рис. 1.2. Ілюстрація логіки обробки записів моделі предметної галузі [5]

Типове рішення у вигляді моделі предметної галузі передбачає створення мережі взаємопов'язаних об'єктів (наприклад, архітектурний шаблон **MVC**, де **M**-модель **V**-вигляд **C**-контролер, мова про який піде у розділі 2), кожен з яких представляє якусь осмислену сутність. В цьому випадку, один об'єкт може імітувати елементи даних (модель), якими оперують в додатку, а інші повинні формалізувати ті чи інші бізнес-правила (контролер – поведінку, вигляд – виведення даних).

Побудова бізнес-логіки

Розберемо предметну галузь для побудови бізнес-логіки на прикладі *міської бібліотеки*, де бібліотека є частиною реального світу, а її діяльність - облік наявних книг і видача їх читачам [6].

Створюючи додаток для автоматизації діяльності бібліотеки, необхідно визначити якими даними, він буде маніпулювати. Ось приблизний список **бізнес-даних** взятих з предметної галузі:

- *Назва книги*
- *П.І.Б. автора*
- *Кількість примірників цієї книги*

• *П.І.Б. читача*

• *Дата народження читача*

• *Примірники книг, які є у нього на руках*

• *Абонемент.*

Де і як можна маніпулювати цими даними в програмі?

Дані будуть зберігатися в таких класах-сутності як Читач, Книга, Абонемент тощо. Саме в них і буде міститися велика частина бізнес-логіки.

Будь-яка Бізнес-логіка реалізує **бізнес-правила**.

Бізнес-правило - це положення, яке визначає чи обмежує будь-які сторони бізнесу (предметної галузі). Його призначення - захистити структуру бізнесу, контролювати або впливати на його операції.

Бізнес-правила поділяють орієнтовно на шість основних категорій:

1. ***Бізнес-терміни*** - фундаментальна форма ***бізнес-правила***. Це фрази, слова, абревіатури з предметної галузі (мова завдань). Зазвичай такі терміни зберігаються в документі під назвою «бізнес глосарій» або «словник термінів».

Приклади бізнес-термінів:

• читач; книга;

• ISBN (англ. International Standard Book Number, скорочено - англ. ISBN) - унікальний номер книжкового видання;

• читацький абонемент - документ, де враховуються книги видані читачеві.

2. ***Факти*** (facts) - це вірні твердження про бізнес дії (є частиною функціонального наповнення). Найчастіше вони описують зв'язки і відносини між важливими бізнес-термінами. Факти також є інваріантами - незмінними істинами про сутність даних і їх атрибути. ***Бізнес-правила*** в багатьох випадках можуть посилатися на певні факти, однак останні зазвичай не перетворюються безпосередньо в функціональні вимоги до програмного забезпечення. Відомості про сутність даних, важливих для системи, застосовують в моделях даних, створюваних аналітиком або архітектором бази даних.

Приклади фактів:

• кожна книга повинна мати міжнародний стандартний книжковий номер;

• клієнт оплачує доставку кожного замовлення;

• замовлення повинен містити не менше однієї позиції.

3. ***Обмеження*** (constraints) - визначають, які операції може виконувати система та її користувачі (є частиною функціонального наповнення). Ось деякі слова і фрази, які застосовуються при описі обмежень ***бізнес-правил***: повинен / не повинен, може / не може, тільки тощо.

Приклади обмежень:

• постійний відвідувач бібліотеки може замовити для себе до 10 книг;

• всі комп’ютерні програми бібліотеки повинні відповідати урядовим постановам, що стосуються їх використання людьми з ослабленим зором.

Як правило, існують політики безпеки, що визначають порядок доступу до інформаційних систем. Зазвичай вони констатують, які паролі слід застосовувати, як часто їх необхідно міняти, чи можна застосовувати старі паролі тощо. Всі ці обмеження, що стосуються доступу до додатка, можна вважати ***бізнес-правилами***. Введення кожного такого правила в конкретний код спрощує оновлення систем, необхідне для відповідності їх новим правилам. Наприклад, зміна необхідної частоти оновлення паролів з 90 до 30 днів.

4. ***Активатор операції*** (action enabler) - це бізнес-правило, що призводить до виконання будь-яких дій за певних умов (є частиною функціонального наповнення). Людина може виконувати ці дії вручну. Іноді правило може керувати деякими функціями програми, завдяки яким додаток при виконанні певних умов реалізує потрібну модель поведінки. Вираз виду:

якщо <деяка умова вірна або настала певна подія>, то <щось станеться>
є ключем, який описує активатор операції.

Приклади активаторів:

• якщо клієнт замовив книгу автора, який написав декілька книг, клієнту слід запропонувати інші книги цього автора, перш ніж прийняти замовлення;

• якщо клієнт подав дистанційно коректне замовлення на отримання книги, необхідно зв'язатися з ним по E-mail для узгодження деталей.

5. ***Висновки*** (inference), іноді називають ще можливим знанням - це правила, що встановлюють нові реалії на основі достовірності певних умов. ***Висновок створює новий факт на основі інших фактів або обчислень***. Висновки часто записують у форматі «якщо - то», що застосовується також під час запису бізнес-правил, що активують операції. Проте, розділ «то» виведення містить в собі факт або припущення, але не дію.

Приклади висновків:

• якщо постачальник не може поставити книгу протягом п'яти днів з моменту отримання замовлення, замовлення вважається невиконаним;

• якщо привезена під замовлення книга не була забрана клієнтом протягом 10 днів, книга вважається такою, що є у вільній наявності;

• якщо виставлений замовнику рахунок на книгу не був сплачений протягом трьох діб з моменту подання замовлення, рахунок вважається простроченим.

6. ***Обчислення*** (computations) представляють собою ще один тип ***бізнес-правил***, що реалізуються з використанням ***математичних формул*** або ***алгоритмів***. Багато обчислення виконуються за зовнішніми для підприємства правилам, наприклад, за формулами утримання прибуткового податку. На відміну від активаторів, реалізація яких передбачає створення специфічних функціональних вимог до системи, правила обчислень в тій формі, в якій вони виражені, можна розглядати в якості вимог до програмного забезпечення.

Приклади бізнес-правил для обчислень на прикладі придбання товару (як варіант, їх можна представити в символьній формі, наприклад у вигляді математичного виразу) [6]:

• ціна одиниці товару знижується на 10% при замовленні від 6 до 10 одиниць, на 20% - при замовленні від 11 до 20 одиниць і на 30% - при замовленні понад 20 одиниць;

• вартість автомобіля «Opel» в салоні розраховується шляхом додавання: вартості базової моделі, вартості обраних покупцем пакетів, вартості опцій, за мінусом знижки при повторній покупці в салоні, і знижки корпоративного клієнта, але за умови, що продаж не реалізується по акції «КАСКО в подарунок".

Окреме обчислення може включати безліч елементів, однак у останньому прикладі для розрахунку загальної вартості автомобіля використовується алгоритм підрахунку суми, знижки та додаткові умови. Таке правило є досить заплутаним і складним для розуміння. Щоб позбутися цього недоліку, рекомендується писати бізнес-правила на елементарному рівні, а не об'єднувати безліч деталей в одне правило.

Створення web-шаблону за допомогою HTML, CSS, JavaScript

Шаблон web-додатку фокусується на проблемах розміщення контексту і служать орієнтиром для розробників, у плані того, коли, як і для чого застосовувати обрані рішення.

Для вирішення завдань проектування почали використовувати **шаблони проектування** (design patterns). Пізніше цей термін застосували і **до програмного забезпечення**, а через кілька років сфера застосування даного терміну розширилася за рахунок області проектування призначених **для користувача інтерфейсів**.

Досить часто відбувається так, що кілька шаблонів разом застосовуються для створення зручного проектного рішення. Це може стати в нагоді проектувальникам тому, що вони застосовуються спільно доповнюючи один одного, або тому, що вони впливають на застосування деякого шаблону на початковому рівні архітектури. Таке об’єднання отримало назву - **пов'язані шаблони проектування**.

Шаблони мають практичне застосування і відповідають вимогам «гарного» проектування, при цьому вони **втілюють високорівневі принципи і стратегії**. Останнім часом шаблони все частіше стали застосовуватися розробниками користувача інтерфейсів і програмного забезпечення.

Компонентний підхід

Сучасний рівень Web програмування - це компонентний підхід: інтерфейс проекту ділиться на компоненти (кнопки, вкладки, записи тощо), які можна використовувати багато разів, вкладати один в одного, модифікувати. Найпростіше добитися цього - використовувати **методологію БЕМ (Block-Element-Modifier: Блок, Елемент, Модифікатор),** в якій компоненти називаються Блоками.

Зручно ділити вихідний код на кілька папок (по імені компонента), в кожній з яких лежить все, що потрібно компоненту для роботи:

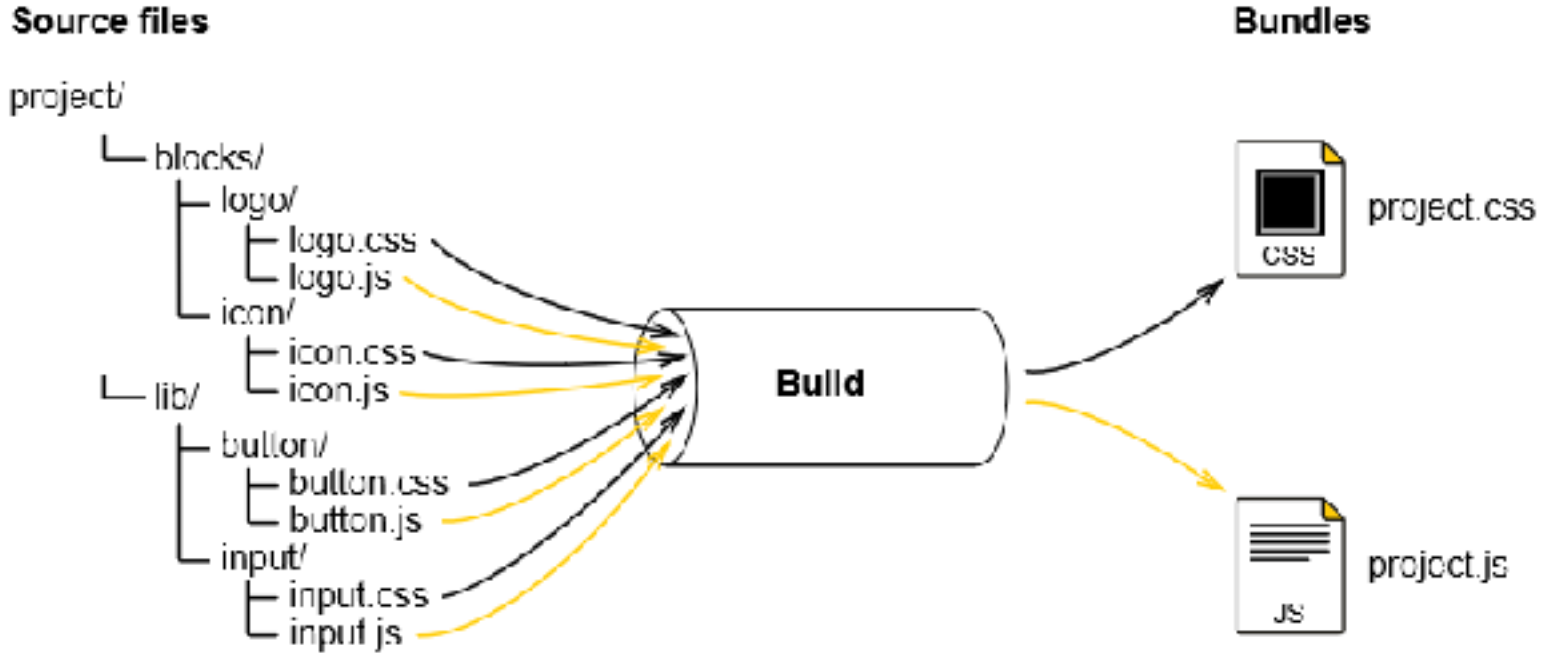


Рис. 1.3. Ілюстрація компонентного підходу БЕМ [7]

Це дозволяє:

Безпечно вмикати або вимикати компоненти. Коли компонент не використовується, його стилі, скрипти та спрайти (об'єкти, що виконують будь-які дії) не завантажуються користувачу (залежить від організації складання).

Мати кілька «збірок» (стильових та інших файлів), що пересилаються до браузера для різних частин проекту. Наприклад, якщо на сайті є публічна і приватна частини, то стилі, скрипти і спрайти у приватній частині проекту можна підключати окремо, щоб не витрачати час на їх завантаження і обробку для неавторизованих відвідувачів.

Безпечно модифікувати компоненти. Якщо змінювати стилі для одного компонента, це не спричинить руйнування стилів для інших компонентів (внаслідок не співпадіння імен елементів).

Швидко знаходити точку редагування. Коли відома назва редагованого блока-компонента, знаходимо файл з тим же ім'ям і редагуємо. Орієнтуватися по коду швидше і простіше, тому що сам файл має менший розмір.

Використовувати компоненти в інших проектах. Базові стилі багатьох елементів (тих же кнопок) однакові від проекту до проекту. Копіюємо блок з стартовою бібліотеки і дописуємо потрібні проекту стилі.

Збирати свою бібліотеку компонентів. Компоненти, з яких складаються сторінки сайтів, часто повторюються: кнопки, списки, поля введення тексту з описом тощо. Тому можна один раз написати для компонента універсальну розмітку, стилі, **JavaScript** для застосування у будь-якому проекті.

БЕМ - методологія web-розробки, що становить основу наборів інтерфейсних бібліотек, фреймворків і допоміжних інструментів.

В основі лежить принцип поділу інтерфейсу на незалежні блоки. Він дозволяє легко і швидко розробляти інтерфейси будь-якої складності та повторно використовувати існуючий код, уникаючи «Copy-Paste».

Об'єктна модель web-сторінок

Об'єктна модель документа (Document Object Model - DOM) – це програмний інтерфейс (API) для HTML документів. DOM утворює структуроване подання документа та визначає те, як ця структура може бути доступна з програм, які можуть змінювати вміст, стиль та структуру документа. Подання DOM складається із структурованої групи блоків (об'єктів), які мають властивості та методи. Фактично, DOM з'єднує web-сторінку з мовами опису сценаріїв чи мовами програмування.

Web-сторінка – це документ. Документ може бути представлений як у вікні браузера, так і в HTML-коді. DOM повністю підтримує об'єктно-орієнтоване представлення web-сторінки, уможливлює її зміну за допомогою мови опису сценаріїв JavaScript.

Відповідно до **DOM** у **БЕМ** блок – це функціонально незалежний компонент сторінки, який може бути повторно використаний. В HTML блоки представлені атрибутом class.

Особливості:

Назва блоку характеризує його призначення («що це?» - «меню»: menu, «кнопка»: button), а не стан («який, як виглядає?» - «червоний»: red, «великий»: big).

Приклад:

<!- Вірно. Семантично осмислений блок `error` ->

 </ div>

<!---------------------------- >

<!- Невірно. Описується зовнішній вигляд ->

 </ div>

Блок не повинен впливати на своє оточення, тобто блоку не слід задавати зовнішню геометрію (у вигляді відступів, кордонів, що впливають на розміри) і позиціонування.

В CSS по БЕМ також не рекомендується використовувати селектори за тегами або id. Таким чином забезпечується незалежність, при якій можливе повторне використання або перенесення блоків з місця на місце.

Принцип роботи з блоками [8]

Вкладеність

Блоки можна вкладати один до одного. Допустима будь-яка вкладеність блоків.

Приклад:

```
<!- Блок "header" ->
<header class = "header">
    <!- вкладений блок "logo" ->
   <div class = "logo"> </ div>
   <!- вкладений блок "search-form" ->
  <form class = "search-form"> </ form>
</ header>
```

Елемент

Складова частина блоку, яка не може використовуватися у відриві від нього.

Особливості:

Назва елемента характеризує його призначення («що це?» - «пункт»: item, «текст»: text), а не стан («який, як виглядає?» - «червоний»: red, «великий»: big).

Структура повного імені елемента **відповідає схемі**:

ім'я-блока__імя-елемента.

Ім'я елемента відокремлюється від імені блоку двома підкресленнями (__).

Приклад

```
<!- Блок "search-form" ->
<form class = "search-form">
    <!- елемент "input" блоку "search-form" ->
    <input class = "search-form__input">
   <!- елемент "button" блоку "search-form" ->
   <button class = "search-form__button"> знайти </ button>
</ form>
```

Принципи роботи з елементами

Вкладеність. Елементи можна вкладати один в одного. Допустима будь-яка вкладеність елементів.

Елемент - завжди частина блоку, а не іншого елемента. Це означає, що в назві елементів не можна прописувати ієрархію виду block__elem1__elem2.

Приклад :

```
<!-   Вірно. Структура повного імені елементів відповідає схемі:
   «Ім'я-блока__імя-елемента»      ->
<form class = "search-form">
   <div class = "search-form__content">
```

```
        <input class = "search-form__input">
        <button class = "search-form__button"> знайти </ button>
    </ div>
</ form>
<!-  Невірно. Структура повного імені елементів не відповідає схемі:
    «Ім'я-блока__імя-елемента»      ->
<form class = "search-form">
    <div class = "search-form__content">
        <input class = "search-form__content__input">
            <!-   рекомендується:
            "search-form__input" або "search-form__content-input"   ->
        <button class = "search-form__content__button"> знайти </ button>
  <!-   рекомендується:
            "search-form__button" або "search-form__content-button"   ->
            </ div>
</ form>
```

Ім'я блоку задає простір імен, яке гарантує залежність елементів від блоку (block__elem).

Блок може мати вкладену структуру елементів в DOM-дереві:

Приклад:

```
 <div class = "block">
    <div class = "block__elem1">
        <div class = "block__elem2">
            <div class = "block__elem3"> </ div>
        </ div>
    </ div>
</ div>
```

Однак ця ж структура блоку в CSS за методологією БЕМ завжди буде представлена плоским списком елементів:

Приклад:

```
.block {}
.block__elem1 {}
.block__elem2 {}
.block__elem3 {}
```

Це дозволяє змінювати DOM-структуру блоку без внесення правок в коді кожного окремого елемента:

Приклад:

```
<div class = "block">
    <div class = "block__elem1">
        <div class = "block__elem2"> </ div>
```

```
    </ div>
    <div class = "block__elem3"> </ div>
</ div>
```

Структура блоку змінюється, а правила для елементів і їх назви залишаються колишніми.

Належність. Елемент - завжди частина блоку і він не повинен використовуватися окремо від нього.

Приклад:

```
<!- Вірно. Елементи лежать всередині блоку "search-form"  ->
<!- Блок "search-form"  ->
<form class = "search-form">
    <!- елемент "input"  блоку "search-form"  ->
    <input class = "search-form__input">
    <!- елемент "button" блоку  "search-form"   ->
    <button class = "search-form__button"> знайти </ button>
</ form>
<!- Невірно. Елементи лежать поза контекстом блоку "earch-form"   ->
<!- блок "search-form"  ->
<form class = "search-form">
</ form>
    <!- елемент "input" блоку  "search-form" ->
    <input class = "search-form__input">
    <!- елемент "button" блоку "search-form" ->
    <button class = "search-form__button"> знайти </ button>
```

Необов'язковість. Елемент - необов'язковий компонент блоку. Не у всіх блоків повинні бути елементи.

Приклад:

```
<!- блок "search-form"  ->
<div class = "search-form">
    <!- блок "input"  ->
    <input class = "input">
    <!-   блок "button"   ->
    <button class = "button"> знайти </ button>
</ div>
```

CSS препроцесор

CSS препроцесор (англ. Cascading Style Sheets preprocessor) - це надбудова над CSS, яка додає раніше недоступні можливості для CSS, за допомогою нових синтаксичних конструкцій.

Основне завдання препроцесора - це надання зручних синтаксичних конструкцій для розробника, щоб спростити, і тим самим, прискорити розробку та підтримку стилів в проектах.

SASS (SCSS)

Найпотужніший з CSS-препроцесорів. Можливості самого препроцесора розширюються за рахунок багатофункціональної бібліотеки Compass, яка дозволяє вийти за рамки CSS і працювати, наприклад, з спрайтами в автоматичному режимі. Має два синтаксиса:

- SASS (Syntactically Awesome Style Sheets) - спрощений синтаксис CSS;
- SCSS (Sassy CSS) - заснований на стандартному для CSS синтаксисі.

Поділ на невеликі файли за компонентним підходом. Коли в одному стильовому файлі описаний лише один компонент (блок), це прискорює сприйняття коду і пошук точки редагування.

Імпорт у CSS

/ * Два рівно можливих варіанти використання * /

1) @import url ("additional-style.css") screen and (orientation: landscape);
2) @import "print.css" print;

Браузер, зустрічаючи імпорт, йде по зазначеному шляху, завантажує і обробляє CSS-файл.

Якщо в одному CSS-файлі написати імпорт іншого файлу, то й перший й другий будуть завантажуватися послідовно, а не паралельно, а це збільшення часу до початку рендера на повільних каналах. Це гірше, ніж написати в розмітці два рази <link>.

В CSS-препроцесорах обсяги імпорту обробляються на етапі компіляції в CSS: якщо ми імпортуємо препроцесорний файл, його вміст «вставляється» замість імпорту. У результаті маємо - один CSS файл, який містить стилі всіх імпортованих файлів.

Імпорт у SCSS

```
@import "btn.scss";  // «вставка» вмісту файлу btn.scss або btn.sass
@import "page-header"; // «вставка» вмісту файлу page-header.scss,
                       //  або page-header.sass, або page-header.css
```

Змінні в CSS-препроцесорах

Препроцесорні змінні працюють тільки на етапі компіляції. Це ніяк не заважає привласнити їх значення в CSS custom properties, щоб використовувати їх же як CSS-змінні.

```
$font-size: 12px;
body {
  font-size: $font-size;
}
```

Робота з псевдокласом: root.

За допомогою нього можна зручно змінювати стилі масово. Наприклад, якщо у вас зазначений один і той же колір в декількох стилях, то для того, щоб його поміняти не потрібно переглядати увесь код.

```
: root {
  --color: red;
}
body {
  margin: 0;
  padding: 0;
}
h2 {
  color: var ( --color);
}
span {
  color: var (  --color);
}
```

Тег призначений для перевизначення малих елементів документа. Наприклад, всередині абзацу (тега <p>) можна змінити колір і розмір першої літери, якщо додати початковий і кінцевий тег і визначити для нього стиль тексту.

Математичні операції

В CSS є calc (). Він підтримується в IE (хоча і не без багів).

```
.tabs {
  width: calc (100% - 3em);
}
h1 {font-size: $font-size * 2; }
h2 {font-size: $font-size * 1.618; }
```

Оскільки це спрацьовує в браузері, в момент виконання попереднього прикладу браузер знає чому дорівнює 3em і ширина дійсно буде 100% - 3em.

```
: root {
  --margin: 3em;
}
.tabs {
  width: calc (100% - var (--margin));
}
```

Основи мови JavaScript

JavaScript - це мова програмування, яка перетворює статичний web-сайт у web-додаток, що має інтерактивність за рахунок зворотного реагування на дії користувача. Наприклад: реакція при натисканні кнопок або на помилки при введенні даних у форми; створення динамічних стилів; анімації та ігрових дій.

Скрипт на мові JavaScript зазвичай вбудовують прямо в HTML-файл [9].

JavaScript ("JS" скорочено) це повноцінна динамічна мова програмування, яка, у застосуванні до HTML документу, може надати динамічну інтерактивність на web-сайтах. Вона була винайдена Бренданом Ейхом, співзасновником проекту Mozilla, the Mozilla Foundation, та Mozilla Corporation.

Мовою JavaScript трохи складніше опанувати, ніж з HTML і CSS.

JavaScript – скриптова мова, що інтерпретується. Це означає, що для виконання програми не потрібна попередня компіляція (перетворення вихідного тексту програми в системно-залежний машинний код). Текст програми інтерпретується, тобто аналізується і відразу ж виконується.

Найбільш поширеним середовищем виконання JavaScript є браузер, який надає можливість опрацьовувати DOM. Браузер виконує код JavaScript за допомогою браузерного рушія, тобто спеціальної програми, яка працює з web-сторінками. До функціоналу браузерних рушіїв () обробляє завантажену з Internet HTML-сторінку і перетворює її код у звичний для користувачів вигляд.

Основні браузерні рушії JavaScript:

• Mozilla - SpiderMonkey;

• Opera - Carakan;

• Google - V8.

Основні поняття JavaScript

Починаючі з специфікації ES2015 (ES6) для оголошення змінних у JavaScript поряд з оператором представлення var введено ще два нових оператори для створення змінних: let та const. Оператори оголошення var, let, і const мають певні відмінності, які необхідно зрозуміти на початку вивчення мови JavaScript. Тому, потрібно визначити такі поняття, як: оголошення та ініціалізація змінних, область видимості (особливо функції) та підняття змінної.

Змінні - контейнери, у яких ви можете зберігати значення. Ви можете розпочати з оголошення змінної за допомогою ключового слова var або let , після якого потрібно вказати ім'я, яким ви хочете назвати змінну.

Оголошення змінної здійснює оператор – var.

Створюємо нову змінну:

```
var declaration;
```

Тобто, цей код створює новий ідентифікатор з ім'ям declaration. Коли JavaScript створюється змінна, вона ініціалізується зі значенням undefined. Якщо ми зараз спробуємо викликати змінну declaration, то повернеться undefined.

Ініціалізація - це надання значення змінної:

declaration = 'Це ініціалізація';

Область видимості визначає, де в коді програми будуть доступні змінні та функції. У JavaScript є два типи області видимості - глобальна та локальна (global scope і function scope). Якщо змінна створюється всередині оголошення функції, її область видимості визначається як локальна і обмежується цією функцією. Тобто, якщо ви створюєте змінну всередині функції з оператором var, то вона буде доступна тільки всередині цієї функції та вкладених у неї функціях.

Найпростішими виразами мови JavaScript є константи (звані також літералами) і змінні.

Якщо ви хочете створити **константу**, то використовуйте **const**.

const myBirthday = '17 .06.1992 ';

Константу не можна перезаписати, тому код

myBirthday = '01 .01.2001 '; // викличе помилку

Присвоєння значення змінної (вказування типу не є потрібним – на відміну від мови C++) відбувається введенням відповідних даних:

var myVariable = 'Bob';

String - послідовність символів, відома як рядок. Щоб визначити, що значенням є рядок, ви мусите занести його в лапки.

var myVariable = 'Bob';

Number Число – будь-яке число записуємо без лапок.

var myVariable = 10;

Boolean Значення True/False. Слова true та false є спеціальними словами в JS, та не потребують лапок.

var myVariable = true;

Array Масив. Дозволяє зберігати багато значень за одним посиланням.

var myVariable = [1, 'Bob', 'Steve', 10];

Виклик значень елементів здійснюється так:

myVariable[0]; myVariable[1];

Object Загалом, об’єктом може бути будь-що. Об’єкт у JS може міститись у змінній. Наприклад:

var myVariable = document.querySelector ('h1');

Поширена практика використання констант в якості псевдонімів для значень, що важко запам'ятовуються. Назви таких констант пишуться з використанням заголовних букв і підкреслення.

Наприклад:

```
const COLOR_ORANGE = "# FF7F00";
// ... коли нам потрібно вибрати колір
let color = COLOR_ORANGE;
alert (color);  // # FF7F00
```

Відмінності між **var** і **let**. Головна відмінність **let** в тому, що область видимості змінної обмежується блоком {...}, а не функцією. Іншими словами, змінна, створена за допомогою оператора **let**, доступна всередині блоку, в якому вона була створена і в будь-якому вкладеному блоці. «Блок» - це все що вкладено між фігурними дужками {}, як ,наприклад, в циклі for або умови if.

Змінна var - одна на всі ітерації циклу і її видно після закінчення циклу:

```
for (var i = 0; i <10; i ++) {/ * ... * /}
alert (i);     // виведе 10
```

З змінної, що оголошена за допомогою **let** - все по-іншому.

Область видимості змінної let - блок {...}.

```
for (let i = 0; i <10; i ++) {/ * ... * /}
alert (i);    // помилка: глобальної змінної і немає.
```

При використанні в циклі, для кожної ітерації створюється своя змінна.

Кожному повторенню циклу відповідає своя незалежна змінна let. Якщо всередині циклу є вкладені оголошення функцій, то в замиканні кожної буде та змінна, яка була при відповідній ітерації.

```
for (let i = 0; i < 4; i++) {
   let j = 10;
}
console.log(i);      //  "ReferenceError: i is not defined"
console.log(j);      //  "ReferenceError: j is not defined"
```

На верхньому рівні програм та функцій let на відміну від var не створює властивості глобального об'єкта. Наприклад:

```
var x = 'глобальна';
let y = 'глобальна';
console.log (this.x);     // "глобальна"
console.log (this.y);      // undefined
```

Сформулюємо правила використання var, let і const у JavaScript

1. Змінна var обмежена областю видимості функції і якщо ви спробуєте звернутися до такої змінної до її оголошення ви отримаєте undefined.

2. Змінна let обмежена областю видимості блоку і якщо ви спробуєте звернутися до цих змінним до їх оголошення ви отримаєте ReferenceError.

3. Відмінність між const і let - це те, що значення, яке було присвоєно const не може бути перезаписано, на відміну від let.

Таблиця 1.1. Оператори у JavaScript

Оператор	Пояснення	Символ	Приклад
Додавання/ конкатинація	Вживається для додавання двох чисел чи злиття двох рядків разом.	+	6 + 9; "Hello " + "world!";

Віднімання, множення, ділення	Те ж, що і в простій математиці.	-, *, /	9 - 3;
			8 * 2; //Множення познач. зірочкою
			9 / 3;

Присвоєння	Присвоєння змінній значення.	=	var myVariable = 'Bob';

Порівняння	Перевіряє чи дві змінні рівні та повертає true/false (Boolean).	===	var myVariable = 3;
			
			myVariable === 4; //false

Заперечення (обернений до порівняння)	Повертає логічний вираз, протилежний значенню операнда; true стає false і т.п. Повертає true, якщо значення не рівні.	!, !==	Початковий вираз true, але повертається false , оскільки це заперечене порівняння:
			
			var myVariable = 3;
			
			!(myVariable === 3);
			
			Ми перевіряємо "чи myVariable НЕ рівне 3". Повертає false, оскільки myVariable = 3.
			
			або
			
			var myVariable = 3;
			myVariable !== 3;

Керуючі конструкції JavaScript у більшості випадків подібні до конструкцій C++. **JavaScript чутлива до регістру!**

Умова - називають структури коду, що дозволяють вам перевіряти, повертає вираз true, чи ні, виконуючи певні частини коду, залежно від результату. Поширеною формою умовної конструкції є if ... else. Наприклад:

```
var iceCream = 'chocolate';
if (iceCream === 'chocolate') {
  alert ('Yay, I love chocolate ice cream!');  }
else {
  alert ('Awwww, but chocolate is my favorite...');    }
```

Особливості **JavaScript** – це, наприклад, оператор **in** та цикл **for in**.

Оператор **in** повертає true, якщо вказана властивість присутня у вказаному об'єкті або у його ланцюжку прототипів.

```
const car = {make: 'Honda', model: 'Accord', year: 1998};
console.log('make' in car);
// видасть значення  true
```

Керуюча конструкція циклу **for in** призначена для проходу по масивах, колекціям і об'єктам:

```
  // телефонні коди в форматі "код країни": "назва"
var codes = { "38": "Україна",   "1": "США, "48": "Польща"  };
for (var code in codes) alert (code);    // виведе  38   1    48
```

Оператори виходу з циклу **break** і **continue** в **JavaScript**. Оператор **exit**

Оператор **break** перериває виконання всього тіла циклу, тобто здійснює вихід з циклу.

У той час як оператор **continue** перериває виконання поточної ітерації циклу, але, продовжуючи при цьому виконання циклу з наступного ітерації.

У мові JAVASCTIPT передбачений оператор виходу з програмного коду - оператор **exit**.

Найчастіше оператор використовується для виключення помилки введення користувача.

За правилом "**false**" організуємо перевірку значення змінної x. Якщо значення не числове виведемо відповідне повідомлення та завершимо програму:

```
x = isNaN (number);
// функція isNaN повертає значення true у разі, якщо змінна не є числом
if (x) {
alert ("Необхідно число!");
exit;      // вихід з програми
}
```

Функціональне програмування на мові JavaScript

Деякі стандартні функції.

Коли вам потрібна стандартна функція, ви можете просто викликати відповідну функцію. Наприклад, на сторінці html, що являє собою document:

```
var myVariable = document.querySelector ('h1');
                                    // функція буде визначати тег 'h1'
var myClassName= document.getElementsByClassName ('test');
                                    // функція шукає class = ' test'
var elementId = document.getElementById ('elem');
                                    // функція шукає id = 'elem'
var myElementClass = document.querySelector (' .elemClass ');
                                // функція шукає тег з класом class = ' elemClass '
alert ('hello!');           // функція показує діалогове вікно
```

Ці функції є вбудованими у браузер та виконуються рушієм JavaScript. Повний перелік стандартних функцій JavaScript можна знайти у документаціях рушіїв.

Приклад hello world.

Як додати деякі основні елементи JavaScript на вашу сторінку, щоб створити "hello world!".

1. Перейдіть до місця знаходження файлу сайта і створіть там папку з ім'ям 'scripts' (без лапок). Потім, в новій папці скриптів, яку ви тільки що створили, створіть новий файл з ім'ям main.js. Збережіть його у папці scripts.

2. Далі перейдіть до файлу index.html і введіть наступний елемент на новому рядку прямо перед закриттям тега </ body>:

<script src = "scripts / main.js"> </ script>

В основному цей код виконує ту ж саму функцію, що й елемент <link> для CSS. Цей код додає зв’язок з кодом JavaScript у зовнішньому файлі на web-сторінку, дозволяючи йому взаємодіяти з HTML.

3. Додайте наступний код у файл main.js:

```
var myHeading = document.querySelector ( 'h1');
myHeading.textContent = 'Hello world!';
```

4. З коду видно, що файл HTML повинен мати тег 'h1'.

5. Переконайтеся, що HTML і JavaScript файли збережені, і завантажте index.html в браузері.

Використання власних функцій [14]

Проста функція

Оголошення функції складається з ключового слова function та наступних її складових:

- ім'я функції;

- список параметрів, що приймаються функцією. Прописуються в круглих дужках () та розділюються комами;

- інструкції-команди, що будуть виконані після виклику функції. Вкладають у фігурні дужки { }.

Наприклад, проста функція (виду - function declaration statement), що приймає два числа аргументами та перемножує їх:

```
function multiply (num1, num2) {
  var result = num1 * num2;
  return result;
}
```

Викликати функцію в коді можна так:

```
multiply (4,7);
```

Будь-яка функція це об'єкт, і отже нею можна маніпулювати як об'єктом.

Функції типу **"function definition expression"**. Така функція може бути анонімною (не має імені).

Оголошення іменованої функції для обчислення числа в степені 3:

```
var f = function (x) {
   return Math.pow(x, 3);
}
```

Такі функції є зручними, коли вона передається аргументом іншої функції. Наприклад, функція map, яка має отримати функцію першим аргументом та масив другим:

```
function map (f, a) {
  var result = []; // Утворюємо масив
  var i;              // Оголошуємо змінну
  for (i = 0; i != a.length; i++)
    result [i] = f (a[i]);
      return result;
}
var numbers = [0, 1, 2, 5, 10];
var cube = map (f, numbers);
console.log(cube);  //  функція повертає: [0, 1, 8, 125, 1000].
```

Однак, функціям типу "function definition expression" ім'я може бути присвоєно для виклику самої себе всередині самої функції та для відладчика (debugger). Розглянемо, це на більш складнішому прикладі обчислення числа введеного в поле input в кубічну степінь з використанням функції f(), що описана у попередньому прикладі, та з використанням методу перехоплення події onClick на елементах HTML DOM - element.addEventListener().

Для цього створимо HTML-блок:

```
<div>
Write a namber here:
   <input type="text" id="read_namber" name="get_num" value />
 </div>
 <button id="exponentiateBy3" >Функція f(x^3)</button>
<div>
          <h2 id="elem5"></h2>
```

Запрограмуємо налаштування перехоплення події 'click' на елементі <button> управління HTML-блоку :

```
          var addValue = document.getElementById ('exponentiateBy3');
                    addValue.addEventListener ('click', getN);
```

Функція виведення з використанням функції f(), що описана у попередньому прикладі :

```
 function getN() {
     var getNum = document.getElementById('read_namber').value;
     var elementId5 = document.getElementById('elem5');
     elementId5.textContent = "Виведення значення функції f(" +
                                  getNum +  "^3)  =  " + f(getNum);
   }
```

Вкладені функції (nested functions)

Можна вкласти одну функцію до іншої. Вкладена функція приватна (private) і вона поміщена в іншу функцію (outer). Так утворюється замикання (closure). Closure - це вираз (зазвичай функція), який може мати вільні змінні разом із середовищем, яке пов'язує ці змінні (що "закриває" ("close") вираз).

```
function outside (x) {
          function inside (y) {
               return x + y;
             }
  return inside;
}
// функція додасть 3 до будь-якого введеного значення
fn_inside = outside (3);
     result = fn_inside (5);                  // повертає 8
        result1 = outside (3)(5);             // повертає 8
```

Зовнішня функція не має доступу до змінних та функцій, оголошених у внутрішній функції. Це забезпечує свого роду інкапсуляцію для змінних усередині вкладеної функції, яка отримала назву **замикання** (англ. closure). Замикання створюється, коли вкладена функція стала доступною в деякому scope поза зовнішньою функцією.

Стрілочні функції (arrow function expression)

Стрілочні функції завжди анонімні. Наприклад маємо масив:

```
var a = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];
```

Опрацювати довжину символів кожного значення можна застосувавши звичайну функцію:

```
var a2 = a.map (function (s) { return s.length; });
console.log(a2);        // виведе [8, 6, 7, 9]
```

або стрілочну:

```
var a3 = a.map (s => s.length);
console.log(a3);          // виведе [8, 6, 7, 9]
```

Метод map ( ) масивів JavaScript застосовує зазначену функцію послідовно до кожного елементу масиву та створює новий масив із результатів.

Стрілочні функції мають укорочений синтаксис порівняно з function expression і лексично пов'язані з значенням **this**.

До стрілочних функцій кожна нова функція визначала своє значення this (новий об'єкт у разі конструктора, невизначений в strict mode, контекстний об'єкт, якщо функція викликана як метод об'єкта).

```
function Person() {       //  у тілі функції Person() визначає `this` як сам себе
                  this.age = 0;
  setInterval (function growUp() {
    // без strict mode функція growUp () визначає `this`
    // как global object, який відрізняється від `this`, що
    // визначений конструктором Person()
    this.age++;
  }, 1000);
}
var p = new Person();
```

Виправити таку невідповідність можна стрілочною функцією:

```
function Person() {
          this.age = 0;
  setInterval(() => {
            this.age++; // this тепер правильно посилається на об’єкт Person
  }, 1000);
}
var p = new Person();
```

Подія, що прив'язана до елемента. Обираємо <html> елемент, виставляючи для події onclick на виконання безіменну функцію, яку ми хочемо запустити при натисканні.

```
document.querySelector ('html').onclick = function() {};
```

еквівалентне цьому:

```
var myHTML = document.querySelector ('html');
myHTML.onclick = function() {};
```

Об’єктно-орієнтоване програмування на мові JavaScript

У об’єктно-орієнтованому програмуванні основним елементом коду є клас. Клас - це шаблон коду, який є розширюваним та використовується для створення об’єктів, що вміщують початкові значення даних (змінні-поля) та код-реалізацію поведінки об’єктів (функції-методи).

Основний синтаксис класу [25]:

```
class MyClass {
  // class methods
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  ...
}
```

Після створення класу для його виклику для застосування використовуємо код:

```
new MyClass();
```

У цьому випадку метод constructor() автоматично викликається new, після чого у пам’яті створюється новий об’єкт з усіма переліченими методами класу. Цей етап отримав назву **ініціалізація об’єкту**.

Приклад використання:

```
class User {
  constructor (name) {
    this.name = name;
  }
  sayHi () {
    alert(this.name);
  }
}
// Використання
let user = new User ("John");
user.sayHi ();
```

Після виклику new User ("John") :

1. Створюється новий об'єкт.
2. Запускається constructor з заданим аргументом і призначає його this.name.
3. Тоді ми можемо викликати об’єктні методи, такі як user.sayHi().

Використання JavaScript фреймворку - jQuery.

Для того щоб розуміти як працює селектор розробнику потрібно мати базові знання CSS, тому що саме від принципів CSS відштовхує селектор jQuery:

$("#header") - отримання елемента з id = "header»

$("h3") - отримати все <h3> елементи

$("div #content .photo") - отримати всі елементи з класом = «photo» які знаходяться в елементі div з id = "content"

$("ul li") - отримати все <li> елементи зі списку

$("ul li:first") - отримати тільки перший елемент <li> зі списку

```
<script type="text/javascript">
// у JavaScript сторінка є об'єктною моделлю DOM
$(document).ready (function (){
// Код, включений всередині $ (document) .ready (), буде працювати
// тільки тоді, коли DOM готова до виконання коду JavaScript
$("p").click(function(){
// Основний синтаксис: $ ("селектор") .Дія ()
// $ - знак визначення / звернення до jQuery
// (селектор) служить, щоб "вибрати (або знайти)" елементи HTML
$ (This) .hide ();
// дію () - дія, яке треба виконати з елементом (елементами)
alert ("Клік на тезі P і він зник \ n для появи клік 2 рази!");
// тут йде дія !!
});
$(".button").mouseenter (function(){
// Функція виконується, коли курсор миші з'являється над елементом HTML
alert ("Покажчик миші над кнопкою!");
// mousedown - Кнопка миші затиснута; mouseleave - покинули
});
}
);
</script>
```

Обробник події натискання на клавішу миші - MouseEvent

```
$(“element_selector”).click(function(e) {
        // does something
   alert (e.type);  //will return you click
}
```

e - є скороченою частиною ширшого діапазону посилань (var для об'єкта event) у JavaScript, що називається функцією обробки подій.

Інструменти та засоби розробки

Для розробки навчального проекту на базі стеку MEAN/MERN використовуємо такі інструменти та засоби, що мають безкоштовний доступ:

- Visual Studio Code, безкоштовний кросплатформний редактор вихідного коду для OS X, Linux та Windows [18]. Він включає вбудований налагоджувач, засоби автодоповнення типових конструкцій та підказки, функцію навігації за кодом та інструменти для роботи з Git;

- Node.js - платформу з відкритим кодом для виконання високопродуктивних мережевих додатків, написаних мовою JavaScript;

- MongoDB - документо-орієнтовану базу даних, що зберігає дані у форматі бінарного JSON.

1.1.*2. Практичний модуль*

Практична частина побудована у вигляді проекту розробки web-додатку Tours City (екскурсії містом) через який можна рекламувати подорожі будь-яким містом у галузі культурного міського туризму.

Етап 1.1. Створення технічного завдання проекту web-додатку

Дія 1.1.1. Опис загальних вимог

Прикладна галузь web-додатку є міський культурний туризм для якого розв’язується проблема діджиталізації. Основним завданням розроблюваного web-додатку є забезпечення інформаційного супроводу туристів під час проходження турів відвідування культурних об’єктів у місті.

Функціональне наповнення. ***Моделювання*** (рис. 1.4).

Завдання інформаційного супроводу турів міського культурного туризму розв’язується шляхом надання інформації щодо туристичних маршрутів у місті на основі визначення геолокації місцеположення користувача або з використанням сервісів пошуку за номером маршруту, за назвою району

міста для обрання пропонованих сформованих туристичних маршрутів або від визначеного користувачем місця за назвою. Після чого користувач отримує опис туристичного маршруту на стороні клієнта - front-end.

Робота з інформацією супроводу в базі даних відбувається на серверному рівні - back-end, на якому у режимі адміністрування доступом, що надається адміністратором, менеджер контенту реалізує чотири базові функції CRUD: створення (англ. create), читання (read), модифікація (update), видалення (delete) контенту.

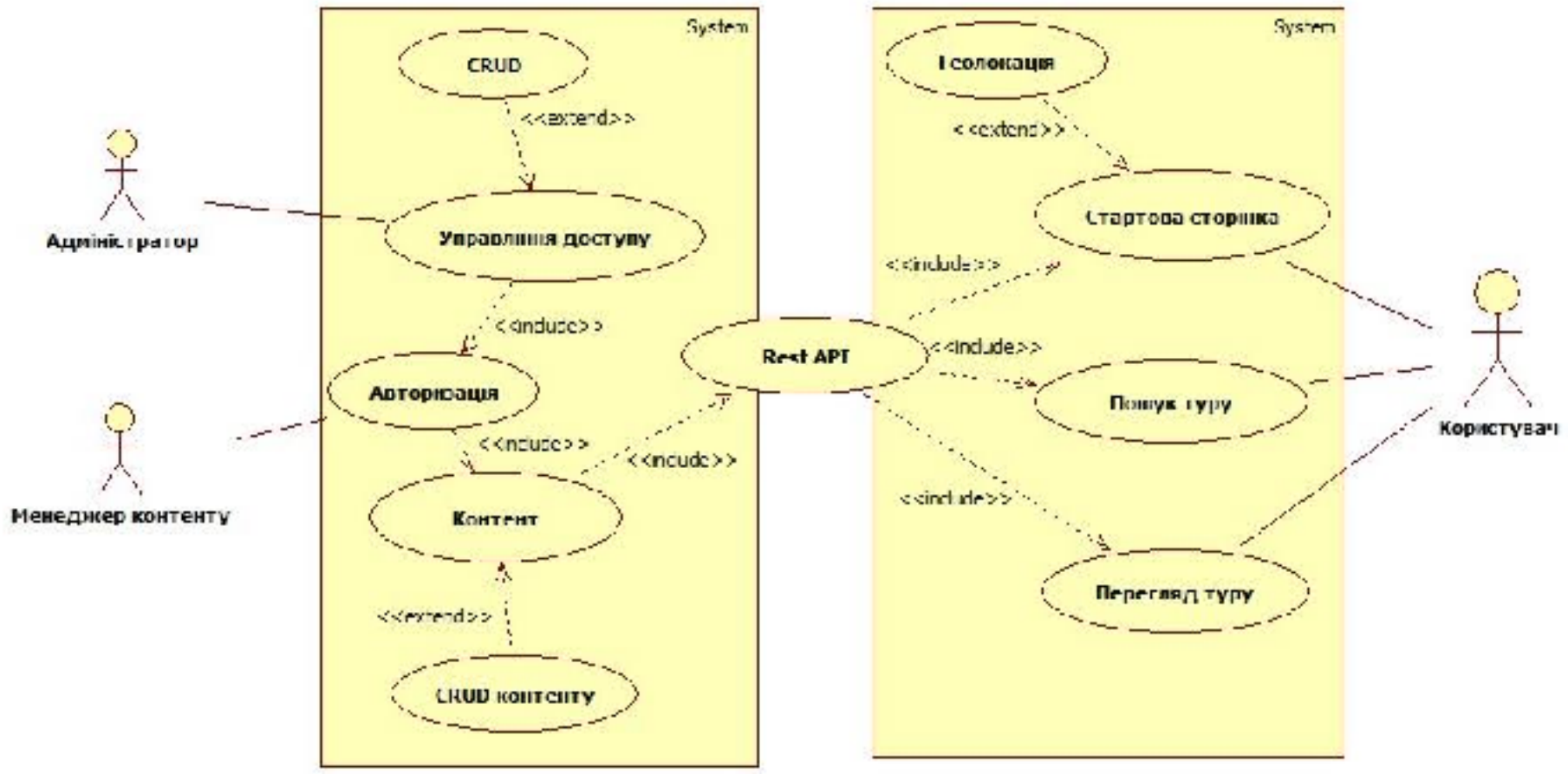


Рис. 1.4. UML діаграма прецинденті в Web-додатку

Мова завдань.

Складаємо репозітарій термінів, що будуть використані для створення інтерфейсу користовача:

Туризм - тимчасовий виїзд людини з місця постійного проживання в оздоровчих, пізнавальних цілях;

Маршрут - заздалегідь намічений або встановлений шлях мандрівників (туристів) або транспортних засобів.

Туристичний маршрут - це географічно визначена і прив'язана до конкретної місцевості й об'єктів траси подорож із можливим текстовим описом.

Тур - це комплекс послуг, оформлених у вигляді поєднання ресурсів туристичного пакету, які можуть бути реалізованими на певному маршруті.

Туристичні ресурси - сукупність природно-кліматичних, оздоровчих, історико-культурних, пізнавальних та соціально-побутових ресурсів відповідної території, які задовольняють різноманітні потреби туриста.

Дестинація – це географічний об'єкт (територія, пункт призначення), який є привабливим для туристів завдяки наявності унікальних або специфічних туристсько-рекреаційних ресурсів та відповідної інфраструктури.

Технологія розробки проекту – стек MERN. База даних, що формує данні інформаційного супроводу, вміщує знання щодо туристичних об'єктів, їх особливості та параметри огляду.

Системні вимоги.

На стороні сервера (back-end) застосовується фреймворк Express.js, документно-орієнтована база даних з відкритим вихідним кодом MongoDB.

На стороні клієнта (front-end) web-додаток будується на засадах односторінкової реалізації (Single-Page Applications, SPA) з використанням JavaScript-бібліотеки React.js. SPA завантажується як одна HTML-сторінка, що динамічно оновлюється при її взаємодії з користувачем. Дизайн шаблону сайту повинен бути виконаним з використання мови HTML, CSS та JS. Сайт повинен коректно відображатися в браузерах Microsoft Internet Explorer 6.0, 7.0; Mozilla FireFox, Google Chrome.

Взаємодія клієнтського web-додатку із сервером реалізується методом REST API. Де API (англ. Application Programming Interface) - це код, який дозволяє обмінюватися даними сторони сервера зі стороною клієнта. REST (англ. Representational State Transfer - "передача репрезентативного стану") являє набір правил того, як організувати написання коду, щоб усі системи web-додатку легко обмінювалися даними.

Дія 1.1.2. Вимоги до контенту Web-додатка та його наповненості на рівні front-end:

Мова контенту Web-додатка - Українська.

Адаптивність. Розробляється з адаптацією для комп'ютерної та мобільної версій. Шаблон повинен відповідати вимогам для адаптивності.

Контент Web-додатка:

Сторінка (шаблон SPA) Web-додатка повинна складатись з таких елементів:

1. "ШАПКА" (хедер) вміщує розміщені зліва направо:
   - напис "*Оберіть маршрут*";
   - напис "*за номером*" та поле вибору select за номером маршруту;
   - напис "у районі" та поле вибору select за назвою району міста;

- напис “*від місця*” та поле вибору select за назвою місця;
- кнопку “*з цього місця*”.

2. Блок КАРТА МАРШРУТУ. Призначення - надати інформацію місцезнаходження туристичних об’єктів поточного маршруту.

Контентний блок має такі атрибути :

відображає Google Maps з маркуванням маршруту.

3. Блок ОБ’ЄКТИ МАРШРУТУ. Призначення - надати інформацію щодо об’єктів туристичного маршруту, їх порядковий номер, зображення та скорочений опис. Подати у два стовпчики.

Контентний блок має такі атрибути :

Номер відвідування.

Назва об’єкту.

Фотографія – одна.

Скорочений опис.

Кнопка “Більше” для розгортання повного опису.

4. Блок ДОДАТКОВА ІНФОРМАЦІЯ МАРШРУТУ Призначення - надати інформацію щодо того як переміщуватись за маршрутом від об’єкта до об’єкта, історичні відомості щодо району через який проходить маршрут, архітектурні особливості, історичні данні тощо. Подати у по ширині.

5. “ПІДВАЛ” (футер) вміщує розміщені зліва направо коротку контактну інформацію з відповідним гіперпосиланням.

Етап 1.2. Створення HTML/CSS шаблон web-додатка

Дія 1.2.1. Вибір шаблону на ресурсах Internet і адаптування його під технічне завдання.

У пошуковику браузера виконуємо запит на знаходження сайтів з безкоштовними шаблонами HTML.

Обираємо шаблон, що відповідає вимогам адаптивності, реалізований за принципом SPA та має фіксоване меню зі скролінгом. Шаблон повинен *використовувати* бібліотеки bootstrap [24] та jquery.js [25].

Нижче подано файл index.html та файл стилів розмітки структуровані згідно принципу БЕМ - компонентного підходу до web-розробки у лістингах 1.1-1.5.

Розмітка файлу index.html утворююче плаваюче меню сайту у частині header з очищенням вмісту у контейнері при додаванні утиліти-хаку "clearfix" з Bootstrap (лістинг 1.1).

Лістинг 1.1 : Вміст файлу index.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>City Tours</title>
<meta name="viewport" content="width=device-width, initial-scale=1">
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link href="css/bootstrap.css" rel='stylesheet' type='text/css' />
<script src="js/jquery-1.11.1.min.js"></script>
<link href="css/style.css" rel='stylesheet' type='text/css' />
<link href='http://fonts.googleapis.com/css?family=PT+Sans+Narrow:400,700'
rel='stylesheet' type='text/css'>
</head>
<body>
  <div class="header-home">
       <div class="fixed-header">
            <div class="logo">
                 <a href="index.html">
            <span class="secondary">City </span>
             <span class="main">Tours</span>
           </a>
            </div>
<div class="top-nav">   <span class="menu"> </span>
            <ul>  <span>Оберіть початок маршруту:</span>
       <li> <select name="tur" style="width: 132px; height:25px" >
 <option selected disabled>За номером туру</option>
 <option value="Тур 1">Тур 1</option>
 <option value="Тур 2">Тур 2</option>
 <option value="Тур 3">Тур 3</option>
        </select>
       </li>
        <li>
      <select style="width: 132px; height:25px" >
 <option selected disabled>Тур у районі міста</option>
 <option value="Пункт 1">Пункт 1</option>
 <option value="Пункт 2">Пункт 2</option>
```

```
<option value="Пункт 3">Пункт 3</option>
            </select>
          </li>
 <li><input type="text" class="text" value="Від зазначеного місця"
onfocus="this.value = '';" onblur="if (this.value == '') {this.value = 'Введіть назву
місця';}">
 </li>
    <li class="active"><a href="#">Від цього місця</a>  </li>
                  </ul>
<!--  script - розгортається при натисканні на картинку мобільне меню   -->
<script>
   $("span.menu").click(function(){
   $(".top-nav ul").slideToggle(500, function(){
                  });
          });
</script>
          </div>
<div class="clearfix"> </div>   <!-- очищає обтікання div хаком "clearfix" -->
             </div>
          </div>
<!--     Скрипт для скролінгу вікна сайту   -->
<script>
$(document).ready(function() {
var navoffeset=$(".header-home").offset().top;
$(window).scroll(function(){
var scrollpos=$(window).scrollTop();
if(scrollpos >=navoffeset){
$(".header-home").addClass("fixed");
                  }else{
$(".header-home").removeClass("fixed");
                  }  });
          });
</script>
```

```
 <!--    Контентна частина сайту   -->
 <div class="container">
        <div class="row">
<h1 class="blog_head">Маршрути</h1>
   <div class="map">
<iframe src="https://www.google.com/maps/... "> </iframe>
        </div>
 <h1 class="blog_head">Об'єкти маршруту</h1>
<section> <div class="row_text">
   <div class="col-md-6">
   <h3 class="tour__title title-mod">Древній Луцьк (Лучеськ)</h3>
<p>Виник на острові, який омивався водами широкоплинного Стиру та його
старого русла Глушця.  </p>
 <p>   <a class="btn btn-default" href="#" target="_blank">Детальніше</a>   </p>
   </div>
   <div class="col-md-6">
   <h3 class="tour__title title-mod">Старе місто</h3>
   <p>Це - колиска Луцька. Саме тут, на острові серед заплав Стиру, виникло
давнє поселення, яке поступово почало переростати у місто.   </p>
<p>   <a class="btn btn-default" href="#" target="_blank">Детальніше</a> </p>
   </div>
</div> </section>
   <h1 class="blog_head">Додаткова інформація маршруту</h1>
<section> <div class="row_text">
   <div>  <h3 class="tour__title title-mod">Древня історія Луцька</h3>
<p>У другій половині XII ст., коли волинський князь Ізяслав Мстиславович
ділив свої землі між синами, то столицею східно-волинського наділу вибрав
 місто над Стиром. </p>
  <p>  <a class="btn btn-default" href="#" target="_blank">Детальніше</a>   </p>
   </div>
</div>
</section>
        </div>
        </div>
```

```
<!--    Частина footer сайту   -->
    <div class="footer">
      <div class="container">
           <div class="footer_top">
                 <div class="col-6 col-sm-4">
  <ul class="list1">
         <h3>Navigation</h3>
 <li><a href="#"> Google Maps</a></li>
 <li><a href="#">Apply</a></li>
  <li><a href="#">Register</a></li>
  </ul>
         </div>
<div class="col-6 col-sm-4">
  <ul class="socials">
   <li><a href="#"><i class="fa fb fa-facebook">
<svg xmlns="http://www.w3.org/2000/svg" width="24" height="24"
fill="currentColor" class="bi bi-facebook" viewBox="0 0 16 16">
  <path d="M16 …"/>   </svg> </i></a></li>
  <li><a href="#"><i class="fa tw fa-twitter">
<svg xmlns="http://www.w3.org/2000/svg" width="24" height="24"
fill="currentColor" class="bi bi-twitter" viewBox="0 0 16 16">
  <path d="M5…"/> </svg>  </i></a></li>
        </ul>
         </div>
         <div class="clearfix"> </div>
   </div>
 </div>
         </div>
</body>
</html>
```

Лістинг 1.2 : Стилі дизайну частини header web-сторінки у файлі style.css

```
   html, body {
   font-family: 'PT Sans Narrow', sans-serif;
   font-size: 100%;
   background: #fff;
}
 /*--header--*/
.header-home{
         background:#f1cd69;
        z-index: 999;
}
.logo{
        float: left;
        display: inline-block;
        background:#312934;
        margin-bottom: 0;
        padding: 0.5em 4em 1.7em;
}
span.secondary {
  margin-left: 4px;
  font-size: 1.5em;
  text-transform: uppercase;
  letter-spacing: 10px;
  color: #ffffff;
  display: block;
  margin-bottom: 7px;
}
span.main {
  color: #ffffff;
  display: block;
  text-transform: uppercase;
  font-size: 2.5em;
  line-height: 30px;
}
```

```
.top-nav{
        float: right;
        margin: 2.3em 4em 0 0;
        font-size: 1.2em;
}
select {
        font-size: 0.7em;
        margin-left: 14px;
        margin-right: 14px;
}
input{
        font-size: 0.7em;
        margin-left: 14px;
        margin-right: 14px;
}
.top-nav ul{
        padding: 0;
        margin: 0;
}
.top-nav ul li{
        display: inline-block;
}
.top-nav ul li a{
  color: #fff;
  text-transform: uppercase;
  text-decoration: none;
  transition: 0.5s all;
  -webkit-transition: 0.5s all;
  -o-transition: 0.5s all;
  -ms-transition: 0.5s all;
  -moz-transition: 0.5s all;
  padding: 2px 10px 2px 10px;
}
```

Продовження лістингу 1.2

```
.top-nav ul li.active a, .top-nav ul li a:hover{
        color:#fff;
        background:#312934;
        -webkit-transform: rotateY(70deg);
        -moz-transform: rotateY(70deg);
        -ms-transform: rotateY(70deg);
        transform: rotateY(70deg);   }

.fixed{
        position: fixed;
        top: 0;
        width: 100%;
        margin: 0 auto;
        left:0;
}
.top-nav span.menu{
        display:none;
}
.top-nav span.menu:before{
        content:url("../images/menu.png");
        cursor:pointer;
        width:100%;
}
```

кінець лістингу 1.2

Лістинг 1.3 : Стилі дизайну частини content web-сторінки у файлі style.css

```
                        /*--  content --*/
.row {
        padding:8em 0;
        margin: 0;

}
```

Продовження лістингу 1.3

```
h1.blog_head{
        margin-top: 20px;
        font-size:3em;
        text-transform: uppercase;
        color:#000;
        text-align:center;

}
.map iframe{
        width:100%;
        min-height:400px;
        border:none;
        padding:0 15px;
}
a:hover, a:focus {
        text-decoration:none !important;
}
```

кінець лістингу 1.3

Лістинг 1.4 : Стилі дизайну footer web-сторінки у файлі style.css

```
/*--footer--*/
.footer{
        background:#312934;
        padding:3em 0;
}
.footer_top {
        padding-left:50px;
}
ul.list1{
        padding:0;
        list-style:none;
}
```

```
ul.list1 h3{
        color:#fff;
        text-transform:uppercase;
        margin-bottom: 1em;
}
ul.list1 li a{
        color:#888;
        font-size: 0.85em;
        line-height:2.5em;
        text-transform:uppercase;
}
ul.list1 li a:hover{
        text-decoration:none;
        color:#5599ba;
}
ul.socials {
        padding: 0;
        list-style: none;
        margin-bottom: 1em;
}
ul.socials li {
        display: inline-block;
}
.socials li a i.fb {
        margin-right: 5px;
        height: 80px;
        width: 80px;
        display: inline-block;
        color: #fff;
        background: none;
        text-align: center;
        line-height:81px;
        font-size: 30px;
        -webkit-border-radius: 500px;
```

```
        -moz-border-radius: 500px;
        border-radius: 500px;
        background: #4d53be;
}
.socials li a i.fb:hover{
  background:#4147B0;
}
.socials li a i.tw {
        margin-right: 5px;
        height: 80px;
        width: 80px;
        display: inline-block;
        color: #fff;
        background: none;
        text-align: center;
        line-height:81px;
        font-size: 30px;
        -webkit-border-radius: 500px;
        -moz-border-radius: 500px;
        border-radius: 500px;
        background:#00aeef;
}
.socials li a i.tw:hover{
  background:#02A8E7;
}
ul.list2{
        padding:0;
        list-style:none;
}
.phone {
        color:#f1cd69;
        margin-bottom:20px;
}
```

Продовження лістингу 1.4

```
ul.list2 li{
	color:#888;
	font-size:1.1em;
}

ul.list2 li a{
	color:#fff;
}

ul.list2 li a:hover{
	text-decoration:none;
	color:#f1cd69;
}

.copy{
	text-align:center;
	padding:1em 0;
	background:#241C27;
}

.copy p{
	color:#888;
}

.copy p a{
	color:#f1cd69;
}

.copy p a:hover{
	color:#fff;
	text-decoration:none;
}
```

кінець лістингу 1.4

Лістинг 1.5 : Стилі адаптованого дизайну web-сторінки під смартфон

```
/*----responsive-menu-start---*/
@media(max-width:768px){
.top-nav ul li{
        display:block;
        float:none;
        margin: 1.5em 0.7em;
        padding: 0 0;
        text-align: center;
}
.top-nav ul li a{
        color:#fff;
        border: none;
}
.top-nav ul{
        display:none;
        background:#f1cd69;
        position:absolute;
        width:100%;
        z-index: 999;
        left: 0%;
        margin: 0em 0em;
}
.top-nav span.menu{
        display: block;
        width: 100%;
        position:relative;
        padding: 1.1em 1em 0.5em;
        text-align: right;
}
.top-nav ul li  span{
        display: none;
}
}
```

```
/*--responsive design--*/
@media (max-width:1024px){
.header {
        min-height:550px;
}
.header_top {
        padding-top:10em;
}
.logo {
        padding: 0.5em 1em 1.7em;
}
.top-nav {
        margin: 2.3em 1em 0 0;
}
h1.blog_head {
        margin-bottom:1em;
}
.header_top h1 {
  font-size: 3em;
}
}
@media (max-width:930px){
.socials li a i.fb, .socials li a i.tw{
  margin-right:2px;
  height: 50px;
  width: 50px;
  line-height: 55px;
  font-size: 20px;
}
ul.list2 li {
  font-size: 1em;
}
h1.blog_head {
  font-size: 2.5em;
}
}
```

```
@media (max-width:768px){
.header {
  min-height: 400px;
}
.header_top h1 {
  font-size: 2em;
}
.header_top h2 {
  font-size: 1em;
}
.header_top {
  padding-top: 8em;
}
span.secondary {
  font-size: 12px;
}
.top-nav span.menu {
  padding: 8px 0 0 0;
}
.top-nav ul li.active a, .top-nav ul li a:hover {
  background:none;
}
span.main {
  font-size: 3.5em;
  line-height: 20px;
}
span.secondary {
        margin-left: 4px;
}
.top-nav {
        margin: 1em 1em 0 0;
}
.top-nav ul {
        margin-top:20px;
}
```

```
ul.list1 h3 {
  font-size: 1.1em;
}
.map iframe {
  min-height:300px;
}
.socials li a i.fb, .socials li a i.tw{
        margin-right: 5px;
        height: 60px;
        width: 60px;
        line-height: 65px;
        font-size: 20px;
}
.top-nav ul li a {
  padding: 0;
}
}
@media (max-width:640px){
ul.list1 li a {
  line-height: 2em;
}
ul.list1 {
  margin-bottom: 2em;
}
.footer{
  padding:3em 0 0 0;
}
.header {
  min-height: 340px;
}
h1.blog_head {
  font-size: 2em;
}
```

```
.header_top {
  padding-top: 6em;
}
}
@media (max-width:480px){
span.main {
  font-size: 2.5em;
  line-height: 15px;
}
span.secondary {
  letter-spacing: 5px;
}
.logo {
        padding: 0.5em 1em 1em;
}
.top-nav {
        margin: 0.6em 1em 0 0;
}
.top-nav span.menu {
  padding: 6px 0 0 0;
}

h1.blog_head {
  font-size: 1.5em;
}
.map iframe {
  min-height:200px;
}
.top-nav ul {
        margin-top: 10px;
}
}
```

```
@media (max-width:320px){
.header {
  min-height:210px;
}
span.main {
  font-size: 2em;
  line-height: 27px;
}
.header_top h1 {
  font-size: 1.2em;
}
.header_top h2 {
  font-size: 12px;
}
.header_top {
        padding-top:3em;
}
span.secondary {
        display: none;
}
.logo{
        padding:10px;
}
.container {
        padding:0 10px;
}
.top-nav {
  margin: 0;
}
.top-nav span.menu {
        padding:8px 10px 0 0;
}
.top-nav ul {
        margin-top: 4px;
}
```

Продовження лістингу 1.5

```
.socials li a i.fb, .socials li a i.tw {
  margin-right: 0px;
  height: 40px;
  width: 40px;
  line-height: 42px;
  font-size: 15px;
}
.footer{
        padding:2em 0 0 0;
}
.map iframe {
        min-height:200px;
}
ul.list1 li a {
  line-height: 1.8em;
  font-size: 13px;
}
ul.socials {
  margin-bottom: 10px;
}
ul.list2 li {
  font-size: 0.95em;
}
.copy p {
  font-size: 14px;
}
.top-nav ul li {
  margin: 10px;
}
}
```

кінець лістингу 1.5

З Internet закачуємо файли: bootstrap.css та jquery-1.11.1.min.js. Після чого, зовнішній вигляд доопрацьованого шаблону index.html подано на рис.1.5.

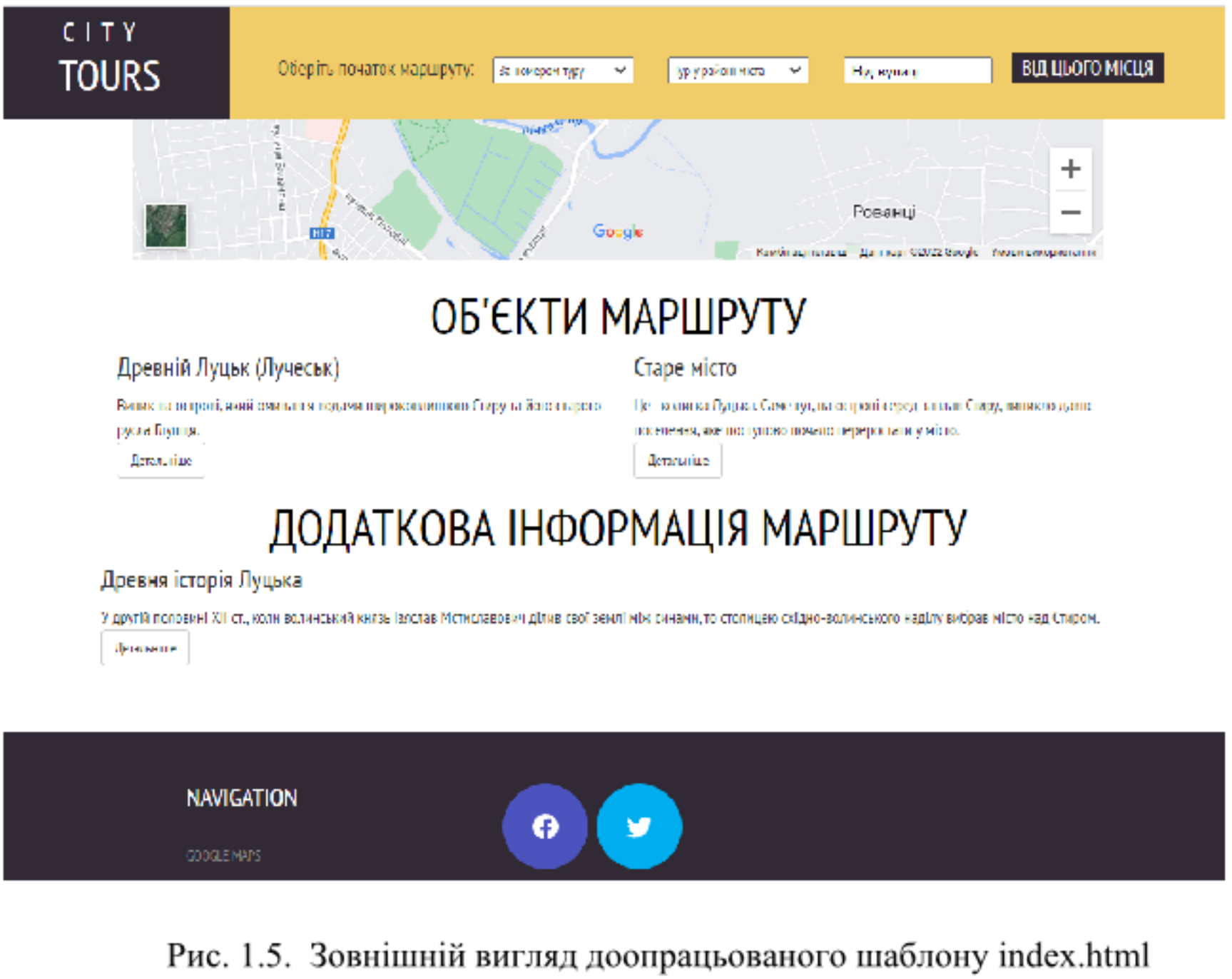


Рис. 1.5. Зовнішній вигляд доопрацьованого шаблону index.html

1.2. СТВОРЕННЯ WEB-ДОДАТКІВ ІЗ ВИКОРИСТАННЯМ ХМАРНИХ ОБЧИСЛЕНЬ. ТЕХНОЛОГІЯ MERN

1.2.1. *Теоретичний модуль*

Хмарні обчислення

Під терміном "***Хмара***" розуміється образ складної інфраструктури за якою приховуються всі технічні деталі сучасних методів обробки та обміну інформаційними даними. Безпосередньо, "***хмарна обробка даних***" - це парадигма IEEE-2008 (англ. Institute of Electrical and Electronics Engineers - міжнародна організація інженерів у галузі електротехніки, радіоелектроніки та радіоелектронної промисловості, світовий лідер в галузі розроблення стандартів), в рамках якої інформація постійно зберігається на серверах у

Internet, тимчасово кешується на боці клієнта (ПЕОМ, ноутбуки, планшети тощо).

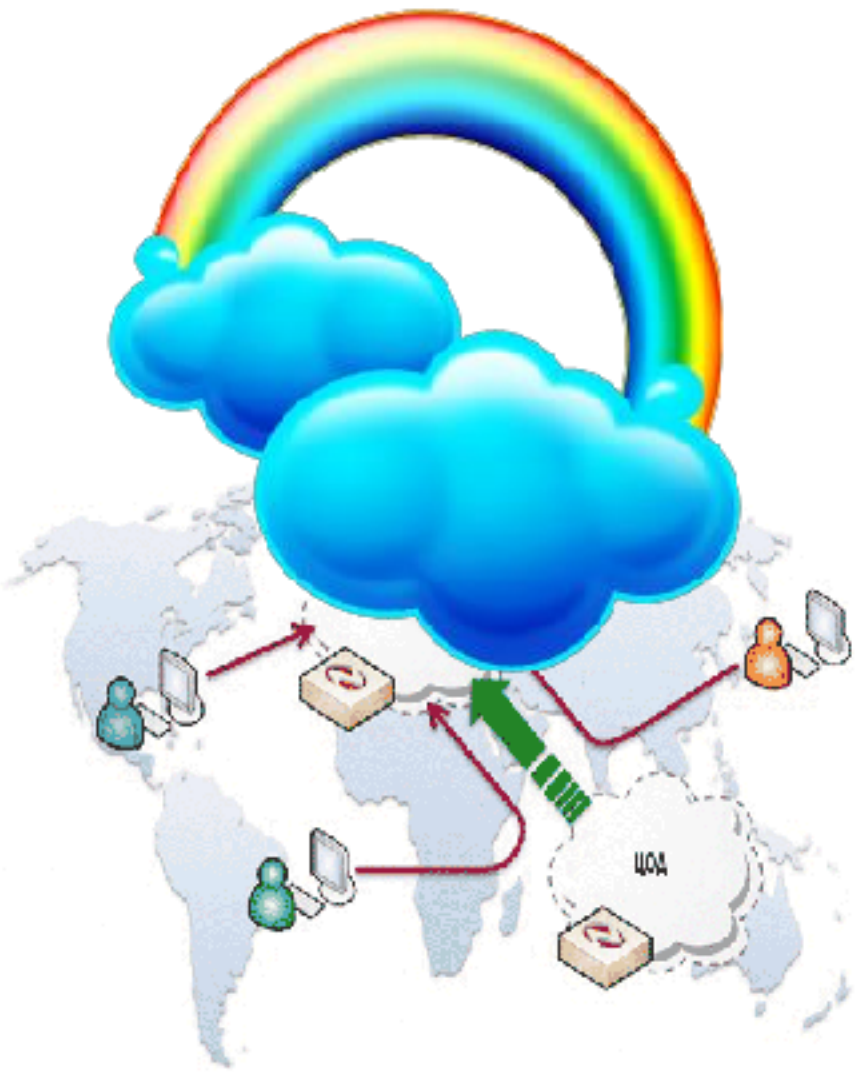


Рис. 1.4. Хмарна обробка даних

Хмарна обробка даних зорієнтована на доступі до глобальних сховищ даних у режимі on-line, їх обчислення та подальшої синхронізації для збереження на віддалених серверах, подання цих даних за викликом користувача.

Хмарні обчислення (англ. cloud computing) – це модель забезпечення повсюдного та зручного мережевого доступу до даних та до обчислювальних ресурсів згідно з конфігурацією користувацького програмного додатку, які можуть з мінімальними експлуатаційними витратами, оперативно надаватися сервісами хмарних обчислень. В цілому, сервіси хмарних обчислень є додатками, доступ до яких забезпечується через Internet за допомогою браузерів та інших мережевих застосувань доступу до серверів додатків, серверів баз даних та FTP серверів.

Сучасна ***розробка web-додатків*** орієнтована на застосуванні під час їх складання, так званих, ***ліній розроблення програм***, які ***об'єднуються у фабрику програм*** (рис. 1.5). Одним з найважливіших технічних ресурсів фабрик програм є алгоритми та розроблені на їх базі програми та ***компоненти повторюваного використання*** (КПВ) [10].

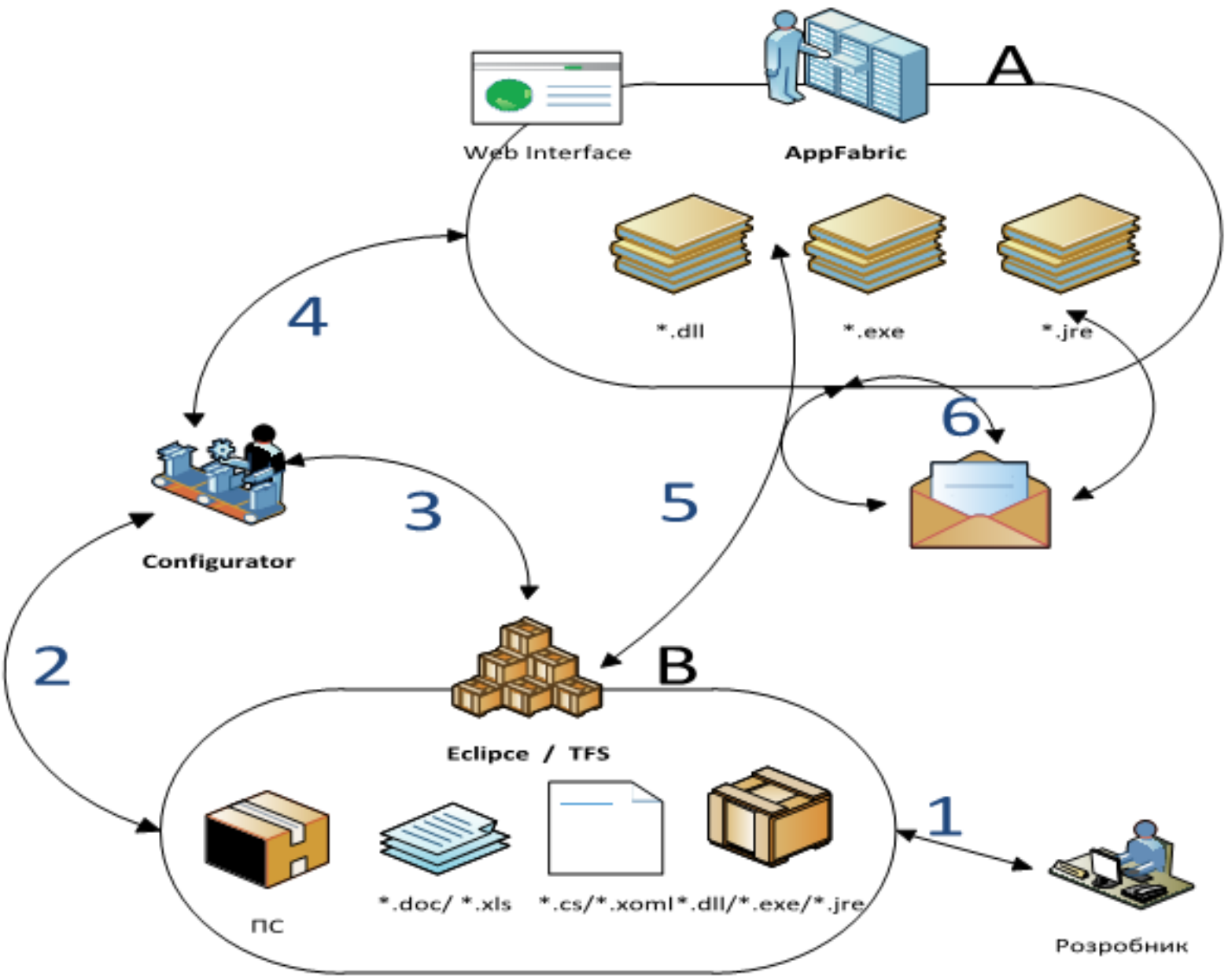


Рис. 1.5. Загальна структура фабрики програм [11]

Під час web-програмування організація обчислень потребує визначення таких методів складання та об'єднання ресурсів, які полягають у координації, кооперації та взаємодії різних сервісів і інших готових ресурсів через ***конфігураційний файл*** для виконання обчислень відповідних задач [11]. Тому, ***до компетентностей розробників web-додатків*** відноситься знання щодо використання готових програмних ресурсів та вміння їх поєднання, вибудовування бізнес логіки за результатами об'єктного аналізу прикладної галузі та функціонального наповнення розроблюваної web-системи, вибору компонентного методу реалізації цих об'єктів, завдання інтерфейсів, необхідних під час складання у середині різних фабрик програм.

У сучасному web-програмуванні широке застосування знаходить стек ресурсів MEAN/MERN, що є комбінацією трьох фреймворків JavaScript і технології баз даних на основі NoSQL (рис. 1.6). Стек MEAN/MERN ґрунтується на переважному застосуванні однієї мови програмування JavaScript впродовж розробки web-додатку.

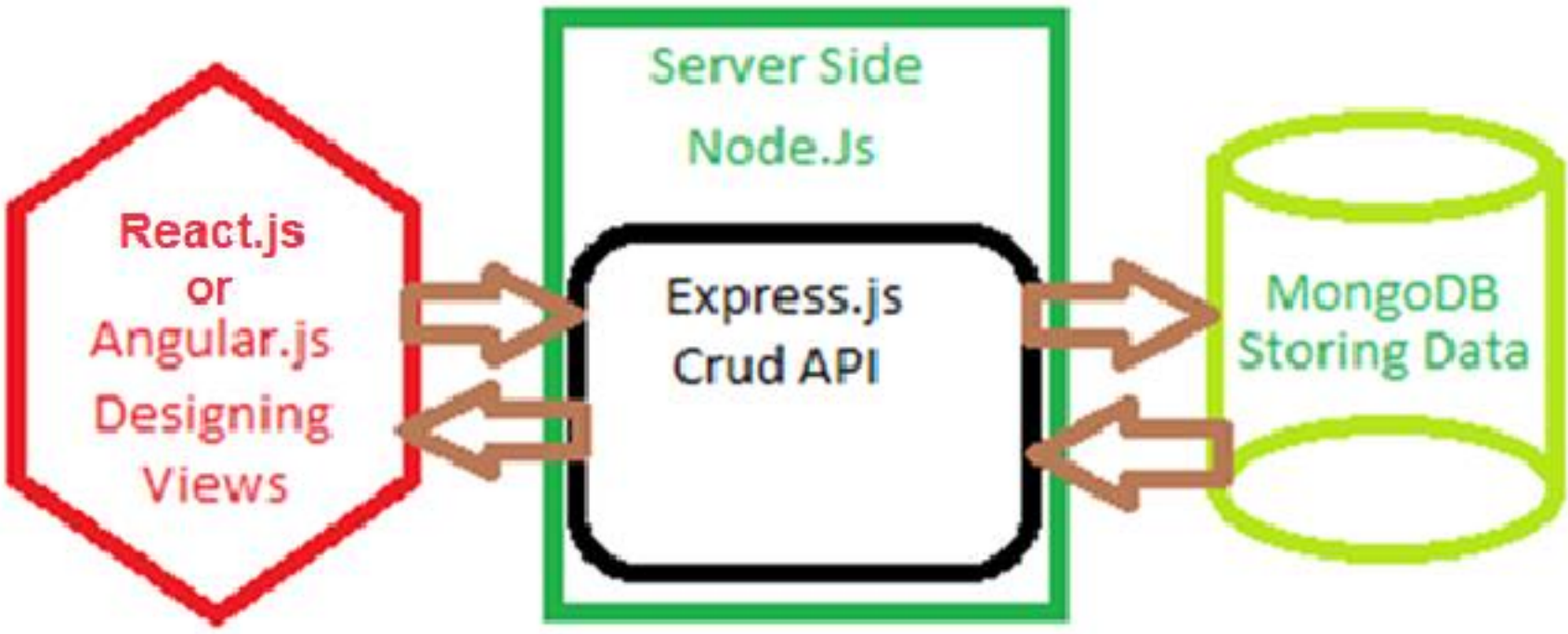


Рис. 1.6. Загальна структура MEAN/MERN [12]

MEAN (абревіатура від MongoDB, Express.js, Angular.js, Node.js) – стек (набір, комплекс) серверного програмного забезпечення, що використовується для web-розробки.

У стеці MERN Angular.js замінений на React.js – JavaScript бібліотеку для створення користувацького інтерфейсу (на рівні front-end).

MongoDB - це документно-орієнтована база даних з відкритим вихідним кодом, розроблена з урахуванням масштабованості та швидкості розробника. Замість того, щоб зберігати дані в таблицях і рядках, як у реляційній базі даних, у MongoDB документи зберігаються у колекціях (collections) у форматі даних, що є динамічними схемами-об'єктами, подібних до JSON.

Express.js - це фреймворк сервера web-додатків Node.js, розроблений для створення односторінкових, багатосторінкових і гібридних web-додатків. Це - стандартна серверна платформа для Node.js (рівень back-end).

Angular.js/React.js - це бібліотечна структура для динамічних web-програм (на рівні front-end). Вони дозволяють використовувати HTML як мову шаблонів і дозволяють розширювати синтаксис HTML, щоб чітко і лаконічно утворювати компоненти web-додатків.

Node.js - це кросплатформне середовище виконання з відкритим вихідним кодом для розробки web-додатків на стороні сервера. Програми Node.js написані на JavaScript і можуть запускатися в середовищі виконання Node.js на Mac OS X, Microsoft Windows, Linux, FreeBSD, NonStop, IBM AIX, IBM System.

Важлива ***особливість стеку MEAN/MERN***:

- перехід від генерації web-сторінок на стороні сервера до створення переважно односторінкових додатків;
- перенесення ядра реалізації архітектурного шаблону MVC зі сторони сервера на сторону клієнта, що забезпечується включенням в склад стеку Angular.js/React.js;

• включений в склад фреймворк Express.js забезпечує і традиційну маршрутизацію, і генерацію сторінок на стороні сервера.

Програмна платформа Node.js

Node.js - платформа з відкритим кодом для виконання високопродуктивних мережевих додатків, написаних мовою JavaScript. Платформу розробив у 2009 р. Раян Дал (англ. Ryan Dahl). Якщо раніше Javascript застосовувався для обробки даних в браузері користувача, то node.js надав можливість виконувати JavaScript-***скрипти*** на сервері та відправляти користувачеві результат їх виконання. Платформа Node.js перетворила JavaScript на мову широкого загального використання з великою спільнотою розробників на рівнях back-end та front-end.

Раніше розробники могли запускати JavaScript тільки в браузері, але від тепер, коли вони розширили його, ви можете запускати JS на своєму комп'ютері в якості окремого (на ПК віртуального) серверного додатка. Цей додаток і буде використовувати платформу Node.js.

Node.js - ***це платформа, а не середовище розробки JavaScript-додатків***. Платформа окрім роботи із серверними скриптами для web-запитів, також використовується для створення клієнтських та серверних програм.

Із застосуванням Node.js розробляються DIRT-***додатки***. Абревіатура DIRT (data-intensive real-time) визначає додатки, що працюють у реальному часі та спроможні обробляти великі обсяги даних.

Платформою Node.js використовується розроблений компанією Google ***рушій V8***. V8 - рушій з відкритим вихідним кодом, написаний на C ++ (рис.1.7).

JavaScript-***рушій*** – це програма, або, іншими словами, інтерпретатор, що виконує код, написаний на JavaScript. Рушій може бути реалізований з використанням різних підходів: у вигляді звичайного інтерпретатора, у вигляді динамічного компілятора (або JIT-компілятора), який перед виконанням програми перетворює вихідний код на JS в байт-код якогось формату.

Рушій JavaScript виконує скрипти JavaScript, переважно, в браузерах, а з появою Node.js також і у серверній реалізації.

V8 використовує метод - приховані класи. Приховані класи схожі на звичайні класи в типовій об'єктно-орієнтованій мові програмування, на зразок Java, крім того, що створюються вони під час виконання програми.

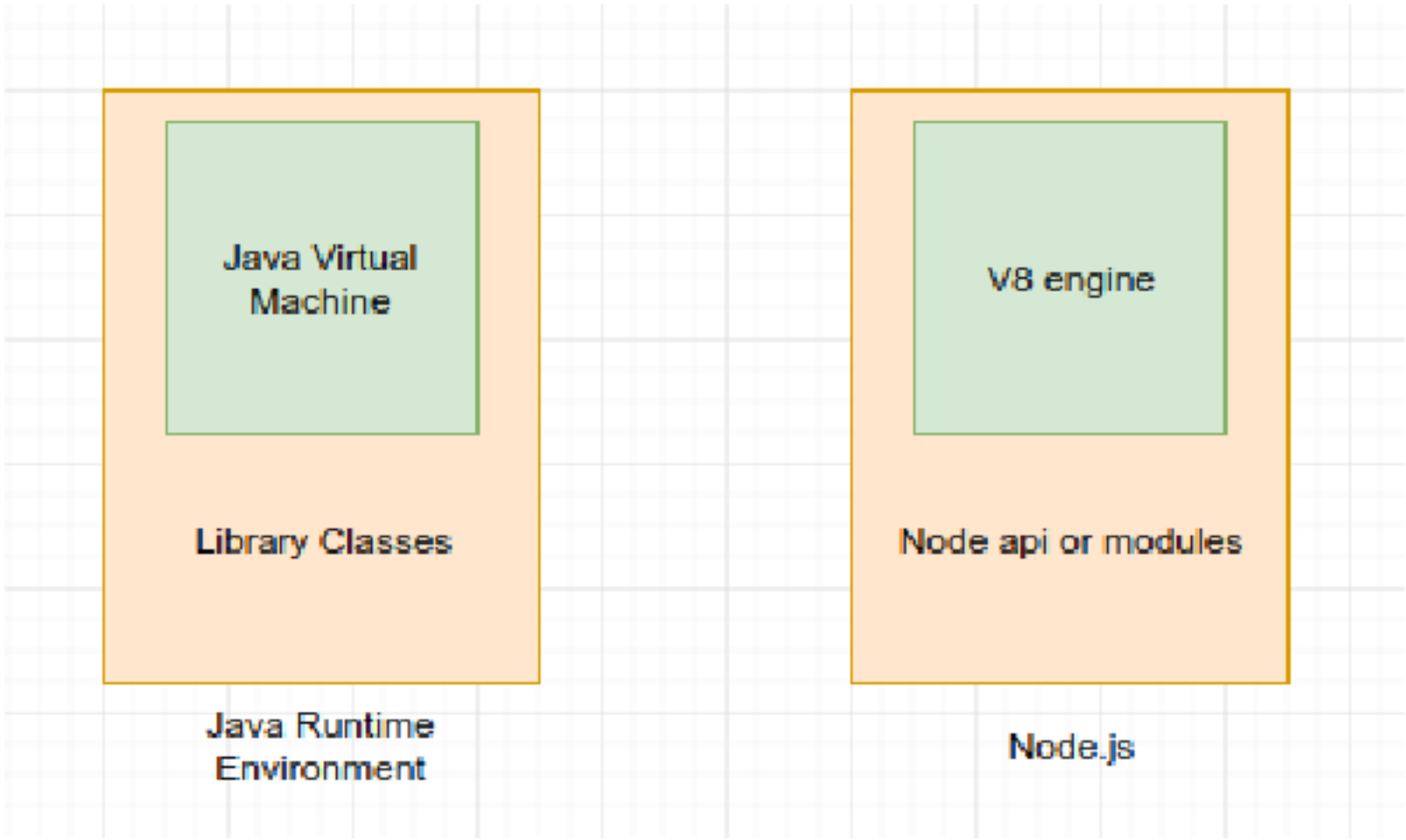


Рис. 1.7. Аналог JVM - рушій V8 [15]

Рушій V8 має такі **особливості** [21]:

- компіляція вихідного коду JavaScript безпосередньо у власний машинний код, минаючи стадію проміжного байт-коду;

- ефективна система керування пам'яттю, яка дозволяє швидко резервувати місця для об'єкту та зменшити очікування на прибирання сміття;

- V8 призупиняє виконання коду під час виконання прибирання сміття;

- V8 регулює використання пам’яті та точно визначає, де містяться в пам'яті об'єкти й вказівники/посилання, що дозволяє запобігати витоку інформації з пам'яті при помилковій ідентифікації об'єктів як посилань;

- використовує приховані класи й вбудований кеш, що значно прискорює доступ до властивостей та виклики функцій.

Node.js має наступні **властивості**:

- асинхронна одно-ланцюгова модель виконання запитів;
- неблокуюче введення/виведення;

Якщо додаток працює синхронно, то користувач не зможе взаємодіяти зі сторінкою, поки не прийде результат. Обробка може тривати досить довго.

Для забезпечення обробки великої кількості паралельних запитів у Node.js використовується **асинхронна модель запуску коду**, яка заснована на обробці подій в неблокуючому режимі та визначенні обробників зворотніх викликів (callback). В цьому випадку головний потік виконання поділяється на дві гілки. Одна з них продовжує займатися інтерфейсом, а інша виконує запит.

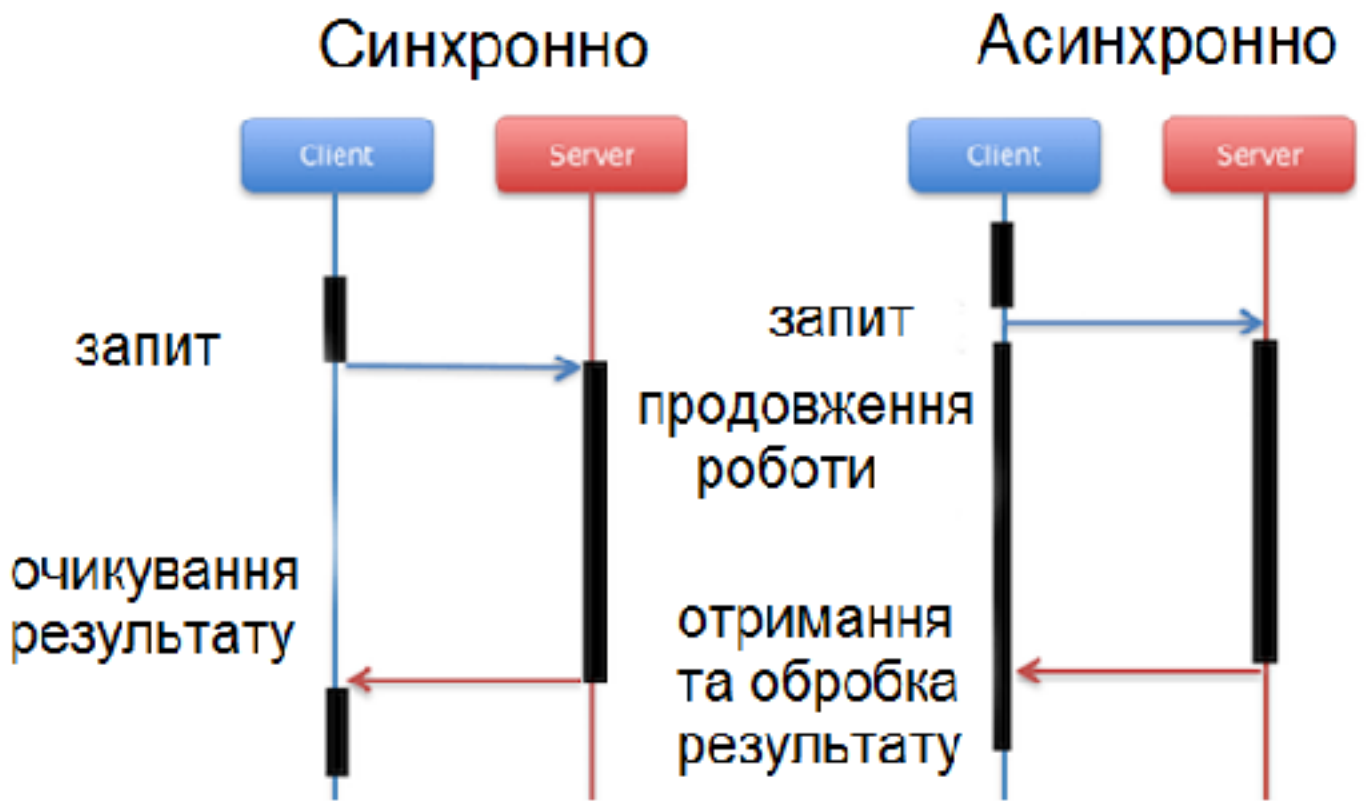


Рис. 1.8. Відмінність синхронної та асинхронної моделей виконання запитів [22]

Асинхронність

Ідея асинхронного виконання полягає в тому, що початок і кінець однієї операції може відбуватися в різний час та в різних частинах коду. Щоб отримати результат синхронно, необхідно почекати закінчення іншої операції, причому час очікування є непередбачуваний.

Коли мова заходить щодо асинхронності, використовують ще три близькі поняття. Це конкурентність (concurrency), паралелізм (parallel execution) і багатопотоковість (multithreading) [23]. Всі вони пов'язані з одночасним виконанням завдань, проте це не одне і те ж (рис.1.9).

Рис. 1.9. Ілюстрація виконання послідовного, паралельного та конкурентного виконання

Паралелізм

Паралельне виконання використовується для поділу одного завдання на частини для прискорення обчислень. Наприклад, потрібно зробити кольорове зображення чорно-білим. Обробка верхньої половини не відрізняється від

обробки нижньої. Отже, можна розділити цю задачу на дві частини і роздати їх різним потокам, щоб прискорити виконання в два рази.

Наявність двох фізичних потоків тут принципово важливо, так як на комп'ютері з одним обчислювальним пристроєм (процесорним ядром) такий прийом провести неможливо.

Багатопотоковість

Тут потік є абстракцією, під якою може ховатися і окреме ядро процесора, і тред ОС. Деякі мови навіть мають власні об'єкти потоків. Таким чином, ця концепція може мати принципово різну реалізацію.

Може здатися очевидним, що багатопотокове середовище зможе обробляти більше запитів за секунду, ніж однопотокове середовище. Це, як правило, справедливо для запитів із інтенсивними обчисленнями, які інтенсивно використовують виділені ресурси.

Однак у випадках, коли запити включають багато введення/виведення, наприклад виклики бази даних або web-служб, кожен запит повинен чекати, поки зовнішній механізм відповість на зроблені виклики, і, отже, виділені ресурси CPU і пам’яті не використовуються під час цього очікування. Тому, у програмах обробки даних, де сервер повинен обробляти велику кількість одночасних запитів, кожен з яких включає час очікування введення/виведення, однопотокова модель асинхронного вводу-виводу NodeJS забезпечує значну перевагу над багатопоточною моделлю синхронного введення-виводу.

Конкурентність

Поняття конкурентного виконання саме загальне. Воно буквально означає, що безліч завдань вирішуються в один час. Можна сказати, що в програмі є кілька логічних потоків - по одному на кожну задачу.

При цьому потоки можуть фізично виконуватися одночасно, але це не обов'язково. Завдання при цьому не пов'язані один з одним. Отже, не має значення, яке з них завершиться раніше, а яке пізніше.

Node модулі

Node модулі це - багаторазово використовувані блоки коду (рівень КПВ), існування яких не впливає на інший код.

Ви можете написати свої власні модулі та їх використовувати в різних додатках. Node.js має набір вбудованих модулів, які можна використовувати без спеціальної установки.

Система модулів CommonJS - проект, метою якого є визначення екосистеми мови програмування JavaScript поза межами браузера (наприклад, серверний JavaScript або рідні додатки).

Обидва - браузерний JavaScript та Node.js запускаються в середовищі виконання Google рушієм V8 (вже згаданий вище). Цей движок використовує ваш JS код і перетворює його в більш швидкий машинний код. Машинний – це низькорівневий код, який комп'ютер може запускати без необхідності спочатку його інтерпретувати.

Для керування модулями Node.js використовується пакетний менеджер **npm** (node package manager).

Як способи мультиплексування з'єднань підтримується epoll, kqueue, /dev/poll і select. Для мультиплексування з'єднань використовується бібліотека libuv, для створення пулу нитей (thread pool) задіяна бібліотека libeio, для виконання DNS-запитів у неблокуючому режимі інтегрований c-ares.

Всі системні виклики, що спричиняють блокування, виконуються всередині пулу ланцюгів і потім, як і обробники сигналів, передають результат своєї роботи назад через неіменовані канали (pipe).

JavaScript Object Notation (JSON) - це стандартний текстовий формат для представлення структурованих даних на основі синтаксису об’єкта JavaScript.

JSON - це текстовий формат даних, що відповідає синтаксису об’єктів JavaScript, який був популяризований Дугласом Крокфордом.

Незважаючи на те, що він дуже нагадує синтаксис об'єктів JavaScript у літералі, його можна використовувати незалежно від JavaScript, а багато середовищ програмування мають можливість читати (аналізувати) та генерувати JSON.

Властивості JSON:

1. JSON існує у вигляді рядка - корисно, коли потрібно передавати дані через мережу. Його потрібно перетворити на власний об’єкт JavaScript, коли ви хочете отримати доступ до даних. Це не велика проблема - JavaScript надає глобальний об'єкт JSON, який має доступні методи для перетворення між ними.

2. Рядок JSON може зберігатися в окремому файлі, який є в основному просто текстовим файлом з розширенням **.json**.

3. JSON - це суто рядок із заданим форматом даних - він містить лише властивості, а не методи.

4. JSON вимагає використання подвійних лапок навколо рядків та назв властивостей. Поодинокі лапки не є дійсними, крім тих, що оточують весь рядок JSON.

5. Навіть одна неправильна кома або двокрапка може призвести до того, що файл JSON пішов неправильно і не працював. Ви повинні бути обережними, щоб перевірити будь-які дані, які ви намагаєтесь використовувати (хоча створений комп'ютером JSON рідше включає помилки, якщо програма

генератора працює належним чином). Ви можете перевірити JSON за допомогою такої програми, як JSONLint .

6. JSON насправді може приймати форму будь-якого типу даних, який допустимий для включення всередину JSON, а не лише масивів або об'єктів. Так, наприклад, один рядок або число буде дійсним JSON.

7. На відміну від коду JavaScript, у якому властивості об'єкта можуть не цитуватися, у JSON як властивості можуть використовуватися лише рядки, що цитуються.

Структура JSON

Формат JSON - це рядок, структура якого дуже нагадує літеральний формат об'єкта JavaScript. Ви можете включити ті самі основні типи даних всередині JSON, що і стандартний об’єкт JavaScript - рядки, числа, масиви, логічні значення та інші об’єкти. Це дозволяє побудувати ієрархію даних приблизно так [13]:

```
'{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
"active": true,
"members": [
  {
   "name": "Molecule Man",
   "age": 29,
   "secretIdentity": "Dan Jukes",
   "powers": [
    "Radiation resistance",
    "Turning tiny",
    "Radiation blast"
   ]
  },
  {
   "name": "Madame Uppercut",
   "age": 39,
   "secretIdentity": "Jane Wilson",
   "powers": [
    "Million tonne punch",
    "Damage resistance",
    "Superhuman reflexes"
```

```
      ]
    },
    {
      "name": "Eternal Flame",
      "age": 1000000,
      "secretIdentity": "Unknown",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ]
    }
  ]
}'
```

Якщо завантажити цей об’єкт до програми JavaScript, створивши змінну під назвою superHeroes, то можна потім отримувати доступ до даних всередині неї, використовуючи нотації “крапка” та “квадратна дужка” як для об’єкту JavaScript. Наприклад:

```
superHeroes.homeTown
superHeroes ['active']
```

Щоб отримати доступ до даних нижче за ієрархією, потрібно об’єднати необхідні імена властивостей та індекси масивів разом. Наприклад, щоб отримати доступ до третьої надможливості другого героя, перерахованого у списку, слід зробити таке:

```
superHeroes['members'][1]['powers'][2]
```

Розкриємо зміст такого запису коду JavaScript:

1. Спочатку задаємо ім'я змінної - superHeroes.

2. Всередині для того, щоб отримати доступ до властивості members - використовуємо ["members"].

3. members вміщує масив, заповнений об’єктами. Для того щоб отримати доступ до другого об’єкта всередині масиву, тому використовуємо данні з індексом 1.

4. Усередині цього об’єкта отримуємо доступ до властивості powers із застосуванням запису ["powers"].

5. Безпосередньо усередині властивості powers розміщується масив, що містить надможливості обраного супергероя. Для отримання третьої надможливості використовуємо данні з індексом 2.

NPM (менеджер пакетів Node)

Пакет (може також називатися модулем або бібліотекою) - є набором функцій або класів, які виконують визначене для них завдання.

NPM пакет - проект, який знаходиться у відкритому доступі для використання сторонніми програмістами у реєстрі NPM.

npm - найбільший у світі реєстр програмного забезпечення.

Розробники з відкритим кодом з усіх континентів використовують npm для спільного використання та запозичення пакетів, а багато організацій використовують npm для управління власними розробками.

Це бібліотеки, побудовані співтовариством. Вони вирішують більшість проблем, що часто зустрічаються.

npm містить пакети, які ви можете використовувати в своїх додатках, щоб зробити вашу розробку більш швидкою та ефективною.

npm складається з трьох різних компонентів:

Перш за все, це Internet-сховище для публікації проектів з відкритим кодом Node.js.

По-друге, це інструмент CLI, який допомагає встановлювати ці пакети та керувати їх версіями та залежностями. Існує сотні тисяч бібліотек і програм Node.js на npm, і багато інших додаються щодня.

По-третє – реєстр є великою громадської базою даних програмного забезпечення JavaScript і мета-інформації навколо нього.

npm сам по собі не запускає жодних пакунків-модулів.

Якщо ви хочете запустити пакет за допомогою npm, ви повинні вказати цей пакет у своєму package.json файлі.

Коли виконувані файли встановлюються через пакети npm, npm створює посилання на них:

- **локальні установки** мають посилання, створені в ./node_modules/.bin/каталозі;

- **глобальні встановлення** мають посилання, створені з глобального bin/каталогу (наприклад: /usr/local/biny Linux або в /AppData/npmWindows).

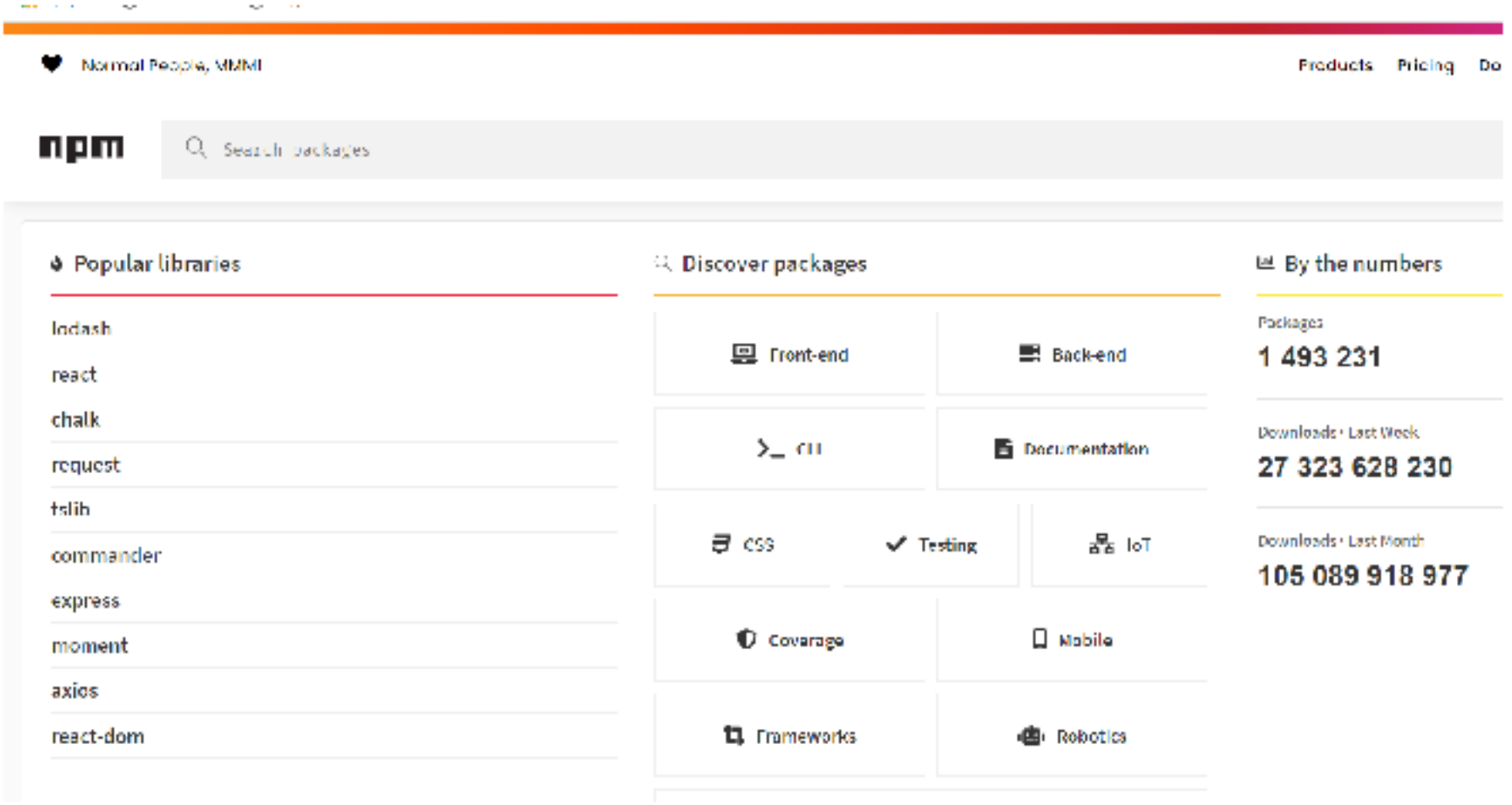


Рис. 1.10. Вигляд ресурсу[20]

Принцип роботи Node.js

Розглянемо принцип роботи **Node.js** на прикладі програми, що запускає web-сервер, виводить в консоль повідомлення, та на кожен HTTP запит відповідає повідомленням «Hello World» [15]:

```
        var http = require('http');      // Завантажуємо модуль http
// Створюємо web-сервер і вказуємо функцію обробки запиту
        var server = http.createServer(function (req, res) {
            console.log ('Початок обробки запиту');
// Передаємо код відповіді і заголовки
            res.writeHead (200, {
                'Content-Type': 'text/plain; charset=UTF-8'
        });
            res.end ('Hello world!');
    });
// Запускаємо web-сервер
            server.listen (1991, "127.0.0.1", function () {
                console.log ('Сервер запущено за адресою http://127.0.0.1:1991/');
        });
```

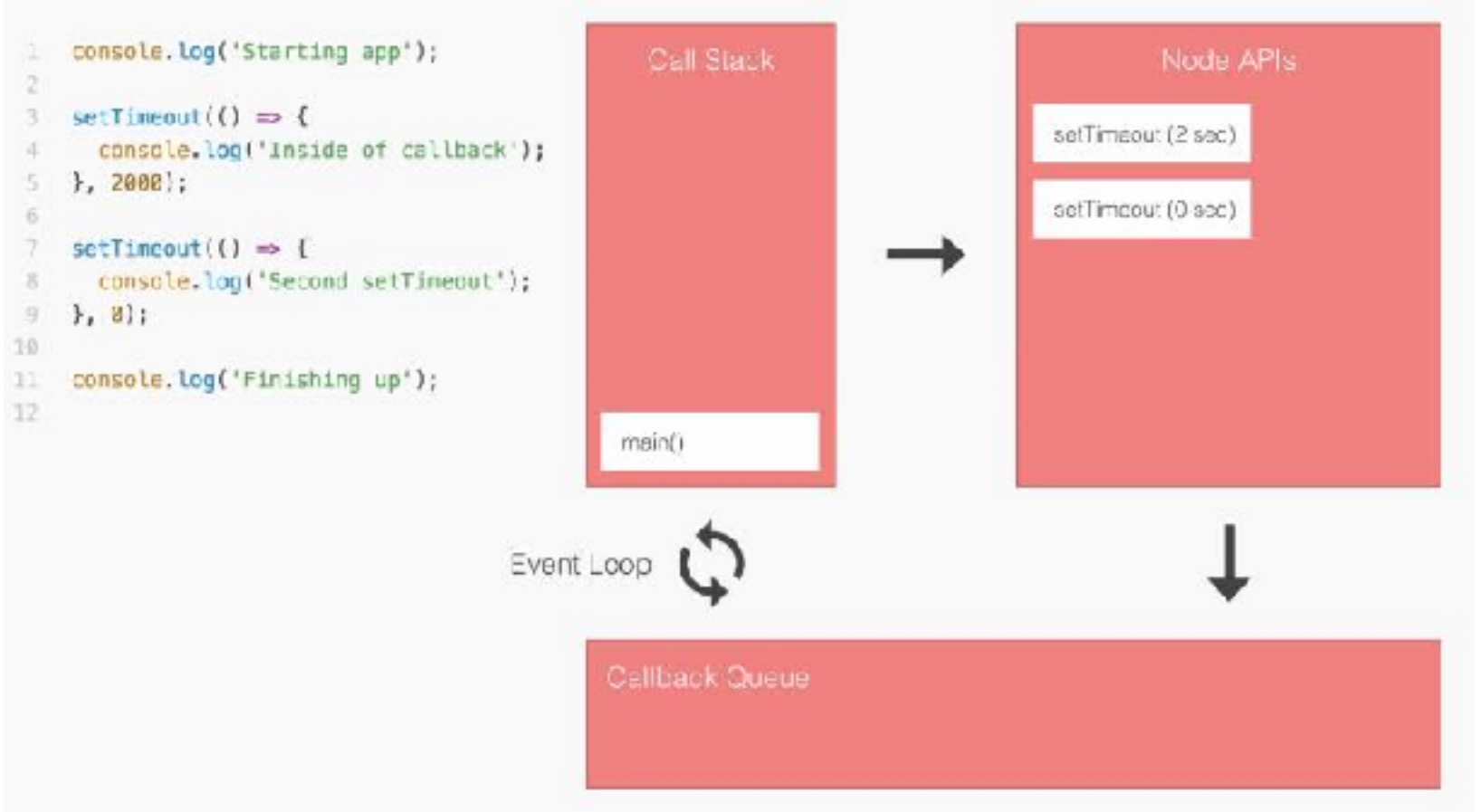


Рис. 1.11. Ілюстрація того, як працює цикл подій JavaScript [15]

Код виконує такі дії [15]:

1. Посилаєте main() в стек викликів.
2. Посилаєте console.log() в стек викликів. Він запускається відразу і з'являється.
3. Посилаєте setTimeout(2000) до стеку. setTimeout(2000) - це Node API. Коли ми викликаємо його, ми реєструємо пару подія-коллбек. Подія буде чекати 2000 мілісекунд, а потім викличе коллбек.
4. Після реєстрації, setTimeout(2000) з'являється в стеку викликів.
5. Тепер другий setTimeout(0) реєструється таким же чином. Тепер у нас є два API-інтерфейси Node, які чекають виконання.
6. Після очікування 0 секунд setTimeout(0) переміщується в чергу виконання коллбеків (callback queue), і те ж саме відбувається з setTimeout(2000).
7. У черзі виконання коллбеків функції чекають, коли стек викликів буде порожнім, тому що тільки одна функція може виконуватись одночасно. Це забезпечує event loop.
8. Остання викликається console.log(), а main() викликається з стека викликів.
9. Цикл подій бачить, що стек викликів порожній, а черги зворотного виклику - немає. Таким чином, він переміщує зворотні запити (по порядку) у стек викликів для виконання.

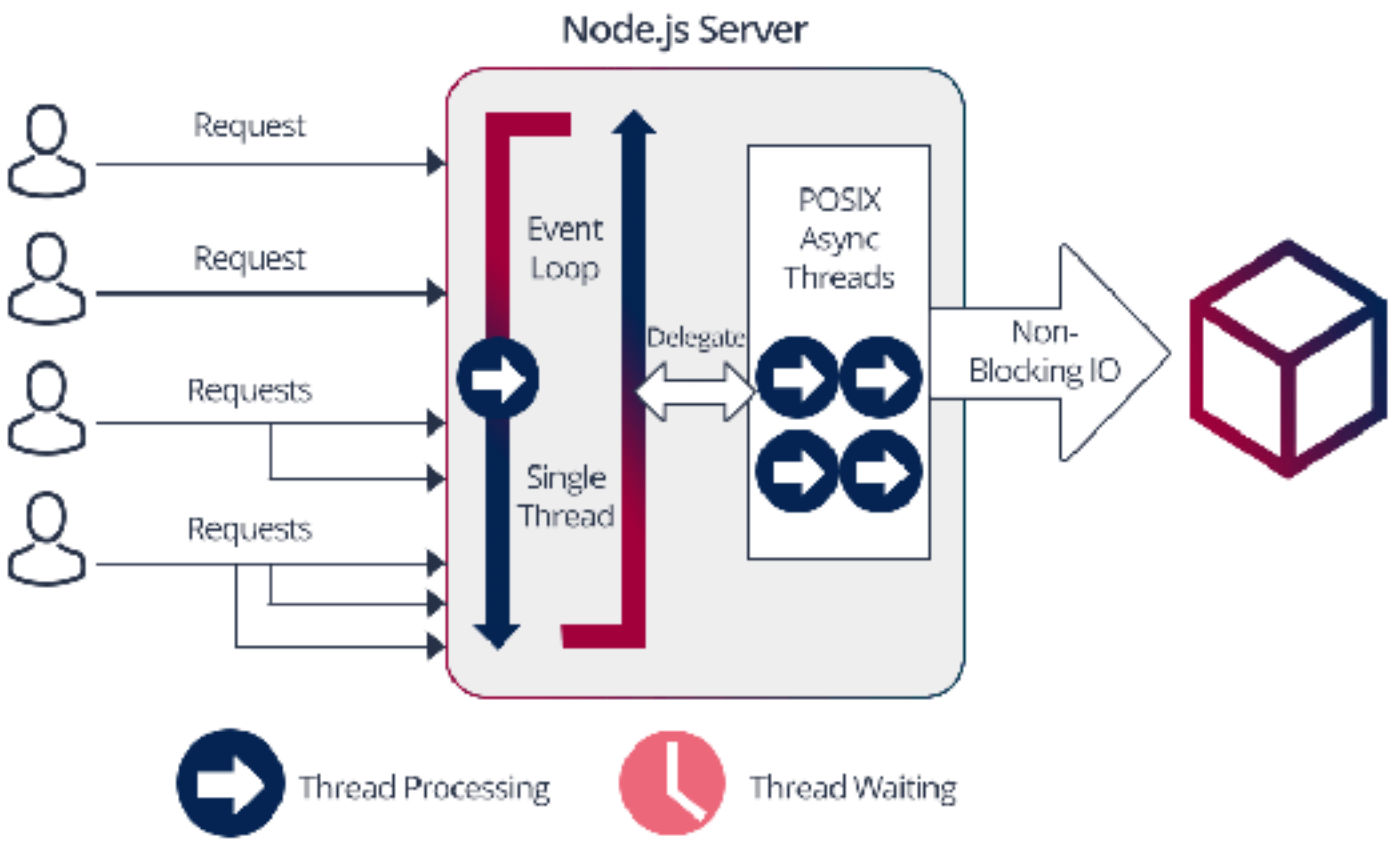


Рис. 1.12. Приклад використання node.js для виконання запиту POST request функції Google (запит до певного користувача, підключеного до кількох пристроїв) [16]

Require виконує три функції [16]:

1. Завантажує модулі, що поставляються в комплексі з Node.js наприклад файлової системи або HTTP з API Node.js.
2. Завантажує сторонні бібліотеки, такі як Express і Mongoose, які ви встановлюєте з npm.
3. Дозволяє створювати власні файли і ділити проект на модулі.

Require - це функція, і вона приймає параметр «шлях» і повертає module.exports.

Сервер на основі Node.js. Фреймворк Express.js

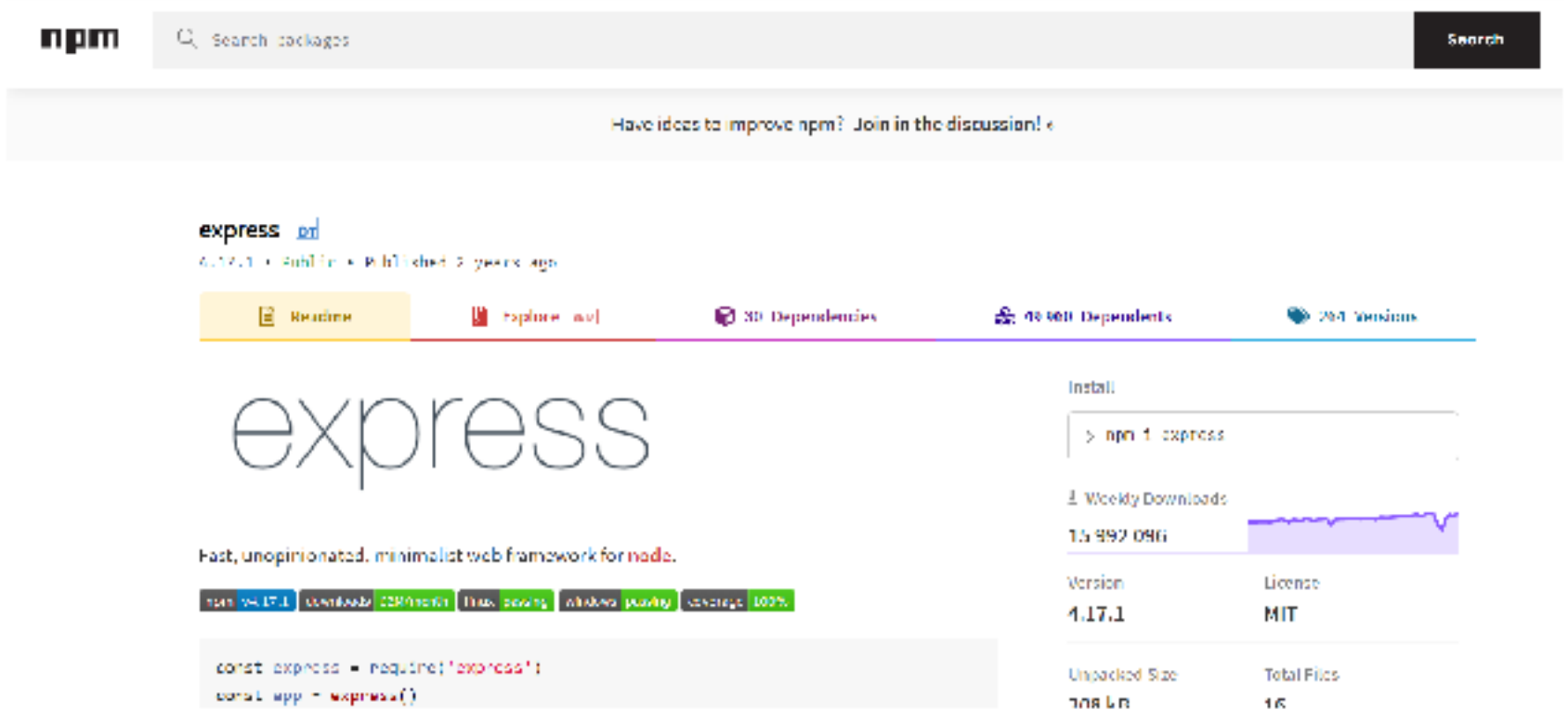


Рис. 1.13. Знаходження Express.js на [20]

Комбінація розробки Node.js та Express.js. Перша платформа займається побудовою додатків на сервері, а другий фреймворк публікує їх як web-сайти.

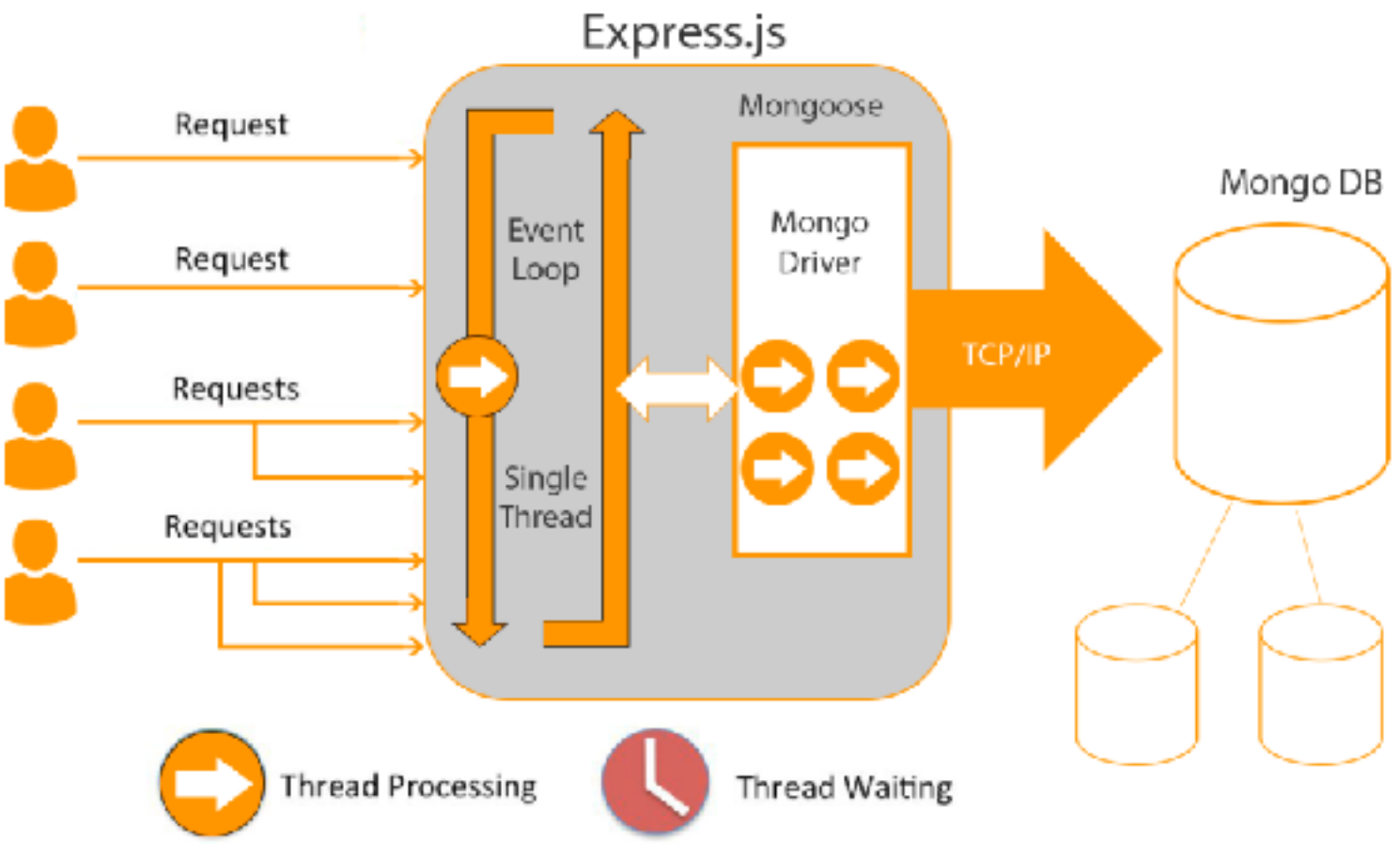


Рис. 1.14. Використання Express як сервера web-додатку [17]

Express - це мінімальна та гнучка рамка web-додатків Node.js, яка забезпечує надійний набір функцій для розробки web-та мобільних додатків. Це сприяє швидкому розвитку web-додатків на основі вузла.

Нижче наведено деякі основні риси Express Framework:

1. Дозволяє налаштовувати середні програми для відповіді на запити HTTP.

2. Визначає таблицю маршрутизації, яка використовується для виконання різних дій на основі методу та URL-адреси HTTP.

3. Дозволяє динамічно відтворювати HTML-сторінки на основі передачі аргументів у шаблони.

ExpressJS має наступні переваги:

- Якщо ви використовуєте Node.js для виробництва додатків, цей процес можна пришвидшити та вдосконалити за допомогою Express.
- На основі методів URL та HTTP можна визначити маршрути існуючої програми.
- Розробники Express.js можуть використовувати універсальні модулі програмного забезпечення для виконання додаткових завдань.

- Доступна інтеграція з різними шаблонів- препроцесорів Jade, Vash та інші.
- Ресурси та статичні файли програми легко обслуговувати.
- Створення сервера API REST - одна з найпривабливіших особливостей розробки додатків на Express.js.
- Ви можете скористатися з'єднанням з такими базами даних, як MySQL, Redis та MongoDB.

Продуктивність Express.js

Кількість запитів, що обробляються в секунду, є найбільш підходящим показником для визначення продуктивності в рамках. Оскільки час затримки в мілісекунді менш важливий, ніж абсолютна продуктивність, показник справедливий для багатьох web-додатків.

Ви можете знайти численні тести, зроблені на універсальних системах та за певних умов. Кількість запитів, зроблених протягом секунди, змінюється залежно від типу перевірених операцій. Проте вони показують базові показники.

Express.js дозволяє оптимізувати ефективність за допомогою певних прийомів як у коді, так і в середовищі.

Інструменти та засоби розробки

Додатково до інструментів та засобів, що запропоновані в попередніх підрозділах, для продовження розробки навчального проекту на базі стеку MEAN/MERN використовуємо такі інструменти та засоби, що мають безкоштовний доступ:

- MongoDB - документоорієнтована система управління базами даних, яка не потребує опису схем таблиць [19]. Є NoSQL-системою, використовує JSON-подібні документи та схему бази даних. Широко застосовується у web-розробці для роботи з JavaScript-орієнтованим стеком MEAN/MERN для створення web-додатків;

- npm - менеджер пакетів, який входить до складу Node.js, що використовується як репозиторій для публікації проектів з відкритим вихідним кодом, та як інструмент командного рядка, який допомагає взаємодіяти з браузерами та серверами, допомагає в установці та видаленні пакетів, керуванні версіями та залежностями, необхідними для роботи проекту [20].

- nodemon - модуль для перезавантаження сервера після зміни коду в проекті. Це інструмент для налагодження програм на основі node.js, який може виявляти зміни у файлах проекту та автоматично перезапускає процес.

1.2.2. ***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 1.3. Створення проекту web-додатку та налагодження зв'язку з ресурсними модулями Node.js. Тестовий запуск

Дія 1.3.1. Встановлення Node.js

- для Windows завантажуємо з сайту https://nodejs.org/download/release/latest/ останню версію з розширенням .msi або можете оновити за допомогою команди у командному рядку:

```
npm install npm -g
```

- після завершення установки Node.js і NPM будуть оновлені до останньої версії, а потім ви зможете очистити пакет виконавши команди:

```
npm cache clean -f
npm update -g
```

Дія 1.3.2. Встановлення Express.js.

Необхідно встановити глобально генератор додатків Express.

```
npm install -g express-generator
```

Дія 1.3.3. Створення папки проекту та перехід по неї

- на диску операційної системи Windows створюємо папку з ім'ям tours_city:

```
mkdir tours_city
```

- переходимо до цієї папки:

```
cd tours_city
```

Дія 1.3.4. Ініціалізація проекту.

- ініціалізуємо проект шляхом створення файлу конфігурації web-додатку package.json:

```
npm init
```

- далі вводимо в консолі:

```
package name: (tours_city)  // погоджуємось, натискаємо Enter
version: (1.0.0)   // пропускаємо, натискаємо Enter
description: Mern stack  // може бути будь що
entry point (index.js)  : app.js    // буде роутером і основним файлом проекту
                    // за замовчуванням пропонується index.js замінюємо на app.js
test command:   // пропускаємо
git repositoty:    // пропускаємо
keywords: mern, react     // може бути будь що
author: Yurii Tulashvili < y.tulashvili@lutsk-ntu.com.ua>
license:  // пропускаємо
 ->OK
```

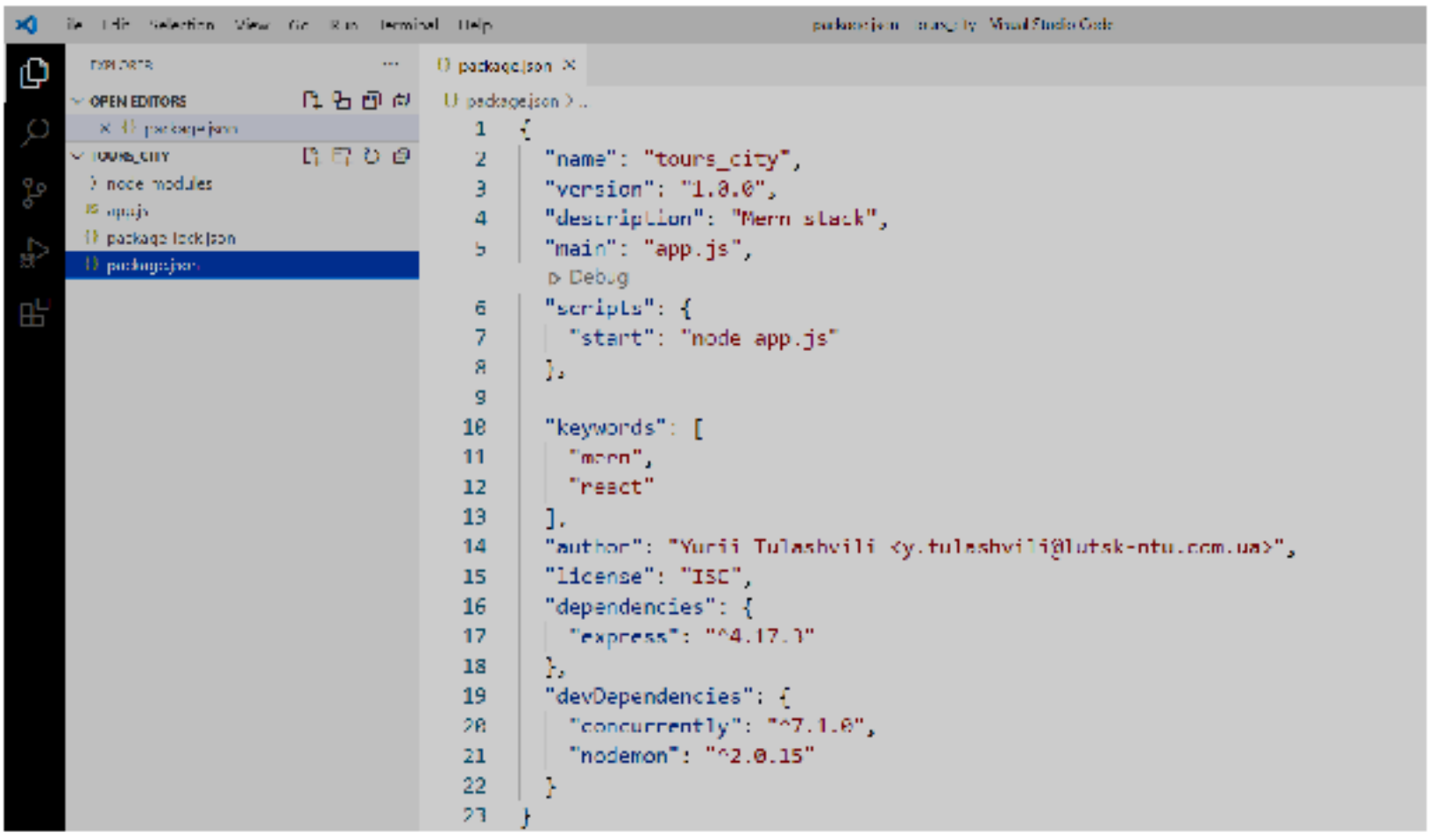


Рис. 1.15. Відкритий файл package.json у Visual Studio Code

Дія 1.3.5. Підключення конфігурації ресурсів проекту

- додаємо до проекту Express.js:

npm install express

- встановлюємо утиліти для роботи з проектом, що будуть потрібні під час розробки:

npm install -D nodemon concurrently

де -

nodemon – пакет, який дозволяє перезавантажувати сервер;

concurrently - буде потрібен пізніше, щоб запускати одночасно back-end та front-end;

npm install -D - щоб записалися залежності package.json у розділ devDependencies { }

Дія 1.3.6. Доопрацьовуємо файл package.json

- допрацьовуємо файл package.json для запуску з консолі. Переписуємо розділ "scripts":

```
"scripts": {
  "start": "node app.js"
},
```

Файл package.json набува вигляду - рис. 1.15.

Дія 1.3.7. Програмуємо перший тестовий запуск проекту

- створюємо в корні проекту за допомогою Visual Studio Code основний файл для back-end під ім'ям, як було вказано в package.json - app.js. Пишемо в середині файлу код:

```
const express = require('express')
const app = express()            // створюємо сервер під ім'ям app

app.get ('/', (req, res) => {    // відкриття web-сторінки
                res.send ('This is my project');
            });
app.listen (5000)  // запускаємо прослуховувач проекту
```

Дія 1.3.8. Виконуємо тестовий запуск проекту (сервера)

- у консолі вводимо:

  npm start

- у браузері вводимо:

  localhost:5000

- зупиняємо проект:

  Ctrl + C.

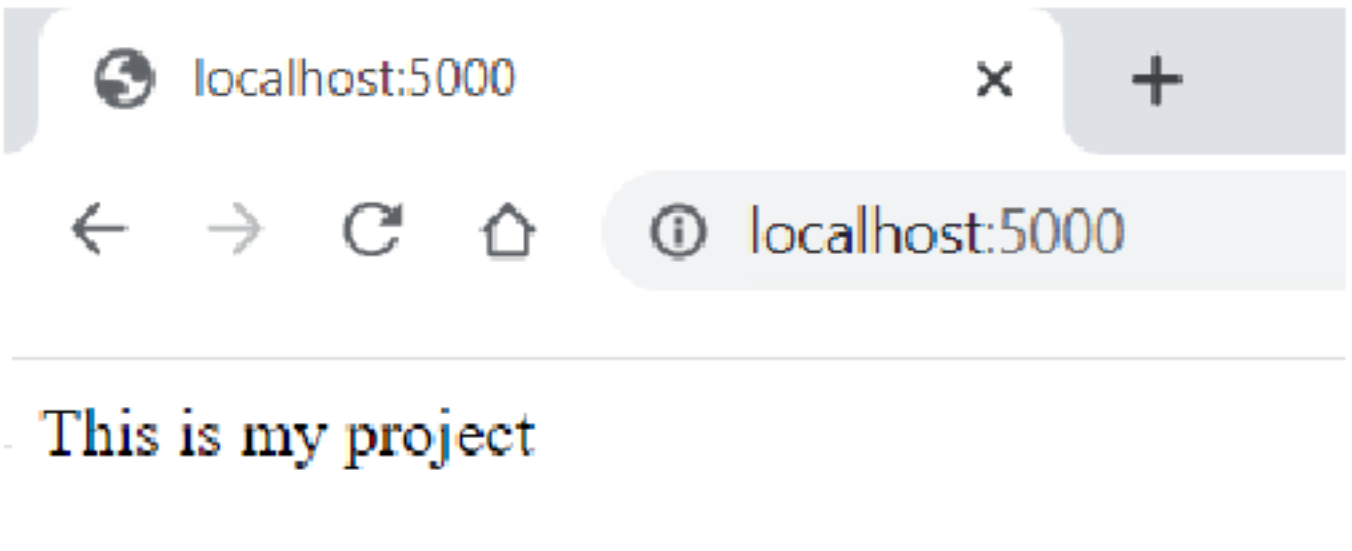


Рис. 1.16. Тестовий запуск проекту на http://localhost:5000/

1.3. СТВОРЕННЯ БАЗИ ДАНИХ WEB-ДОДАТКУ. MONGODB

1.3.1. Теоретичний модуль

Документо-орієнтована система керування базами даних (СКБД) з відкритим вихідним кодом - MongoDB

Зазвичай дані для web-додатку Express зберігають у який-небудь базі даних. Сьогодні є технології баз даних на самі різні випадки - традиційні сховища на основі SQL, документо-орієнтовані бази даних без використання SQL, прості сховища ключів і значень або web-служби запитів типу YQL.

Найпоширеною базою даних для платформи Node є документо-орієнтована база даних MongoDB. Основні можливості MongoDB:

MongoDB - відкрите програмне забезпечення, яке є розширюваною, високопродуктивною, вільною від схем, документо-орієнтованою базою даних, яка написана на C++. Розробляється з жовтня 2007 року компанією 10gen. Зберігає всі ваші дані у форматі бінарного JSON (BSON). Вже має широке застосування у реальних проектах.

Основні можливості MongoDB:

- Документо-орієнтоване сховище (проста та потужна JSON-подібна схема даних)
- Повна підтримка індексів
- Підтримка відмовостійкості і масштабованості
- Гнучка мова для формування запитів
- Динамічні запити
- Профілювання запитів
- Швидкі оновлення
- Журналювання операцій, що модифікують дані в БД
- Підтримка MapReduce

Документоорієнтована СУБД MongoDB призначена для зберігання даних в документах колекцій в форматі JSON (як звичайний текст), які не мають чіткої структури та призначені для зберігання будь-якого типу даних. Такий формат і організація зберігання даних в СУБД MongoDB забезпечує оперативну обробку даних, їх запис і виведення результатів. СУБД MongoDB призначена для роботи з нереляціоннимі даними, наприклад, для роботи з базою даних Website, яка може включати в себе наступні колекції (сутності): статті, новини, фото, архів тощо.

У MongoDB застосовується безсхемна організація неструктурованих або слабоструктурованих даних, тобто організація, яка не потребує опису схеми бази даних. Цей спосіб організації даних не призначений для того, щоб пов'язувати документи з даними однієї колекції з документами, що містять дані, інший колекції через ключові поля. У MongoDB немає власних ресурсів створення відносин між колекціями і документами.

Документоорієнтована СУБД MongoDB призначена для зберігання даних в документах колекцій в форматі JSON (як звичайний текст), які не мають чіткої структури та призначені для зберігання будь-якого типу даних. Такий формат і організація зберігання даних в СУБД MongoDB забезпечує оперативну обробку даних, їх запис і виведення результатів. СУБД MongoDB призначена для роботи з нереляціоннимі даними, наприклад, для роботи з базою даних Website, яка може включати в себе наступні колекції (сутності): статті, новини, фото, архів тощо.

СУБД MongoDB пропонує документоорієнтовану модель даних. Якщо в традиційних реляційних БД основними компонентами є двовимірні таблиці

(кожної сутності або базового типу інформації призначається таблиця) з записами (рядками) і полями (стовпцями), то в MongoDB - тематичні колекції (сутності), які складаються з документів, що включають в себе набір текстових полів в форматі JSON, тобто пар: ключ-значення ({ "key": "value"}).

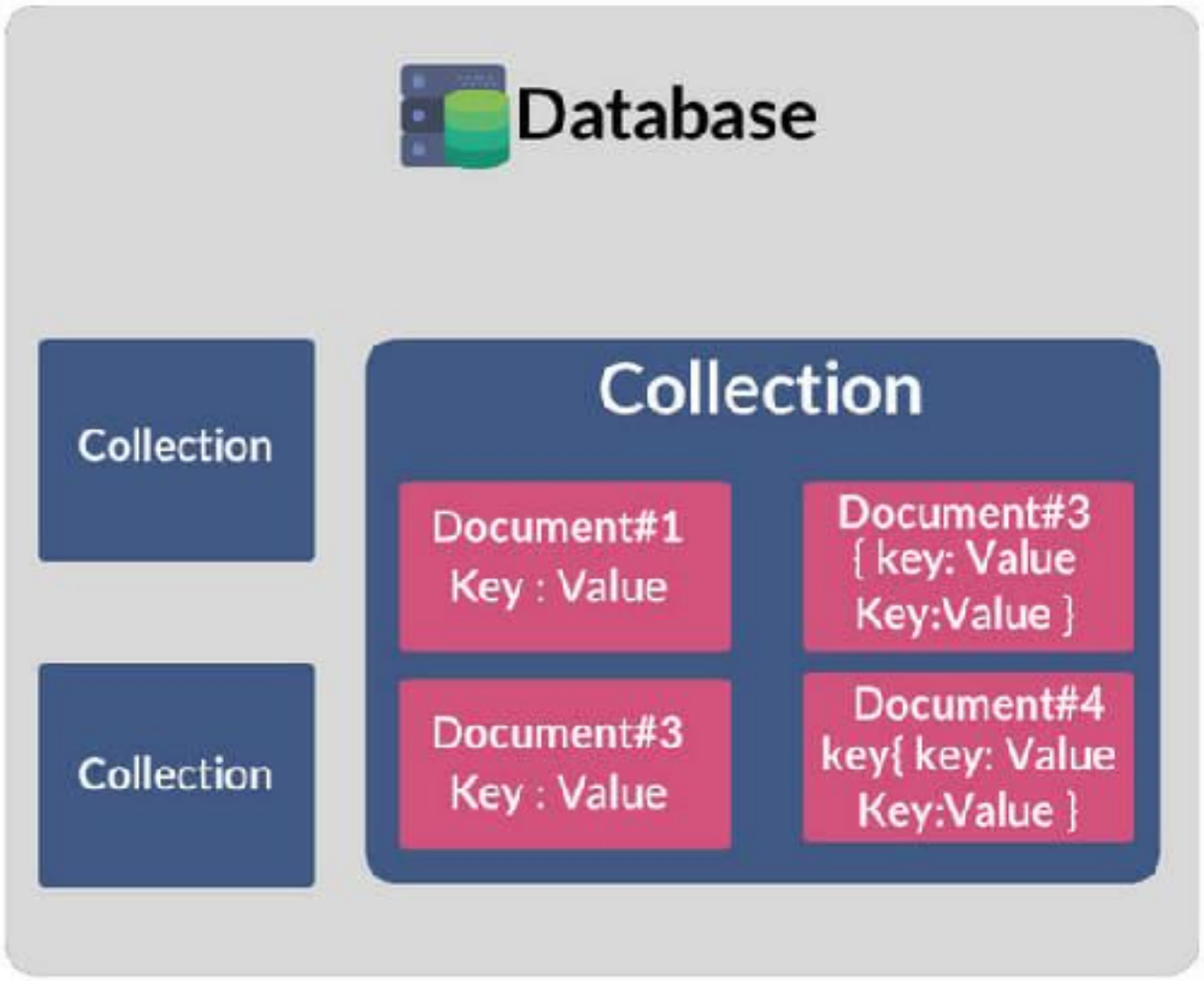


Рис. 1.17. Структура бази даних MongoDB [26]

Основною одиницею зберігання, доступу та обробки в базі MongoDB є документ. Документ має набір властивостей - атрибутів. Кожен атрибут в документі задається парою

<ім'я (ключ) атрибута>: <значення атрибута>.

Звідси випливає, що MongoDB добре працює з наборами даних (колекціями і документами), які не пов'язані між собою, і її можна використовувати для створення Web додатків, що забезпечують високу продуктивність і необмежену горизонтальне масштабування колекцій даних [27].

Кожна **колекція** має своє унікальне ім'я - довільний ідентифікатор, що складається з не більше ніж 128 різних алфавітно-цифрових символів і знака підкреслення.

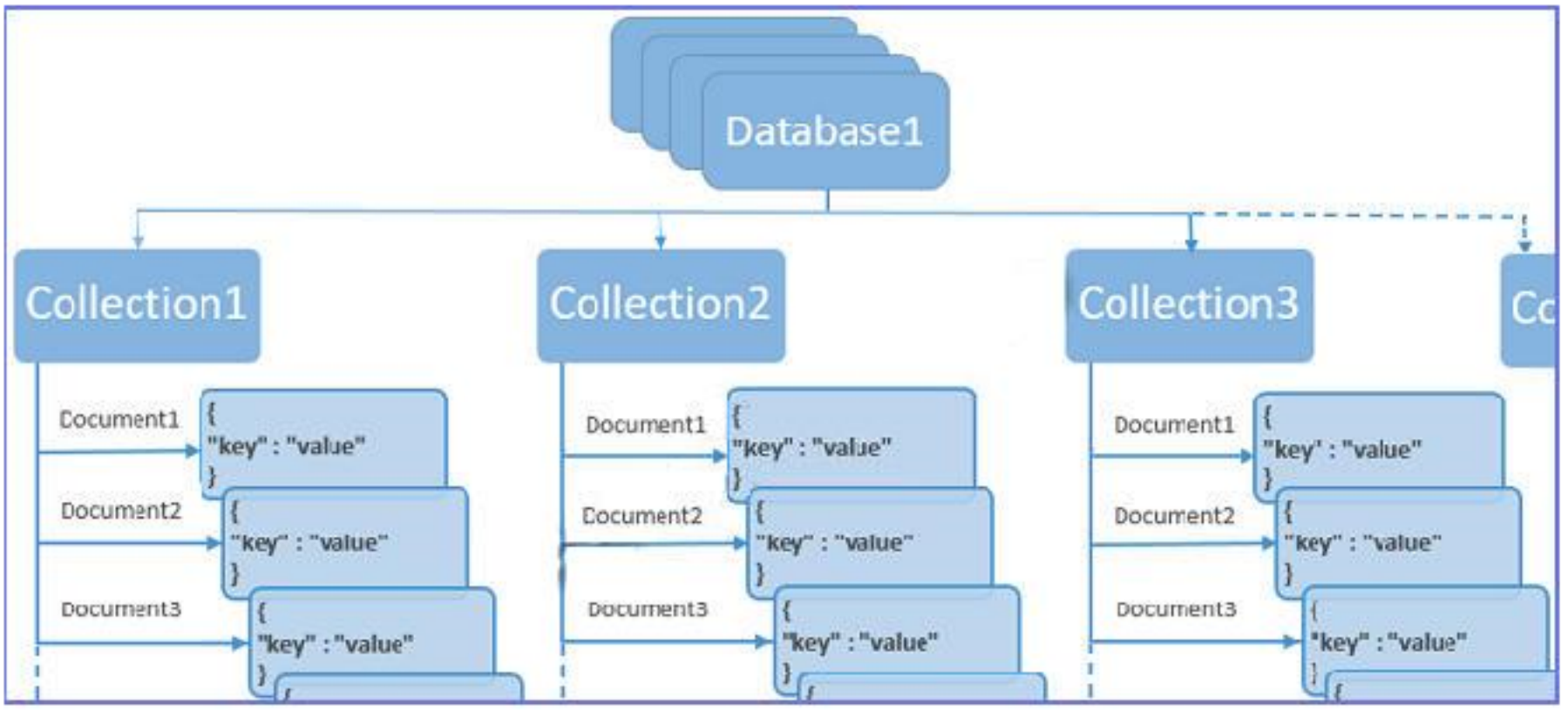


Рис. 1.18. Структура бази даних MongoDB [28]

На відміну від реляційних баз даних MongoDB не використовує табличний пристрій з чітко заданим кількістю стовпців і типів даних. MongoDB є документо-орієнтованої системою, в якій центральним поняттям є документ .

При використанні стандартної реляційної БД структура таблиць була б, приблизно, такою:

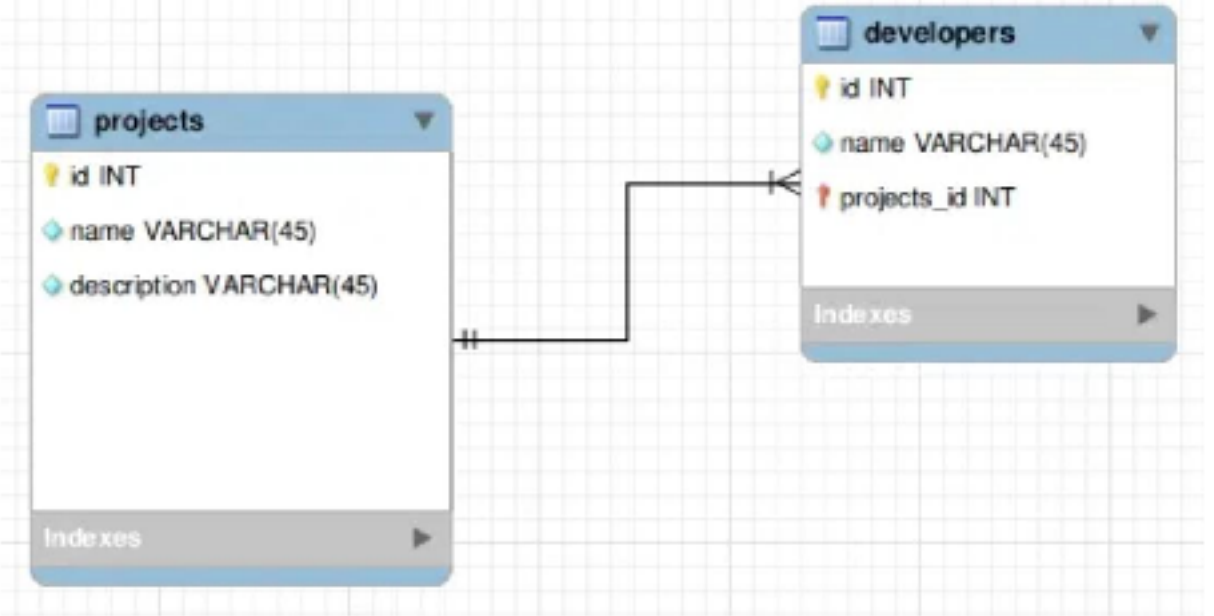


Рис. 1.19. Структура реляційної бази даних

У MongoDB у нас буде одна колекція проектів з наступною структурою:

```
{
  _id: PROJECT_ID
  name: NAME_OF_PROJECT,
  developers: [
    {
      name:'DEVELOPER_NAME',
    },
```

```
    {
      name: 'DEVELOPER_NAME'
    }
                ]
}
```

Для зв'язку MongoDB із додатком застосовується **mongoose** - інтерфейс між Node. Mongoose - одна з лідируючих «nosql» СУБД (nosql означає, що вона не базується на мові SQL). Mongoose - це бібліотека JavaScript, часто використовувана в додатках Node.js з базою даних MongoDB. В описі говориться, що це «масштабована, високопродуктивна, документо-орієнтована СУБД з відкритим вихідним кодом». Вона дозволяє зберігати документи в форматі, близькому до JSON, без чітко визначеної схеми, і має цілу низку передових можливостей.

За докладнішою інформацією та документацію можна знайти на сайті проекту http://www.mongodb.org/.- один з декількох модулів для доступу до MongoDB, що представляє собою засіб об'єктного моделювання, тобто ваша програма визначає об'єкти Schema, що описують дані, а mongoose бере на себе турботу про їх збереженні в MongoDB. Це надзвичайно потужний інструмент, що володіє такими засобами, як вбудовані документи, гнучка система типізації полів, контроль введення полів, віртуальні поля тощо.

На відміну від реляційних баз даних MongoDB не використовує табличний пристрій з чітко заданим кількістю стовпців і типів даних. MongoDB є документо-орієнтованої системою, в якій центральним поняттям є документ .

Документ можна уявити як об'єкт, який зберігає деяку інформацію. У певному сенсі він подібний до рядкам в реляційних СУБД, де рядки зберігають інформацію про окремий елемент. Наприклад, типовий документ:

```
{
   "name": "Bill",
   "surname": "Gates",
   "age": "48",
   "company": {
      "name" : "microsoft",
      "year" : "1974",
      "price" : "300000" }
}
```

Документ являє набір пар **ключ-значення**. Наприклад, в вираженні

```
"name": "Bill"
```

name представляє ключ, а Bill - значення.

Mongoose - це ODM (Object Document Mapper – об’єктно-документний утворювач). Це означає, що Mongoose дозволяє вам видобувати об’єкти за чітко визначеною схемою, що описує документ (колекцію) MongoDB.

Ключі подаються як рядковий тип даних. Значення ж можуть відрізнятися за типом даних. В даному випадку у нас майже всі значення також представляють строковий тип, і лише один ключ (company) посилається на окремий об'єкт.

Mongoose надає величезний набір функціональних можливостей для створення та роботи зі схемами. На даний момент Mongoose містить список SchemaTypes (типи даних схем), які можуть мати властивість, що зберігається в MongoDB.

Типи значень:

String : рядковий тип даних, як в наведеному вище прикладі (для рядків використовується кодування UTF-8);

Binary data (двійкові дані) : тип для зберігання даних в бінарному форматі

Boolean : булевий тип даних, який зберігає логічні значення TRUE або FALSE, наприклад,{"married": FALSE};

Date : зберігає дату в форматі часу Unix ISODate, наприклад, ("1831-02-16");

Double : числовий тип даних для зберігання чисел з плаваючою точкою;

Integer : використовується для зберігання цілочисельних значень, наприклад,{"age": 29};

JavaScript : тип даних для зберігання коду JavaScript;

Min key / Max key : використовуються для порівняння значень з найменшим / найбільшим елементів BSON;

Null : тип даних для зберігання значення Null;

Object : строковий тип даних, як в наведеному вище прикладі;

ObjectID : тип даних для зберігання id документа, наприклад,

_id: ObjectId ("559a8c8d250c6e372a67abbd");

Regular expression : застосовується для зберігання регулярних виразів;

Symbol : тип даних, ідентичний строковому. Використовується переважно для тих мов, в яких є спеціальні символи;

Timestamp : застосовується для зберігання часу;

Приклад документа, що містить відомості про автора Джеймса Олдріджа і заданих в масиві Books двох його книгах:

{
“au_id”: "000–11–0001",
“au_lname”: "Олдридж, Джеймс",
“years”: [1918, 2015],
“contract”: false,

```
“books”: [
{“title_id”: "bb1111",
 “title”: "Морський орел"},
 {“title_id”: "bb2222",
“title”: "Останній дюйм"}
            ]
}
```

На відміну від рядків документи можуть містити різнорідну інформацію. Так, поруч з документом, описаним вище, в одній колекції може знаходитися інший об'єкт, наприклад:

```
{
   "name": "Tom",
   "birthday": "1985.06.28",
   "place" : "Berlin",
   "languages" :[
      "english",
      "german",
      "spanish"
   ]  }
```

Здавалося б різні об'єкти за винятком окремих властивостей, але всі вони можуть перебувати в одній колекції.

Ще пара важливих зауважень: в MongoDB запити мають чутливі до регістру і строгою типізацією. Тобто такі два документу не будуть ідентичні:

```
{"age" : "28"}
{"age" : 28}
```

Якщо в першому випадку для ключа age визначена в якості значення рядок, то в другому випадку значенням є число.

Ідентифікатор документа

У процесі створення кожного документа (ObjectID) призначається "_id" унікальний ідентифікатор документа, який призначений для автоматичного сортування документів.

При додаванні документа в колекцію даний ідентифікатор створюється автоматично. Однак розробник може сам явно задати ідентифікатор, а не покладатися на автоматично генеруються, вказавши відповідний ключ і його значення в документі.

Дане поле повинно мати унікальне значення в рамках колекції. І якщо ми спробуємо додати в колекцію два документа з однаковим ідентифікатором, то додасться лише один з них, а при додаванні другого ми отримаємо помилку.

Якщо ідентифікатор не заданий явно, то MongoDB створює спеціальне бінарне значення розміром 12 байт. Це значення складається з декількох

сегментів: значення типу timestamp розміром 4 байта, ідентифікатор машини з 3 байт, ідентифікатор процесу з 2 байт і лічильник з 3 байт. Таким чином, перші 9 байт гарантують унікальність серед інших машин, на яких можуть бути репліки бази даних. А наступні 3 байта гарантують унікальність протягом однієї секунди для одного процесу. Така модель побудови ідентифікатора гарантує з високою часткою ймовірності, що він буде мати унікальне значення.

Структура таблиць в MongoDB

Якщо створення relational DB починається зі створення структури таблиць і установки зв'язків між таблицями, а потім заповнення таблиць і формування SQL запитів або інструкцій SQL, то в MongoDB спочатку створюють нову базу даних і задають її ім'я, потім при додаванні в неї даних вона автоматично створює колекцію і об'єкт (документ) з описом, що додається об'єкта в форматі JSON. Для зберігання даних в MongoDB застосовується формат, який називається BSON, скорочення від binary JSON.

В СУБД MongoDB для створення, читання, оновлення, видалення об'єктів застосовуються операції CRUD (create, read, update, delete).

Нижче наведено таблицю відповідності між основним термінами MySQL та MongoDB [29]:

Таблиця 1.2. Взаємозв'язок термінології СУБД із MongoDB

MySQL term	Mongo term/concept
database	database
table	collection
index	index
row	BSON document
column	BSON field
join	embedding and linking
primary key	_id field
group by	aggregation

Якщо у реляційних БД таблиця має вигляд:

Last Name	First Name	Date of Birth
DUMONT	Jean	01-22-1963
PELLERIN	Franck	09-19-1983
GANNON	Dustin	11-12-1982

тоді колекція документів у MongoDB виглядає абсолютно інакше:

```
{
   "_id": ObjectId("4efa8d2b7d284dad101e4bc9"),
   "Last Name": "DUMONT",
   "First Name": "Jean",
   "Date of Birth": "01-22-1963"
},
{
   "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
   "Last Name": "PELLERIN",
   "First Name": "Franck",
   "Date of Birth": "09-19-1983",
   "Address": "1 chemin des Loges",
   "City": "VERSAILLES"
}
```

Кожне поле документа може містити складну підструктуру довільної глибини вкладеності:

```
{
   "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
   "Last Name": "PELLERIN",
   "First Name": "Franck",
   "Date of Birth": "09-19-1983",
   "Address": {
      "Street": "1 chemin des Loges",
      "City": "VERSAILLES"
   }
}
```

За рахунок такої структури запити є простішим та логічнішими. Виконання операцій на великих об'ємах даних займає не більше (а, зазвичай, значно менше) часу.

Інструменти та засоби розробки

Додатково до інструментів та засобів, що запропоновані в попередніх підрозділах, для продовження розробки навчального проекту на базі стеку MEAN/MERN використовуємо такі інструменти та засоби, що мають безкоштовний доступ:

- MongoDB - документоорієнтована система управління базами даних, яка не потребує опису схем таблиць [19]. Є NoSQL-системою, використовує JSON-подібні документи та схему бази даних. Широко застосовується у web-розробці для роботи з JavaScript-орієнтованим стеком MEAN/MERN для створення web-додатків;

- mongoose - це бібліотека JavaScript, часто використовувана в додатках Node.js з базою даних MongoDB [30];

- config – модуль Node.js, що організовує ієрархічні конфігурації для розгортання програмного додатку. Дає змогу задавати набір параметрів за замовчуванням та розширити їх для різних середовищ розгортання. Конфігурації зберігаються у файлах конфігурації [31].

1.3.2. 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 1.4. Підключення до проекту web-додатку MongoDB

Дія 1.4.1. Встановлюємо пакет **config**

- організовуємо ієрархічну конфігурацію для зручності роботи з портами та з константами конфігурації програми, встановтвши пакет config. Він дає змогу визначати набір параметрів за замовчуванням та розширити їх для різних середовищ у середині проекту. У консолі вводимо:

```
npm i config
```

Дія 1.4.2. Встановлюємо пакет **mongoose**

- для підключення до бази даних MongoDB за допомогою пакету mongoose. У консолі вводимо:

```
npm i mongoose
```

Дія 1.4.3. Створення БД на mongodb.com

- для створення бази даних MongoDB виконуємо декілька послідовних дій на сайті ресурсу mongodb.com (рис. 1.20 – рис. 1.33) після реєстрування на ньому.

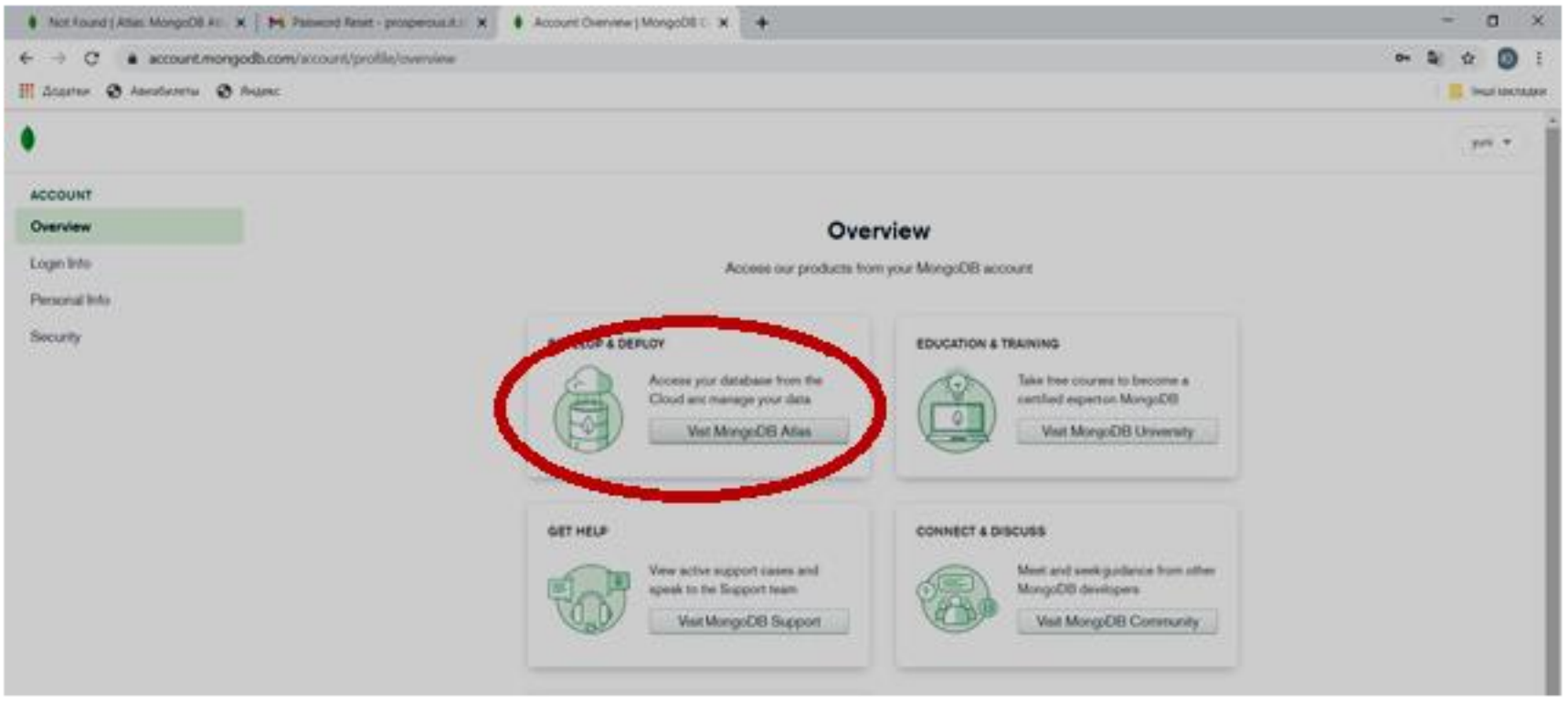


Рис. 1.20. Визначаємо прозначення бази даних

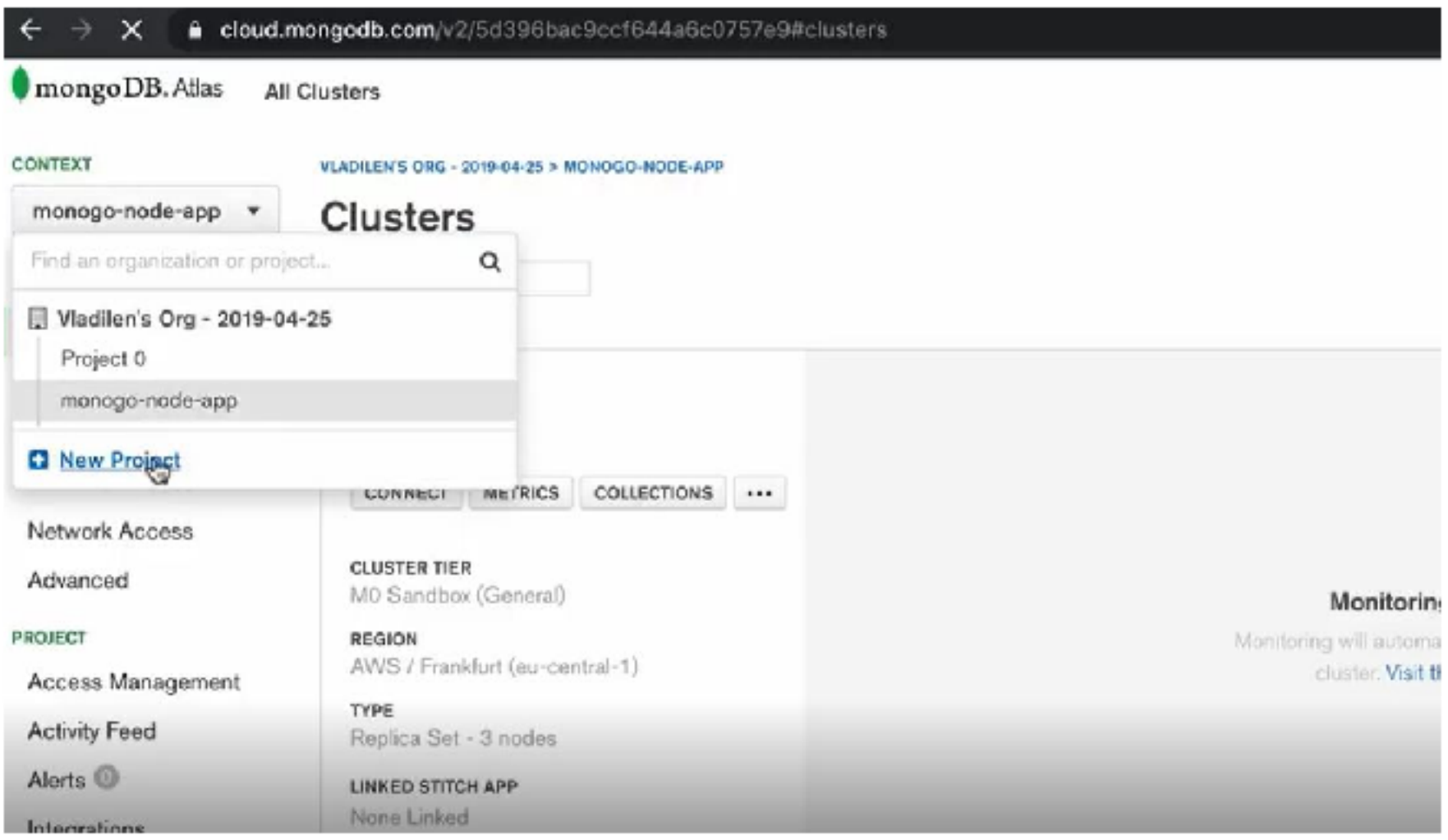


Рис. 1.21. Ініціюємо створення нового проекту

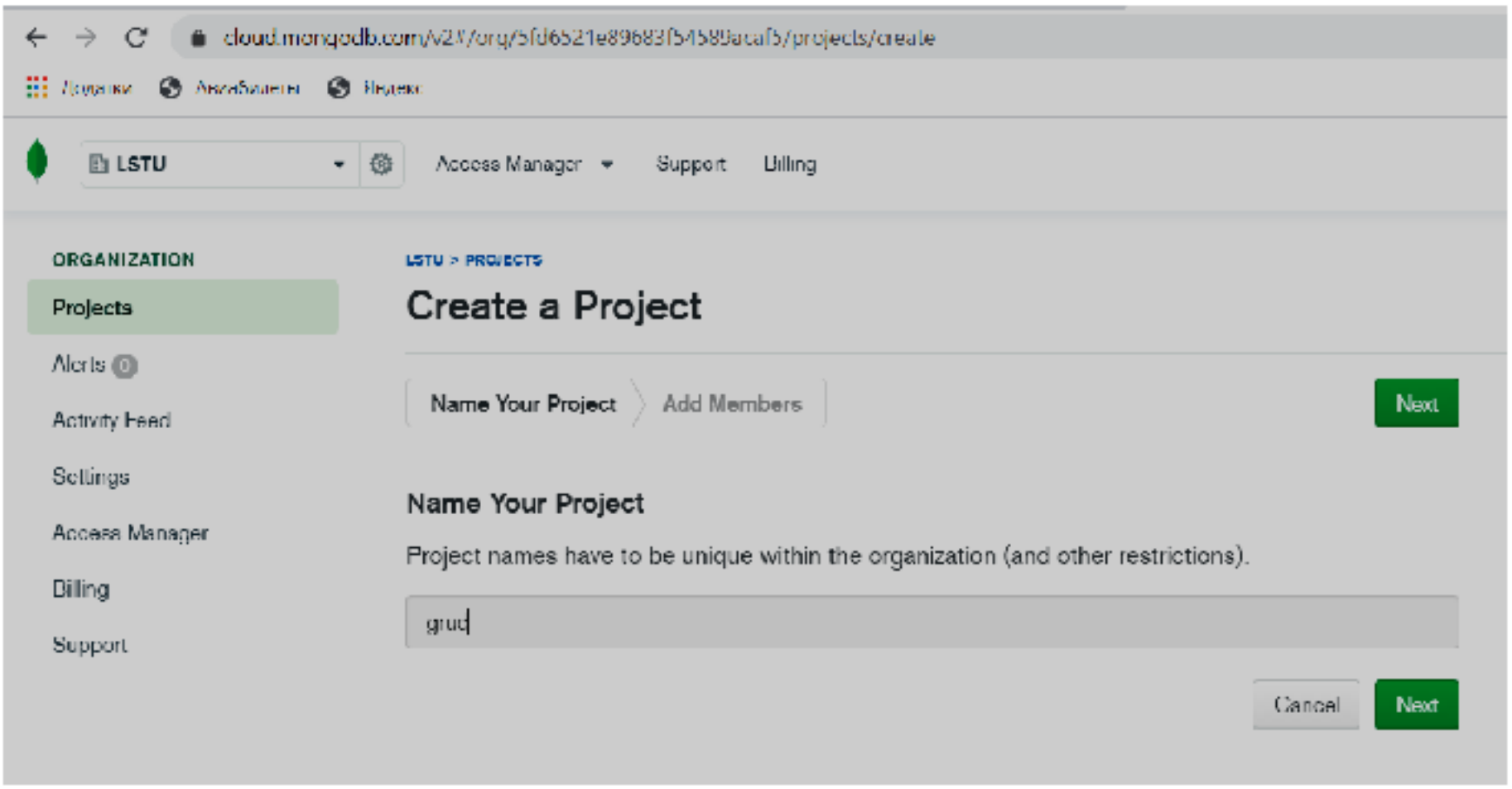


Рис. 1.22. Називаємо проект бази даних

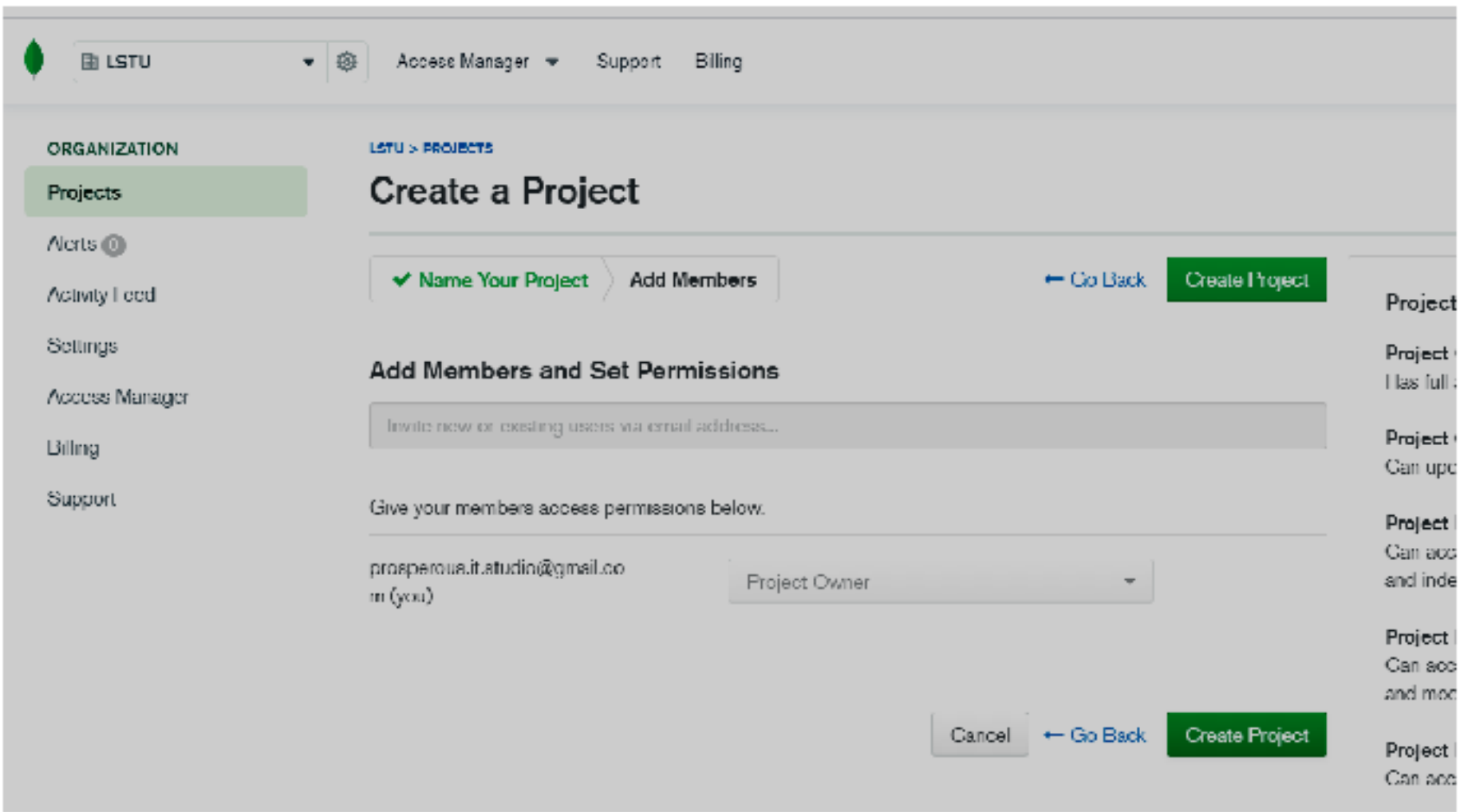


Рис. 1.23. Натискаємо Create Project

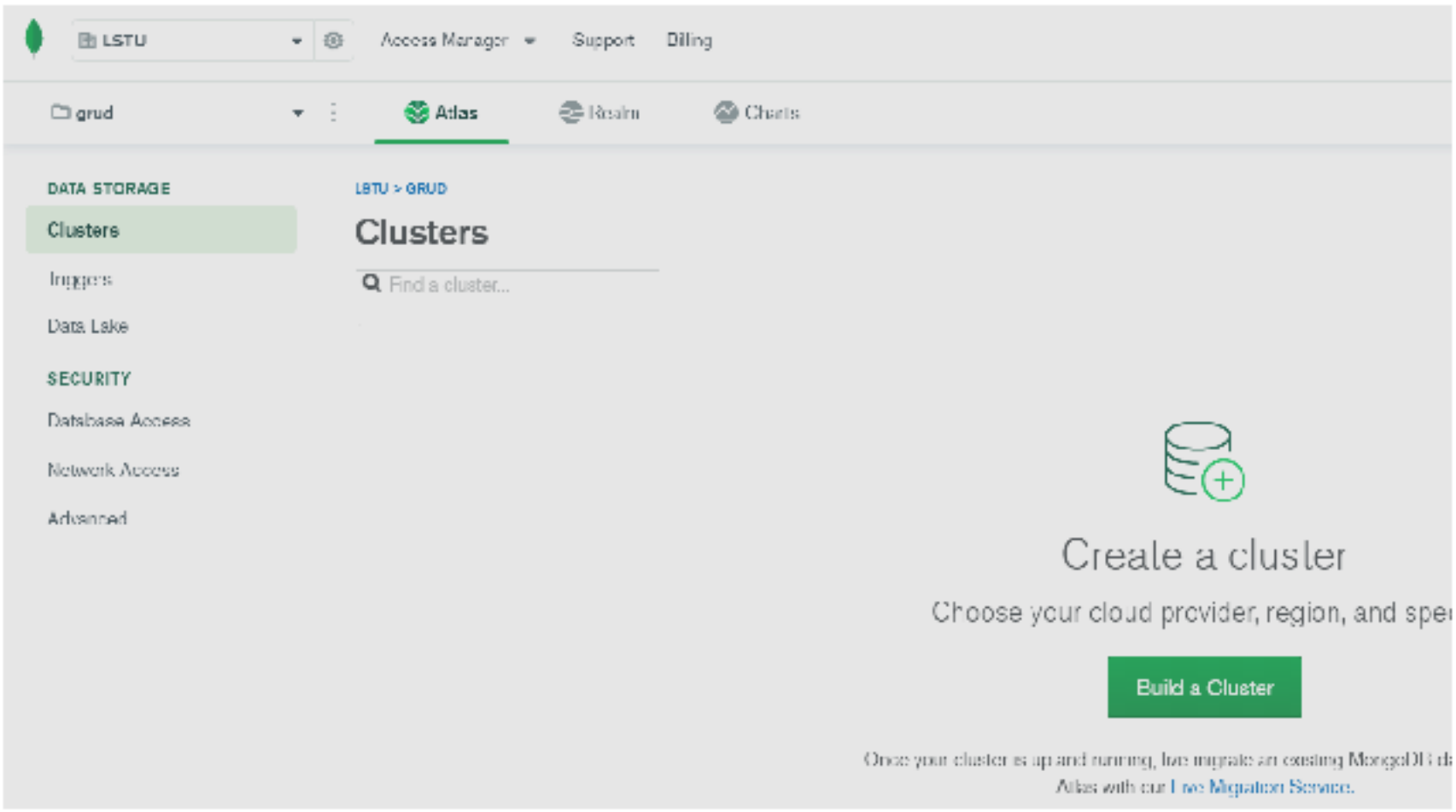


Рис. 1.24. Ініціюємо побудову нового кластеру. Далі обираємо Free

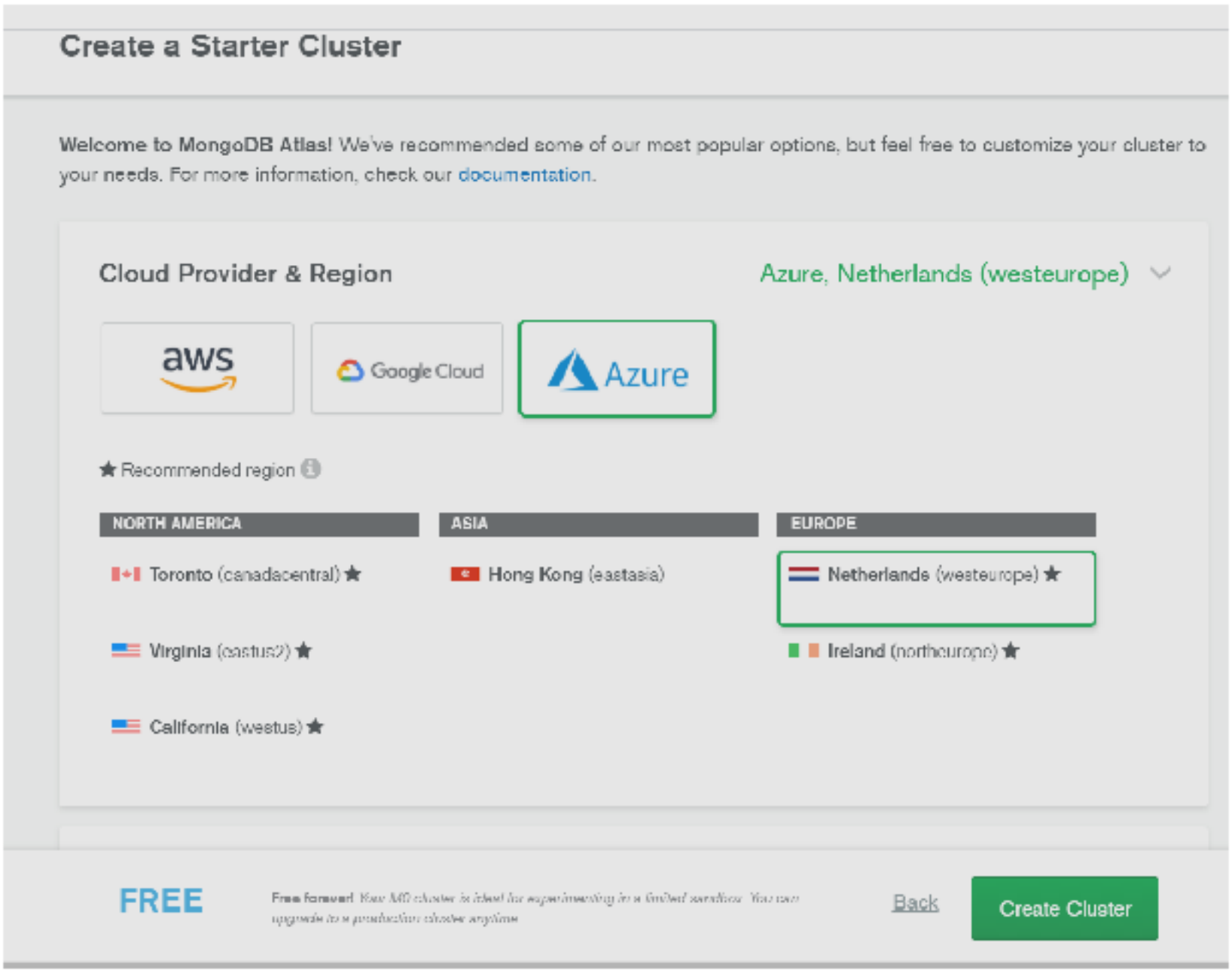


Рис. 1.25. Обраємо найближчий ресурс за географічним положенням.

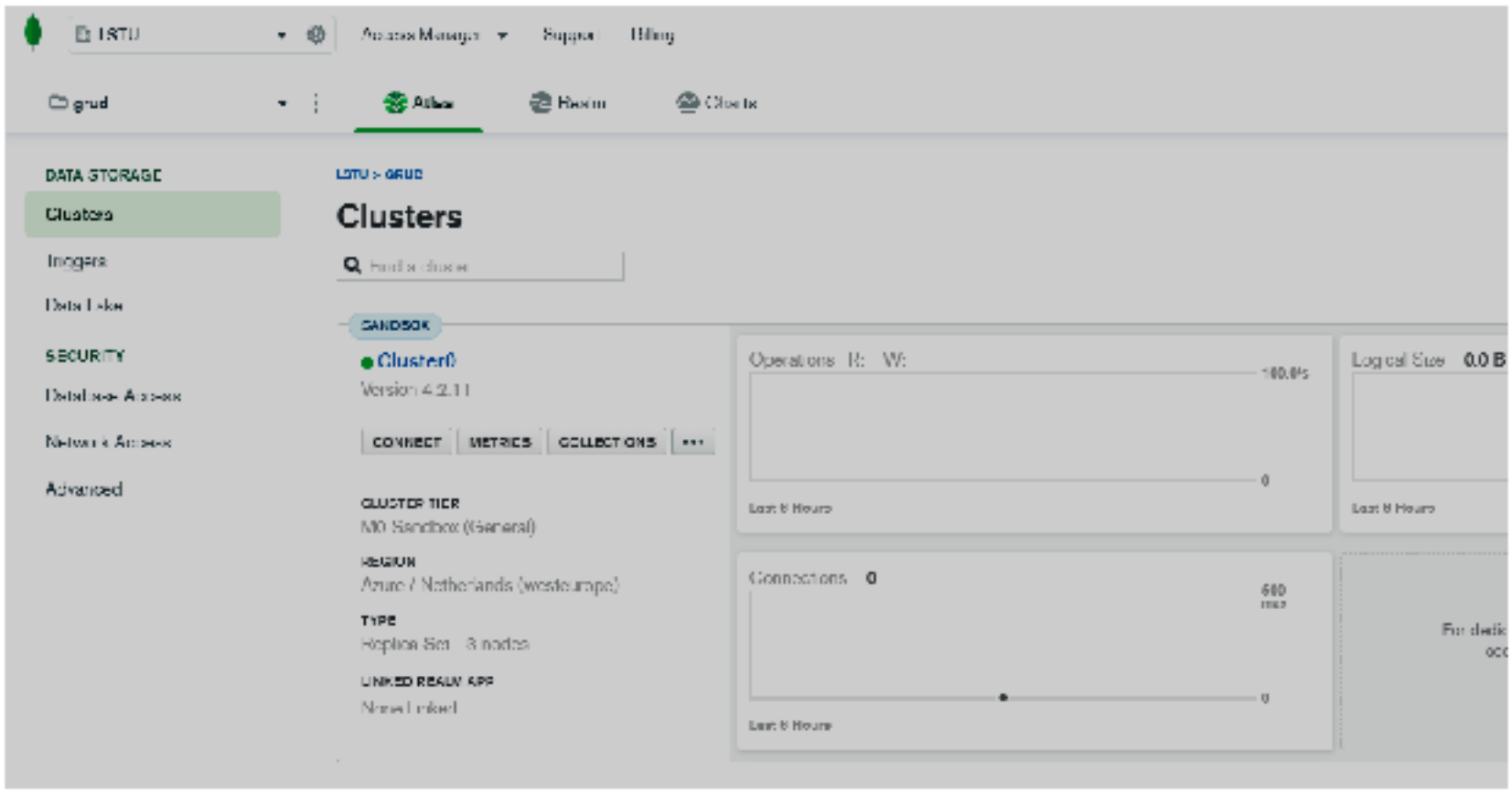


Рис. 1.26. Створюємо командою CONNECT

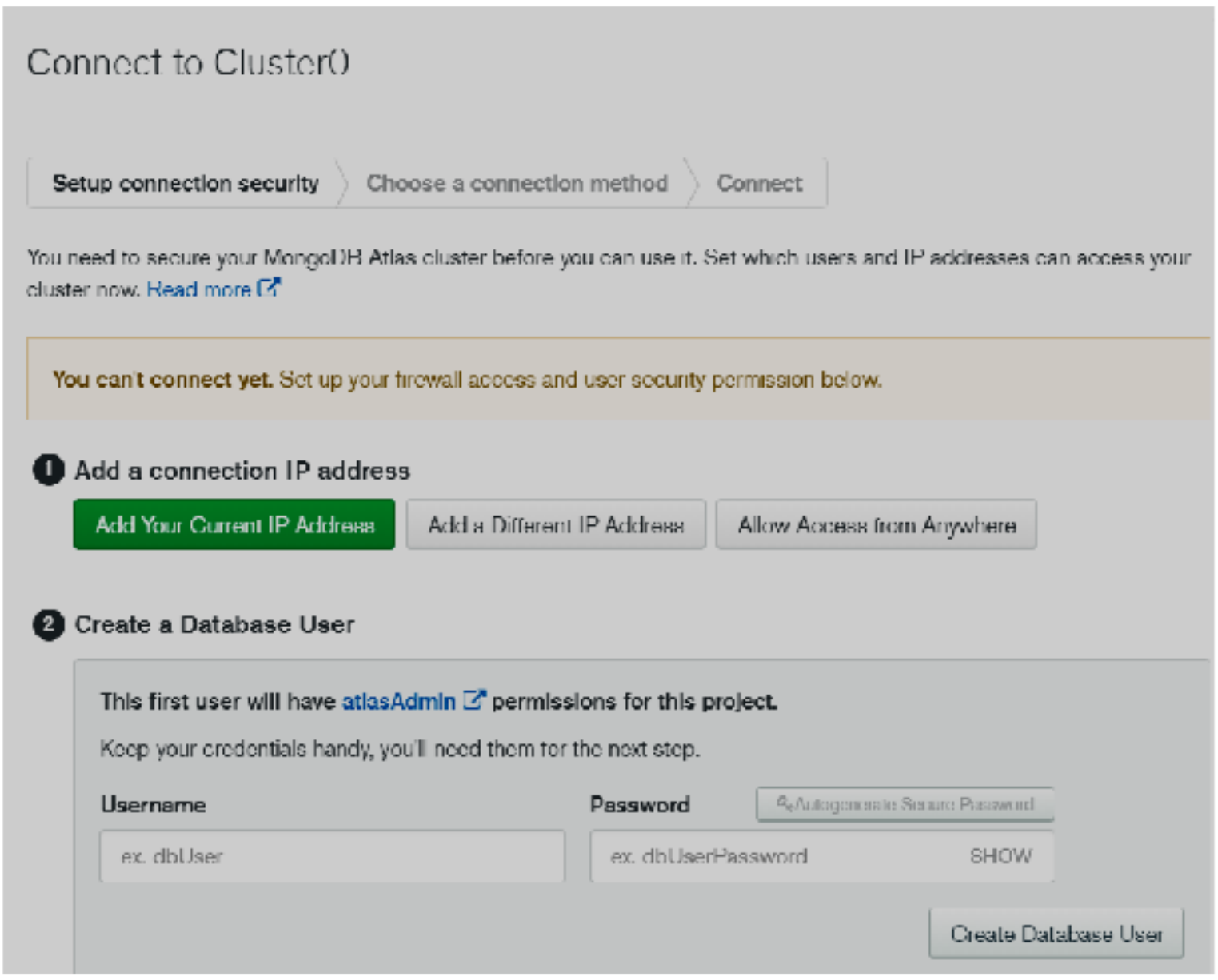


Рис. 1.27. Розпізнаємо наш IP

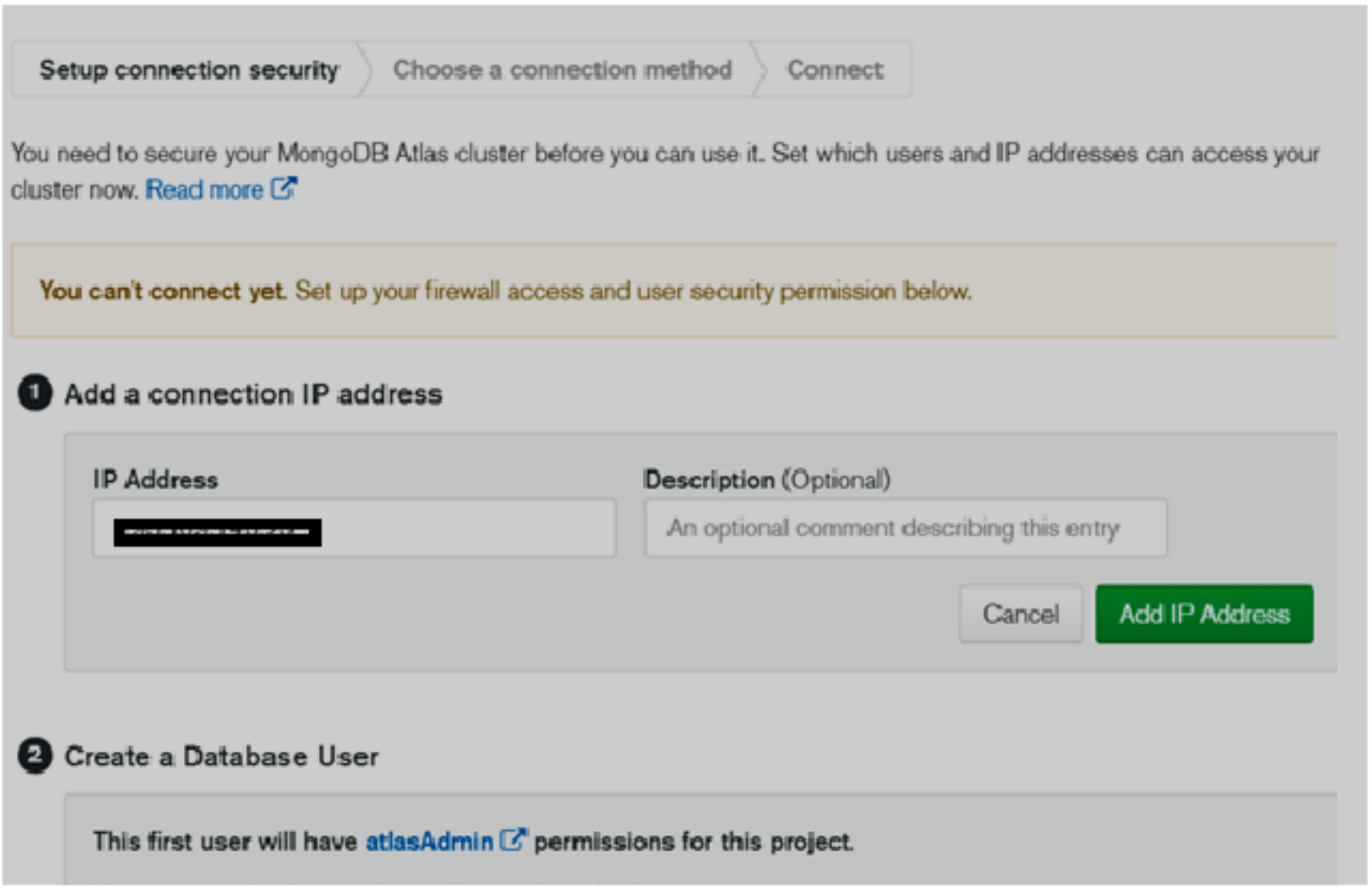


Рис. 1.28. Додаємо наш IP

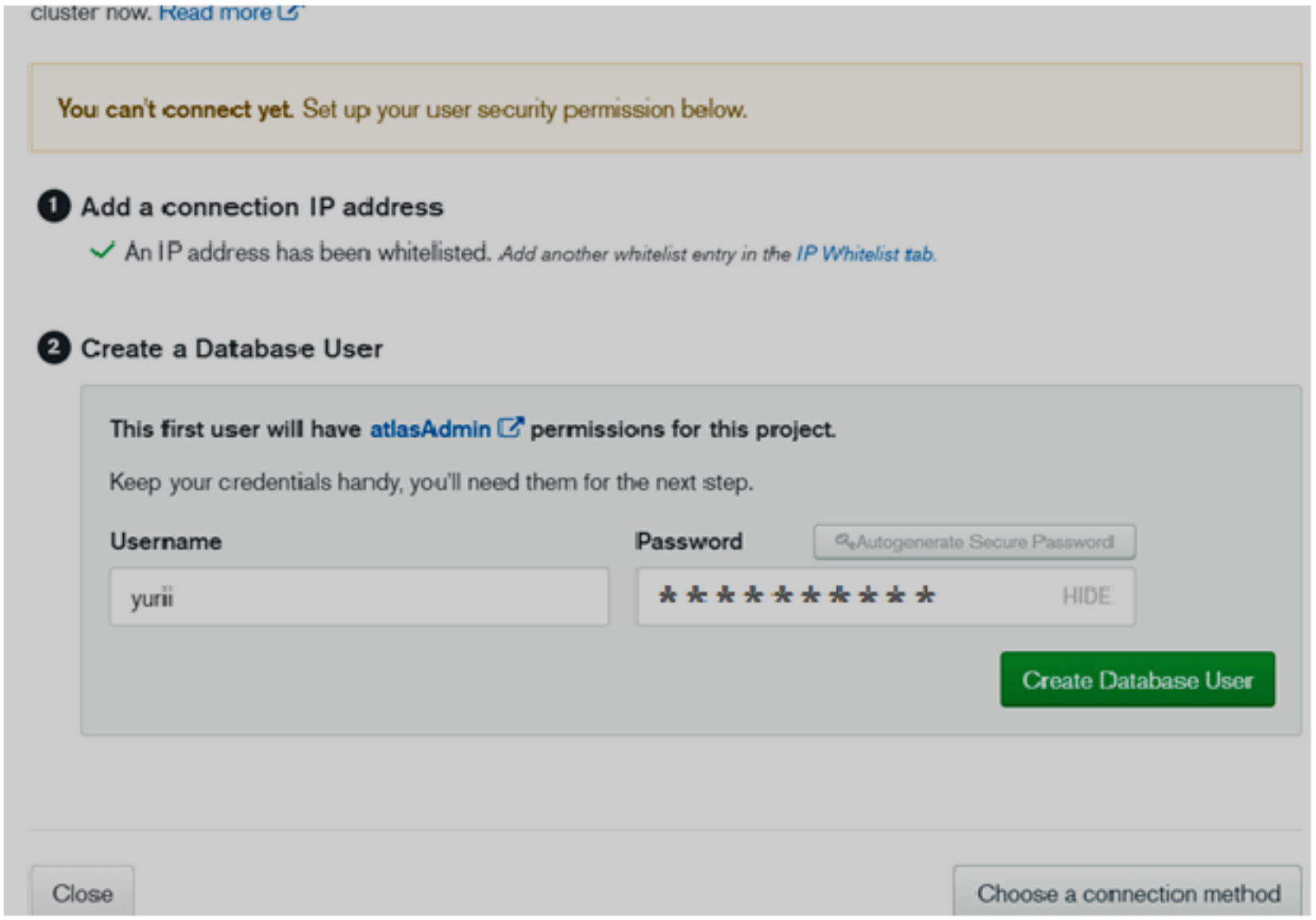


Рис. 1.29. Задаємо користувача та пароль

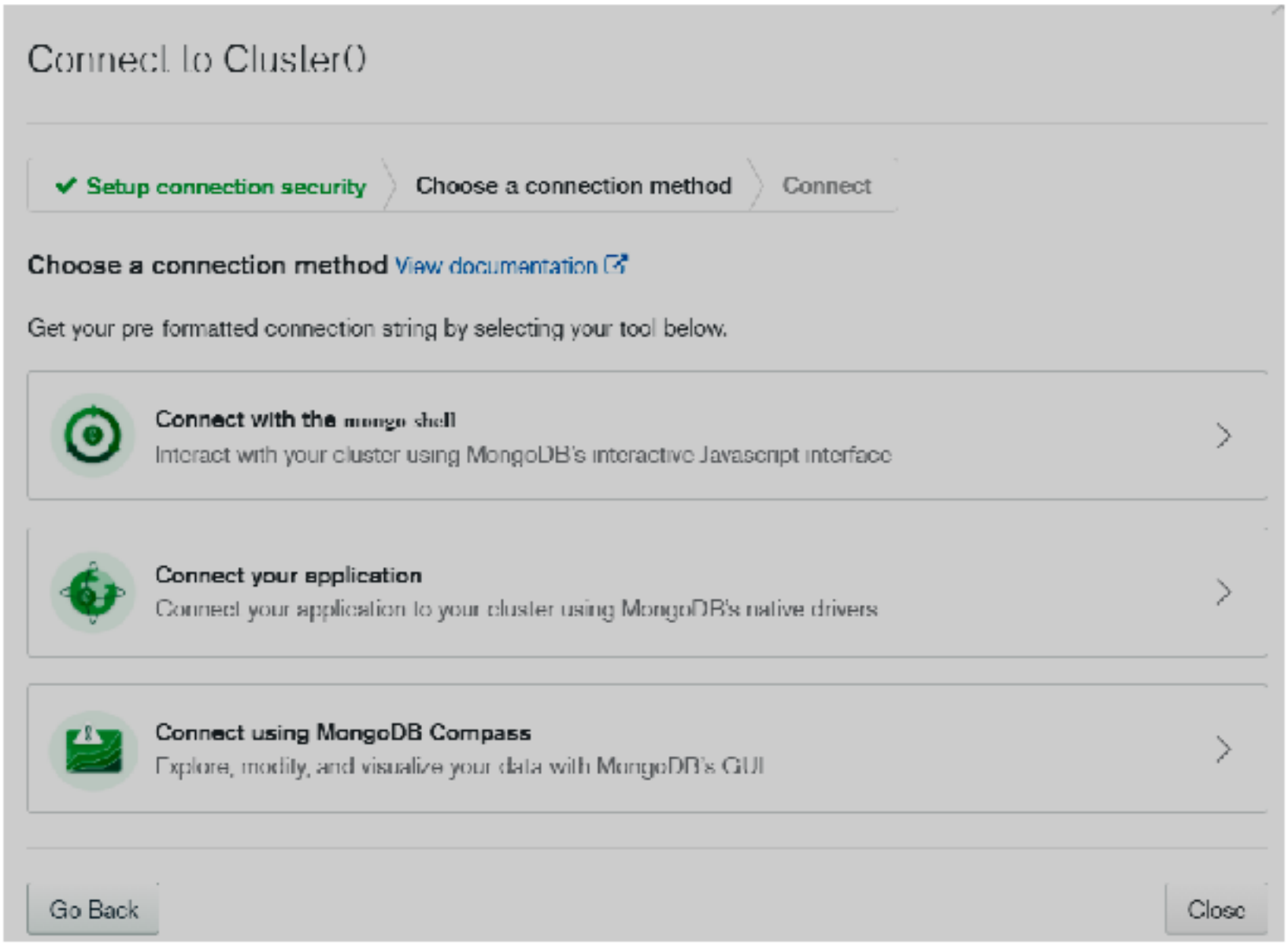


Рис. 1.30. Обираємо метод Connect your application

Рис. 1.31. Копіюємо код для зв’язку з базою даних тура

- замінюємо написи password та dbname на пародь та назву бази даних.

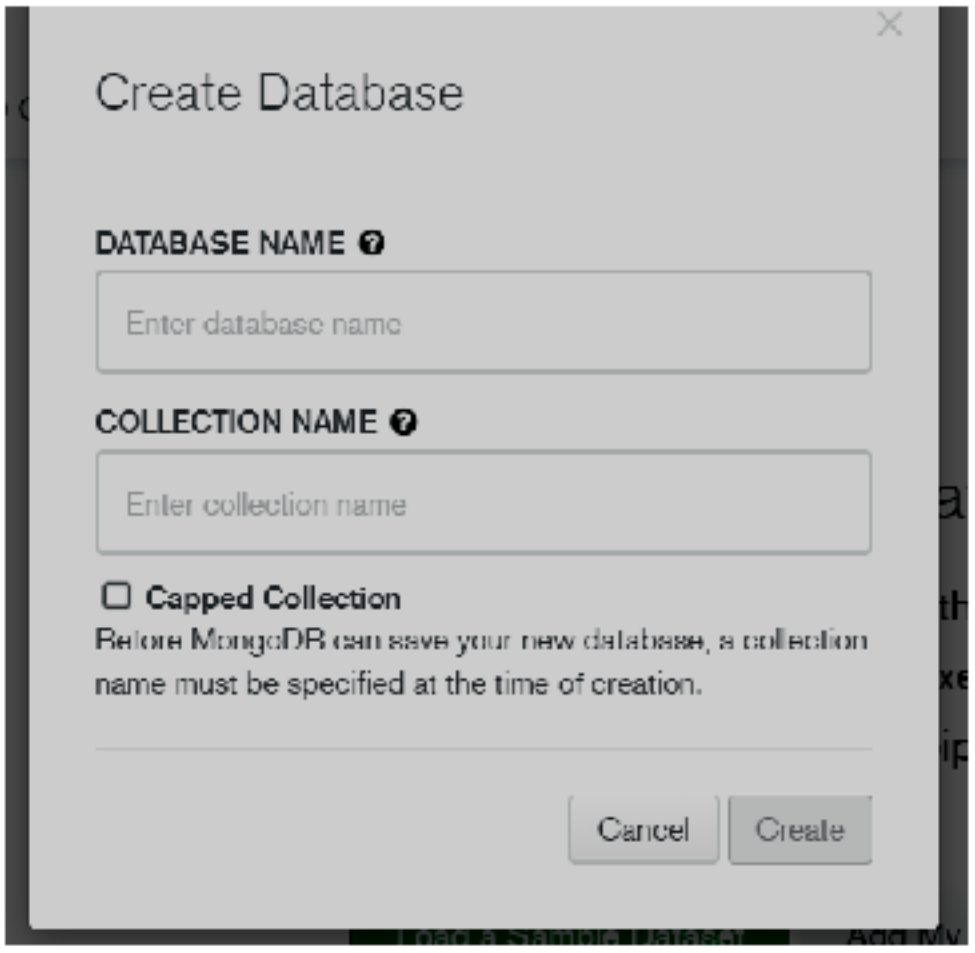


Рис. 1.32. Створюємо назви DB та Collection

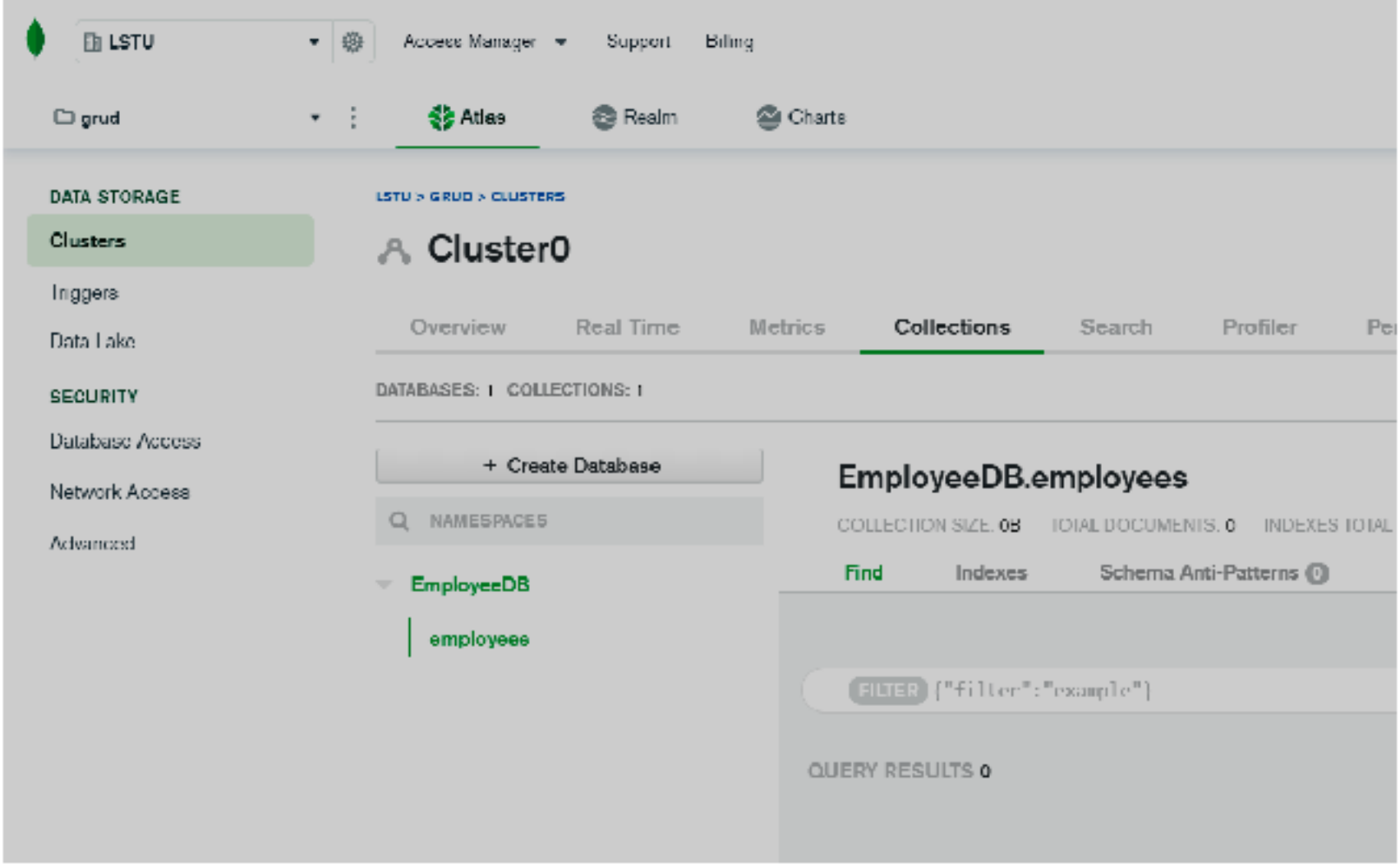


Рис. 1.33. Отримуємо готові базу даних та колекцію

Дія 1.4.4. Підключення до MongoDB у файлі app.js

- для підключення у файлі app.js:

```
const express = require('express')
const config = require('config')
const mongoose = require('mongoose')
const app = express()
// сервер повертає promis
const PORT = config.get('port') || 5000
// створюємо асинхронну функцію
async function start() {
  try {
    await mongoose.connect(config.get('mongoUri'), {
      useNewUrlParser: 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()
```

Дія 1.4.5. У config/default.json створюємо константи

```
{
   "port": 5000,
   "mongoUri": "mongodb+srv://yurii: …  "
}
```

Дія 1.4.6. Виконуємо тестовий запуск проекту (сервера)

- у консолі вводимо:

```
npm start
```

У консолі VS Code отримуємо відповідь:

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

Дія 1.4.7. Тестуємо, у браузері

- вводимо:

```
localhost:5000
```

- зупиняємо проект:

Ctrl + C.

1.4. ТЕСТОВІ ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Яким тегом оголошується заголовок web-сторінки?
   a) <head> </ head>
   b) <html> </ html>
   c) <title> </ title>
   d) <body> </ body>
2. HTML - це:
   a) мова редагування
   b) мова структурної розмітки
   c) мова програмування
   d) мова гіпертекстової розмітки
3. Для вставки зображення в документ HTML використовується команда:
   a) <img src = "ris.jpg”>
   b) <body background = "ris.jpg”>
   c) <a href=”ris.jpg”>
   d) <input = “ris.jpg”>
4. Гіперпосилання задається тегом:
   a) <font color = “file.htm”
   b) <img src = "http://www.chat.ua”>
   c) текст
   d) <embed = “http://www.da.ua”>
5. Які теги задають розмір заголовка?
   a) <h1> </ h1>
   b) <p> </ p>
   c) <img src = "name">
   d) <body> </ body>
6. Які теги вказують браузеру, що це HTML документ?
   a) <html> </ html>
   b) <body> </ body>
   c) <title> </ title>
   d) <p> </ p>
7. Які теги створюють абзац у документі?
   a) <p> </ p>
   b) <body> </ body>
   c) <img src = "name">
   d) <html> </ html>
8. За допомогою якого елемента можна створювати прокручувати списки в формах?
   a) TEXTAREA
   b) TR
   c) SELECT
   d) INPUT

9. Селектор jQuery викликом з кодом $("#header") отримує
   a) елемент з id
   b) елемент з класом
   c) елемент тега
   d) елемент JavaScript
10. Селектор jQuery викликом з кодом $(“h3”) отримує
   a) елемент з id
   b) елемент з класом
   c) елемент тега
   d) елемент JavaScript
11. На мові програмування JavaScript == це ?
   a) порівняння з приведенням типів (результат - дорівнює)
   b) порівняння з без приведення типів (результат - дорівнює)
   c) порівняння з приведенням типів (результат – більше дорівнює)
   d) порівняння з приведенням типів (результат – не дорівнює)
12. На мові програмування JavaScript !== це
   a) порівняння з приведенням типів (результат - дорівнює)
   b) порівняння з без приведення типів (результат - дорівнює)
   c) порівняння з приведенням типів (результат – менше дорівнює)
   d) порівняння з приведенням типів (результат – не дорівнює)
13. На мові програмування JavaScript === це
   a) порівняння з приведенням типів (результат - дорівнює)
   b) порівняння з без приведення типів (результат - дорівнює)
   c) порівняння з приведенням типів (результат – більше дорівнює)
   d) порівняння з приведенням типів (результат – не дорівнює)
14. Node.js має такі особливості
   a) компіляція сирцевого коду JavaScript безпосередньо у власний машинний код, минаючи стадію проміжного байт-коду
   b) регулює використання пам’яті та точно визначає, де містяться в пам'яті об'єкти й вказівники/посилання, що дозволяє запобігати витоку інформації
   c) відкритий код для виконання високопродуктивних мережевих додатків, написаних мовою JavaScript
   d) окрім роботи із серверними скриптами для web-запитів, також використовується для створення клієнтських та серверних програм
15. Рушій V8 має такі особливості
   a) компіляція сирцевого коду JavaScript безпосередньо у власний машинний код, минаючи стадію проміжного байт-коду
   b) регулює використання пам’яті та точно визначає, де містяться в пам'яті об'єкти й вказівники/посилання, що дозволяє запобігати витоку інформації
   c) відкритий код для виконання високопродуктивних мережевих додатків, написаних мовою JavaScript
   d) окрім роботи із серверними скриптами для web-запитів, також використовується для створення клієнтських та серверних програм

16. Напишіть команду консольного режиму, що здійснює виклик менеджера пакетів Node …

17. Node.js. Якщо робота програми полягає в тому, що початок і кінець однієї операції відбуваються в різний час в різних частинах коду в неблокуючому режимі, то – це

a) Асинхронність
b) Конкурентність
c) Паралелізм
d) Багатопотоковість

18. Напишіть назву та розширення конфігураційного файлу проекту Node (проекту Express) …

19. Node.js. Якщо робота програми полягає в тому, що безліч завдань вирішуються в один час, коли можна казати, що в програмі є кілька логічних потоків - по одному на кожну задачу, то – це

a) Асинхронність
b) Конкурентність
c) Паралелізм
d) Багатопотоковість

20. Напишіть конфігураційний файл проекту Express має розділ, що вміщуватиме опис підключених модулів Node та їх версії … .

21. Node.js. Якщо робота програми полягає в тому, що задачу можна розділити на більше ніж дві частини і роздати їх різним потокам, щоб прискорити виконання, то – це

a) Асинхронність
b) Конкурентність
c) Паралелізм
d) Багатопотоковість

22. Node.js. Якщо робота програми полягає в тому, коли частини програми є абстракціями, під якими може ховатися і окреме ядро процесора, і тред ОС, то – це

a) Асинхронність
b) Конкурентність
c) Паралелізм
d) Багатопотоковість

23. Чи передбачена автоматична установка фреймворка Express JS?

a) Ні, тільки ручна установка з архіву
b) Так, можна скачати інсталятор з офіційного сайту
c) Так, через NPM
d) Так, але тільки для ОС Windows

24. У проектах Node JS для роботи з документо-орієнтованою базою даних MongoDB використовується

a) server
b) express
c) nodemon
d) mongoose

25. Данні у документо-орієнтованій базі даних MongoDB зберігаються в

a) таблицях
b) коллекціях
c) списках
d) масивах

26. Данні у документо-орієнтованій базі даних MongoDB зберігаються в форматі

a) JSON
b) SQL
c) DATA
d) KEY

27. У документо-орієнтованій базі даних MongoDB одиницею зберігання, доступу та обробки даних є

a) JSON
b) SQL
c) DATA
d) VALUE

28. Документ у MongoDB має набір властивостей, що звуться

a) атрибути
b) типи
c) коллекції
d) імена

29. У документо-орієнтованій базі даних MongoDB доступ до даних здійснюється через

a) ключ
b) ім’я
c) тип
d) розмір

30. Напишіть назву інтерфейсу між MongoDB та Node для зв’язку з даними …

1.5. ЗАВДАННЯ ДО САМОСТІЙНОЇ РОБОТИ

1. Ознайомитись з прийомами роботи з використанням HTML.
2. Ознайомитись з прийомами роботи з використанням CSS.
3. Ознайомитись з прийомами роботи з використанням JavaScript.
4. Ознайомитись з прийомами створення односторінкової реалізації web-сторінки з вимогою до адаптивності.
5. За допомогою Internet виконати пошук шаблонів web-сторінок. Програмувати у скачаному шаблоні каскадне меню web-сторінки.
6. Виконати запуск Visual Studio Code. У Visual Studio Code створити папку проекту Node.js. За допомогою NPM створити файл конфігурації проекту Node.js package.json.
7. Розгорнути серверну частину (фреймворк Express.js) для створення Web-додатку.
8. Виконати запуск серверної частини web-додатку.
9. Ознайомитись з прийомами роботи у momgodb.com.
10. Виконати створення бази даних у momgodb.com.

ВИКОРИСТАНІ ДЖЕРЕЛА

1. ДСТУ 2481-94. Інтелектуальні інформаційні технології. Терміни.
2. Марченко А.В. Проектування інформаційних систем : навчальний посібник. URL: http://kist.ntu.edu.ua/textPhD/PIS_Marchenko.pdf (дата звернення: 17.03.2023).
3. ДСТУ 3278-95. СИСТЕМА РОЗРОБЛЕННЯ ТА ПОСТАВЛЕННЯ ПРОДУКЦІЇ НА ВИРОБНИЦТВО. Основні терміни та визначення.
4. Patterns of Enterprise Application Architecture : Martin, Fowler: URL: http://ce.sharif.edu/courses/97-98/2/ce418-1/resources/root/Books/Patterns%20of%20Enterprise%20Application%20Architecture%20-%20Martin%20Fowler.pdf (дата звернення: 17.03.2023).
5. http://proggerdotnet.blogspot.com/p/blog-page_26.html (дата звернення: 17.03.2023).
6. Блог .NET C# - программиста. Что такое бизнес-логика в программировании? URL: http://proggerdotnet.blogspot.com/2014/04/blog-post.html (дата звернення: 17.03.2023).
7. Building a BEM project. URL: https://en.bem.info/methodology/build (дата звернення: 17.03.2023).
8. Quick start / Methodology / BEM. URL: https://en.bem.info/methodology/quick-start (дата звернення: 17.03.2023).
9. JavaScript Tutorial. URL: https://learntutorials.net/de/javascript (дата звернення: 17.03.2023).
10. Лаврищева Е.М. Концепція індустрії наукового софтвера і підхід до обчислення наукових задач // Проблеми програмування. – 2011. – № 1. – С. 3–16.
11. Лавріщева К.М. Інструментально-технологічний комплекс для розроблення й навчання прийомам виробництва програмних систем / К. М. Лавріщева // Вісн. НАН України. - 2012. - № 3. - С. 67-79.
12. URL: https://www.codeproject.com/Articles/1052703/MEAN-Stack-Beginner-Tutorial (дата звернення: 17.03.2023).
13. Работа с JSON URL: https://developer.mozilla.org/ru/docs/Learn/JavaScript/Objects/JSON (дата звернення: 17.03.2023).
14. JavaScript. Function. MDN WEB DOCS. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function (дата звернення: 17.03.2023).
15. What exactly is Node.js? URL: https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5 (дата звернення: 17.03.2023).

16. NODE JS HTTP CREATESERVER EXAMPLE NIANGALA URL: https://thenaturalfuneral.org/niangala/node-js-http-createserver-example.php (дата звернення: 17.03.2023).
17. Top 25+ Express.Js Interview Questions & Answers URL: https://learnfrenzy.com/blog/top-25-expressjs-interview-questions--answers (дата звернення: 17.03.2023).
18. Visual Studio Code - Code Editing. Redefined URL: https://code.visualstudio.com (дата звернення: 17.03.2023).
19. MongoDB: The Application Data Platform URL: https://www.mongodb.com (дата звернення: 17.03.2023).
20. URL: https://www.npmjs.com (дата звернення: 17.03.2023).
21. Lars Bak. Google Chrome's Need for Speed. September 2, 2008. URL: https://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html (дата звернення: 17.03.2023).
22. Введення в асинхронний JavaScript. URL: http://webdiz.com.ua/vvedenie-v-asinkhronnyy-javascript-kolbyeki-fun (дата звернення: 17.03.2023).
23. Java EE threads v/s Node.js. Jun 9, 2013. URL: https://bijoor.me/2013/06/09/java-ee-threads-vs-node-js-which-is-better-for-concurrent-data-processing-operations (дата звернення: 17.03.2023).
24. Bootstrap The most popular HTML, CSS, and JS library. URL: https://getbootstrap.com (дата звернення: 17.03.2023).
25. JAVASCRIPT.INFO. Class basic syntax. URL: https://javascript.info/class (дата звернення: 17.03.2023).
26. Everything you need to know about NoSQL databases. URL: https://dev.to/lmolivera/everything-you-need-to-know-about-nosql-databases-3o3h (дата звернення: 17.03.2023).
27. Simon Holmes. Getting MEAN with Mongo, Express, Angular, and Node. Manning Publications, 2016 - 416p.
28. METANIT.COM Сайт о программировании. Работа с базой данных MongoDB. Устройство базы данных. Документы. URL: https://metanit.com/nosql/mongodb/2.1.php (дата звернення: 17.03.2023).
29. MongoDB – Краткое руководство. URL: https://coderlessons.com/tutorials/bazy-dannykh/uchitsia-mongodb/mowwwwwwwwngodb-kratkoe-rukovodstvo
30. Envatotuts+. Введение в Mongoose для MongoDB и Node.js URL: https://code.tutsplus.com/ru/articles/an-introduction-to-mongoose-for-mongodb-and-nodejs--cms-29527 (дата звернення: 17.03.2023).
31. Configure your Node.js Applications. URL: https://www.npmjs.com/package/config (дата звернення: 17.03.2023).

Розділ 2. BACK-END ПРОЕКТУВАННЯ WEB-ДОДАТКУ

2.1. ПАТЕРНИ WEB-ІНТЕРФЕЙСІВ. ПАТЕРН MVC

2.1.1. Теоретичний модуль

Back-end розробник - це людина, яка займається створенням та підтримкою технологій, які є прихованими від очей користувача, тобто знаходяться поза його браузером і комп'ютером, а саме, на сервері.

Back-end-розробка - це набір апаратно-програмних засобів, за допомогою яких реалізована логіка роботи сайту, його взаємодія із сервером. Backend - це процес об'єднання сервера з користувачем.

До обов’язків Back-end розробника входить:

- розробка програмно-адміністративної частини web-додатків;
- робота з базами даних, продумування архітектури проєкту, його програмної логіки.

Діяльність back-end IT-фахівця, в цілому, має такі кроки:

- спершу він вивчає технічне завдання, яке формулює клієнт чи бізнес-аналітик компанії;
- моделює майбутній продукт та його функціонал;
- обирає певний стек технологій, який застосовуватиметься в проєкті. Тобто, робить вибір архітектурного підходу та оптимального стека (групи технологій): бази даних, пошукового рушія, сервісу опрацювання завдань, способу кешування тощо;
- пише код;
- займається виправленням багів після релізу.

Компоненти back-end-розробки

Back-end-розробник застосовує ті інструменти, що доступні на його сервері. Він має право вибрати будь-яку з універсальних мов програмування. Все залежить від конкретного проєкту і завдання замовника.

Для зберігання даних контенту back-end-розробники використовують такі системи управління базами даних:

- MySQL;
- PostgreSQL;
- SQLite;
- MongoDB.

Залежно від продукту обов'язки back-end-розробника сильно змінюються. Такий фахівець може створювати та інтегрувати бази даних, забезпечувати безпеку або налаштовувати технології резервного копіювання та відновлення.

Під час проектування web-додатку на рівні back-end розробнику потрібно мати такі знання та навички, наприклад:

- знати як працює Internet - вивчити алгоритми роботи HTTP, браузерів, DNS, розібратися, що таке домен і хостинг;
- опанувати одну з операційних систем, спробувати працювати в терміналі, почитати про керування пам'яттю, концепції побудови мережі;
- визначитися з мовою програмування та фреймворком (у цьому посібнику це JavaScript);
- опанувати системи контролю версій та їхні хостинги (GIT та GitHub);.
- вивчити реляційні бази даних;
- опанувати API REST, способи автентифікації;
- вивчити кешування;
- вивчити webсервери, наприклад Nginx;

також не слід забувати про безпеку даних, томузнання хеш-алгоритмів є обов'язковим;

- вміти завантажувати та оновлювати дані у базі даних на сервері;
- валідувати дані із форми завантаження даних;
- обробляти відповіді користувача тощо.

Код, який відповідає за різні завдання, відповідно до модульного підходу, потрібно, по можливості, розділити на окремі програмні модулі. З цією метою використовують архітектурні патерни, що допомагають розділити код на модулі за критеріями та принципами їх функціонального призначення.

Патерн MVC

При проектуванні web-інтерфейсів в якості архітектурного рішення часто застосовується шаблон Model-View-Controller (MVC). Він вперше був описаний в 1979 році Трюгве Реенскауг при розробці мови Smalltalk. MVC передбачає поділ всієї програми на три незалежні об'єкта:

• Модель (Model) - надає інформацію: дані та методи роботи з цими даними, реагує на запити, змінюючи свій стан.

• Представлення (View) - відповідає за відображення інформації (візуалізацію). Часто в якості представлення виступає форма з графічними елементами.

• Контролер (Controller) - забезпечує зв'язок між користувачем і системою: контролює введення даних користувачем і використовує модель і представлення для реалізації необхідної реакції.

Користувач подає команди web-додатку, як результат взаємодії з елементами керування. Контролер отримує ці команди та перетворює дані в моделі. Модель оновлюється і повідомляє про те, що потрібно перемалювати інтерфейс, щоб відобразити зміни в даних на представленні (рис.2.1).

Важливою особливістю даного підходу є неможливість прямого взаємодії моделі і представлення. Будь-яка дія користувача, обробляється контролером, який в свою чергу взаємодіє з моделлю.

Схема MVC розділяє об'єкти в призначених для користувача інтерфейсах, що підвищує гнучкість і повторне використання програми.

MVC відокремлює представлення від моделі, встановлюючи між ними протокол взаємодії. Представлення має гарантувати, що зовнішнє представлення відображає стан моделі. При зміні внутрішніх даних, модель оповіщає всіх залежних від неї види, в результаті чого, вид оновлює себе. Це дозволяє створювати кілька представлень для моделі, не змінюючи її.

Принцип дії

Типове рішення MVC має на увазі виділення трьох окремих ролей (рис.2.1).

Модель - це об'єкт, що надає деяку інформацію про домен. У моделі немає візуального інтерфейсу, вона містить в собі всі дані і поведінку, не пов'язані з призначеним для користувача інтерфейсом. В об'єктно-орієнтованому контексті найбільш "чистою" формою моделі є об'єкт моделі предметної області (Domain Model). У якості моделі можна розглядати і сценарій транзакції (Transaction Script), якщо він не містить в собі ніякої логіки, пов'язаної з призначеним для користувача інтерфейсом. Подібне визначення не дуже розширює поняття моделі, однак повністю відповідає розподілу ролей в розглянутому типовому рішенні.

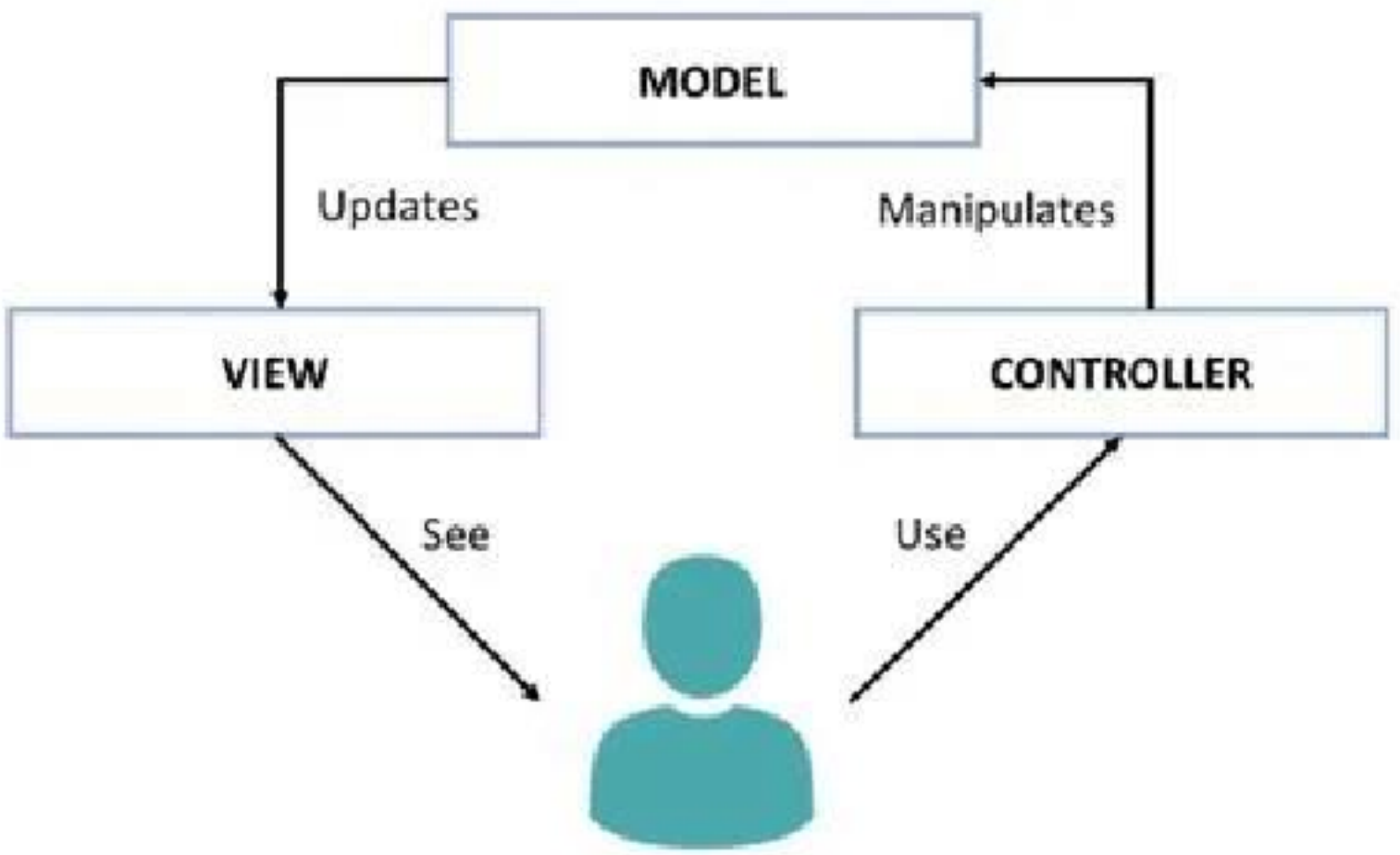


Рис. 2.1. Структура та принцип взаємодії патерна MVC [1]

Представлення відображає вміст моделі засобами графічного інтерфейсу.

Наприклад, якщо модель MVC - це подання даних для покупця Internet магазину, то відповідне представлення може бути фреймом з купою елементів управління або HTML-сторінкою, заповненої інформацією для покупця. Функції представлення полягають тільки в відображенні інформації на екрані. Всі зміни інформації обробляються третім елементом шаблону MVC - контролером.

Контролер отримує вхідні дані від користувача, виконує операції над моделлю і вказує поданням на необхідність відповідного оновлення. В цьому плані графічний інтерфейс можна розглядати як сукупність представлення і контролера.

Говорячи про типове рішення MVC потрібно підкреслити два принципових типи поділу: відокремлення представлення від моделі та відокремлення контролера від представлення (рис.2.1).

Відокремлення представлення від моделі - це один з фундаментальних принципів проектування програмного забезпечення. Наявність подібного поділу є дуже важливим з ряду причин. Представлення та модель відносяться до абсолютно різних сфер програмування. Розробляючи представлення, ви думаєте про механізми побудови інтерфейсу, що призначений для користувача, і про те, як зробити інтерфейс програми максимально зручним для нього. У свою чергу, при роботі з моделлю ваша увага зосереджена на бізнес-даних і на взаємодії з базою даних. Очевидно, при розробці моделі та представлення застосовуються різні бібліотеки-ресурси. Крім того, більшість розробників спеціалізуються тільки в одній з цих областей.

Досить часто користувачі хочуть, щоб, в залежності від ситуації, одна і та ж сама інформація могла бути відображена різними способами. Відділення представлення від моделі дозволяє розробити для одного і того ж коду моделі кілька представлень, а точніше, кілька абсолютно різних інтерфейсів.

Ключовим моментом у відділенні представлення від моделі є напрямок залежностей: представлення залежить від моделі, але модель не залежить від представлення.

Для Web-інтерфейсів відокремлення контролера від представлення є надзвичайно корисним. Типове рішення - контролер сторінок передбачає наявність окремого контролера для кожної логічної сторінки Web-сайту. В основі контролера лежить ідея створення компонентів, які будуть виконувати роль контролерів для кожної сторінки Web-сайту. Передавання бізнес-даних з моделі до представлення, у більшості випадків, реалізується компонентами контролера.

Схема MVC розділяє об'єкти в призначених для користувачів різних інтерфейсах, що підвищує гнучкість і повторне використання програми. MVC відокремлює представлення від моделі, встановлюючи між ними протокол взаємодії.

Потік даних у класичному MVC вибудовується таким чином:

- дані від користувача передаються на елементи представлення;
- від представлення до контролера;
- контролер оновлює данні через модель;
- модель повідомляє представлення про те, що змінилися данні.

Представлення відображає дані таким чином, щоб відповідати стану моделі. При зміні внутрішніх даних, модель оповіщає всі залежні від неї представлення, за результатом чого, кожне представлення оновлюється. Це дозволяє створювати кілька представленнь для однієї моделі, не змінюючи її.

У Express.js для взаємодії контролера-моделі-представлення використовуються стандартні функції JavaScript. Основними з яких в архітектурному шаблоні MVC є:

app.set/get() - для отримання змінних на запит, друкувати їх у представленні (view)

app.listen() - використовується для зв'язування та прослуховування з'єднань на вказаному хості та порту. Цей метод ідентичний методу http.Server.listen() у Node.JS.

Наприклад:

```
const title = app.get  (' My Site');
app.set  ('title', 'My Site');                                    //
app.set ('jabberwocky', 'correct battery horse staples');
app.set('views', path.join (__dirname, 'views'));
        //  папка перегляду  /Users/jilles/Project/myApp/views
```

Модуль Path є вбудованим у Node.JS та надає набір функцій для роботи з шляхами у файловій системі.

Підключення модуля: const path = require('path');

Найчастіше використовувані методи **Path**:

basename() - повертає кінцеву частину шляху, першим параметром приймає шлях, другим необов'язковим аргументом - розширення файлу, яке потрібно прибрати з результату, що повертається;

```
path.basename('/srv/app/app.js');                 // app.js
path.basename('/srv/app/app.js', '.js');          // app
```

dirname() - повертає директорію переданого шляху;

```
path.dirname('/srv/app/app.js');                  // \srv\app
```

extname() - повертає розширення файлу переданого шляху;

```
path.extname('/srv/app/app.js');                  //.js
```

isAbsolute() - булеве значення true, якщо переданий шлях є абсолютним;

```
path.isAbsolute('/srv/app/app.js');          //true
path.isAbsolute('srv/app/app.js');           //false
```

join() - приймає необмежену кількість складових частин шляху, включаючи повернення в батьківські директорії, та повертає отриманий в результаті шлях;

```
path.join('/srv/app', '../config/..', 'app/app.js');    // \srv\app\app.js
```

normalize() - призводить до коректного та оптимального виду переданий шлях;

```
path.normalize('/srv//app///app.js');          // \srv\app\app.js
```

parse() - розбирає переданий шлях на елементи та повертає об'єкт із такими властивостями:

root - корінь шляху;
dir - директорія;
base - кінцева частина шляху;
name - ім'я файлу (директорії) без розширення;

relative() - приймає два шляхи та повертає відносний шлях від першого до другого:

```
path.relative(
  '/srv/app/app.js',
  '/srv/config/default.conf'
);                                          // ..\..\config\default.conf
```

resolve() - приймає складові шляху і повертає абсолютний шлях отриманого в результаті обробки переданих сегментів шляху:

```
path.resolve('/srv/app', 'app.js');          // D:\srv\app\app.js
```

JS-фреймворк Express. Шаблонізатори Node js

У Node js для генерації і рендерингу HTML-сторінки застосовуються шаблонізатори.

Node js шаблонізатор є спеціальний модуль, який використовує більш зручний синтаксис для формування HTML на основі динамічних даних і дозволяє розділяти представлення від контролера.

Шаблонізатор для представлень Handlebars

Механізми застосування Handlebars.

1. Підтримка каталогу частин (шаблонів) представлення, наприклад:

```
{{> foo/bar}}
```

який існує у файловій системі за адресою views/partials/foo/bar.handlebars за замовчуванням.

2. Розумна файлова система вводу-виводу та кешування шаблонів. Під час розробки шаблони завжди завантажуються з диска.

3. Усі асинхронні та неблокувальні. Введення/виведення файлової системи працює повільно, і сервери не повинні блокувати обробку запитів під час читання. Чергу вводу/виводу використовується, щоб уникнути зайвої роботи.

4. Можливість легкої попередньої компіляції шаблонів та частин для використання на клієнті, що дозволяє спільне використання шаблонів та повторне використання.

Налаштування Node js шаблонізатору здійснюється встановленням двох параметрів:

views - шлях до директорії, у якій знаходяться шаблони;

view engine - вказівка самого шаблонізатора.

Для встановлення цих параметрів використовується метод Express set ().

```
app.set ('views', './views');
app.set ('view engine', 'handlebars');
```

Представлення лежать за замовчуванням в папці views, лайаути в layouts , а в шаблони можна передавати дані другим аргументом методу render.

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. 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)

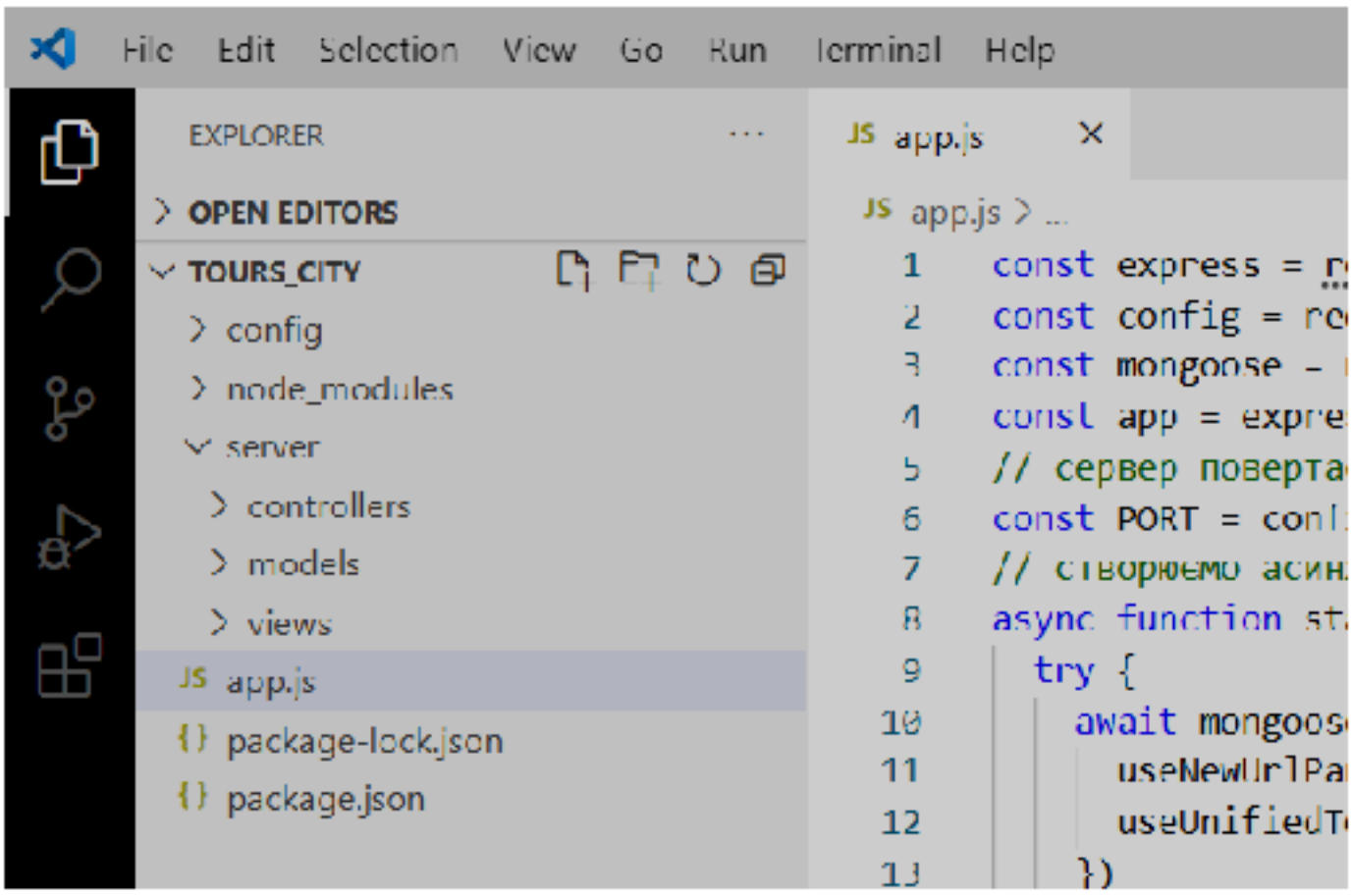


Рис. 2.2. Перегляд архітектури MVC у Visual Studio Code

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).

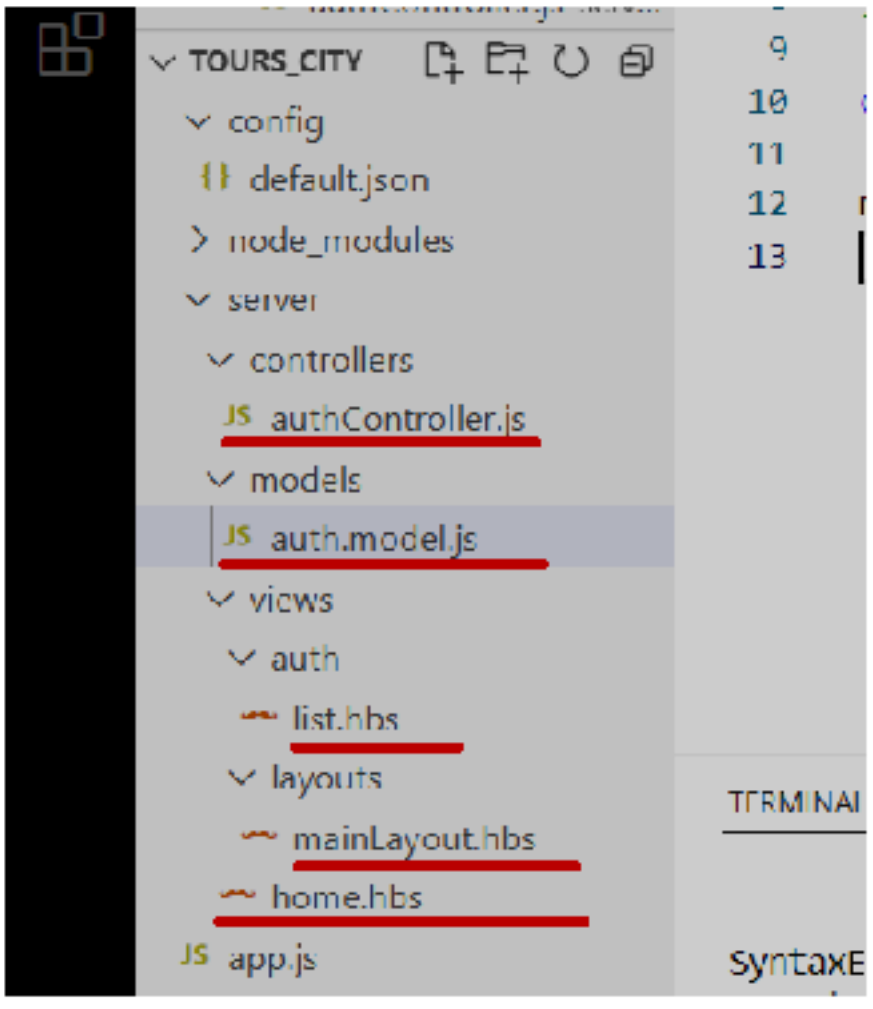


Рис. 2.3. Файли архітектури MVC

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 команнду New File (рис. 2.4 зона 2)
- вводимо назву файлу authController.js (рис. 2.4 зона 3)
- натискаємо Enter

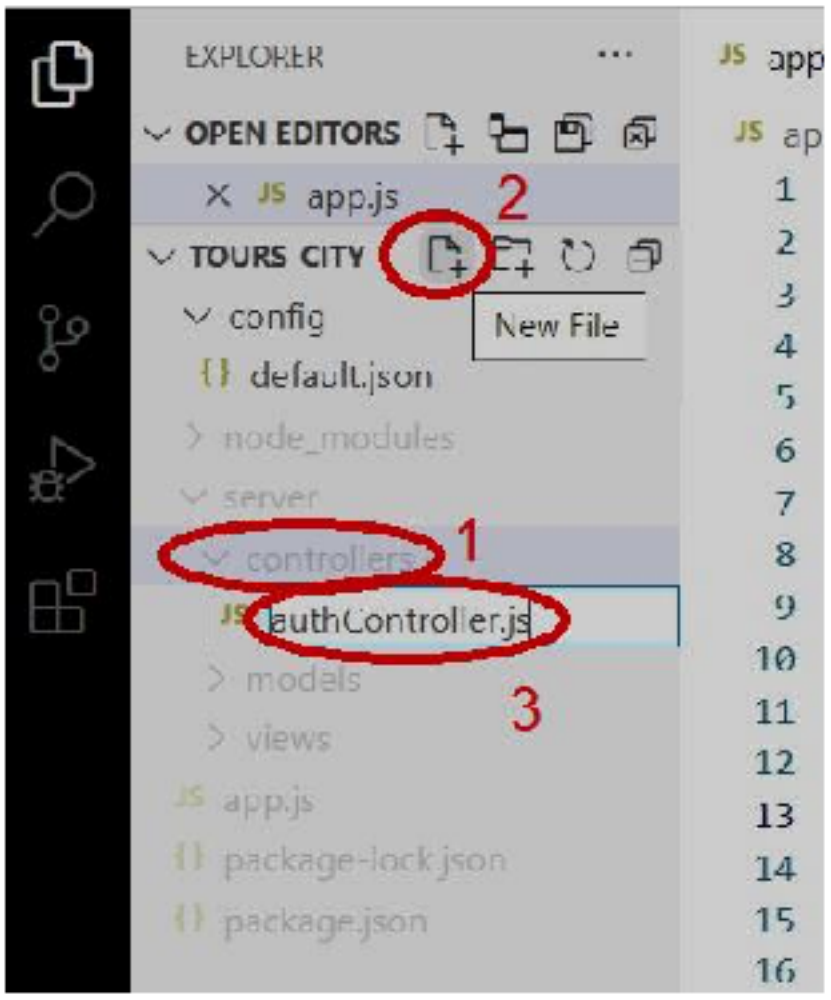


Рис. 2.4. Створення нового файлу в Visual Studio Code

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 представлення. 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 представленнями 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 представлення 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 представленнями
    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'), {
          useNewUrlParser: 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. У **package.json** замінюємо скрипти на

"start": "node app.js",

"server": "nodemon app.js" // запускає бекент

Дія 2.1.12. Виконуємо тестовий запуск сервера проекту

npm run server

У консолі VS Code отримуємо відповідь:

…

Server App has been started on port 5000...

- у браузері вводимо:

localhost:5000

- відкривається стартова сторінка- представлення beck-end

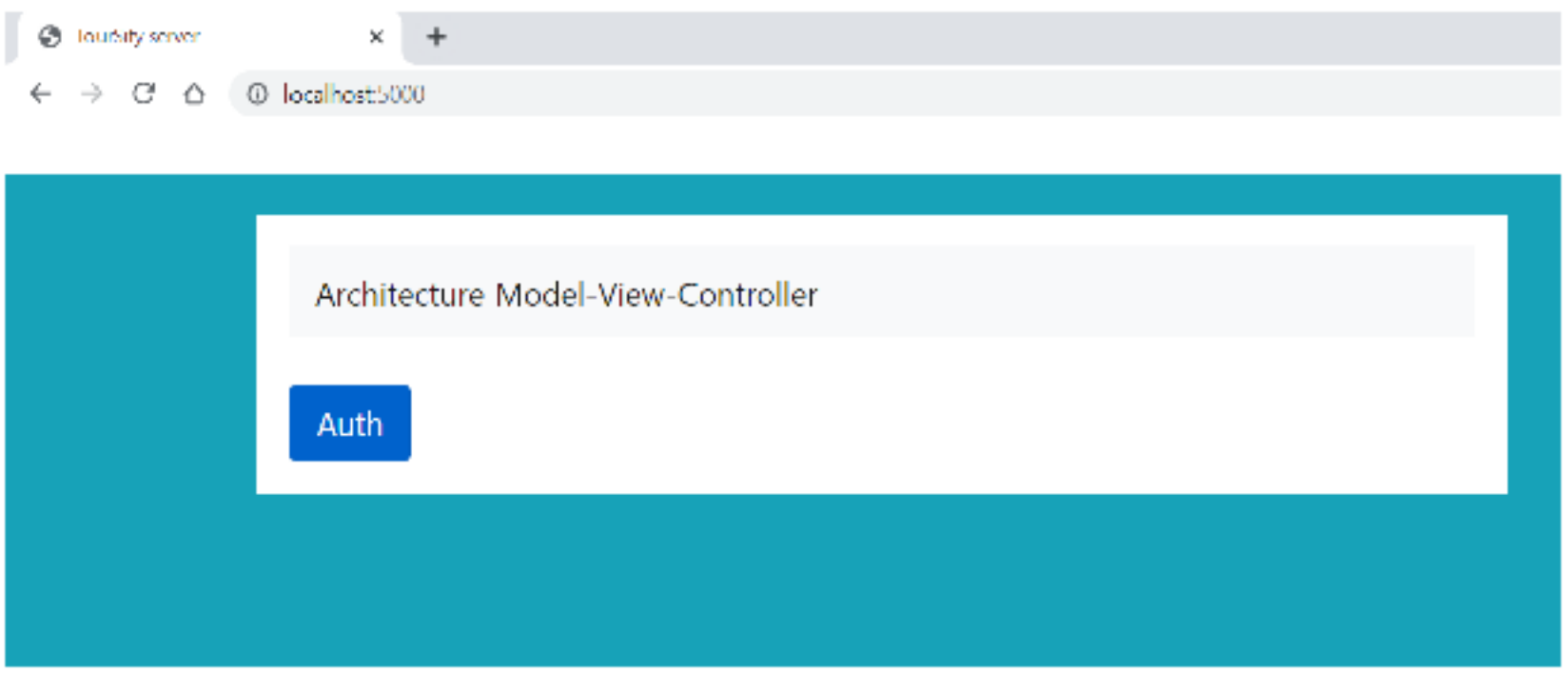


Рис. 2.5. Вигляд стартової сторінки-представлення beck-end

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

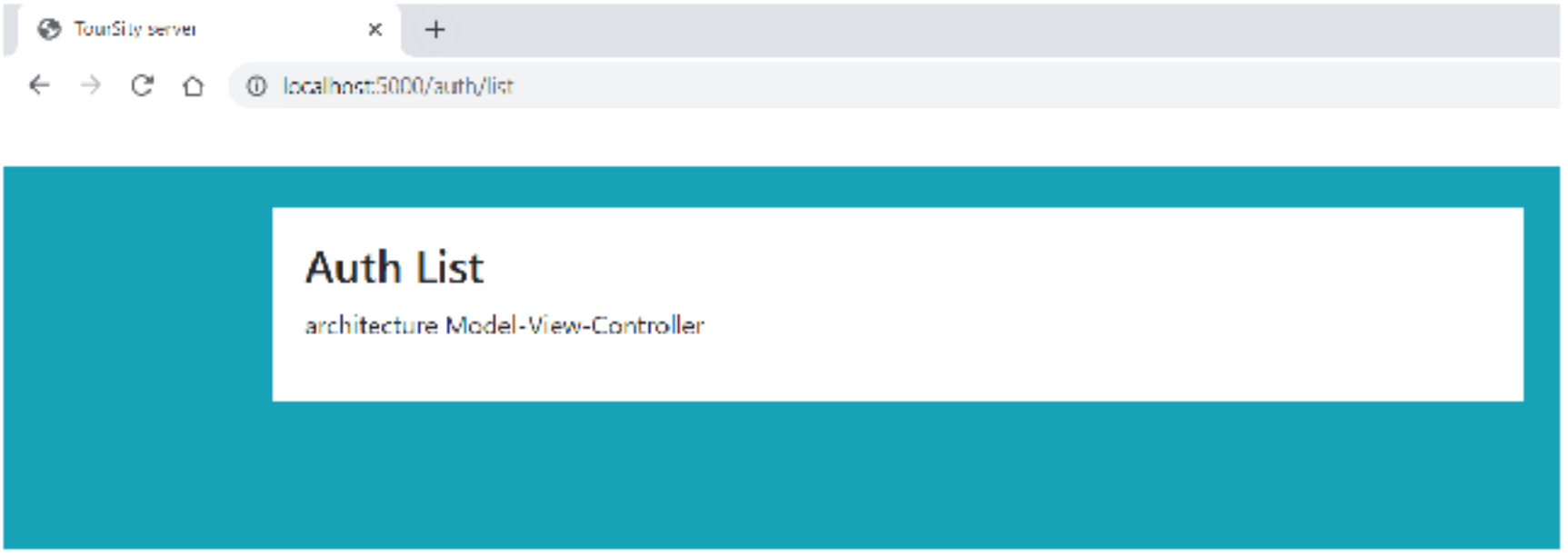


Рис. 2.6. Вигляд сторінки-представлення list.hbs

- зупиняємо проект:

Ctrl + C.

2.2. АРХІТЕКТУРА КОДУ MVC. ПОБУДОВА CRUD

2.2.1. Теоретичний модуль

Правильна архітектура коду web-додатку значно спрощує довгостроковий його супровід та розширення. В основі вдалої архітектури коду лежить основне правило системного підходу до проектування web-додатків - потрібно проектувати таким чином, щоб максимізувати кількість об'єктів-модулів та мінімізувати кількість зв'язків між ними. Тобто, будь-який програмний додаток (складна система) дробиться на модулі (підсистеми) в'язок з якими забезпечується за допомогою інтерфейсів.

Архітектура коду безпосередньо залежить від функціональних вимог web-додатку. Для з'єднання модулів програмного коду, наприклад частин, що забезпечують зв'язок між базою даних та рівнем back-end web-додатку, потрібно створити програмний інтерфейс API (англ. Application Programming Interface).

API забезпечує взаємодію модулів між собою та дозволяє без особливих зусиль здійснювати обмін контентом в середині web-додатку. Основним завданням API є створення зв'язку між двома підпрограмами. API дозволяє надсилати запити на передачу або отримання інформації. Взаємодія здійснюється через JavaScript Object Notation (JSON), а дані отримуємо у модулях програми за допомогою API-запитів. API-запит включає 4 компоненти: endpoint (точка прийому запиту), header (заголовок), method (метод) і data (дані). Після виклику всіх компонентів, ми можемо побудувати API-запит.

API може бути реалізоване з використанням різної архітектури коду, але важливо розуміти, що будь-яке API має під собою твердий фундамент, званий бібліотекою. Наприклад, у Express.js API суворо прив'язане до рівня back-end web-додатку, і є поєднанням модулів Node.js Path [2], mongoose [3] та bodyParser [4].

Під час розробки web-додатків використовується архітектурний стиль взаємодії компонентів REST (англ. Representational State Transfer) – передавання репрезентативного стану, який є надмножиною CRUD, що використовується для ресурсів HTTP. Функції в стилі API REST реалізуються через HTTP-методи PUT, POST, GET, PATCH, DELETE [5].

Наприклад, потрібно відправити на сервер GET-запит за певною адресою та отримати список завдань. Далі додаємо нове завдання: відправляємо на ту ж адресу, але вже методом POST. Потім вирішуємо, що для редагування завдання потрібно відправити вже метод PUT, який спрямовуємо на ту ж адресу. Для видалення якогось завдання відправляємо метод DELETE.

Формування GET-запиту є дуже простим – достатньо набрати адресу у браузері. Для POST вже потрібна html-форма з method = post. Що стосується PUT та DELETE, то ні HTML, ні браузер про них нічого не знають. Тут ми підходимо до реалізації цієї концепції CRUD у back-end [7].

Робота з супроводом інформації в базі даних відбувається на серверному рівні - back-end, на якому в режимі адміністрування доступом, що надається адміністратором, менеджер контенту реалізує чотири базові функції CRUD: створення (англ. create), читання (read), модифікація (update), видалення (delete) контенту. Це не що інше, як основні операції для переважної більшості даних. CRUD потрібний для того, щоб розділити програмний код на окремі частини, кожна з яких відповідає за свої дії. Таким чином, хоч HTTP і підтримує довільні методи, але насправді працюють лише два: GET та POST.

Операції CRUD використовуються для управління контентом бази даних: створення, читання, редагування та видалення. CRUD пропонує безліч переваг, у тому числі:

- полегшення контролю безпеки;
- задовольняє різні вимоги доступу;
- спрощує розробку додатків, роблячи їх масштабованішими;
- має кращу продуктивність у порівнянні зі спеціальними операторами SQL.

CRUD - це не що інше, як основні операції для переважної більшості даних. Кожна функція пов’язана з своїм http-методом:

для POST буде виконано метод create();

для GET буде виконано метод read();

для PUT буде виконано метод update();

для DELETE буде виконано метод delete().

Якщо не використовувати REST, то швидше за все довелося б вказувати дію або за окремою url-адресою або в get-параметрах. Якось так:

сайт/task/create.

Модуль **Path** надає багато дуже корисних функцій для доступу та взаємодії з файловою системою [2]. Він використовується для обробки та перетворення шляхів до файлів. Цей модуль надає path.sep роздільник сегментів шляху (у Windows /Linux / macOS). Модуль можна імпортувати, використовуючи такий синтаксис:

```
const path = require('path');
```

Модуль Path має методи:

path.basename() - повернути останню частину шляху. Другий параметр може відфільтрувати розширення файлу:

```
require('path').basename('/test/something'); // something
require('path').basename('/test/something.txt'); // something.txt
```

```
require('path').basename('/test/something.txt', '.txt'); // something
```

path.dirname() - повертає частину шляху каталогу:

```
require('path').dirname('/test/something'); // /test
require('path').dirname('/test/something/file.txt'); // /test/something
```

path.extname() - повертає додаткову частину шляху:

```
require('path').extname('/test/something'); // ''
require('path').extname('/test/something/file.txt'); // '.txt'
```

path.parse() - розбирає шлях до об’єкта з сегментами, які його складають:

root: корінь
dir: шлях до папки, починаючи з кореня
base: назва файлу + розширення
name: ім'я файлу
ext: розширення файлу

приклад:

```
require('path').parse('/users/test.txt');
```

дає результат:

```
{
    root: '/',
    dir: '/users',
    base: 'test.txt',
    ext: '.txt',
    name: 'test'
}
```

path.format() - повертає рядок шляху з об’єкта. Це протилежно path.parse

path.format прийняттю об’єкта як аргументу з такими ключами:

root: корінь
dir: шлях до папки, починаючи з кореня
base: назва файлу + розширення
name: ім'я файлу
ext: розширення файлу
rootігнорується, якщо dir надається ,
exti nameігнорується, якщо base існує

```
// POSIX
require('path').format({ dir: '/Users/joe', base: 'test.txt' });
                    //  '/Users/joe/test.txt'
require('path').format({ root: '/Users/joe', name: 'test', ext: '.txt' });
                    //  '/Users/joe/test.txt'
// WINDOWS
require('path').format({ dir: 'C:\\Users\\joe', base: 'test.txt' });
                    //  'C:\\Users\\joe\\test.txt'
```

Об’єкт bodyParser [4] відкриває різні можливості для створення проміжного програмного забезпечення. Усі проміжні програми заповнять властивість req.body проаналізованим тілом, якщо Content-Type заголовок запиту збігається з type опцією, або порожнім об’єктом ({}), якщо немає тіла для аналізу, Content-Type не було зіставлено або сталася помилка.

Встановлюється:

const bodyParser = require('body-parser');

bodyParser.json([параметри])

Повертає проміжне програмне забезпечення, яке аналізує json та переглядає лише запити, Content-Type заголовок яких відповідає type параметру. Цей синтаксичний аналізатор приймає будь-яке кодування Юнікод тіла та підтримує автоматичне розширення gzip та deflate кодування. Новий body об’єкт, що містить аналізовані дані, заповнюється request об’єктом після проміжного програмного забезпечення (req.body).

Опції bodyParser.json

Функція json приймає необов’язковий options об’єкт, який може містити будь-який із таких ключів:

inflate - якщо встановлено значення true, то спущені (стиснуті) тіла будуть роздуті; при false, спущені тіла відкидаються. За замовчуванням true.

limit - контролює максимальний розмір тіла запиту. Якщо це число, то значення визначає кількість байтів; якщо це рядок, значення передається в бібліотеку байтів для аналізу. За замовчуванням '100kb'.

reviver - параметр reviver передається безпосередньо JSON.parse як другий аргумент.

strict - якщо встановлено значення true, прийматимуться лише масиви та об’єкти; коли false прийме будь-який JSON.parse. За замовчуванням true.

type - цей параметр використовується для визначення типу носія, який аналізуватиме проміжне програмне забезпечення. Цей параметр може бути рядком, масивом рядків або функцією. Якщо це не функція, type параметр передається безпосередньо до бібліотеки type-is, і це може бути ім’ям розширення (як json), тип mime (як application/json) або тип mime із символом підстановки (як */*або */json). Якщо функція, type параметр викликається як fn(req), то запит аналізується, якщо він повертає правдиве значення. За замовчуванням application/json.

verify - якщо його надано, подається як verify(req, res, buf, encoding), де buf – буфер тіла необробленого запиту, а encoding - кодування запиту. Запит переривається, отримавши помилку.

Приклад Express/Connect загальнього рівня. У цьому прикладі демонструється додавання загального аналізатора JSON і URL-адреси як

проміжного програмного забезпечення верхнього рівня, яке аналізуватиме тіла всіх вхідних запитів. Це найпростіше встановлення body-parser:

```
var express = require('express')
var bodyParser = require('body-parser')
var app = express()
// parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({extended: false }))
// parse application/json
app.use(bodyParser.json())
app.use(function (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.write('you posted:\n')
  res.end(JSON.stringify(req.body, null, 2))
})
```

Mongoose виступає як засіб взаємодії з моделями об'єктів MVC бази даних. Взаємодіє через спеціальні об’єкти-схеми. Схема в Mongoose [3] визначає метадані моделі - її властивості, типи даних та низку іншої інформації. Як тип даних можна вказувати одне з наступних значень:

String – рядок;
Number – число;
Date – дата;
Buffer – буфер;
Boolean – логічний;
Mixed – змішаний;
Objectid - ;
Array – масив;
Decimal128 – десятковий;
Map – карта.

Якщо не визначено властивість, то використовується значення за промовчанням.

Mongoose має низку вбудованих правил валідації, які ми можемо вказати у схемі:

required : вимагає обов'язкового значення для властивості;

min та **max** : задають мінімальне та максимальне значення для числових даних;

minlength та **maxlength** : задають мінімальну та максимальну довжину для рядків;

enum : рядок повинен представляти одне із значень у зазначеному масиві рядків;

match : рядок повинен відповідати регулярному виразу/

Приклад кодування схеми для колекції User у БД:

```
    const mongoose = require("mongoose");
    const Schema = mongoose.Schema;
// підключення
    mongoose.connect("mongodb://localhost:27017/usersdb",                    {
    useUnifiedTopology: true, useNewUrlParser: true });
// встановлення схеми
    const userScheme = new Schema({
        name: {
            type: String,
            required: true,
            minlength:3,
            maxlength:20
        },
        age: {
            type: Number,
            required: true,
            min: 1,
            max:100
        }
    });
    const User = mongoose.model("User", userScheme);
    const user = new User({name: "Li"});
    user.save(function(err){
        mongoose.disconnect();
        if(err) return console.log(err);
        console.log("Збережено об’єкт user", user);
    });
```

Для шифрування даних користувача, а особливо паролів авторизації, застосоується модуль Express.js Crypto, який підтримує криптографію [7]. Цей модуль надає криптографічні функції, що включають набір оболонок для відкритого хешу HMAC SSL, шифрування, дешифрування, підпису та перевірки функцій.

Хеш (від англ. Hash) - це рядок бітів фіксованої довжини, тобто процедурно та детерміновано згенерований з деякого довільного блоку вихідних даних.

HMAC означає код автентифікації повідомлення на основі хешу. Це процес застосування алгоритму хешування як до даних, так і до секретного ключа, що призводить до єдиного остаточного хешу.

Приклад шифрування з використанням хешу та HMAC:

```
const crypto = require('crypto');
const secret = 'abcdefg';
const hash = crypto.createHmac('sha256', secret)
                   .update('Welcome to JavaTpoint')
                   .digest('hex');
console.log(hash);
```

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-Z\-0-9]+\.)+[a-zA-Z]{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.

Лістинг 2.4 : Вміст файлу authController.js

```
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) => {
```

Продовження лістингу 2.4

```
      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. Представлення 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.

Лістинг 2.6 : Вміст файлу list.hbs

```
<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>
```

кінець лістингу 2.6

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>
```

Продовження лістингу 2.7

```
  <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-представленнях beck-end (рис. 2.7 – 2.10).

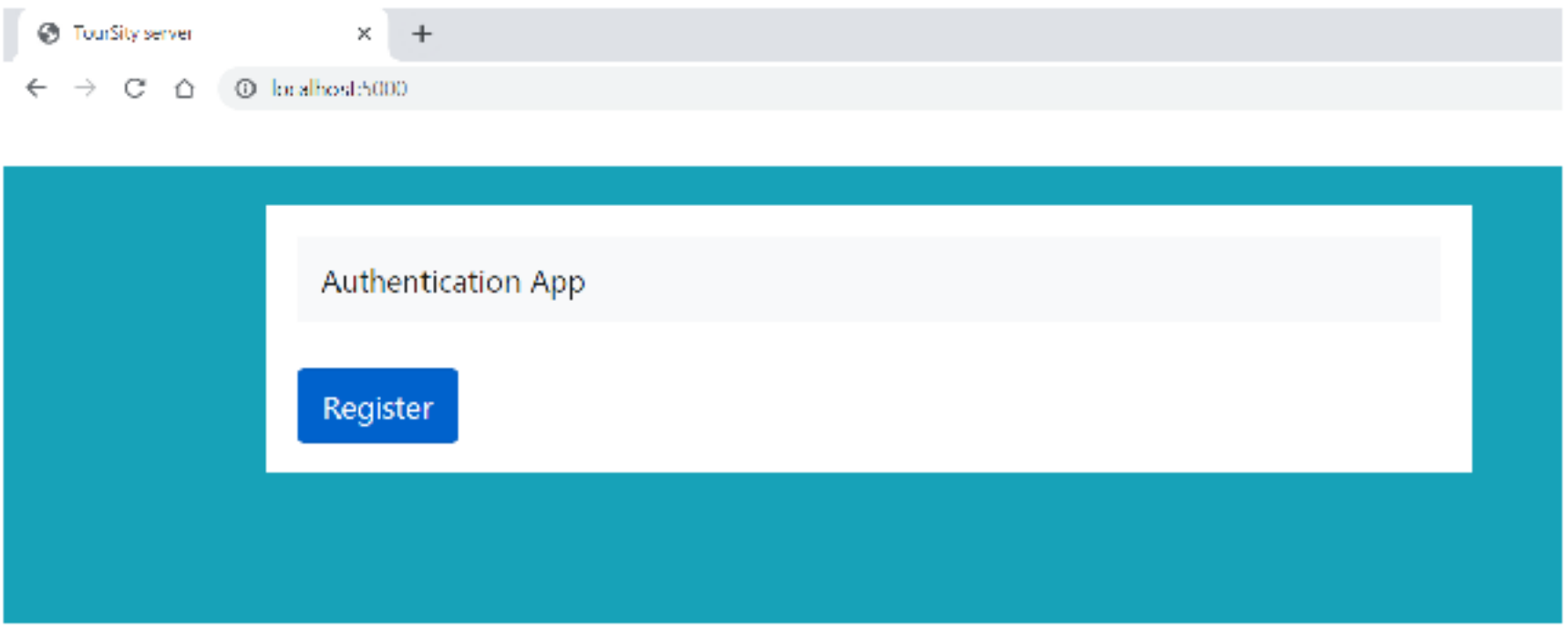


Рис. 2.7. Вигляд сторінки home.hbs

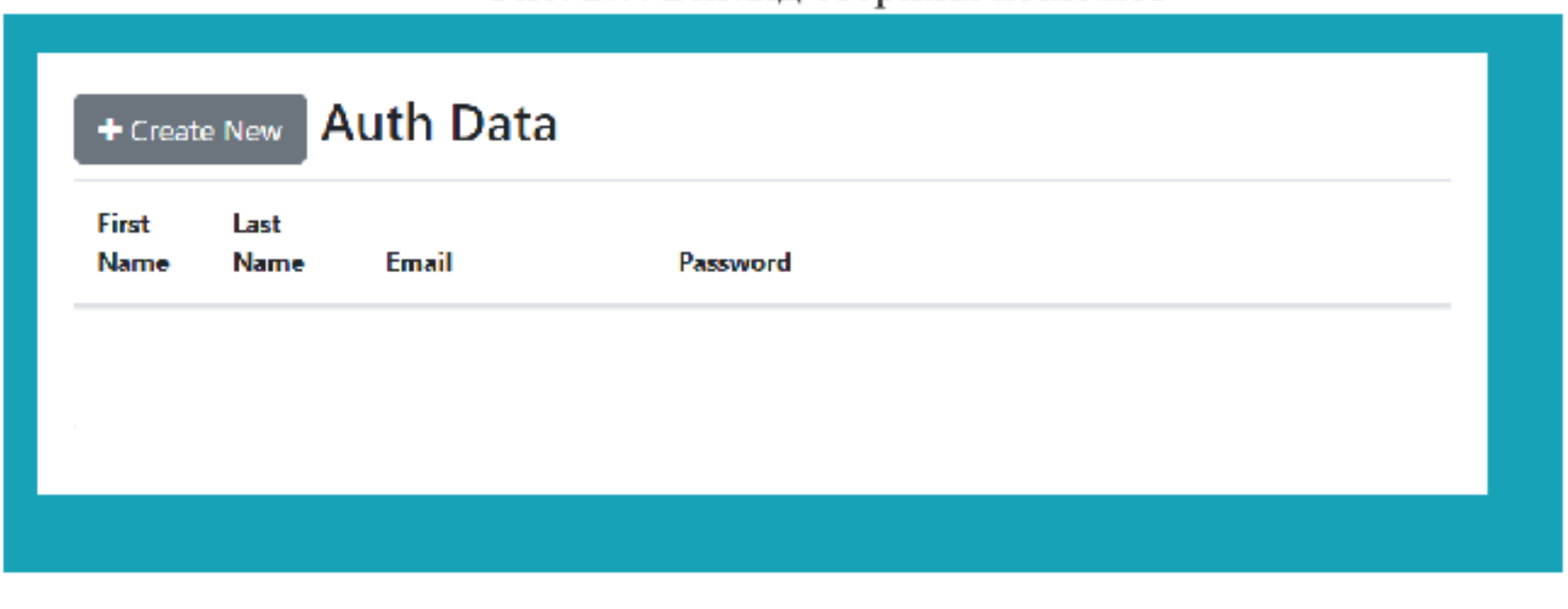


Рис. 2.8. Контент з представлення list.hbs до введення даних

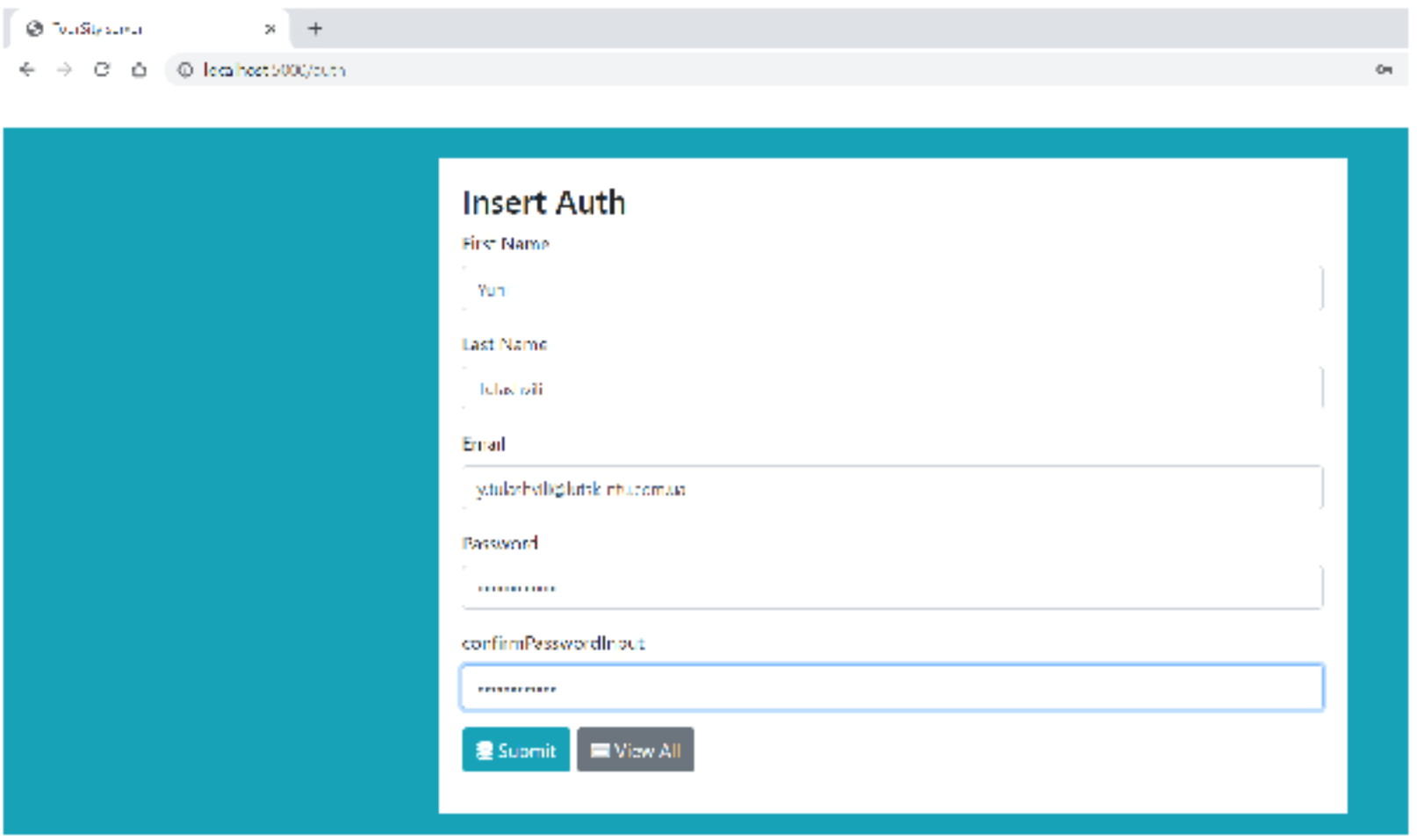


Рис. 2.9. Вигляд сторінки register.hbs для введення даних

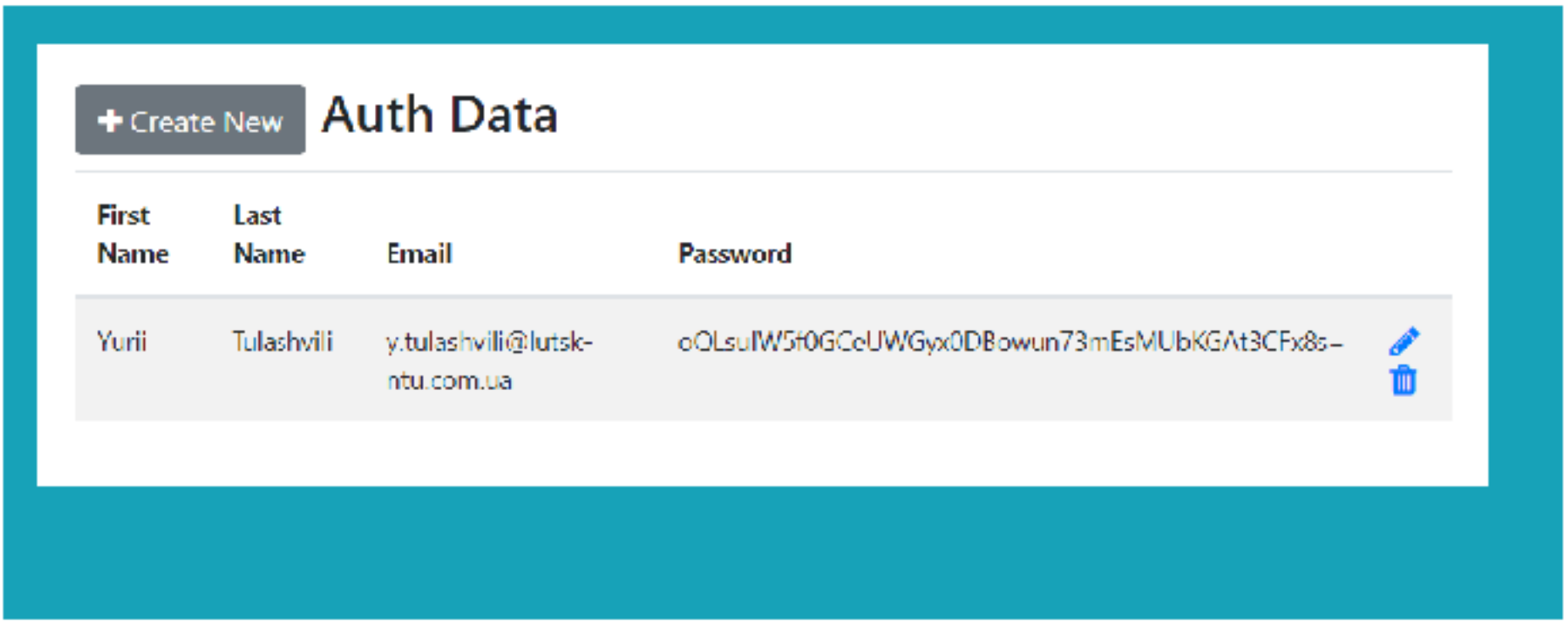


Рис. 2.10. Контент з представлення list.hbs після введення даних

- зупиняємо проект:

Ctrl + C.

2.3. АВТОРИЗАЦІЯ НА СЕРВЕРІ. ЗАХИСТ ІНФОРМАЦІЇ

2.3.1. Теоретичний модуль

Зайшовши на головну сторінку web-додатка на стороні back-end користувач повинен пройти певну перевірку. Система може перевіряти:

- чи існує користувач з таким ім'ям;
- чи збігається введений пароль з його обліковим записом;
- може запитувати ще наявність сертифіката, IP-адресу або додатковий код верифікації.

Автентифікація та авторизація – дві ключові функції сервісної інфраструктури для захисту конфіденційних даних та операцій від несанкціонованого доступу з боку зловмисників.

Автентифікація використовується для підтвердження особи зареєстрованого користувача. Це процес перевірки облікових даних: ідентифікатора користувача (імені, адреси електронної пошти, номера телефону) та пароля.

Авторизація відбувається після того, як особистість користувача успішно аутентифікується системою. Процес авторизації визначає, чи має людина, що пройшла перевірку, доступ до певних ресурсів: інформації, файлів, бази даних. Фактори автентифікації, необхідні для авторизації, можуть відрізнятися залежно від рівня безпеки.

Авторизація допомагає системі визначити, що дозволено тим чи іншим користувачам:

- право писати повідомлення;
- право додавати в повідомлення певні матеріали;
- право переглядати фотографії інших користувачів;
- право на редагування або копіювання документів.

Авторизація перевіряє наявність прав на конкретні дії. Це відбувається не тільки під час входу в систему, але і при будь-якій спробі вчинити будь-які маніпуляції з даними.

Два підходи, що найчастіше використовуються у web-додатках для реалізації механізму авторизації, засновані на файлах cookie і токенах [8].

Авторизація на основі файлів cookie [8-9]

Cookies – це невеликі рядки даних, які зберігаються безпосередньо у браузері. Вони є частиною протоколу HTTP, визначеного в специфікації RFC 6265. Куки зазвичай встановлюються web-сервером за допомогою заголовка Set-Cookie. Потім браузер автоматично додаватиме їх у кожен запит на той же домен за допомогою заголовка Cookie.

Один з найчастіших випадків використання cookie – це аутентифікація:

При вході на сайт сервер надсилає у відповідь HTTP-заголовок Set-Cookie для того, щоб встановити cookie із спеціальним унікальним ідентифікатором сесії («session identifier»).

Під час наступного запиту до цього домену браузер посилає на сервер HTTP-заголовок Cookie.

Таким чином сервер розуміє, хто зробив запит.

Ми також можемо отримати доступ до cookie безпосередньо з браузера, використовуючи властивість document.cookie.

Більшість браузерів автоматично приймають файли cookie, але, за бажанням, ви можете змінити налаштування таким чином, щоб не отримувати подібні файли. Після відмови від отримання файлів cookie, ви не зможете надалі користуватися всіма перевагами, які пропонує наш сайт. Cookie-файли є унікальними для сервера, який їх створив, інші сервери не можуть отримати до них доступ. Це означає, що файли cookie неможливо використовувати для відстеження ваших переміщень в Internet. Хоча ці файли й допомагають розпізнати комп'ютер окремого користувача, вони не ідентифікують його особу і не зберігають паролі.

Файли cookie діляться на основні (вони встановлюються безпосередньо Сайтом, що відвідується) і сторонні (встановлюються іншими сайтами).

Використання куки є одним із найпопулярніших способів передачі ідентифікатора (даний спосіб працює за умовчанням). Якщо змінну помістити один раз у куки, тоді її отримуватимуть усі скрипти.

Сервер надсилає файл cookie браузеру, включаючи його в заголовок Set-Cookie.

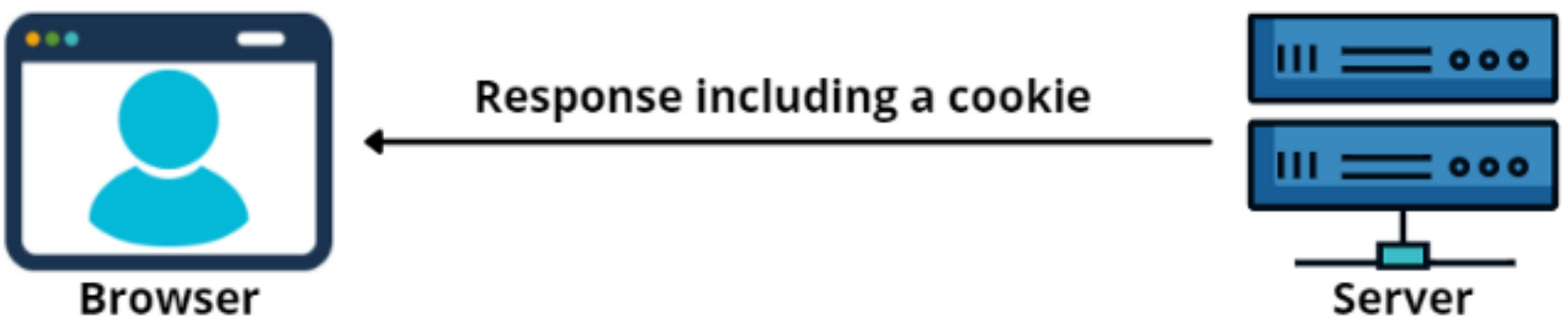


Рис. 2.11. Сервер надсилає файл cookie браузеру [8]

Значення document.cookie складається з пар ключ = значення, розділених ";". Кожна пара є окремим куки. Щоб знайти певне куки, достатньо розбити рядок з document.cookie на ";", а потім знайти потрібний ключ.

Авторизація на основі токенів [8]

Цей спосіб був введений для заміни способу аутентифікації на куках. Особливості підходу – потрібна ручна реалізація та токени, що зберігаються на стороні клієнта. **Токен** - це цифровий сертифікат, що гарантує певні права, зобов'язання тощо. Для авторизації генерується унікальний токен.

Коли авторизований користувач входить у web-додаток, сервер перевіряє облікові дані та відправляє зашифрований токен у браузер. Браузер зберігає токен і додає його до заголовка авторизації майбутніх запитів від цього користувача.

В основному різниця між автентифікацією на основі Cookie (шляхом зберігання sessionIds у файлах cookie на клієнті) та автентифікацією токена полягає в тому, що токен автентифікації відправляється в поле "автентифікація" http-заголовка. Це гнучкіше, оскільки є клієнти REST (клієнти на телефонах тощо), які взагалі не підтримують концепцію cookie.

Файли cookie зберігаються на стороні клієнта. HTTP не має стану. Щоразу, коли відправляється HTTP, буде прикріплено файл cookie під ім'ям домену, тому статус може бути прикріплений тільки до HTTP.

Для розпізнавання стану входу найбільш актуальним є утворення токену. При створенні токена записуються основні права, зашифровується записане, а

потім, при запитах стану входу користувача вже не потрібно шукати інформацію про нього в БД, що дає можливість відразу зчитувати її з токена й довіряти їй без перевірки. Тобто, кожен JWT (JSON Web Tokens) містить конкретну інформацію, яка може бути інтерпретована будь-якою стороною, яка має цей токен. Наприклад, ця інформація може містити ідентифікатор користувача, якому вона видана.

Web-токен JSON, або JWT, є стандартизованим, у деяких випадках підписаним та/або зашифрованим форматом упаковки даних, який використовується для безпечної передачі інформації між двома сторонами [10].

Тобто, формат упаковки даних JWT визначає особливу структуру інформації, що відправляється по мережі. Вона представлена у двох формах – серіалізованій та десеріалізованій. Перша використовується безпосередньо для передачі даних із запитами і відповідями. З іншого боку, щоб читати та записувати інформацію в токен, потрібна його десеріалізація.

У несеріалізованому вигляді JWT складається із заголовка та корисного навантаження, що є звичайними JSON-об'єктами.

Заголовок (заголовок JOSE) в основному використовується для опису криптографічних функцій, які застосовуються для підпису та/або шифрування токена. Тут також можна вказати додаткові властивості, наприклад тип вмісту, хоча це рідко потрібно. Щоб дізнатися більше про заявки заголовка, зверніться до специфікації .

Якщо JWT підписано та/або зашифровано, в заголовку вказується ім'я алгоритму шифрування. Для цього призначена заявлення alg.

Слово "заявлення" у специфікації означає просто частину інформації та є аналогічним ключу JSON-об'єкта. Вона представлена у вигляді пари ім’я: значення, де ім'я завжди є рядком. Значення заявлення може мати будь-який тип даних, що серіалізується. Наприклад, наступний об'єкт JSON складається з трьох заявлень: iss, expі http://example.com/is_root:

```
{
  "iss": "joe",
  "exp": 1300819380,
  "http://example.com/is_root": true
}
```

Заявлення бувають службовими та користувальницькими. Перші зазвичай є частиною будь-якого стандарту, наприклад, реєстру JWT заявлення та мають певні значення. Найбільш поширені службові заявлення:

iss - видавець токена;

sub - описуваний об'єкт;

aud - одержувачі;

exp - дата закінчення строку дії;

iat - час створення.

Корисне навантаження – це корисні дані, тобто частина токена, в якій розміщується вся необхідна інформація користувача. Як і заголовок, корисне навантаження є звичайним об'єктом JSON. Тут жодне поле не є обов'язковим. Зазвичай використовуються вже розглянуті службові заявлення iss, suba aud які є також специфічними для додатку даними. Наприклад, ось так виглядає JWT у фреймворку OpenID:

```
{
 "iss": "https://auth-provider.domain.com/",
 "sub": "auth|some-hash-here",
 "aud": "unique-client-id-hash",
 "iat": 1529496683,
 "exp": 1529532683
}
```

Серіалізовані JSON токени – це рядок наступного формату:

[Header].[Payload].[Signature]

Заголовок (header) і корисне навантаження (payload) присутні завжди, а ось підпис (signature) може бути відсутнім.

Приклад компактної форми:

```
eyJhbGciOiJub25lIn0.
eyJzdWIiOiJ1c2VyMTIzIiwicHJvZHVjdElkcyI6WzEsMl19.
```

Отримується з таких даних:

```
// заголовок
{"alg":"none"}
// корисні дані
{"sub":"user123", "productIds":[1,2]}
```

Тут визначено непідписаний токен, його заявлення alg дорівнює none. Тому в рядку немає третьої частини, однак точка після другого фрагмента все одно додається.

Процес **серіалізації** JWT складається з кодування заголовка, корисного навантаження та підпису, якщо він є, за допомогою алгоритму base64url. Являє собою просту варіацію base64, яка використовує URL безпечний символ _замість небезпечних +і /. Справа в тому, що деякі інструменти обробки даних розпізнають керуючі символи в рядку, тому їх не повинно бути.

Серіалізацію JWT можна представити так:

```
function encode(h, p) {
   const header = base64UrlEncode(JSON.stringify(h));
   const payload = base64UrlEncode(JSON.stringify(p));
   return `${header}.${payload};
}
```

Щоб декодувати токен, досить просто розбити його по точках і конвертувати заголовок та корисне навантаження з коду base64url назад у рядок. Приклад коду, який це робить:

```
function decode  (jwt) {
    const [headerB64, payloadB64] = jwt.split('.');
    const headerStr = base64UrlDecode(headerB64);
    const payloadStr = base64UrlDecode(payloadB64);
    return {
      header: JSON.parse(headerStr),
      payload: JSON.parse(payloadStr)   };
  }
```

Зрозуміло, JSON токени не кодуються вручну. Існує безліч бібліотек, призначених для цього. Наприклад, jsonwebtoken :

```
const jwt = require('jsonwebtoken');
const secret = 'shhhhh';
// серіалізація - кодування
const token = jwt.sign({ foo: 'bar' }, secret);
// перовірка та декодування
const decoded = jwt.verify(token, secret);
console.log (decoded.foo);
```

Цей код створює підписаний JWT із використанням секретного слова. Потім він перевіряє справжність токена і декодує його, застосовуючи той самий секрет.

Процес запитів з використанням токену (рис.2.12):

1. Клієнт запитує сервер відправлення імені користувача і пароля.

2. Сервер генерує токени за допомогою шифрування відповідно до інформації про користувача. Інформація про користувача включає номери облікових записів, термін дії токена тощо, які настроюються сервером.

3. Сервер повертає клієнту токен.

4. Клієнт використовує файли cookie для зберігання токена, і наступні запити принесуть цей токен.

5. Сервер розшифровує токен та підтверджує правильність інформації про користувача. Якщо вона вірна, перевірку пройдено.

6. Сервер повертає клієнту результат перевірки.

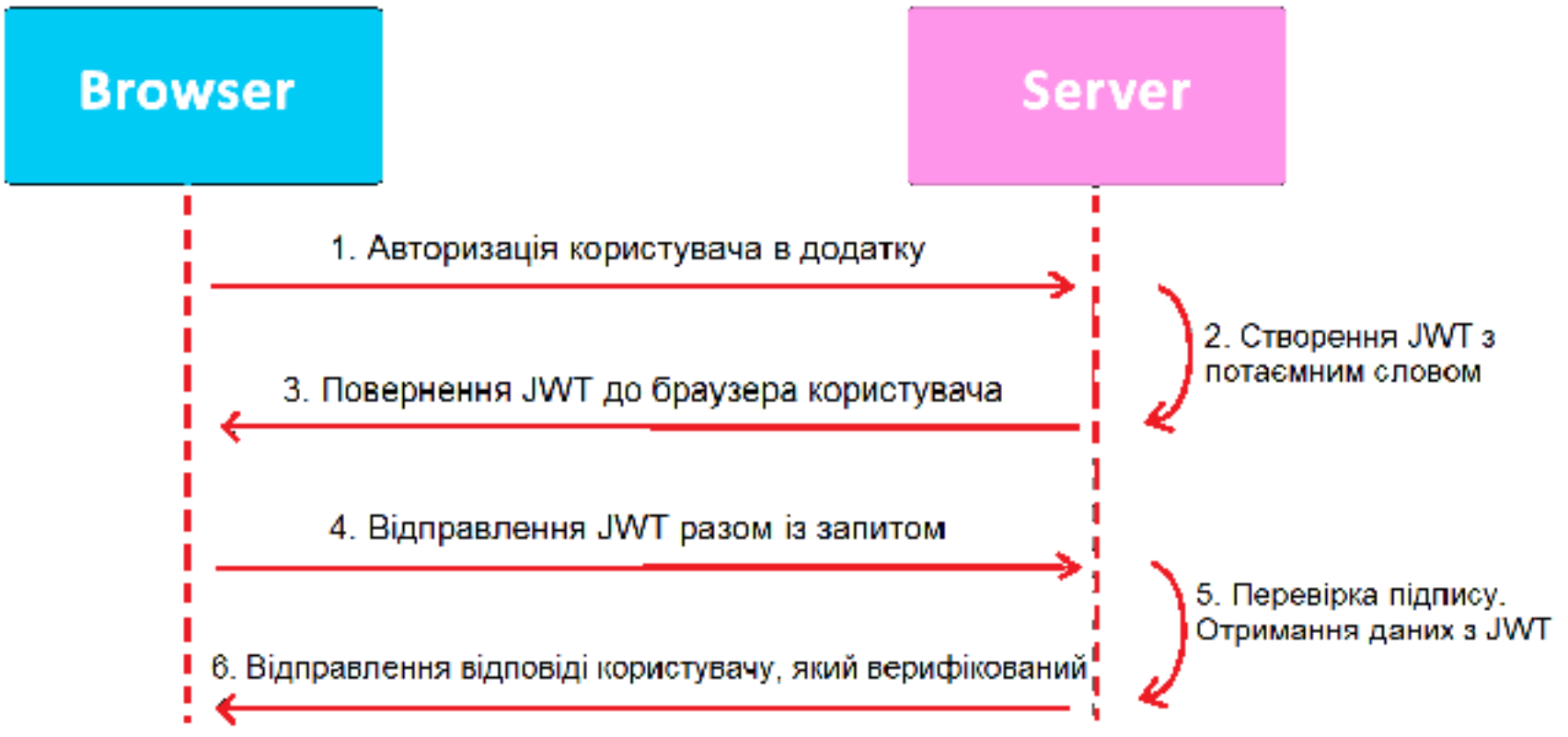


Рис. 2.12. Токен процес [10]

JWT використовує два механізми захисту інформації: підписи і шифрування.

Токен і сеанс - це два різні методи автентифікації сервера. Взагалі кажучи, файл cookie зберігатиме значення локально на клієнті, а потім прикріплюватиме значення до HTTP і відправлятиме його на сервер. Тоді як сервер використовує це значення, щоб визначити, чи увійшов зареєстрований користувач у систему? Це те, що роблять сеанс та токен.

Процес запиту сеансу:

1. Клієнт запитує сервер відправлення імені користувача і пароля.

2. Сервер генерує sessionId, пов'язує дані користувача і зберігає їх у базі даних.

3. Сервер повертає sessionId клієнту.

4. Клієнт використовує файли cookie для зберігання ідентифікатора сеансу, і наступні запити повертатимуть цей ідентифікатор сеансу.

5. Якщо сервер отримує ідентифікатор сеансу, він переходить до бази даних для пошуку даних користувача. Якщо він їх знаходить, це означає, що автентифікація пройдена.

6. Сервер повертає клієнту результат перевірки.

Розглянемо, як можна керувати автентифікацією в Express.js [11]. Для аналізу тіл HTTP-запиту та аналізу необхідних файлів cookie під час виконання автентифікації використовуватимемо два пакети програмного забезпечення Express - body-parser та cookie-parser. Встановлення модулів:

```
npm install --save body-parser cookie-parser
```

Для початку давайте імпортуємо бібліотеки, які ми встановили раніше:

```
const express = require('express');
```

```
const exphbs = require('express-handlebars');
const cookieParser = require('cookie-parser');
const bodyParser = require('body-parser');
```

Для хешування паролів і генерації маркера автентифікації будемо використовувати **crypto** модуль Node, дії з яким було описано в підпункті 2.2. У тому ж самому підпункті також було детально описано створення сторінок реєстрування та login користувача.

Форма login надсилатиме POST запит. Одночасно буде реалізоване надсилання маркера автентифікації для входу. Цей маркер використовуватиметься для ідентифікації користувача, і кожного разу, коли він надсилає HTTP-запит, цей маркер надсилатиметься як файл cookie:

```
const generateAuthToken = () => {
   return crypto.randomBytes(30).toString('hex');
}
```

За допомогою нашого допоміжного методу ми можемо створити обробник запитів для сторінки входу з таким кодом:

```
const authTokens = {};
app.post('/login', (req, res) => {
const { email, password } = req.body;
const hashedPassword = getHashedPassword(password); // розглядалось вище
   const user = users.find(u => {
      return u.email === email && hashedPassword === u.password
   });

   if (user) {
      const authToken = generateAuthToken();
// зберігати токен автентифікації
      authTokens[authToken] = user;
// Налаштування токену автентифікації в cookies
      res.cookie('AuthToken', authToken);
// Перенаправлення користувача на захищену сторінку
      res.redirect('/protected');
   } else
         {
      res.render('login', {
         message: 'Invalid username or password',
         messageClass: 'alert-danger'
      });
   }
});
```

У цьому обробнику запитів об’єкт-карта, що викликається authTokens, використовується для зберігання маркерів автентифікації як ключа відповідного користувача та як значення, що являє простий маркер для пошуку користувача. Ви можете використовувати базу даних, наприклад Redis, або будь-яку базу даних для зберігання цих токенів - ми використовуємо цю об’єкт-карту для простоти.

Поверх усіх обробників запитів і нижче cookie-parser проміжного програмного забезпечення створимо проміжне програмне забезпечення для введення користувачів у запити:

```
app.use((req, res, next) => {
// Get auth token from the cookies
    const authToken = req.cookies['AuthToken'];
// Inject the user to the request
    req.user = authTokens[authToken];
    next();
});
```

Тепер можемо використовувати req.user у обробнику запитів, щоб перевірити, чи автентифікований користувач за допомогою маркера.

Для цього у контролері створимо обробник запитів для сторінки:

```
app.get('/protected', (req, res) => {
  if (req.user) {
    res.render('protected');
  } else {
    res.render('login', {
      message: 'Please login to continue',
      messageClass: 'alert-danger'
    });
  }
});
```

Об’єкт req.user використовується для перевірки автентифікації користувача. Якщо цей об’єкт порожній, користувач не автентифікований.

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-логики, 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 логінізації 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 контроллеру 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  контролеру  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
созданый 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 контроллера authController для переходу на відповідні сторінки прописуємо перевірку токену та при його відсутності переведення на сторінку логінізації:

```
// 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-представленнях beck-end (рис. 2.13 – 2.16).

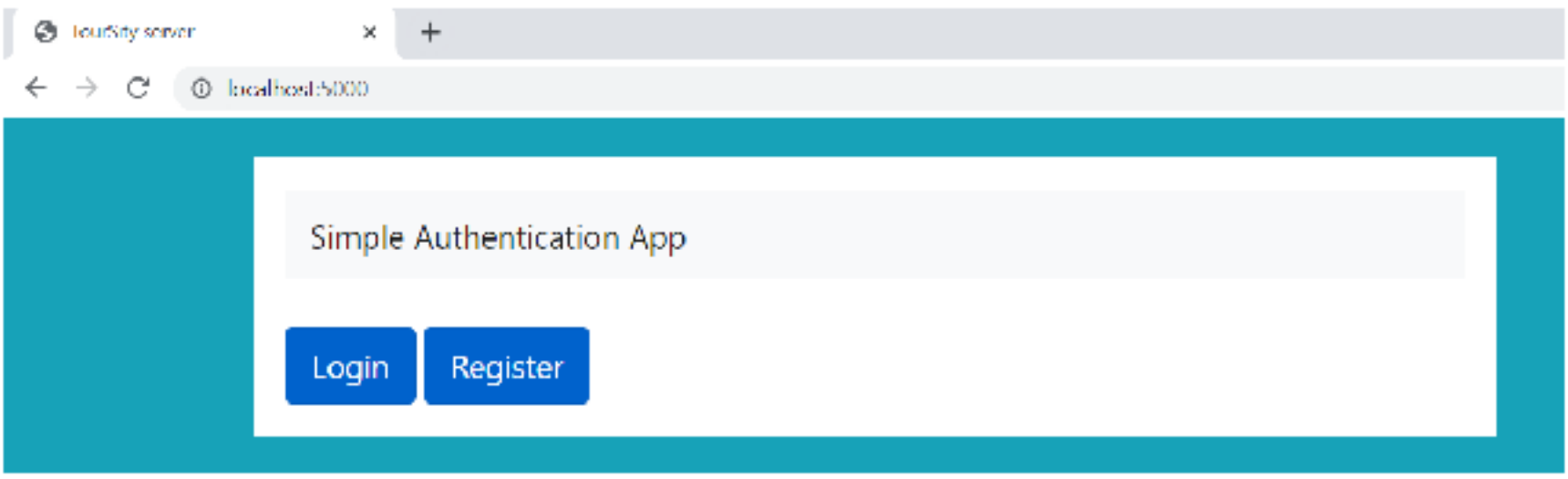


Рис. 2.13. Вигляд сторінки home.hbs

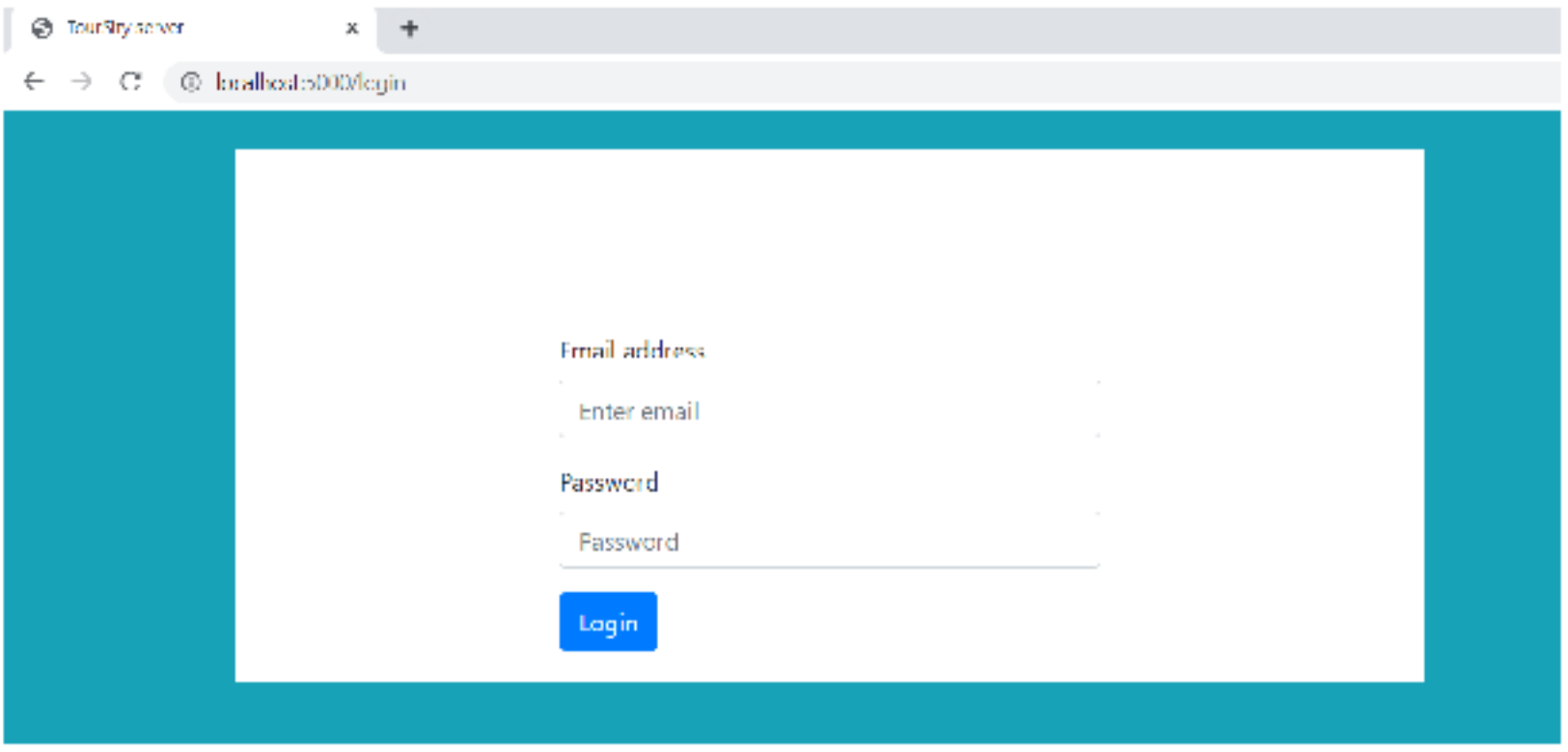


Рис. 2.14. Сторінка login

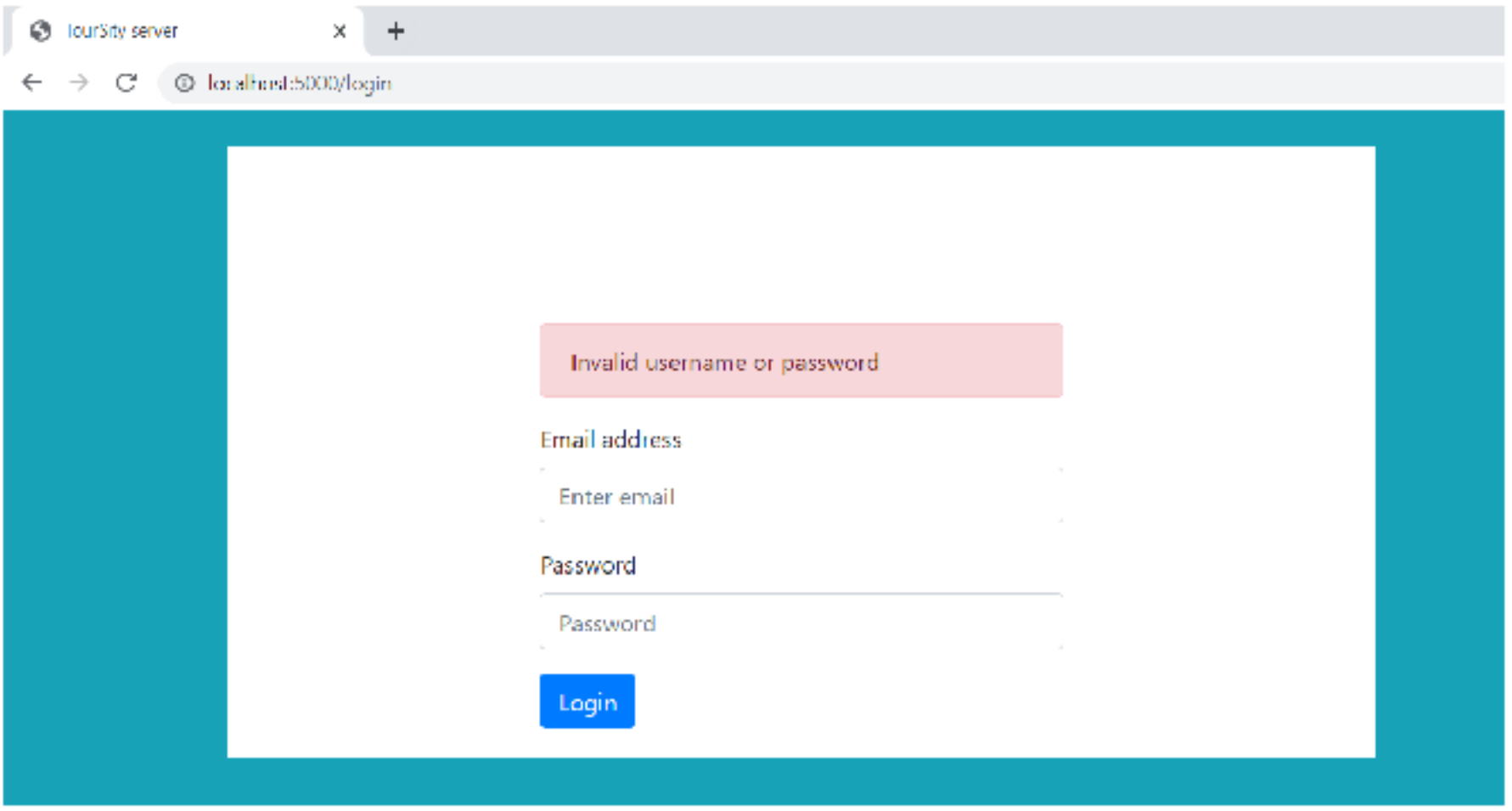


Рис. 2.15. Виведення повідомлення щодо помилки на сторінці login

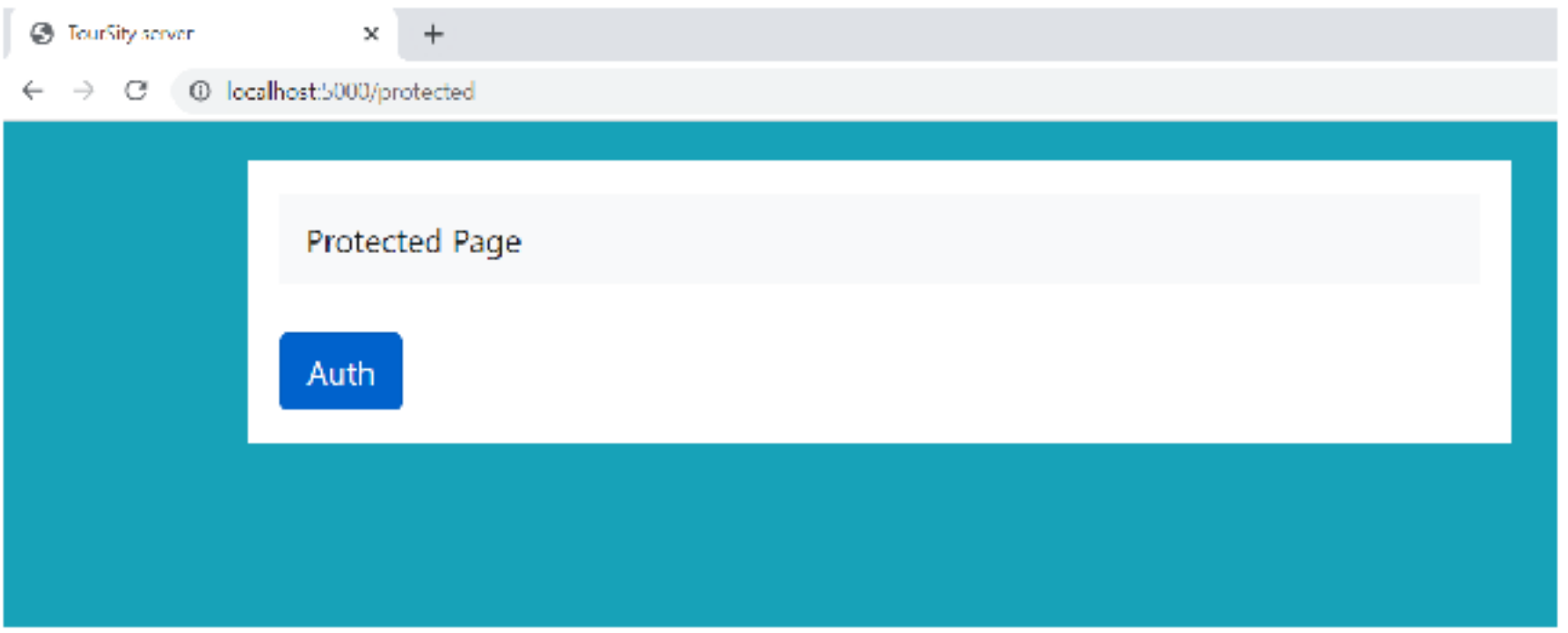


Рис. 2.16. Перехід на сторінку опрацювання даних Auth

- зупиняємо проект:

Ctrl + C.

2.4. РОЗРОБКА АРХІТЕКТУРИ КОНТЕНТУ WEB-ДОДАТКУ

2.4.1. Теоретичний модуль

Архітектура контенту

Контент сайту – це інформаційне наповнення його web-сторінок. Архітектура контенту можлива для будь-якої публікації [13]. При розробці сайтів та їх сторінок потрібно створити комфортну модель взаємодії користувачів з додатком. Важливу роль у цьому відіграють елементи в структурі контенту та навігація по них.

Основне завдання архітектури контенту - зробити так, щоб користувач швидко і зручно досягав мети пошуку інформації. Кожен елемент контенту має бути корисним користувачеві: поділ контенту на поля, акцентування на тих чи інших елементах, створення списків, виділення цитат і багато іншого - це все для того, щоб користувач швидко і із задоволенням знайшов те, за чим він прийшов.

Контент сайту, найчастіше, це текстове наповнення з графічними та по можливості медіа елементами. Текст найбільш популярний елемент, через його простоту та легкість представлення на сайті. Залежно від джерела текстовий контент прийнято розділяти на копірайтинг (написання з нуля), рерайтинг (переписування готових текстів) та плагіат (спарсені чи скопійовані тексти).

Для збільшення швидкості сприйняття контенту елемент текстового контенту доповнюється графічним елементом, а по можливості й аудіо та відео.

Контерт, що розробляється повинний бути придатним. Придатний - це той тип контенту, який має реальну користь для своїх користувачів, який є актуальний на даний момент та висвітлює необхідну інформацію і відповідає очікуванням відвідувача. Контент повинен бути структурованим.

Структурування контенту

Для того, щоб інформація на сайті була структурована, використовуються такі елементи:

- інформаційна архітектура (ієрархія категорій та тегів, продумана навігація);
- моделі контенту (візуалізація того, як контент виглядатиме на сторінці, розмір та функціональність різних елементів);
- структура (спочатку задані поля - заголовки, авторство, тіло статті).

Добре структурований контнент дозволить отримати більший відгук від користувача, ніж банальні кілька абзаців тексту. Тут важливо спростити "споживання" інформації читача. Це можна зробити кількома способами:

- підберіть заголовок, який працюватиме;

- розбийте текст на смислові блоки, обов'язково додайте підзаголовки;

- використовуйте нумеровані та марковані списки для більш комфортного сприйняття інформації;

- подавайте інформацію невеликими абзацами з можливістю розгортання поданого тексту. На початку надайте основну пропозицію, що є достатньою, щоб передати ключову думку.

Заголовки - це окрема тема. Оскільки тут йдеться про залучення аудиторії та отримання миттєвого відгуку, приділіть цьому питанню максимум часу. Користувачеві повинно бути не важко визначити за заголовком, чи варто відкривати вашу публікацію чи ні.

Що потрібно для складання інформативного та ефективного заголовка:

- покажіть, що пост містить інформацію, яка буде корисна користувачеві. Зробіть акцент на тому, що користувач отримає: нову, невідому ще досі йому інформацію; спрямування на виконання певних дій; якісь уміння та навички; бонуси тощо;

- використовуйте в заголовку уточнюючі деталі, наприклад цифри. Якщо ви хочете розповісти користувачеві те, про що він може дізнатися та чого навчитися, додайте в заголовку період часу, за який він їх набуде;

- персоналізуйте заголовок. Вкажіть у заголовку, кому дана інформація може бути корисною, наприклад: подорожуючим, дітям чи для родинного відпочинку тощо.

- заголовок має бути коротким, так він приверне більше уваги. Намагайтеся писати ємні, короткі, а головне такі заголовки, що запам'ятовуються.

- додайте до заголовка пошуковий запит, якщо це можливо.

Заголовки повинні доповнюватись графічними елементами контенту. Користувачі насамперед звертають увагу на оформлення, а потім починають вчитуватись у вміст. Якщо зміст або стиль заголовку та графічного додатку їх не влаштовує і не зворушує, вони перейдуть в інший заголовок.

Створення контенту має здійснюватись із застосуванням маркетингових правил написання SEO-тексту для сайту [14]. SEO (англ. Search Engine Optimization) – це комплекс заходів для покращення сайту для його ранжування в пошукових системах.

Базові вимоги щодо підготовки SEO-тексту

1. Об`єм. Текст має бути щонайменше 2500 знаків (з пробілами). Іноді його можна збільшити до 3500–4000. Такий обсяг дозволить повніше розкрити тему та рівномірніше розподілити ключові запити.

Часто трапляється думка, що текстовий масив – це неефективний підхід. Але статистика наполегливо підтверджує, що наявність SEO-тексту на сторінці категорій товарів на сайті Internet-магазину збільшує загальну релевантність

сторінки. Тому якщо така методика позитивно впливає на SEO, то використовувати її доцільно.

2. Структура текста. Щоб текст не був схожий на «простирадло», необхідно використовувати прийоми структурування:

заголовки (h2 - h4);

розбиття на абзаци;

списки (нумеровані чи звичайні);

виділення шрифтом (можна використовувати курсив, підкреслення).

3. Тематичні посилання. Потрібно додавати тематичні посилання. Наприклад, на сторінках виробників або окремих моделей.

4. Різноманітність контенту. За наявності унікальних зображень та відео, розміщеного на ютуб-каналі, ними можна і потрібно розбавити текст. Впроваджувати додатковий контент необхідно продумано, не перевантажуючи сторінку. Розміщуйте зображення відповідно до змістового наповнення сторінки.

5. Нудота тексту. Нудота (переспам) тексту – це показник, що визначає частоту використання будь-якого слова у текстовому документі. Нудота показує, скільки у статті будь-яких слів або фраз, що повторюються. Чим більше повторень, тим більше текст нудить. Для онлайн перевірки нудоти тексту наябільш поширеними є такі хмарні ресурси [15]:

- Advego. Сервісом зручно перевіряти академічну нудоту тексту та нудоту за словом (//advego.ru/text/seo/). .

- SeoTXT.com (www.seotxt.com/service/optimizer/) - оптимізатор тексту для визначення щільності ключових слів та фраз у тексті. За допомогою його можна оптимізувати сторінку під певний набір слів, використовуючи багато налаштувань та фільтрів.

6. Вода. Намагайтеся уникати загальних та порожніх фраз. Виключайте шаблонні обороти у стилі:

- у наш час дедалі більше…

- на даний момент…

7. Орфографія. Помилок у тексті не повинно бути – ні орфографічних, ні пунктуаційних.

Моделювання потоків даних контенту DFD

Для візуалізації процесу перебігу інформації є дуже корисним в межах системи створювати діаграму DFD (англ. Data Flow Diagram). Діаграми DFD застосовуються для аналізу існуючих та моделювання нових систем.

Побудова DFD здійснюється за правилами та інструкціями, де символи відображають **чотири компоненти діаграми DFD** [16]:

Зовнішні сутності - зовнішні системи, з яких надходить або куди направляється інформація в результаті взаємодії з системою, що зображається.

Іншими словами, це джерела та пункти доставки інформації, яка надходить або йде з системи. Такими сутностями можуть бути зовнішні організації, особи, комп'ютерні чи бізнес-системи. Ці сутності також мають інші назви, наприклад, "термінатори", "джерела", "приймачі" або "агенти", і, як правило, розташовуються по краях схеми.

Процеси - будь-які процеси, які ведуть до зміни інформації та створення вихідних даних. Наприклад, виконання підрахунків, сортування даних згідно з встановленою логікою або направлення інформаційного потоку відповідно до бізнес-правил. Для опису процесів використовуються короткі мітки, наприклад, «Відправлення платежу».

Сховища даних - файли або репозиторії, де зберігається інформація для подальшого використання, наприклад бази даних або форми заявки на участь. Сховища даних супроводжуються простими мітками, наприклад, «Замовлення».

Потоки даних — маршрути, якими інформація переміщається між зовнішніми сутностями, процесами та сховищами даних. Потоки даних ілюструють взаємодію між іншими компонентами та відображаються у вигляді стрілок, як правило, з короткими мітками, наприклад, «розрахункова інформація».

Правила щодо побудови діаграм DFD

Кожен процес має супроводжуватися як мінімум одним вхідним та одним вихідним потоком;

У кожне сховище повинен впадати щонайменше один потік даних і щонайменше один — витікати;

Дані, що зберігаються у системі, повинні проходити через процес;

Кожен процес діаграми DFD повинен вести або до іншого процесу або до сховища даних.

Етапи створення діаграми DFD:

1. Введення та виведення інформації

Більшість процесів і систем починаються з введення інформації із зовнішнього джерела та закінчуються виведенням даних в іншу сутність чи базу. Виявивши ключові пункти введення та виведення інформації отримаємо загальну картину своєї системи та її основних завдань. Важливо визначитися з введенням і виведенням інформації на ранньому етапі роботи, оскільки це той самий фундамент, на якому буде вибудовуватися частина діаграми DFD.

2. Створення контекстної схеми

Після того як ви визначилися з основними пунктами введення та виведення інформації здыйснюэться побудова контекстної схеми. Додаэмо вузол одиничного процесу та з'єднуэмо його з необхідними зовнішніми

сутностями. Цей вузол символізує узагальнений процес, через який проходить інформація від введення до виведення.

3. Розширення контекстної схеми до DFD 1-го рівня

Вузол одиничного процесу, представлений у вашій контекстній діаграмі, дає обмежене уявлення про систему, тому рекомендується розбити його на підпроцеси. Діаграма DFD 1-го рівня повинна включати кілька вузлів процесів, а також основні бази даних та всі зовнішні сутності.

4. Поглиблення діаграми DFD до 2-го рівня і далі

Якщо ви хочете створити більш докладну схему потоку даних, дотримуйтесь інструкцій з пункту 3. Процеси, що входять до складу діаграми DFD 1-го рівня, можна так само розбити на докладніші підпроцеси. Знову ж таки, не забудьте включити в діаграму всі необхідні сховища та потоки даних. На цьому етапі ви маєте отримати досить докладну картину своєї системи. Однак, якщо ви вирішите поглибити діаграму далі 2-го рівня, просто повторіть описаний вище процес. Коли діаграма досягне бажаного рівня деталізації, можна закінчити з розбивкою.

5. Перевірка остаточної схеми

Коли схема буде готова, перевірте її від початку остаточно. Звертайте особливу увагу: чи логічно все викладено? Чи всі сховища даних на місці? Фінальний варіант схеми повинен давати чітке уявлення про те, як буде збудовано система та чи перехід від одних даних контенту до інших призводить до бажаного результату.

Побудова діаграм DFD відбувається ітераційно: від контекстних схем до псевдокоду. За допомогою шарів та рівнів діаграму DFD можна доповнювати все більшими та більшими подробицями, фокусуючи увагу на одній конкретній ділянці. Необхідний рівень деталізації залежить від цілей, що стоять перед вами.

DFD 0-го рівня називається контекстною схемою (рис. 2.17). Це найпростіший спосіб зображення аналізованих або моделюваних систем та процесів. Такі схеми показують загальну картину і представляють систему як єдиного процесу, наділеного зв'язками із зовнішніми сутностями. Схеми 0-го рівня будуть зрозумілі широкій аудиторії, включаючи учасників проекту, бізнес-аналітиків та розробників.

DFD 1-го DFD 2-го рівнів дають детальніше уявлення про елементи контекстної схеми. Розбивши узагальнений процес контекстної схеми на підпроцеси, ви цим зможете виділити основні функції системи для глибшого занурення у систему.

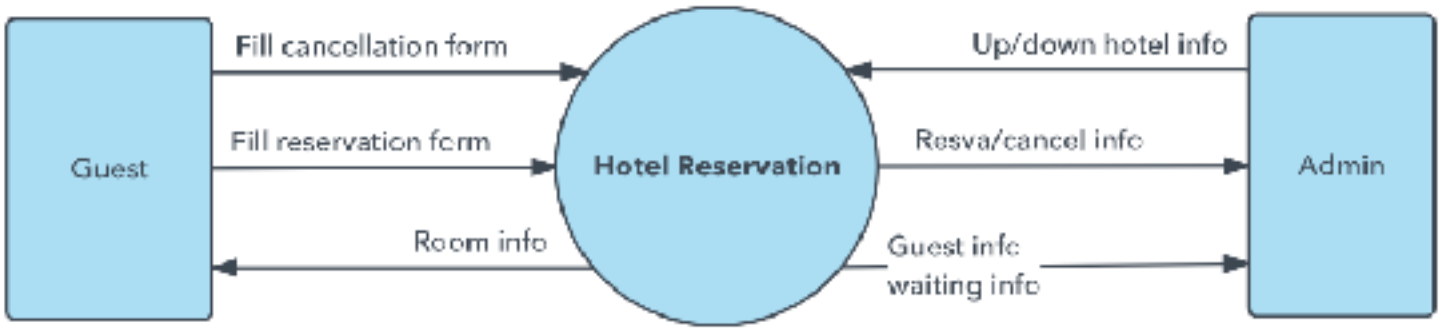


Рис. 2.17. Контекстна схема [16]

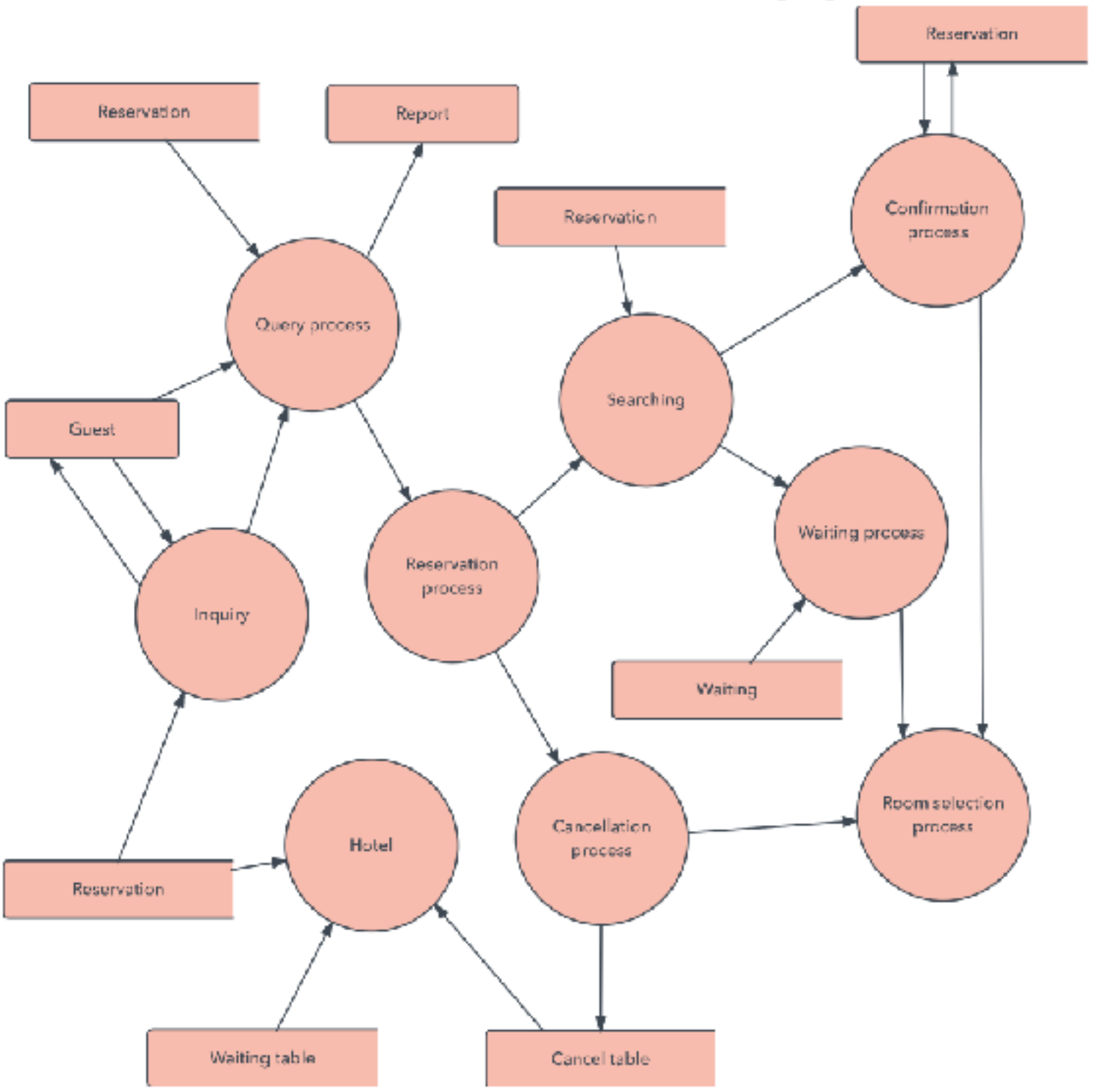


Рис. 2.18. Схема 2-го рівня [16]

Шари дозволяють зібрати рівні, що випадають безпосередньо в DFD-схемі: такий метод поєднує глибокий аналіз з ясністю викладу. При досить високій деталізації розробники та дизайнери можуть застосувати діаграми DFD для написання псевдокода, який є поєднанням програмної та природної мови. Завдання псевдокода - спростити роботу з написання повноцінного коду.

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).

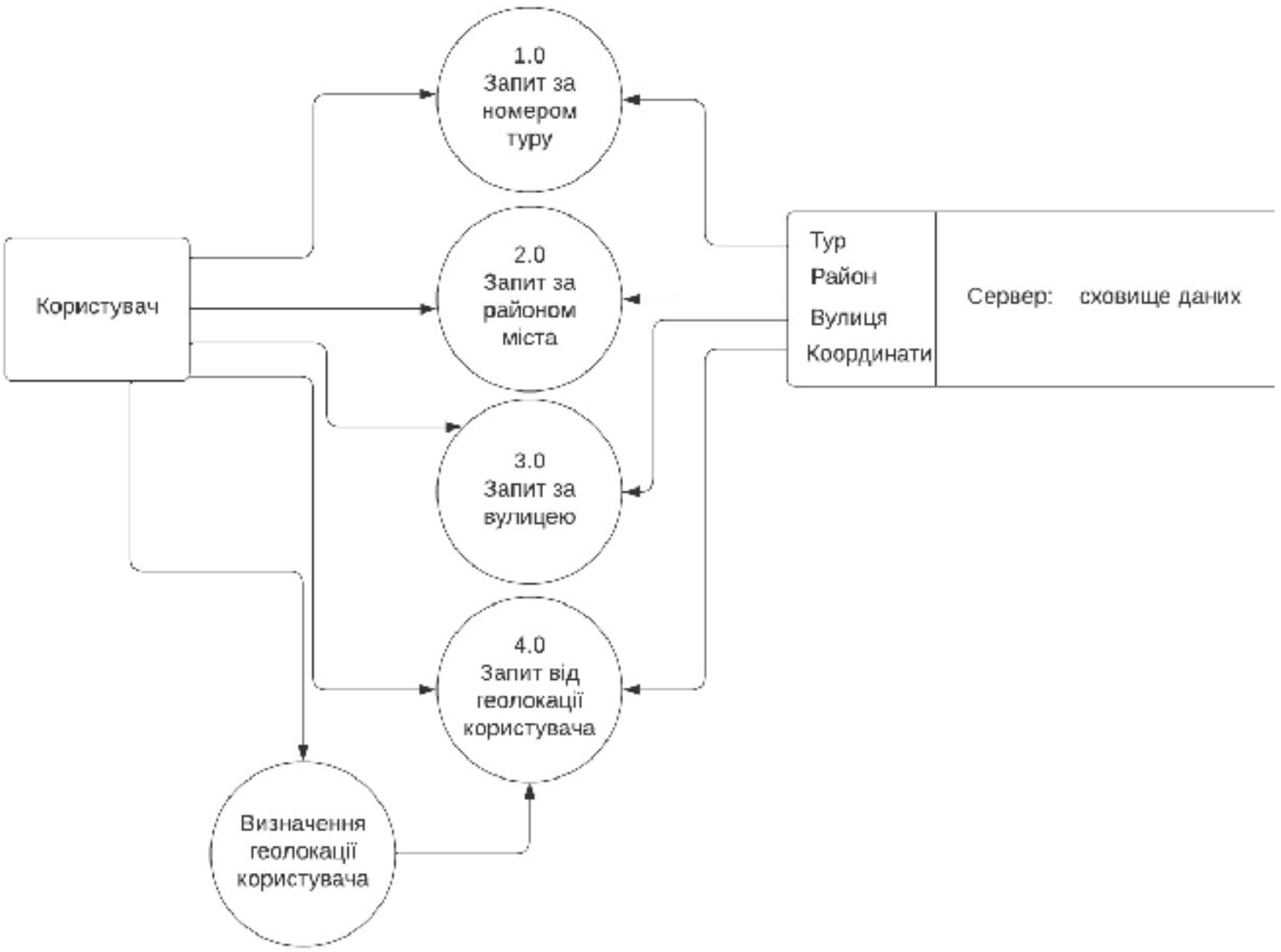


Рис. 2.19. DFD діаграма 2-го рівня

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 данних 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 токену логінізації 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> <img  src="/static/files/{{this.image}}" width="75"  >
            </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 данних контенту потрібна окрема сторінка web-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>
```

```
<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>
```

Продовження лістингу 2.13

```
<div class="mb-3">
  <div>
   <img  src="/static/files/{{destination.image}}" width="75"  >
   <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 beck-end. Переходимо на сторінку колекцій БД (рис. 2.20).

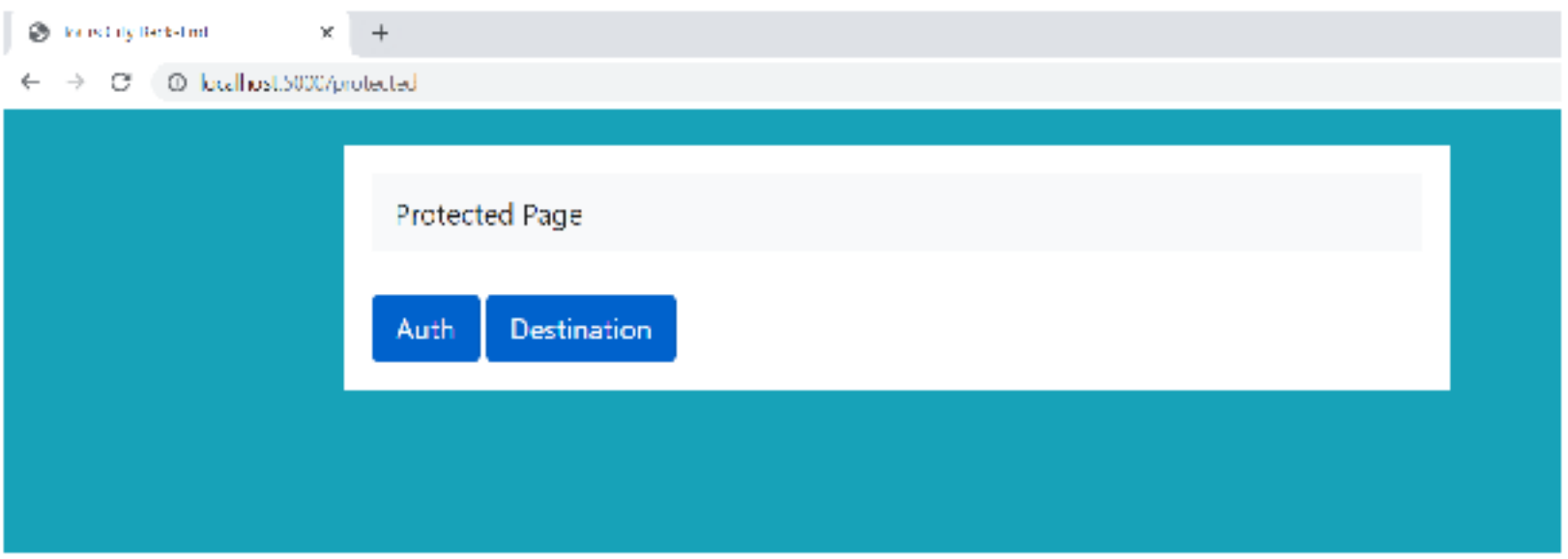


Рис. 2.20. Вигляд сторінки protected.hbs

- на сторінці представлення list.hbs (рис. 2.22) автивізувавши Create New переходимо на сторінку введення данних (рис. 2.21).

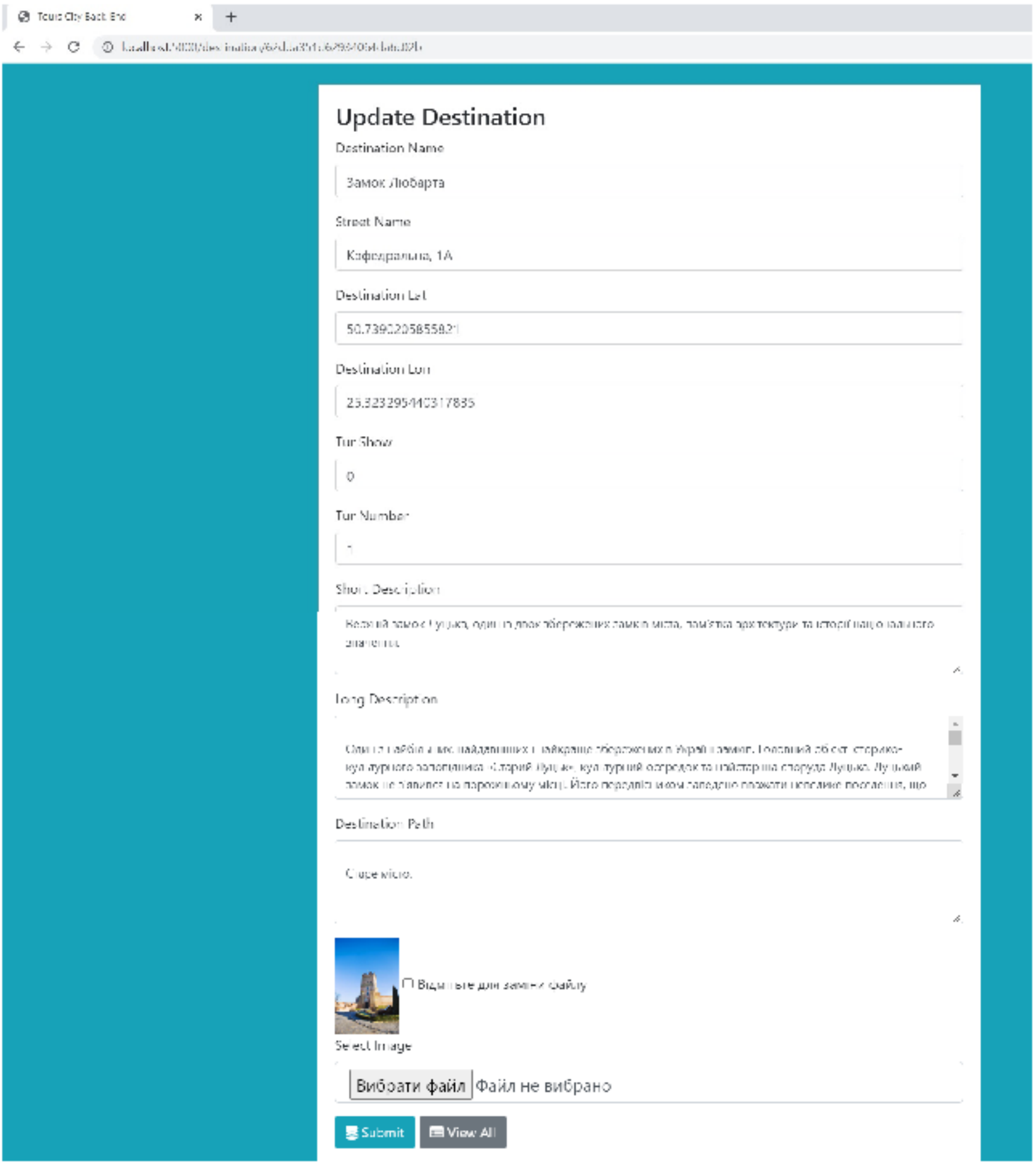


Рис. 2.21. Вигляд addOrEdit.hbs з введеними данними

- На сторінці представлення addOrEdit.hbs натиснувши клавішу Submit переходимо на сторінку збережених данних (рис. 2.22).

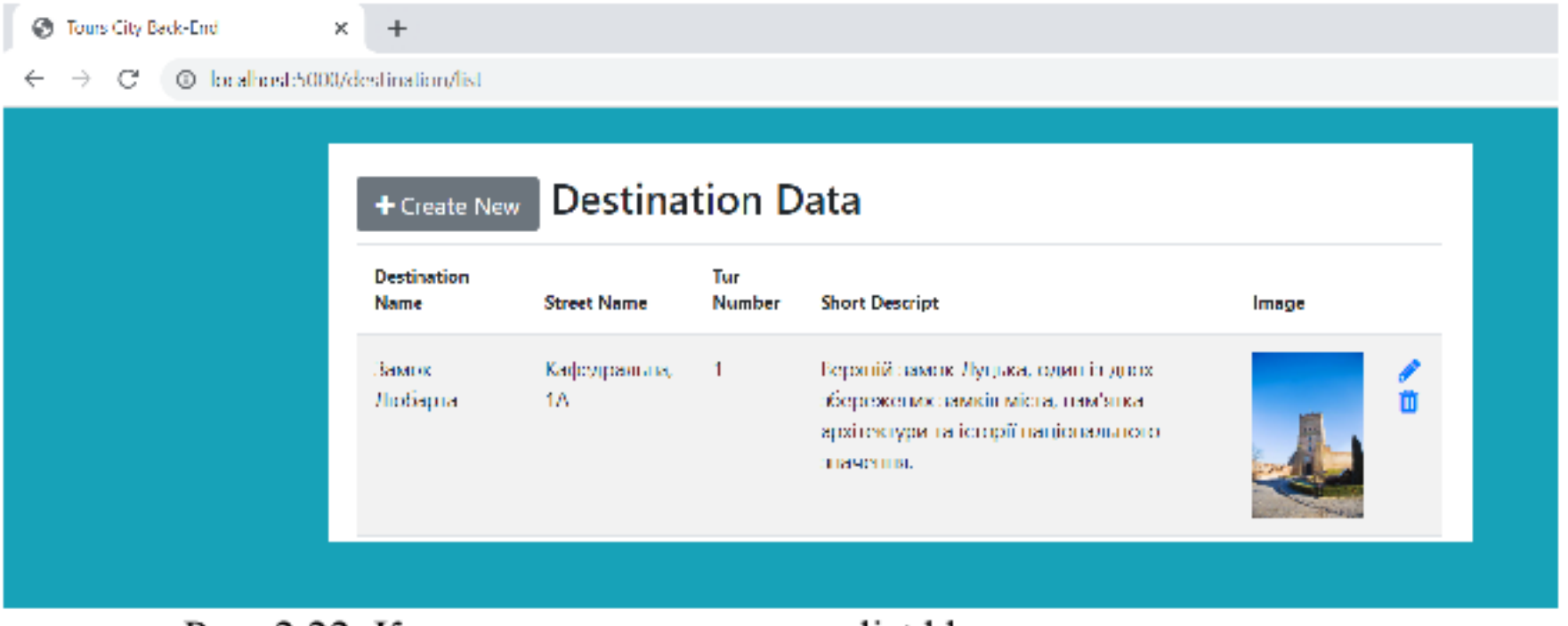


Рис. 2.22. Контент з представлення list.hbs до введення даних

- зупиняємо проект:

Ctrl + C.

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

2.5.*1. Теоретичний модуль*

У попередніх підпунктах цього розділу було описано теорію та створено сервер для введення, зберігання та редагування даних web-додатку. Наступним етапом розгляду є отримання окремих елементів даних із сервера для оновлення розділів веб-сторінки без необхідності завантаження всієї нової сторінки. Тобто для передачи даних на сторону клієнта (front-end) потрібно створити видалений API. Для цього створюємо API на сервері та дозволяємо у подальшому клієнту звертатися до нього. Деякі питання з REST API було попередньо розглянуто у підпункті 2.2 цього розділу під час побудови CRUD. У цьому підпункті зосередимось на утворені тої частини REST API, яка відповідатиме за передавання даних із web-сервера на бік клієнта.

Передавання даних між web-сервером та web-колієнтом розглядається як міжпроцесна взаємодія як мінімум двох процесів: процес-сервера та процес-клієнта. Процес-сервер може віддати дані у форматі json або html, хоча при цьому жоден із них не є реальним типом зберігання всередині сервера.

Побудова REST API за даною методикою розглядається як розмежування потреб частин back-end та front-end web-додатку. Відокремленні потреб інтерфейсу клієнта від потреб сервера, що зберігає дані, підвищується

переносимість коду клієнтського інтерфейсу на інші платформи, а при спрощенні серверної частини покращується масштабованість [17].

API – це Application Programming Interface. Технологія, що описує метод, що програмне забезпечення надає зовнішнім користувачам для комунікації з цим. API передбачає «містки», які дають можливість програмі А отримати доступ до даних програми Б або певного функціоналу.

Під час використання обробників подій створюються запити HTTP до методів дії API. Front-end через API витягуватиме дані з back-end програмного забезпечення. Виглядає це так [18]:

- користувач вводить букву на сайті;
- сторінка відправляє запит до підказок по API;
- відповідні «покажчики» повертають відповідь;
- сайт обробляє отриманий контент та виводить результат клієнту.

REST – це архітектурний підхід, що визначає, як API мають виглядати. Читається як "Representational State Transfer". Цьому набору правил і слід розробник під час створення своєї програми. Одне з цих правил говорить, що при зверненні до певної адреси отримується певний набір даних (ресурс) [19].

REST API - це API, який структурований відповідно до структури REST для API.

Принципи REST:

1. Прихід до моделі клієнт-сервер. Тут закладено обмеження потреб. Потрібно чітко відокремлювати клієнтський інтерфейс від серверних «інтересів», які відповідають за зберігання даних. Принцип допомагає підвищити переносимість кодифікації на інші платформи, а також покращити масштабованість системи.

2. Відсутність стану. Між запитами сервера не потрібно зберігати дані про стан клієнта. Зворотне правило також діє. Запити з боку "користувача" формуються так, щоб server міг отримати необхідні дані для обробки запиту. Це допомагає «не читати між рядками» складеній утиліті.

3. Кешування. Відповіді серверного типу мають бути явними чи неявними. Це допоможе користувачеві не отримувати застарілі та неправильні відомості. Сприяє покращенню продуктивності.

4. Одноманітність інтерфейсу. Фундаментальний принцип. У архітектури має бути одноманітний interface. Клієнт повинен завжди розуміти, які адреси та в якій формі отримуватимуть запит. Сервер – розбиратися у форматах, що видаються як відповіді.

5. Шари. Мається на увазі ієрархія мережевих структур. Іноді клієнт може звернутися безпосередньо до сервера, іноді – через проміжний вузол. Розшарування системи усуне проблему завантаження servers.

6. Код на вимогу. Чи не є обов'язковим «критерієм». Має на увазі, що клієнт зможе розширити функціональність за рахунок завантаження кодів із серверів під виглядом аплетів та сценаріїв.

Структура запиту REST API

Кожна адреса запиту - є маршрутом, пакет даних - запитом, а ресурс - відповіддю. Тому структура запиту буде складатись з:

- типу методу запиту. Визначає один з HTTP- методів;
- маршруту відправки – це адреса, куди буде надіслано запит;
- заголовку. Забезпечує метаінформацію про запит, наприклад, дату надсилання клієнтом цього запиту і розмір його тіла;
- тіла – це набір даних (ресурс), що передаються в запиті.

Тип методу позначає тип запиту, дефакто він є специфікацією операції, яку повинен зробити сервер. Типи запитів (GET, POST, PUT, PATCH, DELETE) та прийоми їх застосування під час побудови CRUD було наочно продемонстровано у підпункті 2.2 цього розділу.

Маршрут – це адреса, за якою надсилається ваш запит. Його структура приблизно така: root-endhoint/?

Root-endpoint - це точка прийому запиту з сервера (API). Наприклад, кінцева точка GitHub - https://api.github.com. Тобто, шлях визначає запитуваний ресурс. Наприклад, припустимо, ви хочете отримати список репозиторіїв для конкретного користувача Git. Згідно з документацією, ви можете використати наступний шлях для цього:

/users/:username/repos

Вам слід підставити під пропуск ім'я користувача. Наприклад, щоб знайти список моїх репозиторіїв, ви можете використати маршрут:

https://api.github.com/users/:username/repos

Остання частина маршруту – параметри запиту. Запити не є частиною REST-архітектури, але на практиці зараз все ґрунтується на них. Параметри запиту дозволяють використовувати у запиті набори пар ключ-значення. Тобто, JSON - загальний формат для надсилання та прийому даних за допомогою REST API. Тому, у цьому прикладі відповідь з Github також буде міститься у форматі JSON.

Заголовки використовуються для надання інформації як клієнту, так і серверу. Взагалі, їх можна використовувати для багато чого, наприклад, для автентифікації та авторизації.

Заголовки являють собою пари ключів-значень. Приклад [19]:

“Content-Type: application/json”. Missing the opening”

Після того як створили сервер та створити власний API можна протестувати запити.

Тестування API за допомогою утиліти Curl.

Curl - це утиліта командного рядка, яка дозволяє виконувати HTTP-запити з різними параметрами та методами. Замість того, щоб переходити до web-ресурсів в адресному рядку браузера, можна використовувати командний рядок, щоб отримання та перегляду ресурсів, що витягуються API у вигляді тексту. Щодо встановлення утиліти Curl та її використання детально описано у джерелі [20].

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 запитами 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":"╨ч╨░╨╗╨╡╨╖╨╜ ╨ы╤О╨▒╨░╤А╤В╨░","streetName":"╨Ъ╨░╤Д╨╡╨┤╤А╨░╨╗╤М╨╜╨░, 1А","destLat":"50.7390205855821 ","destLon":"25.323295440317835","destShow":0,"turNumber":1,"destShort":" …"

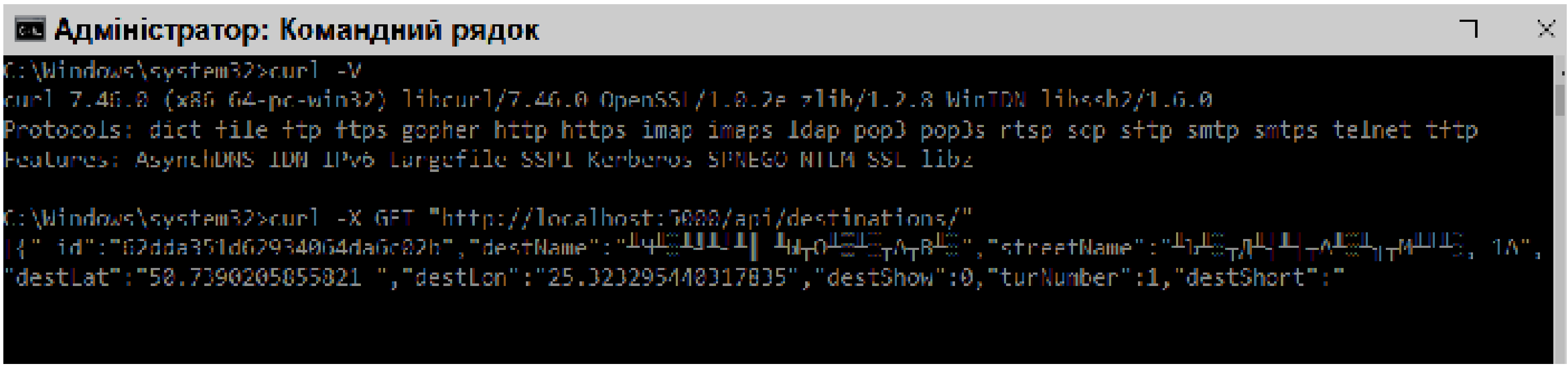


Рис. 2.23. Приблизна відповідь на тестування REST API

2.6. ТЕСТОВІ ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ

1. У архітектурному шаблоні MVC надає інформацію про дані та методи роботи з цими даними

a) model
b) view
c) controller
d) не відноситься до MVC

2. У архітектурному шаблоні MVC відповідає за відображення даних

a) model
b) view
c) controller
d) не відноситься до MVC

3. У архітектурному шаблоні MVC контролює введення інформації та реалізує потрібну реакцію на дії користувача

a) model
b) view
c) controller
d) не відноситься до MVC

4. У архітектурному шаблоні MVC приймає необмежену кількість даних та відправляє до директорії, а в результаті повертає отриманий шлях

a) model
b) view
c) controller
d) не відноситься до MVC

5. У web-фреймворку Express маршрутизація та проміжна обробка здійснюється за рахунок викликів функцій проміжної обробки (middleware). Як позначається об'єкт запиту функції проміжної обробки?

a) res
b) req
c) get
d) set

6. У web-фреймворку Express маршрутизація та проміжна обробка здійснюється за рахунок викликів функцій проміжної обробки (middleware). Як позначається об'єкт відповіді функції проміжної обробки?

a) res
b) req
c) get
d) set

7. Представлення …
   a) відображає вміст контенту засобами графічного інтерфейсу
   b) забезпечує реагування на дії користувача, виконує оновлення контенту
   c) містить всі дані і поведінку, не пов'язані з інтерфейсом користувача
   d) розділяє об'єкти в призначених для користувача інтерфейсах
8. Контролер …
   a) відображає вміст контенту засобами графічного інтерфейсу
   b) забезпечує реагування на дії користувача, виконує оновлення контенту
   c) містить всі дані і поведінку, не пов'язані з інтерфейсом користувача
   d) розділяє об'єкти в призначених для користувача інтерфейсах
9. Модель …
   a) відображає вміст контенту засобами графічного інтерфейсу
   b) забезпечує реагування на дії користувача, виконує оновлення контенту
   c) містить всі дані і поведінку, не пов'язані з інтерфейсом користувача
   d) розділяє об'єкти в призначених для користувача інтерфейсах
10. API - це
   a) створення зв'язку між двома підпрограмами
   b) архітектурний підхід
   c) управління, читання, вставки, видалення та редагування бази даних
   d) набір корисних функцій для доступу та взаємодії з файловою системою
11. REST - це
   a) створення зв'язку між двома підпрограмами
   b) архітектурний підхід
   c) читання, вставка, видалення та редагування бази даних
   d) набір корисних функцій для доступу та взаємодії з файловою системою
12. CRUD - це
   a) створення зв'язку між двома підпрограмами
   b) архітектурний підхід
   c) читання, вставка, видалення та редагування бази даних
   d) набір корисних функцій для доступу та взаємодії з файловою системою

13. Модуль Path - це
    a) створення зв'язку між двома підпрограмами
    b) архітектурний підхід
    c) читання, вставка, видалення та редагування бази даних
    d) набір корисних функцій для доступу та взаємодії з файловою системою
14. Базова функція CRUD create() пов’язана з http-методом:
    a) POST
    b) GET
    c) PUT
    d) DELETE
15. Базова функція CRUD read() пов’язана з http-методом:
    a) POST
    b) GET
    c) PUT
    d) DELETE
16. Базова функція CRUD update() пов’язана з http-методом:
    a) POST
    b) GET
    c) PUT
    d) DELETE
17. Базова функція CRUD delete() пов’язана з http-методом:
    a) POST
    b) GET
    c) PUT
    d) DELETE
18. Результат дії модуля Node.js body-parser
    a) проміжні програми заповняють властивість req.body
    b) взаємодія через спеціальні об’єкти-схеми
    c) надає криптографічні функції, які включають набір певних оболонок
    d) забезпечує синтаксичний аналіз спеціальних файлів файлів

19. Результат дії модуля Node.js mongoose
    a) проміжні програми заповняють властивість req.body
    b) взаємодія через спеціальні об’єкти-схеми
    c) надає криптографічні функції, які включають набір певних оболонок
    d) забезпечує синтаксичний аналіз спеціальних файлів файлів

20. Результат дії модуля Node.js crypto

a) проміжні програми заповняють властивість req.body
b) взаємодія через спеціальні об’єкти-схеми
c) надає криптографічні функції, які включають набір певних оболонок
d) забезпечує синтаксичний аналіз спеціальних файлів файлів

21. Результат дії модуля Node.js cookie-parser

a) проміжні програми заповняють властивість req.body
b) взаємодія через спеціальні об’єкти-схеми
c) надає криптографічні функції, які включають набір певних оболонок
d) забезпечує синтаксичний аналіз спеціальних файлів

22. Текстове наповнення з графічними та по можливості медіа елементами- це

a) контент
b) токен
c) кукі
d) хеш

23. Цифровий сертифікат, що гарантує певні права, зобов'язання тощо - це

a) контент
b) токен
c) кукі
d) хеш

24. Невеликі рядки даних, які зберігаються безпосередньо у браузері- це

a) контент
b) токен
c) кукі
d) хеш

25. Рядок бітів фіксованої довжини, який процедурно згенерований з деякого довільного блоку вихідних даних - це

a) контент
b) токен
c) кукі
d) хеш

26. Що не відноситься до компонентів діаграми DFD?

a) зовнішні сутності
b) процеси
c) сховища даних
d) прецеденти

27. Що не відноситься до компонентів діаграми DFD?
    a) зовнішні сутності
    b) сценарії
    c) сховища даних
    d) потоки даних
28. У структурі запиту REST API маршрут відправлення - це
    a) точка прийому запиту
    b) один з HTTP- методів
    c) метод
    d) ресурс
29. У структурі запиту REST API тип запиту - це
    a) точка прийому
    b) один з HTTP- методів
    c) метод
    d) ресурс
30. У структурі запиту REST API заголовок - це
    a) точка прийому
    b) один з HTTP- методів
    c) метаінформація про запит
    d) ресурс

2.7. ЗАВДАННЯ ДО САМОСТІЙНОЇ РОБОТИ

1. Ознайомитись із структурою та принципом взаємодії патерна MVC.
2. Ознайомитись як у Express.js для взаємодії контролера-моделі-представлення використовуються стандартні функції JavaScript.
3. Ознайомитись з прийомами роботи з шаблонізатором для представлень Handlebars.
4. Ознайомитись з прийомами роботи з API REST, що реалізуються через HTTP-методи PUT, POST, GET, PATCH, DELETE.
5. Ознайомитись з опціями bodyParser.json.
6. Ознайомитись з Хеш-функціями, що перетворюють вхідні дані будь-якого розміру в дані фіксованого розміру.
7. Ознайомитись з механізмами авторизації, що засновані на файлах cookie і токенах.
8. Ознайомитись з механізмом архітектури контенту, завданнями та правилами структурування контенту.
9. Ознайомитись базовими вимогами щодо підготовки SEO-тексту.
10. Ознайомитись з методами побудови REST API.

ВИКОРИСТАНІ ДЖЕРЕЛА

1. Програма Model-View-Controller MVC - що це, особливості і опис. URL: https://hi-news.pp.ua/kompyuteri/14628-programa-model-view-controller-mvc-scho-ce-osoblivost-opis.html (дата звернення: 07.04.2023).
2. NodeJs. The Node.js path module. URL: https://nodejs.dev/learn/the-nodejs-path-module (дата звернення: 07.04.2023).
3. METANIT.COM Сайт про програмування. Визначення схеми у Mongoose. URL: https://metanit.com/web/nodejs/6.7.php (дата звернення: 07.04.2023).
4. Express. Fast, unopinionated, minimalist web framework for Node.js. Body-parser. URL: https://expressjs.com/en/resources/middleware/body-parser.html (дата звернення: 07.04.2023).
5. AWS. Что такое RESTful API? https://aws.amazon.com/ru/what-is/restful-api/ (дата звернення: 07.04.2023).
6. MaxSite.org. REST, RESTFul и CRUD. URL: https://maxsite.org/page/restful-crud (дата звернення: 07.04.2023).
7. Java T Point. Node.js Crypto. URL: https://www.javatpoint.com/nodejs-crypto
8. Веб-аутентификация: файлы cookies или токены? URL: https://proglib.io/p/veb-autentifikaciya-fayly-cookies-ili-tokeny-2021-08-14 (дата звернення: 07.04.2023).
9. Javascript.info. Cookies, document.cookie. URL: https://javascript.info/cookie (дата звернення: 07.04.2023).
10. PROGLIB. JWT простою мовою: що таке JSON токени і навіщо вони потрібні. URL: https://proglib.io/p/json-tokens (дата звернення: 07.04.2023).
11. Janith Kasun (Author) Handling Authentication in Express.js. StackAbuse. URL: https://stackabuse.com/handling-authentication-in-express-js/ (дата звернення: 07.04.2023).
12. MDN WEB DOCS. Express Tutorial Part 2: Creating a skeleton website. URL: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/skeleton_website (дата звернення: 07.04.2023).
13. CMDIGITAL. Что такое архитектура контента? URL: https://cmdigital.agency/arhitektura-kontenta/ (дата звернення: 07.04.2023).
14. OdesSeo. Написання SEO-тексту для сайту – основні рекомендації. URL: https://odesseo.com.ua/napisanie-seo-teksta/
15. FORTRESS-DESIGN. Оценка «тошноты» страницы: онлайн-сервисы. URL: https://fortress-design.com/ocenka-toshnoty-stranicy-onlajn-servisy/(дата звернення: 07.04.2023).
16. LUCIDCHART. What are your DFD needs? https://www.lucidchart.com/pages/data-flow-diagram#:~:text=A%20data%20flow%20diagram%20(DFD,for%20any%20process%20or%20system (дата звернення: 07.04.2023).

17. Guides on technologies, tools, and practices. What is REST API. URL: https://guides.hexlet.io/rest-api/ (дата звернення: 07.04.2023).
18. OTUS JOURNAL. Архитектура REST и API. URL: https://otus.ru/journal/arhitektura-rest-i-api (дата звернення: 07.04.2023).
19. Zell Liew. Understanding And Using REST APIs. URL: https://www.smashingmagazine.com/2018/01/understanding-using-rest-api (дата звернення: 07.04.2023). (дата звернення: 07.04.2023).
20. Введение в curl и его установка. URL: https://starkovden.github.io/curl-intro-and-instalation.html (дата звернення: 07.04.2023).

Розділ 3. FRONT-END ПРОЕКТУВАННЯ WEB-ДОДАТКУ З ВТКОРИСТАННЯМ БІБЛІОТЕКИ REACT.JS

3.1. ОСНОВИ КОМПОНЕНТНОГО ПІДХОДУ БІБЛІОТЕКИ REACT.JS

3.*1.1. Теоретичний модуль*

Front-end – являю собою ту частину web-додатку, яка є публічною та завжди знаходиться на стороні користувачів, надаючи їм потрібну інформацію та сервісну підтримку. Основними завданнями front-end є: відображення контент-інформації; надання функціоналу, що подається розробниками web-додатку на стороні клієнта; створення інтерфейсу для запитів користувачів на отримання потрібних послуг, що опрацьовуються за цими запитами на стороні back-end. Іншими словами, front-end – це все те, що є на web-сторінках web-додатку та те з чим користувач може взаємодіяти. Таким чином, web-додаток складається з двох частин – клієнт-серверного додатку, де клієнтом виступає браузер на стороні користувача; серверного додатку, коли web-сервер обслуговує запити користувача та реалізує інтерфейс сервер-браузер.

Front-end розробка є дуже відповідальною частиною проектування web-додатку. Від дизайну публічних web-сторінок додатку залежить не тільки привабливість програмного продукту в цілому, а й зручність функціоналу з яким безпосередньо контактує користувач та який зазвичай виконується на стороні клієнта. Тобто, front-end розробник є відповідальним за те, що розміщено на кожній web-сторінці: як розміщені інформаційний текст, керуючі гіперпосилання (меню, кнопки, діалогові вікна тощо), за те керуючи елементи web-додатку виконували свої функціональні дії. Простіше кажучи, front-end - це те, що бачить користувач, які дії він може виконати після відвідування web-сторінки, який функціонал надає web-додаток коли він взаємодіє з мережею Internet.

Front-end-розробник на початку створення своєї частини web-додатку повинен визначити таке: який дизайн матиме публічна частина web-додатку; які мови програмування будуть застосовані; які данні будуть відображатись, а які будуть передаватись на серверну частину; яким чином буде побудований інтерфейс взаємодії між браузером та сервером тощо.

Основними компетенціями якими повинен володіти front-end-розробник web-додатку на основі стеку MERN є знання та розуміння з:

- HTML (HyperText Markup Language) – мова розмітки всіх елементів і документів на web-сторінці;

- CSS (Cascading Style Sheets) – мова стилізації зовнішнього вигляду web-сторінки. CSS визначає шрифти, кольори, розташування блоків сайту тощо;

- JavaScript – мова програмування, створена для забезпечення інтерактивності web-сторінки на стороні користувача, опрацьовує запити дій користувача (обробляє переміщення курсора, натискання клавіш, кліки мишкою тощо);

- бібліотеки React.js (далі за текстом React), яка на стороні сервера створює можливості верстки web-сторінок мовою шаблонів HTML-in-JavaScript під назвою JSX (JavaScript XML) як розширення синтаксису мови програмування JavaScript; опрацьовувати документ web-додатку з використанням технології віртуальної DOM; групувати дезайнерських та функціональних елементів web-додатку як компонентів (components); використовувати бібліотеку стилів React-компонентів React-Bootstrap; опрацьовувати стани компонентів Props та State; керувати станом даних і інтерфейсом користувача в web-додатку використовуючи менеджер станів Redux; функції middlewares для послідовного оновлення даних у сховищі; здійснювати маршрутизацію потоків даних (data flow) з використанням модулю React Router; підтримувати стан web-додатку, використовуючи структури даних модулю immutable.js; тестування React-додатків за допомогою Jest і Enzyme.

У цьому розділі основний акцент поставимо на можливостях бібліотеки React та тих можливостях, які вона надає для front-end-розробника.

React – це JavaScript-бібліотека для роботи з користувацьким інтерфейсом (User Interface - UI), яку створили розробники соціальної мережі Facebook в 2011 році. Відкритий вихідний код React широко почали використовувати з 2013 року. За допомогою React розробники створюють реактивні web-додатки, що змінюють відображення без перезавантаження web-сторінки вцілому, а лише тої частини яка реагує на дії користувача. Завдяки цьому web-додаток значно пришвидшує свою роботу.

Ключові особливості React

Одна з ключових особливостей React - **універсальність**. Бібліотеку можна використовувати на сервері та на мобільних платформах за допомогою React Native. Це принцип Learn Once, Write Anywhere або «Навчіться один раз, пишіть де завгодно».

React заснований на **компонентах**. Кожен компонент повертає частину призначеного для користувача інтерфейсу зі своїм **станом**. Об'єднуючи компоненти, програміст створює завершений інтерфейс web-додатку.

Важлива особливість бібліотеки - **декларативність**. За допомогою React розробник описує, як компоненти інтерфейсу виглядають в різних станах. Декларативний підхід скорочує код і робить його зрозумілим.

Базовий принцип React – це застосування компонентів для створення користувацького інтерфейсу. Також бібліотека може повністю управляти front-end. В цьому випадку React застосовує бібліотеки для управління станом і роутінгом, наприклад, Redux та React Router.

Компоненти React – це самостійні, незалежні частини коду, що можуть описувати будь-яку частину інтерфейсу користувача на front-end. React дозволяє створювати компоненти як функції чи класи [1].

Найпростіше оголосити React-компонент як функцію:

```
function Welcome (props) {
        return <h1>Привіт, {props.name}</h1>;
    }
```

Ця функція є компонентом, тому що вона отримує дані в одному об'єкті (**props** - «пропси») як параметр і повертає React-елемент.

Або створимо компонент у вигляді класу:

```
class Welcome extends React.Component {
        render() {
          return <h1>Привіт, {this.props.name}</h1>;
              }
        }
```

Метод **render** () є обов'язковим методом у класі компоненту. Функція render() має бути чистою, тобто вона не змінює стан компонента, повертає той самий результат кожного разу, коли її викликають, і не взаємодіє безпосередньо з браузером.

Під час виклику він має перевірити this.props та this.state повернути один із таких типів:

- елемент, що реагує. Створюється за допомогою JSX. Наприклад, <MyComponent /> буде елементом React, який відображатиметься в React як вузол DOM;

- масиви та фрагменти. Це шаблони React, які дозволяють компоненту повертати декілька елементів;

- портали. Надають спосіб відтворення дочірніх елементів у вузлі DOM, який існує поза ієрархією DOM батьківського компонента. Тобто, дає можливість відображати дочірні елементи в іншому піддереві DOM;

- рядок і числа, які відображаються як текстові вузли в DOM;

- логічне значення або nullaboundefined, яке вказує, що непотрібно нічого рендерити. Наприклад, у шаблоні return test && <Child /> test буде логічне значення.

Створивши компоненти створюємо код we, додатку як компонент App можна відрендити попередньо створений компонент Welcome (функція чи клас) один раз або декілька разів:

```
function App() {
        return (
        <div>
            <Welcome name="Аліса" />
            <Welcome name="Базиліо" />
            <Welcome name="Піноккіо" />
      </div>
       );
}
```

Мова шаблонів JSX

React використовує JSX - це розширення синтаксису JavaScript, що робить опис інтерфейсу що зручніше. **JSX** схожий на HTML, однак це є JavaScript.

Розберемо деталі написаного вище коду. Елементи та компоненти React створює за допомогою мови шаблонів JSX. Мова JSX являє собою поглиблений синтаксис JavaScript, який зовні має схожість з HTML та нагадує XML, однак це є JavaScript.

Наприклад, код на JSX [2]:

```
const header = text ? <h1>{text}</h1> : null;
const vdom = (
  <div>
      {header}
        <Hello />     // компонент
  </div>
);
```

JSX дозволяє писати JavaScript-код за допомогою готових компонентів, які практично повністю повторюють HTML. Це спрощує розробку.

Наприклад, використовуючи JSX **оголосимо змінну**:

```
const element = <h1>Привіт, світ!</h1>;
```

Синтаксис цього дивного коду, що вміщує тег <h1> не є ні рядком, ні HTML. Він називається JSX . JSX створює «елементи» React.

Вбудовування виразів в JSX. Наведемо нижче приклад, у якому оголошується змінна з ім'ям **name**, а потім ця змінна використовується всередині коду JSX обернена в фігурні дужки:

```
const name = 'Юрій Тулашвілі';
const element = <h1>Привіт, {name}</h1>;
ReactDOM.render(
      element,
            document.getElementById ('root')
       );
```

У фігурних дужках JSX ви можете помістити будь який вираз JavaScript. Наприклад, 2 + 2 (для обчислення), user.firstName (для виведення даних з JSON-об’єкту) тощо.

Наступний приклад вставляє результат виклику функції JavaScript, formatName(user) до елементу <h1>:

```
function formatName (user) {
    return user.firstName + ' ' + user.lastName;
                                }
const user = {                              //   створюємо об’єкт
    firstName: ' Юрій ',
    lastName: 'Тулашвілі'
                };
const element = (
  <h1>   Привіт, {formatName (user)}!  </h1>
);
ReactDOM.render(
  element,
  document.getElementById ('root')
);
```

Рекомендується, як показано у прикладі вище, обгортати код JSX в круглі дужки, щоб уникнути помилок, під час автоматичного вставлення редактором коду крапки з комою.

Після компіляції **вирази JSX** стають звичайними викликами функцій JavaScript і обчислюються в об'єкти JavaScript. Тому, можемо використовувати JSX всередині операторів if та for, привласнювати його змінним, приймати його в якості аргументів, а також повертати з функцій. Наприклад:

```
function getGreeting (user) {
  if (user) {
      return <h1>Привіт, {formatName (user)} ! </h1>;
        }
      return <h1>Привіт, незнайомий користувач.</h1>;
}
```

Визначення атрибутів тегів за допомогою JSX. Для встановлення строкових літералів в якості атрибутів використовуються лапки. Наприклад:

```
const element = <div tabIndex = "0" ></div>;
```

Також можна використовувати фігурні дужки для вставки JavaScript-виразу як атрибуту:

```
const element = <img src={user.avatarUrl}> </img>;
```

Вбудовувати призначені для користувача дані в JSX **є безпечним**:

```
const title = response.potentiallyMaliciousInput;
const element = <h1>{title}</h1>;
```

За замовчуванням DOM React екранує будь-які значення, що вбудовані до JSX, перед їх рендерингом (промальовуванням). Таким чином, є гарантованим, що у web-додатку ніколи не відобразиться те, чого явно немає в коді. Перед рендерингом все перетворюється в рядок. Це допомагає запобігти атакам міжсайтовогу скриптингу (cross-site-scripting, XSS).

Розглянемо яким чином JSX опрацьовує об'єкти. Компілятор Babel перетворює код JSX у виклики React.createElement().

Наприклад, два ідентичних коди [2]:

```
const element = (
  <h1 className="greeting">
    Привіт, світ!
  </h1>
);
```

та

```
const element = React.createElement (
  'h1',
      {className: 'greeting'},
          'Привіт, світ!'
  );
```

React.createElement() у процесі перетворення коду виконає перевірку на помилки та створить такий об'єкт:

```
// Спрощена структура
const element = {
      type: 'h1',
          props: {
            className: 'greeting',
                children: 'Привіт, світ!'
            }
    };
```

Ці об'єкти **називаються елементами** React. React зчитує ці об'єкти і використовує їх для побудови DOM та його поновлення.

Віртуальна DOM

Використання віртуальної DOM (Virtual DOM) є важливою особливістю React. Віртуальна DOM - об'єкт, в якому зберігається інформація про стан інтерфейсу. При зміні стану, наприклад, після відправлення даних форми або натискання кнопки, React розраховує різницю між старим і новим станом. Після цього бібліотека обробляє рендерингує (промальовує) новий стан компонентів, що піддавались змінам. Використання віртуальної DOM дозволяє бібліотеці ефективно оновлювати реальну DOM. Процес синхронізації віртуальної DOM з реальною DOM за допомогою бібліотеки, такої як ReactDOM, називається узгодженням [3].

Життєвий цикл React-компонента

Будь-який компонент визначається як React-елемент віртуальної DOM. Рендеринг (перемальовування) компонентів може виникати з багатьох причин. В залежності від причини викликаються різні функції, що дозволяють розробнику виконувати оновлення певних частин компонента.

На рисунку 3.1. подана діаграма життєвого циклу компонента, яка складається з таких етапів: монтування, оновлення та демонтування.

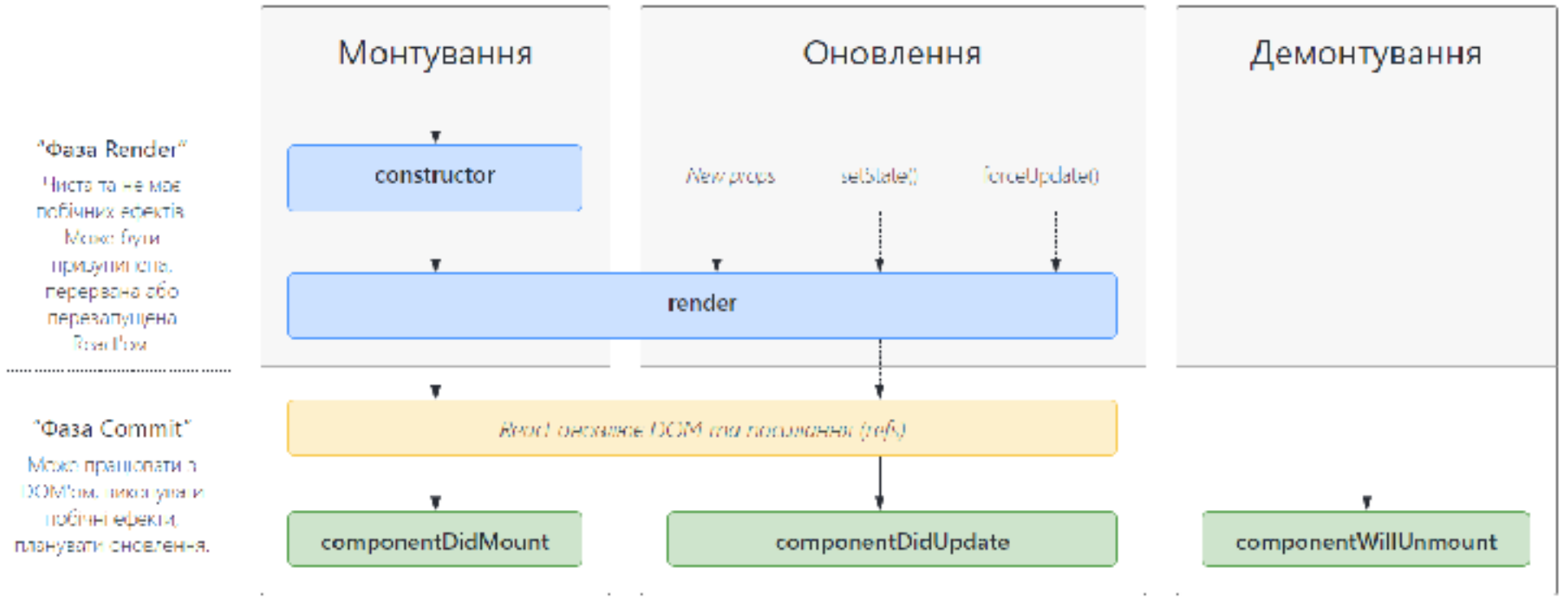


Рис. 3.1. Життєвий цикл React-компонента [4]

Монтування (створення компонента)

Перший цикл - це створення компонента, що, зазвичай, відбувається при першому виявленні компоненту в розпарсеному JSX дереві. При створенні екземпляра компонента та його вставці в DOM методи React викликаються у такому порядку [5]:

1. constructor()

2. static getDerivedStateFromProps()

3. render()

4. componentDidMount()

Оновлення. Оновлення відбувається за зміни пропсів або стану (наприклад виклик this.setState()). Тоді викликаються при повторному рендері компонента такі методи [5]:

1. static getDerivedStateFromProps()

2. shouldComponentUpdate()

3. render()

4. getSnapshotBeforeUpdate()

5. componentDidUpdate()

Демонтування. При видаленні компонента з DOM викликається метод componentWillUnmount()

Якщо сталася помилка в процесі рендерингу компоненту то викликаються такі методи життєвого циклу або конструктор будь-якого дочірнього компонента:

1. static getDerivedStateFromError()

2. componentDidCatch()

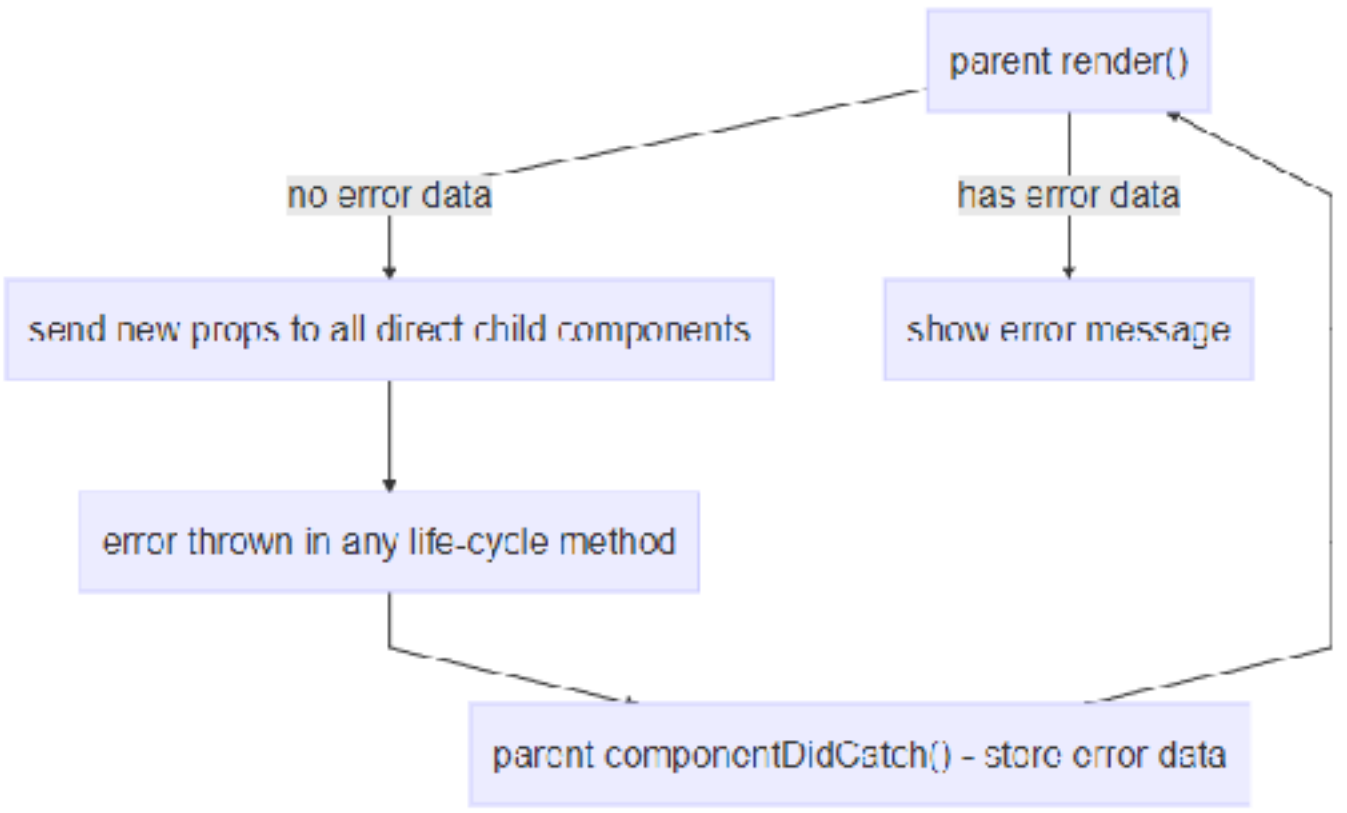


Рис. 3.2. Алгоритм перемальовування при перехопленні помилки [6]

Компонент може визначати спеціальний шар, який може перехоплювати помилки і надавати новий метод життєвого циклу - componentDidCatch - який дає розробнику можливість обробити або залогувати ці помилки.

Основи взаємодії компонентів у React

Front-end web-додаток на React після розгортання матиме структуру як подано на рис. 3.3. Розкриємо структуру проекту за важливістю її елементів:

- папка client вміщує проект front-end web-додатку;

- файл package.json має основне завдання - зберігати різні метадані, пов'язані з проектом. Файл використовується для надання інформації менеджеру пакетів вузлів (NPM), яка дозволяє ідентифікувати проект і його залежності;

- папка node_modules вміщує набір пакетів встановлених у React;

- папка public вміщує декілька файлів, головним серед яких є index.html – файл, який запускає web-додаток на React, що вкладається у тег з id="root":

```
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
 </body>
```

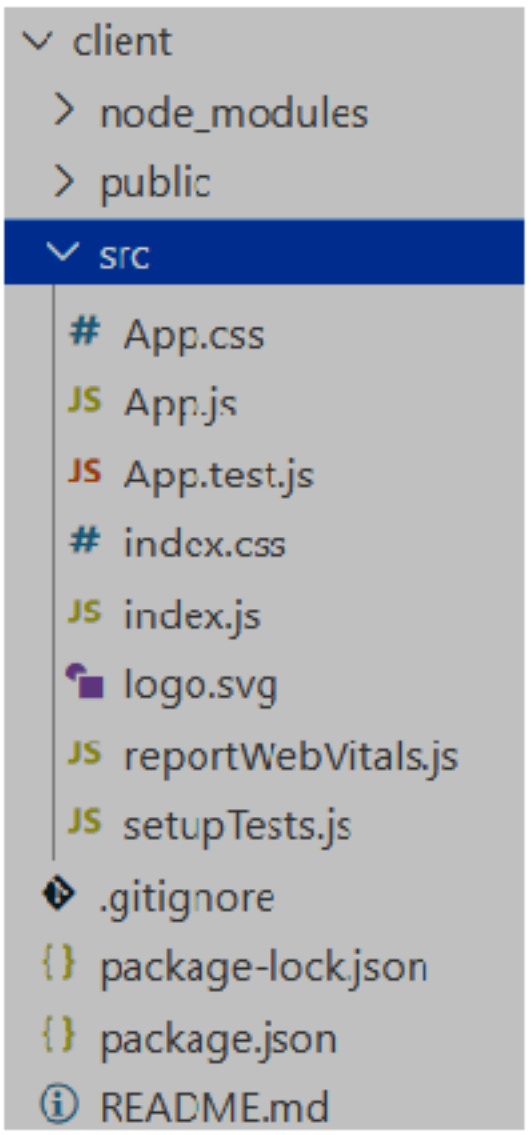


Рис. 3.3. Структура front-end web-додатку на React

- папка src вміщує основні ресурси проекту;

- ресурс index.js – запускається першим, виконавши імпортування ресурсів React, налаштування підтримки ReactDOM, підключивши файл стилів index.css та файл додатку App. Переглянемо код файлу index.js:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
const root = ReactDOM.createRoot(document.getElementById('root'));
    root.render(
      <React.StrictMode>
        <App />
      </React.StrictMode>
    );
    reportWebVitals();
```

Процес запуску web-додатку забезпечується викликом бібліотеки ReactDOM, вказуванням на тег з id="root" та подальшим рендерингом основного компонента App;

- ресурс App.js – основний компонент App побудований як функція чи клас;

- ресурс App.css – основний файл стилів css проекту.

3.1.*2. Практичний модуль*

У практичній частині продовжуємо проектну розробку web-додатку Tours City (екскурсії містом), а саме програмуємо front-end частину.

Етап 3.1. Побудова клієнтської частини та компонент головної сторінки

Дія 3.1.1. Будуємо front-end частину web-додатку. Для цього встановлюємо REACT з MS Code.

- знаходячись в корні проекту в терміналі вводимо:

```
npx create-react-app client
```

Чекаємо установку. Клієнтська частина встановиться до новоутвореної папки client.

Дія 3.1.2. Прописуємо завантаження back-end та front-end одночасно.

У файл package.json у корні проекту дописуємо скрипти для спільного завантаження back-end та front-end. Код повинен виглядати так:

```
"scripts": {
   "start": "node app.js",
```

```
    "server": "nodemon app.js",
    "client": "npm run start --prefix client",
    "dev": "concurrently \"npm run server\" \"npm run client\""
  },
```

Дія 3.1.3. Виконуємо попереднє тестування web-додатку на сумісність back-end та front-end.

- у консолі в корні проекту вводимо

  npm run dev

Чекаємо поки завантажиться back-end (створений у попередньому розділі)
У терміналі пропише App has been started on port 5000...
та та front-end – завантаження (автоматично) у браузер за localhost:3000.
Отримуємо стартову сторінку React в браузері (рис.3.4)

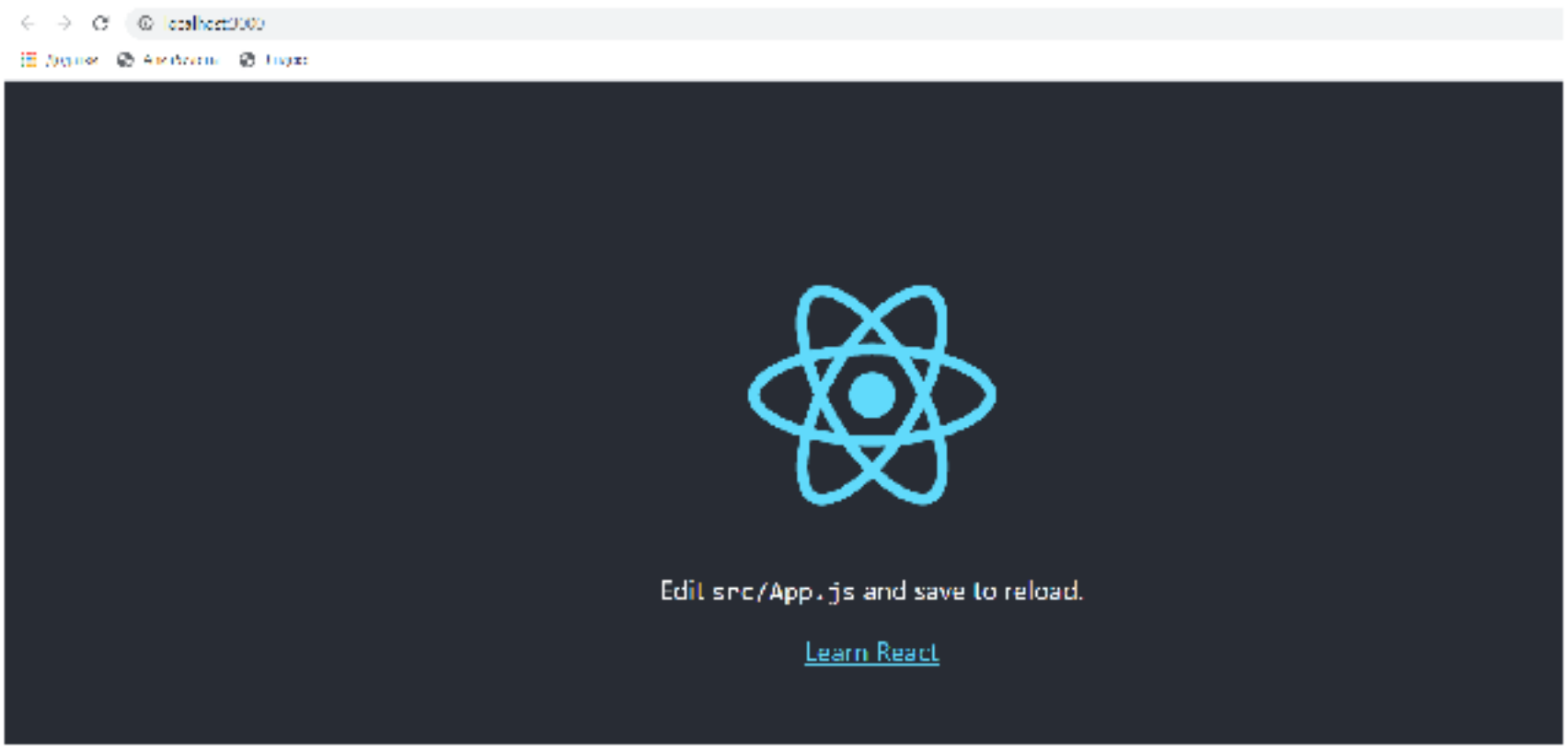


Рис. 3.4. Браузер за localhost:3000 відображає стартову сторінку React

Дія 3.1.4. Засобами MS Code створюємо в середині папки src папку для збереження файлів react компонент – components.

Дія 3.1.5. Відповідно до структури нашого шаблону в папці components засобами MS Code створюємо три файли, що формують цю структуру:

- шапка сторінки:

  файл Header.js

- центральна частини сторінки, що буде завантажуватись при старті:

  файл MainPage.js

- підвалини сторінки:

  файл Footer.js

Дія 3.1.6. Кодуємо кожен файл компонента подібно (у тезі <p> вписуємо текст «Text (назва компоненту)»). Для компонента Header код такий:

```
import React, { Component } from 'react'
        class Header extends Component {
          render() {
            return(
              <p>Text Header </p>
            );
          }
        }
        export default Header;
```

Аналогічно кодуємо два інших компоненти.

Дія 3.1.7. За умовою роботи React web-додатку в основному батьківському-компоненті сусідні елементи JSX (дочірні-компоненти) мають бути обгорнуті зовнішнім тегом, наприклад теги section, div.wrap, div.container, які використовуються для центрування контенту. У нашому випадку, створюємо фрагмент JSX як пустий тег <>...</>, в середині якого пропишемо всі три компоненти.

Файл app.js виглядатиме так:

```
import React, { Component } from 'react';
import './App.css';
import  Header  from './components/Header';
import  MainPage  from './components/MainPage';
import  Footer  from './components/Footer';
      class App extends Component {
        render() {
          return (
            <Header/>
            <MainPage/>
            <Footer/>
          );  }
      }
      export default App;
```

код JSX виглядає в index.js, app.js та у створених компонентів подібно. У початкових рядках завжди потрібно імпортувати залежності. Файл index.js як завантажувальний імпортує основні залежності React, файл проекту App та потрібні пакети CSS. Файл app.js як компонент вищого рівня утворює ієрархію компонент. Після цього можна використовувати класи початкового завантаження всередині наших компонентів програми React.

Дія 3.1.9. Тестуємо виведення компонент.

- у консолі в корні проекту вводимо

```
npm run dev
```

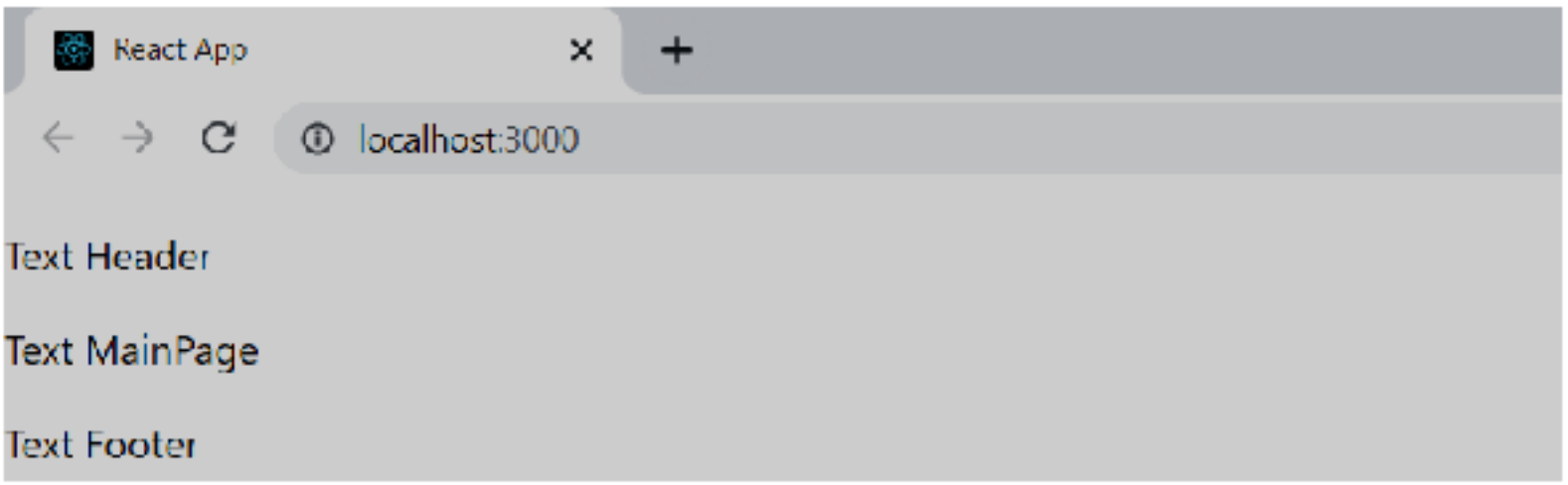


Рис. 3.5. Браузер за localhost:3000 відображає елементи компонент

- зупиняємо проект:
Ctrl + C.

3.2. СТВОРЕННЯ ДИЗАЙНУ UI МОЖЛИВОСТЯМИ REACT.JS

3.2.*1. Теоретичний модуль*

React значно спростив написання коду дизайну сторінки web-додатку за рахунок використання готових компонент бібліотеки. Створення web-додатку React на етапі front-end відображає переваги цієї технології. Ця бібліотека створювалася, щоб UI було написано просто, швидко, зручно. При цьому, універсальність React дозволяє виконувати програму одночасно на сервері та на клієнті. Використання компонентів для формування елементів дизайну UI створює можливості коли один і той же елемент інтерфейсу користувача може мати декілька різних реалізацій. На етапі front-end прописується вся візуальна композиція: іконки, меню, кнопки, анімація, інфографіка та інші візуальні елементи. Коли прототип готовий, web-додаток опрацьовується на клікабельність та створюється система маршрутизації, яка дозволяє зіставляти запити до web-додатку з певними компонентами.

Можливості побудови дизайну з Bootstrap

Для формування вигляду react-компонент, які відповідно до методології БЕМ (дивись розділ 1) являють собою блоки-елементи інтерфейсу web-додатку, широко застосовують популярний фреймворк CSS – Bootstrap. Це потужний, розширюваний і багатофункціональний інтерфейсний набір CSS та JavaScript файлів цікавий тим, що надає простий інструментарій для побудови дизайну та для створення адаптивного web-додатку. React має можливості приєднання інструментів Bootstrap, просто імпортувавши залежність початкового завантаження у файли React. Після імпорту CSS Bootstrap стають доступними

усі його інструменти: колонкова система (сітка Bootstrap), класи та компоненти як залежності, які можна використовувати в компонентах програми React.

Bootstrap дає можливості використовувати:

- інструменти для створення макетів (обгорткових контейнерів, потужної системи сіток, гнучких медіа-об'єктів, адаптивних утилітних класів);

- класи для стилізації контенту: тексту, зображень, таблиць тощо;

- готові компоненти: кнопки, форми, навігаційні панелі, слайдери, списки, що випадають, підказки тощо;

- адаптивну сітку, що базується на Flex-моделі, дозволяє змінювати розміри в залежності від наповнення контейнерів контентом залежно від поточного розміру екрана користувача.

Для встановлення Bootstrap до web-додатку React потрібно виконати його встановлення:

```
npm install bootstrap
```

Після чого прописати глобальний зв’язок у index.css, якій буде діяти у всьому web-додатку:

```
@import "~bootstrap/dist/css/bootstrap.css";
```

Модуль react-bootstrap, також є найпопулярнішим способом додавання максимальних можливостей Bootstrap під час початкового завантаження React web-додатку, що створює можливості перебудови компонентів Bootstrap як компонентів React.

react-bootstrap - це повна реалізація компонентів Bootstrap як компонентів React. Цей модуль потрібен для розширеного позиціонування вмісту, наприклад, спадаючих меню, спливаючих вікон та підказок, формування списків, групування форм тощо. Для додавання Bootstrap до React [7] потрібне встановлення пакета React Bootstrap - react-bootstrap:

```
npm install react-bootstrap
```

React-bootstrap можна використовувати безпосередньо для елементів і компонентів у вашому додатку React, застосовуючи вбудовані класи, як будь-який інший клас. Після встановлення потрібно імпортувати будь-який компонент пакету. Наприклад, імпортування компонентів виглядатиме так:

```
import {Nav, NavDropdown, Item, Form, Button, ListGroup} from 'react-bootstrap';
```

Компоненти для побудови меню та його функціонування

Для будь-якого web-додатку важлива проста в створенні та у використанні навігація. Меню являє собою панель навігації на якій розміщується перелік посилань, що реалізують бізнес-логіку web-додатка.

В залежності від цільового призначення меню може бути: адаптивним, розміщене на бічній панелі, спадним фіксованим, гамбургер тощо. Для кодування навігаційного меню найчастіше використовується Bootstrap -

безкоштовна інтерфейсна платформа розробки з відкритим кодом для створення web-додатків

Для навігації у React є своя система маршрутизації, що дозволяє зіставляти запити до застосування з певними компонентами. Для створення навігаційного меню front-end використовуємо модуль react-router-dom, що поряд з модулум react-router є ключовою ланкою в роботі маршрутизації та містить основний функціонал роботи з маршрутизацією.

Пакет react-router-dom містить прив'язки для використання React Router у web-додатках [8].

Меню є кореневим маршрутом, оскільки решта наших маршрутів відображатимуться всередині нього. Воно слугуватиме кореневим макетом інтерфейсу користувача.

Після імпорту react-bootstrap стає можливим використання його компонентів для навігаційного меню. Для створення навігаційного меню, що вміщує посилання, застосовується об'єкт Link, який визначений в модулі react-router-dom для визначення кожного посилання. Для обгортання посилань у навігаційний блок застосовується компонент Nav чи Navbar. Окрім компонента Link з цього модуля для створення посилань можна використовувати компонент NavLink. Цей компонент є розширеним аналогом Link та дозволяє використовувати стан посилання.

Компоненти Nav та Navbar інкапсулюють (включають) такі дочірні компоненти [9]:

Brand - для брендингу;

```
<Navbar.Brand href="/index.html">
    <img src={logo} className= “d-inline-block align-top” />
</Navbar.Brand>
```

Item, Link - для записів навігації;

```
<Nav.Item>
    <Nav.Link href="/home">Active</Nav.Link>
</Nav.Item>
```

Toggler - для використання з JS-плагіном згортання у формі кнопки розгортання меню;

Collapse - для групування та приховування вмісту меню утвореного Navbar.

Так, використовуючи дочірні компоненти для кодування адаптивного меню, можна використовувати такий шаблон:

```
<Navbar.Toggle aria-controls="basic-navbar-nav" />
 <Navbar.Collapse id="basic-navbar-nav">
  <Nav className="me-auto">
      <Nav.Link href="/index.html">Home</Nav.Link>
        <span className="selectText">Оберіть початок маршруту:</span>
```

```
        <NavDropdown title="За номером туру" id="basic-nav-dropdown">
            <NavDropdown.Item href="1">Тур 1 </NavDropdown.Item>
            <NavDropdown.Item href="2">Тур 2 </NavDropdown.Item>
            <NavDropdown.Item href="3">Тур 3 </NavDropdown.Item>
        </NavDropdown>
    </Nav>
  </Navbar.Collapse>
```

Шаблон реалізує навігаційну панель з найпоширенішими елементами, такими як бренд, посилання і спадне меню.

Панелі навігацій Navbar потребують обгортання для адаптивного згортання, коли їх вміст «схлопується» після натискання кнопки. Стеки навігаційної панелі:

xl | lg | md | sm - для розмірів екрана дуже великі, великі, середні та малі. Bootstrap css клас для центрування навігаційної панелі .justify-content-center.

Для фіксації панелі навігацій Navbar вгорі web-сторінки застосовується атрибут:

fixed='top'

fixed означає, що вона вилучається зі звичайного потоку DOM і можуть потребувати спеціального CSS щоб запобігти накладанню з іншими елементами.

Щоб створити web-додаток MPA з кількома маршрутами сторінок потрібно застосовувати React Router для маршрутизації сторінок на основі URL. Наприклад, у файлі index.js прописати “роутери” [10]:

```
import ReactDOM from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Layout from "./pages/Layout";
import Home from "./pages/Home";
import Blogs from "./pages/Blogs";
import Contact from "./pages/Contact";
import NoPage from "./pages/NoPage";
export default function App() {
    return (
      <BrowserRouter>
        <Routes>
          <Route path="/" element={<Layout />}>
            <Route index element={<Home />} />
            <Route path="blogs" element={<Blogs />} />
            <Route path="contact" element={<Contact />} />
            <Route path="*" element={<NoPage />} />
          </Route>
```

```
        </Routes>
      </BrowserRouter>
    );
  }
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Як бачимо виклик відповідного URL призведе до відкриття відповідної web-сторінки у вигляді компоненту. Це кодується в такий послідовності:

1. Спочатку загортаємо контент-вміст у <BrowserRouter>;

2. Потім визначаємо <Routes>. Програма може мати кілька <Routes>. У прикладі використовується лише один;

3. <Route> є вкладеними у <Routes>. Перший <Route>має шлях "/" і відображає Layout компонент. Вкладені <Route> успадковують і додають до батьківського маршруту. Отже, маршрут Home компонента не має шляху, але має index атрибут. Це визначає цей маршрут як маршрут за замовчуванням для батьківського маршруту, яким є "/". Для маршруту blogs шлях поєднується з батьківським і стає "/blogs". А встановлення path значення "*" буде діяти як універсальне для всіх невизначених URL-адрес, тобто для сторінки помилки 404.

3.2.2. Практичний модуль

У практичній частині продовжуємо проектну розробку web-додатку Tours City (екскурсії містом), а саме будуємо дизайн головної сторінки front-end.

Етап 3.2. Побудова дизайну головної сторінки клієнтської частини

Дія 3.2.1. Встановлюємо потрібні модулі для побудови інтерфейсу користувача.

- переходимо до папки client:

```
cd client
```

- у папці client вводимо у терміналі:

```
npm install bootstrap
npm install react-bootstrap
npm install react-router
npm install react-router-dom
```

Дія 3.2.2. У файл клієнтської частини index.css прописуємо імпорт стилів

```
@import "~bootstrap/dist/css/bootstrap.css";
```

Дія 3.2.3. Головна web-сторінка UI згідно обраного шаблону web-додатку (дивись пункт 1.1. розділу 1) складається з трьох частин:

- шапки з керуючим елементами (меню, випадаючи списки, пошуковий рядок) – компонент Header;

- нижньої підвальної частини, що надає додаткову інформацію з гіперпосиланнями – компонент Footer;
- центральної інформаційно-контентної частини, яка буде в подальшому змінною, а підчас першого запуску відображатиме стартову інформацію - компонент MainPage.

Для оформлення дизайну головної web-сторінки до файлу App.css записуємо відповідні стилі цих частин (дивись лістинги 1.2 - 1.5).

Дія 3.2.4. Виконуємо кодування компонента Header. Він реалізує адаптивне меню web-додатку.

Лістинг 3.1 : Код компонента Header.js

```
import React, { Component } from 'react';
import reactRouterDom, {NavLink, useHistory} from 'react-router-dom';
import {Navbar, Nav, Container, NavDropdown } from 'react-bootstrap';
import { Link } from 'react-router-dom';
    class Header extends Component {
        render() {
            return (
<Navbar className="bg-warning bg-gradient  text-dark " fixed='top'  expand="lg">
      <Container>
        <Navbar.Brand href="/">
        <div className="header-home_logo">
          <a href="/">
            <span className="header-home_logo_second">City </span>
            <span className="header-home_logo_main">Tours</span>       </a>
        </div>
        </Navbar.Brand>
        <Navbar.Toggle aria-controls="basic-navbar-nav" />
        <Navbar.Collapse id="basic-navbar-nav">
  <Nav className="me-auto">
<Nav.Link href="/index.html">Home</Nav.Link>
          <span className="selectText">Оберіть початок маршруту:</span>
<NavDropdown title="За номером туру" id="basic-nav-dropdown">
  <NavDropdown.Item href="1">Typ 1 </NavDropdown.Item>
  <NavDropdown.Item href="2">Typ 2 </NavDropdown.Item>
  <NavDropdown.Item href="3">Typ 3 </NavDropdown.Item>
</NavDropdown>
//   Аналогічно додаємо ще спадаючі меню
    </Nav>
          </Navbar.Collapse>
```

Продовження лістингу 3.1

```
      </Container>
    </Navbar>
     );
  }
}
  export default Header;
```

кінець лістингу 3.1

Дія 3.2.5. Виконуємо кодування компонента MainPage. Він реалізує контент, що буде відображатись на стартовій сторінці.

Лістинг 3.2 : Код компонента MainPage.js

```
 import React, { Component } from 'react'
import Container from 'react-bootstrap/Container';
import Row from 'react-bootstrap/Row';
import Col from 'react-bootstrap/Col';
      class MainPage extends Component {
       render() {
            return (
   <div className="main" style={{ padding: "0 0 0 30px" }}>
    <div className="container">
     <div className="wrap">
      <div className="row">
       <h3 className="blog_head">Маршрути</h3>
       <div className="map">
        <iframe src="https://www.google.com/maps/embed"></iframe>
       </div>
       <h3 className="blog_head">Об'єкти маршруту</h3>
       <Container>
        <Row>
         <Col>       <h3 className="tour__title title-mod">
            Древній Луцьк (Лучеськ)    </h3>
```

Продовження лістингу 3.2

```
<p> Виник на острові, який омивався водами широкоплинного Стиру …/p>
<p>   <a  className="btn btn-default"      href="https://uk.wikipedia.org"    >
                Детальніше   </a>   </p>
            </Col>
            <Col>
              <h3 className="tour__title title-mod">Старе місто</h3>
<p> Це - колиска Луцька. Саме тут, на острові серед заплав Стиру …  </p>
 <p>   <a   className="btn btn-default"   href="https://uk.wikipedia.org"
        target="_blank"  > Детальніше   </a>  </p>
            </Col>
          </Row>
        </Container>
<h3 className="blog_head"> Додаткова інформація маршруту</h3>
      <section>
          <div className="row  text">
    <div> <h3 className="tour__title title-mod"> Древня історія Луцька </h3>
    <p>      У другій половині XII ст.    …     </p>
    <p>   <a   className="btn btn-default"    href="https://uk.wikipedia.org"
                target="_blank"    >   Детальніше   </a>    </p>
            </div>
          </div>
        </section>
      </div>
    </div>
   </div>
  );
}
}
  export default MainPage;
```

кінець лістингу 3.2

Дія 3.2.6. Виконуємо кодування компонента Footer.

Лістинг 3.3 : Код компонента Footer.js

```
import React, { Component } from 'react'
import Container from 'react-bootstrap/Container';
import Row from 'react-bootstrap/Row';
import Col from 'react-bootstrap/Col';
   class Footer extends Component {
       render() {
          return(
<div class="footer">
<Container>
   <Row>
 <Col>   <ul class="list1">
<h3>Navigation</h3>
<li><a href="#"> Google Maps</a></li>
<li><a href="#">Apply</a></li>
<li><a href="#">Register</a></li>
</ul>    </Col>
  <Col> <ul class="socials">
<li><a href="#"><i class="fa fb fa-facebook">
<svg xmlns="http://www.w3.org/2000/svg" width="24" height="24"
fill="currentColor" class="bi bi-facebook" viewBox="0 0 16 16">
<path d=" … "/> </svg>  </i></a></li> <li><a href="#"><i class="fa tw fa-twitter">
<svg xmlns="http://www.w3.org/2000/svg" width="24" height="24"
fill="currentColor" class="bi bi-twitter" viewBox="0 0 16 16">
<path d="… "/> </svg>   </i></a></li>
</ul>       </Col>
   </Row>
</Container>
</div>
    )   }
 }
 export default Footer;
```

кінець лістингу 3.3

Дія 3.1.7. В MS Code створюємо файл routes.js, що буде відповідати за маршрутизацію внутрішньої структури сторінок web-додатку на основі URL. Переходимо до файлу та пишемо в середині файлу код як на лістингу 3.4.

Лістинг 3.4 : Код файлу routes.js

```
import React from 'react'
import {Switch, Route, Redirect} from 'react-router-dom'
import MainPage from './components/MainPage';
export const routes = () => {
 return (
     <Switch>
        <Route path="/" exact>
            <MainPage />
        </Route>
            <Redirect to="/" />
     </Switch>
     )    }
```

кінець лістингу 3.4

Дія 3.1.8. Для коректного виведення контенту у web-додатку створюємо у проектному файлі App.js зв'язок зі створеним "роутером" (рядок коду import {routes} from './routes';) та корегуємо код як показано на лістингу 3.5:

Лістинг 3.5 : Код файлу App.js

```
import React, { Component } from 'react';
import {BrowserRouter as Router} from 'react-router-dom';
import {routes} from './routes';
import './App.css';
import  Header  from './components/Header';
import  Footer  from './components/Footer';
class App extends Component {
  render() {
    return (
  <Router>
```

Продовження лістингу 3.5

```
      <Header/>
      <div  className = 'container' >
                { routes (true) }
          </div>

      <Footer/>
      </Router>
         );
       }
      }
export default App;
```

кінець лістингу 3.5

. **Дія 3.2.9.** Тестуємо виведення компонент.

- у консолі в корні проекту вводимо

npm run dev

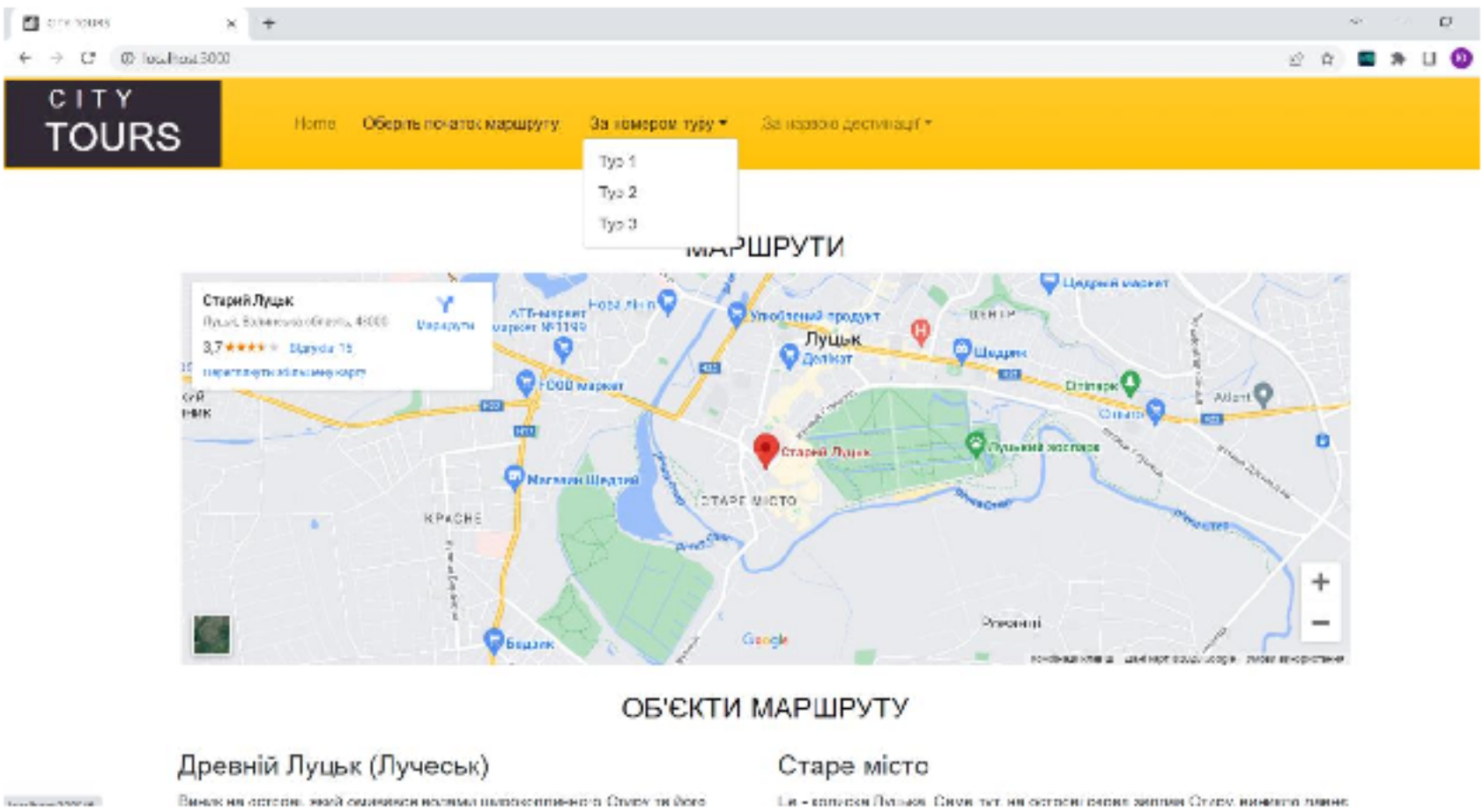


Рис. 3.6. Стартова сторінка проекту

3.3. ОТРИМАННЯ ДАНИХ В ПРОЕКТІ REACT ІЗ ЗОВНІШНЬОГО REST API

3.3.1. Теоретичний модуль

Клієнтська частина Front-end, що побудована на React основним завданням має отримання даних із зовнішнього API для відображення в середині контенту компоненту. Одним із способів отримати дані із зовнішнього API до програми React є використання менеджера станів Redux [11].

Бібліотека **Redux** – це засіб керування станом програми. Вона заснована на кількох концепціях, вивчивши які можна з легкістю вирішувати проблеми зі станом.

Redux ідеально використовувати в середніх та великих додатках [12]. У Redux загальний стан програми представлено одним об'єктом JavaScript - **state** (стан) або **state tree** (дерево станів). Незмінне дерево станів доступне лише для читання. Під час стану **state** змінювати нічого безпосередньо не можна. Зміни можливі лише при надсиланні **action** (дії). Дія (action) - це JavaScript-об'єкт, який лаконічно описує суть зміни. Єдина вимога до об'єкта дії - наявність властивості type, значенням якого зазвичай є рядок. У програмі тип дії задається рядком. Типи дій мають бути константами. По мірі розростання функціональності програми рекомендується використовувати константи:

```
const ADD_ITEM = 'ADD_ITEM'
const action = { type: ADD_ITEM, title: 'Third item' }
```

та виносити дії в окремі файли. А потім їх імпортувати:

```
import { ADD_ITEM, REMOVE_ITEM } from './actions'
```

Створюють дії спеціальні функції (генератори дій - actions creators):

```
function addItem(t) {
 return {
   type: ADD_ITEM,
   title: t
 }
}
```

Після генерації дії завжди виникає зміна стану програми. Обчислювання наступного стану дерева забезпечується на підставі його попереднього стану та дії функції, що отримала назву - редуктор (**reducer**).

```
(currentState, action) => newState
```

Функція працює незалежно від стану програми та видає вихідне значення, приймаючи вхідний і не змінюючи нічого в ньому та в іншій програмі. Редуктор повертає новий об'єкт дерева станів на який замінюється попередній.

Наприклад, є стан [12]:

```
{
 list: [
  { title: "First item" },
  { title: "Second item" },
 ],
 title: 'Grocieries list'
}
```

та список дій:

```
{ type: 'ADD_ITEM', title: 'Third item' }
{ type: 'REMOVE_ITEM', index: 1 }
{ type: 'CHANGE_LIST_TITLE', title: 'Road trip list' }
```

Тоді, редуктор для кожної частини стану виглядатиме таким:

```
const title = (state = '', action) => {
 if (action.type === 'CHANGE_LIST_TITLE') {
  return action.title
 } else {
  return state
 }
}
const list = (state = [], action) => {
 switch (action.type) {
  case 'ADD_ITEM':
   return state.concat([{ title: action.title }])
  case 'REMOVE_ITEM':
   return state.map((item, index) =>
    action.index === index
     ? { title: item.title }
     : item
   default:
    return state
 }
}
```

а редуктор для загального стану:

```
const listManager = (state = {}, action) => {
 return {
  title: title(state.title, action),
  list: list(state.list, action),
 }
}
```

Сховище (**store**) - це об'єкт, який:

- містить стан програми;
- відображає стан через getState();
- може оновлювати стан через dispatch();
- дозволяє реєструватися (або видалятися) як слухач зміни стану через subscribe().

Сховище у додатку завжди унікальне. Наприклад, так створюється сховище для listManager:

```
import { createStore } from 'redux'
import listManager from './reducers'
let store = createStore(listManager)
```

Потік даних у Redux завжди односпрямований. Передача дій з потоку даних відбувається через виклик методу dispatch() у сховищи. Саме сховище передає дії редуктору та генерує наступний стан, а потім оновлює стан та повідомляє про це всіх слухачів.

Сховище можна ініціювати через серверні дані:

```
let store = createStore(listManager, preexistingState)
```

Отримання стану:

```
store.getState()
```

Оновлення стану:

```
store.dispatch(addItem('Something'))
```

Прослуховування змін стану:

```
const unsubscribe = store.subscribe(() =>
  const newState = store.getState()
)
unsubscribe()
```

Усе вище визначене тісно пов’язане з компонентами, які відображають контент. Компонент відповідає за виклик API, оновлення стану через **Redu**x, а потім виконання будь-яких завдань, які компонент повинен робити в першу чергу. Модуль **react-redux** відповідає за зв’язок стану Redux з компонентами React.

Розглянемо послідовність за якою програма отримує данні та робить їх доступними для компонента (рис. 3.3) [13]:

1. Компонент надсилає дію action, яка формує запит fetch(‘data’) до API.
2. Коли ця дія отримає успішну відповідь від API, змінюється стан та надсилається дія loadSuccess.
3. Сховище пересилає оновлений стан store.dispatch () та дію loadSuccess редукторам.

4. Ред'юсер (reducer) відповідає створенням нової копії стану, яка включає всі оновлені данні.

5. Новий стан стає доступним для будь-якого компонента.

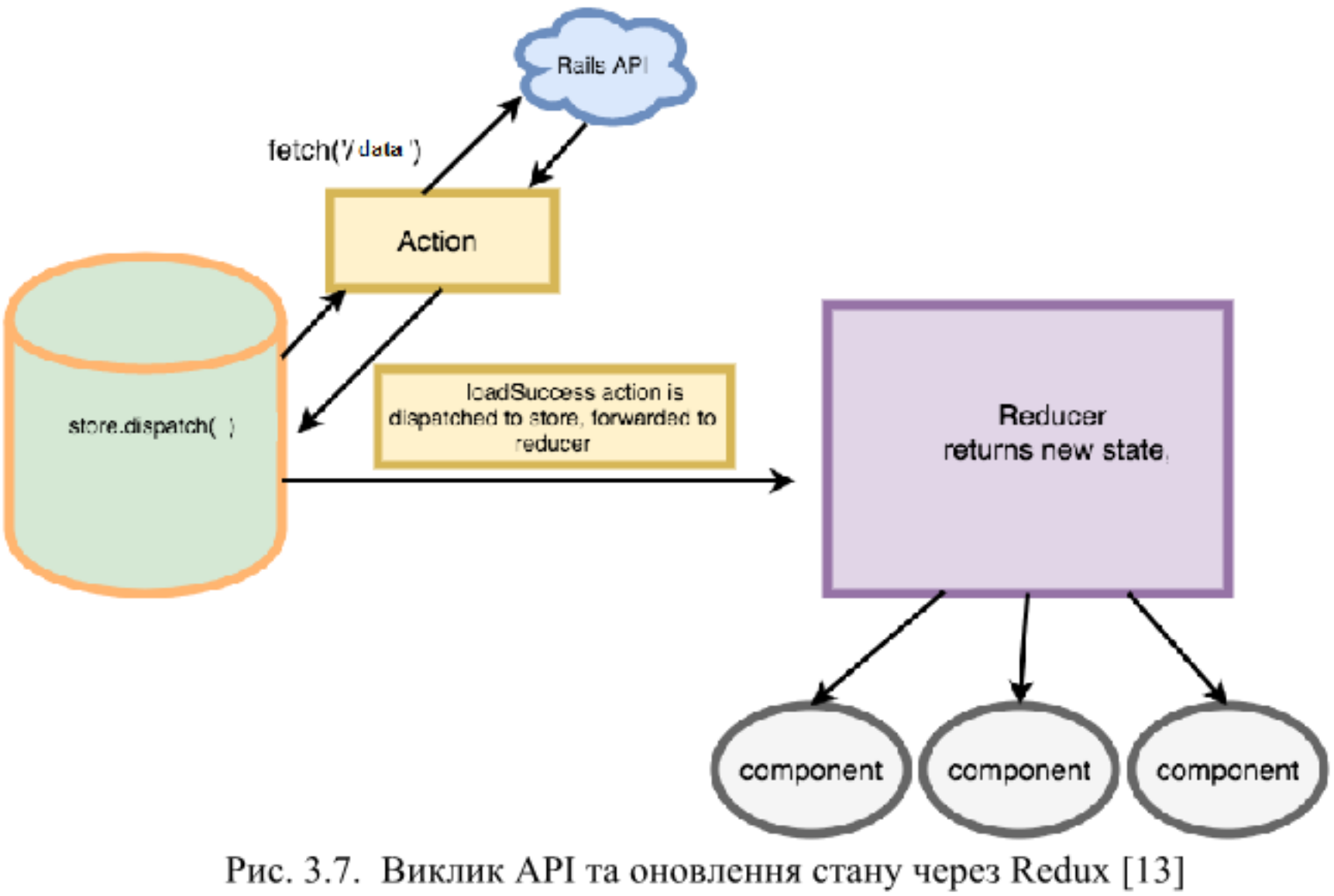


Рис. 3.7. Виклик API та оновлення стану через Redux [13]

Детально проілюструємо як під час отримання з сервера json та його опрацювання може виглядати стан (state) нашої програми. Як було розкрито вище, за зміст всього стану нашої програми відповідає об'єкт Store. Це звичайний об'єкт {}. Важливо розуміти, що у Redux лише один об'єкт Store. Redux - односпрямований потік даних у вашому додатку. Сталася дія від користувача - полетів **action**, екшен був спійманий ред'юсером - змінилися пропси у React-компонента - компонент перемалювався.

Процес створення store розглянемо більш детально. Store об'єднує редьюсер (reducer) і дії (actions), а також має кілька надзвичайно корисних методів, наприклад:

- getState() - дозволяє отримати стан програми;

- dispatch(action) - дозволяє оновлювати стани, шляхом виклику ("диспатча") дії;

- subcribe(listener) - реєструє слухачів.

Actions описують дії. Actions – це об'єкт. Обов'язкове поле - type. Усі дані, які передаються разом з дією. Щоб викликати actions, ми повинні написати функцію, яка в рамках Redux називається - ActionsCreator().

Наприклад:

```
function getPhotos(year) {
  return {
    type: GET_PHOTOS,
    payload: year,
  };
}
```

Actions описує факт, про те що щось сталося, але не вказує, як стан програми повинен змінитися у відповідь, це робота для Reducer. Кожній зміні state відповідає свій reducer. Усі редьюсери об'єднуються у Кореневому Ред'юсері (rootReducer).

У редьюсері завжди повинні повертати новий об'єкт, не змінювати state, а створити новий state. Об'єкт, який ми повертаємо в редьюсері, далі за допомогою функції connect, перетвориться на властивості компонентів. Таким чином, якщо працювати, наприклад, з фото, можна написати такий псевдо-код:

<Page photos={reducerPage.photos} />

Завдяки цьому, всередині компонента <Page />ми зможемо отримати фото, як this.props.photos.

Тобто, у програмі потік даних виглядає так:

1. Користувач, наприклад, натискає кнопку, щоб завантажити список фото;
2. Компонент React викликає обробник даних, щоб надати йому деякі дані;
3. Обробник даних викликає API за допомогою запиту GET, наприклад, '/fotos';
4. API отримує дані та повертає свою обіцянку обробнику даних;
5. Обробник даних надсилає дію Redux із корисним навантаженням API (наприклад, список фото);
6. Redux оновлює стан програми за допомогою списку фото, яким вона була передана
7. Зміну стану помічає компонент, який виконує дії для оновлення, оновлюючи себе блискучим списком фото.

Розглянемо процес отримання даних в React за допомогою Fetch API [14].

Fetch API — це інструмент, який вбудовано в більшість сучасних браузерів на об’єкт вікна (window.fetch) і дозволяє нам дуже легко робити HTTP-запити за допомогою обіцянок JavaScript.

Щоб зробити простий запит GET із fetch, нам просто потрібно включити кінцеву точку URL-адреси, до якої ми хочемо зробити наш запит. Ми хочемо зробити цей запит, коли наш компонент React буде змонтовано.

Для цього ми робимо запит у хуку useEffect і обов’язково надаємо порожній масив залежностей як другий аргумент, щоб наш запит було зроблено

лише один раз (за умови, що він не залежить від будь-яких інших даних у нашому компоненті).

Під час першого .then()зворотного виклику ми перевіряємо, чи була відповідь правильною (response.ok). Якщо так, ми повертаємо нашу відповідь для переходу до наступного, а потім виконуємо зворотний виклик як дані JSON, оскільки це дані, які ми отримаємо від нашого API випадкового користувача.

Якщо це неправильна відповідь, ми припускаємо, що під час надсилання запиту сталася помилка. Використовуючи fetch, нам потрібно самостійно обробляти помилки, тому ми викидаємо response помилку, щоб її обробив наш catch зворотній виклик.

Тут, у нашому прикладі, ми переводимо наші дані про помилки в стан із setError. Якщо є помилка, ми повертаємо текст "Помилка!".

Зауважте, що ви також можете відобразити повідомлення про помилку з об’єкта помилки, який ми перевели в стан, використовуючи error.message.

Ми використовуємо .finally() зворотний виклик як функцію, яка викликається, коли наша обіцянка успішно чи ні. У ньому ми встановлюємо loading значення false, щоб більше не бачити текст завантаження.

Натомість ми бачимо або наші дані на сторінці, якщо запит зроблено успішно, або те, що під час виконання запиту сталася помилка, якщо ні.

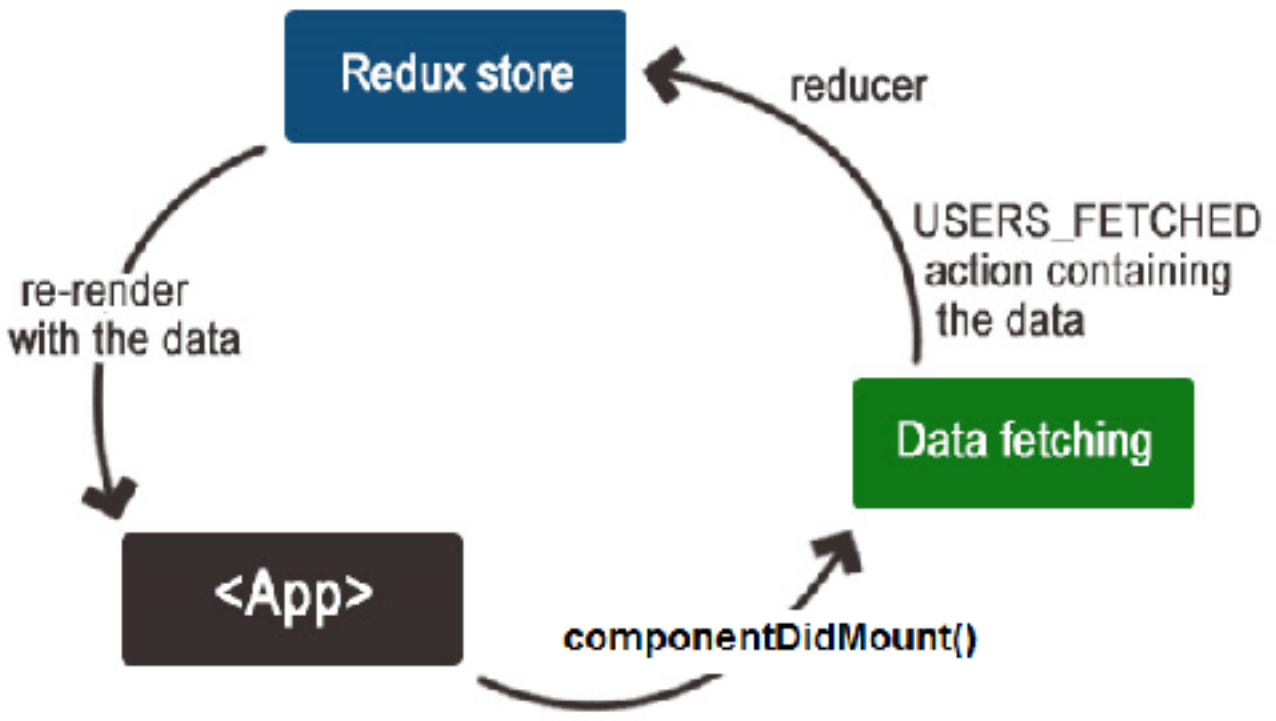


Рис. 3.8. Реалізація шаблону Redux з методом fetch [15]

Як було визначено вище (п. 3.1) componentDidMount() викликається відразу після монтування компонента (вставлення в дерево). Якщо потрібно завантажити дані з віддаленої кінцевої точки, це гарне місце для створення екземпляра мережевого запиту.

Редуктор, який обробляє наш USER_FETCHED масив, може виглядати так [15]:

```
// reducer.js
        import { USERS_FETCHED } from './constants';
        function getInitialState() {
          return { users: null };
        }
        const reducer = function (oldState = getInitialState(), action) {
          if (action.type === USERS_FETCHED) {
            return { users: action.response.data };
          }
          return oldState;
        };
```

Нам також потрібен творець дій, який використовуватиметься в <App> компоненті, і селектор, щоб ми могли отримати users зі стану програми [15]:

```
// actions.js
        import { USERS_FETCHED } from './constants';
        export const usersFetched = response => ({ type: USERS_FETCHED,
response });
// selectors.js
        export const getUsers = ({ users }) => users;
```

Останнім моментом щодо реалізації Redux є створення простої функції-помічника:

```
// store.js
            import { USERS_FETCHED } from './constants';
            import { createStore } from 'redux';
            import reducer from './reducer';
            export default () => createStore(reducer);
```

Під час рендерингу на стороні сервера створюється новий екземпляр store для кожного запиту.

Написання компонента React (<App>). Якщо ми хочемо відобразити щось на стороні сервера, нам потрібно змінити своє мислення. Ми повинні ретельно подумати про те, що робить наш код і чи можливо це на сервері. Наприклад, якщо ми отримуємо доступ до window об'єкта, нам доведеться переосмислити наш компонент або використовувати оболонку, оскільки її немає window на стороні сервера.

Наступний код є реалізацією нашого <App> компонента.

```
// App.js
            import React from 'react';
            import { connect } from 'react-redux';
            import { getUsers } from './redux/selectors';
```

```
import { usersFetched } from './redux/actions';

const ENDPOINT = 'http://localhost:3000/users_fake_data.json';

class App extends React.Component {
 componentWillMount() {
  fetchUsers();
 }
 render() {
  const { users } = this.props;
  return (
   <div>
    { users && users.length > 0 && users.map (
      // ... render the user here
     )
    }
   </div>
  );
 }
}
```

```
const ConnectedApp = connect ( state => ({ users: getUsers(state)}),
        dispatch => ({
          fetchUsers: async () => dispatch(
            usersFetched(await (await fetch(ENDPOINT)).json())
          )
        })
      ) (App);
export default ConnectedApp;
```

Ми використовуємо componentDidMount.

fetchUsers це асинхронна функція, яка передається як проп, яка використовує Fetch API для отримання даних із кінцевої точки. Коли виконані функції fetch(), і json() то, ми відправляємо USERS_FETCHED дію. Пізніше редуктор підбирає її та повертає новий стан, що містить дані користувачів. І оскільки наш App компонент підключено до Redux, він повторно відображається. Код на стороні клієнта завершується розміщенням <App> компонента на сторінці.

Модуль **redux-thunk** - проміжне програмне забезпечення (middleware) Thunk для Redux [16]. Це дозволяє створювати асинхронні и функції з

внутрішньою логікою, які можуть взаємодіяти зі сховищем dispatch, відображати стан через getState та взаємодіяти з іншими методами Redux.

Оскільки редюсери повинні бути «чистими» (вони не змінюють нічого поза межами своєї області), то можливо виконувати жодних викликів API або диспетчерських дій зсередини редуктора. Якщо потрібно, щоб дія щось робила, код thunk має працювати всередині функції. Тому, функція ("thunk") - це пакет роботи, яку потрібно виконати. Thunk підтримує введення користувацьких аргументів у програмне забезпечення проміжного ланцюга. Зазвичай це корисно для таких випадків, як використання рівня служби API, який можна замінити на макет служби в тестах. Для Redux Toolkit getDefaultMiddleware зворотний виклик всередині configureStore дає змогу передати настроюваний extraArgument. Наприклад,

```
import { configureStore } from '@reduxjs/toolkit'
import rootReducer from './reducer'
import { myCustomApiService } from './api'
const store = configureStore({
  reducer: rootReducer,
  middleware: getDefaultMiddleware =>
    getDefaultMiddleware({
      thunk: {
        extraArgument: myCustomApiService
      }
    })
})
// later
function fetchUser(id) {
  // The `extraArgument` is the third arg for thunk functions
  return (dispatch, getState, api) => {
    // you can use api here
  }
}
```

Модуль **redux-devtools-extension** - це спеціальне розширення для браузера, яке дозволяє легко дебажувати (debug - перевірка та налагодження файлів, що виконуються) redux програми. Найчастіше використовується функція переглянути redux store. Вона дозволяє контролювати потоки даних, бачити стан виконання в будь-який момент часу (initial state) нашої програми.

Процес підключення REST API для React додатків проілюстровано на рис. 3.9.

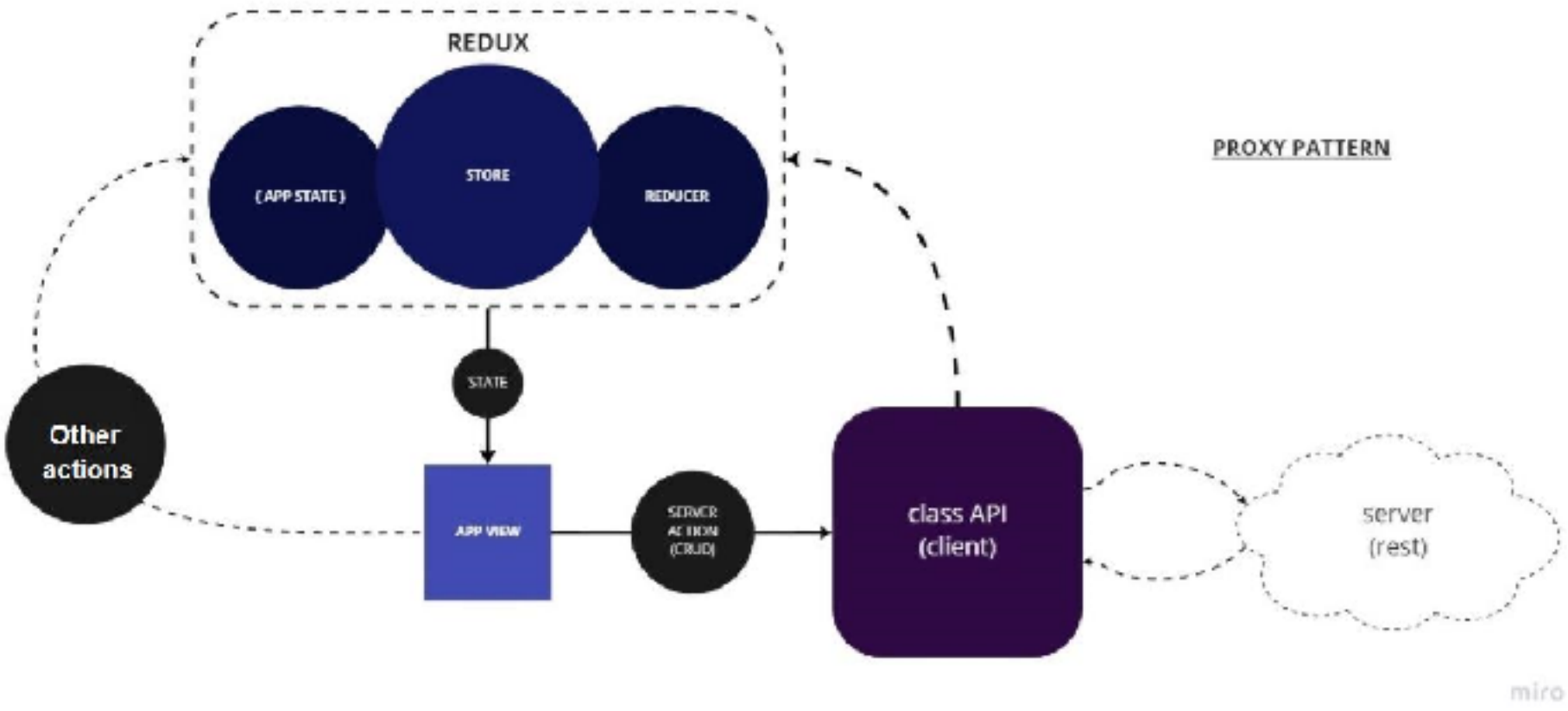


Рис. 3.9. Процес підключення REST API для React додатків [16]

3.3.2. Практичний модуль

У практичній частині продовжуємо проектну розробку web-додатку Tours City (екскурсії містом), а саме, виводимо у front-end данні з REST API.

Етап 3.3. Підключення REST API до головної сторінки

Дія 3.3.1. Встановлюємо потрібні модулі для побудови інтерфейсу користувача.

- переходимо до папки client:

```
cd client
```

- у папці client вводимо у терміналі:

```
npm install redux@4.0.1
npm install react-redux@5.1.1
npm install redux-thunk@2.3.0
npm install redux-devtools-extension
```

Дія 3.3.2. У файл клієнтської частини до файлу package.json у кінець коду прописуємо спеціальну властивість proxy де вказуємо локалхост сервера для доступу до API:

```
"proxy": "http://localhost:5000/"
```

Дія 3.3.3. Заходимо до client\src та засобами VS Code створюємо дві нові папки:

reducers

actions

Дія 3.3.4. У папці reducers створюємо перший файл rootReducer.js. До файлу записуємо код:

```
import { combineReducers } from "redux";
import { destinations } from "./destinations";
const rootReducer = combineReducers({
   destinations
});
export default rootReducer;
```

Дія 3.3.5. У папці reducers створюємо другий файл destinations.js, який буде взаємодіяти із даними колекції destinations. До файлу записуємо код:

```
export function destinations (state = [], action) {
   switch (action.type) {
      case "destinations_FETCH_DATA_SUCCESS":
         return action.destinations;
      default:
         return state;
   }
}
```

Дія 3.3.6. У папці actions створюємо файл destination.js. До файлу записуємо код:

Лістинг 3.6 : Код файлу destination.js

```
export function destinationsFetchDataSuccess(destinations) {
   return {
      type: "destinations_FETCH_DATA_SUCCESS",
      destinations      }
}
export function destinationsFetchData(url) {
   return (dispatch) => {
      fetch(url)
         .then(response => {
            if(!response.ok) {  throw new Error(response.statusText);    }
            return response;    })
         .then(response => response.json())
         .then(destinations                                                    =>
dispatch(destinationsFetchDataSuccess(destinations)))
         .catch(()=>{});
   }
}
```

кінець лістингу 3.6

Дія 3.3.7. Файл index.js змінюємо відповідно до такого коду:

Лістинг 3.7 : Код файлу index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { Provider } from "react-redux";
import { createStore, applyMiddleware } from "redux";
import thunk from "redux-thunk";
import rootReducer from "./reducers/rootReducer";
import { composeWithDevTools } from "redux-devtools-extension";
import { BrowserRouter } from 'react-router-dom';
const store = createStore(
   rootReducer,
   composeWithDevTools(applyMiddleware(thunk))    );
ReactDOM.render(
 <BrowserRouter>
   <Provider store={store}>
      <App />
   </Provider>
</BrowserRouter>,
   document.getElementById('root')
);
reportWebVitals();
```

кінець лістингу 3.7

Дія 3.3.8. У файлі Header.js в частині навігаційної панелі змінюємо код:

```
<NavDropdown title="За назвою дестинації" id="basic-nav-dropdown">
          {this.props.destinations.map((destination, index)=> {
return  <NavDropdown.Item key={index}><Link
to={`/detail/:${destination._id}`}>{destination.destName}</Link></NavDropd
own.Item>
          })}
      </NavDropdown>
```

а наприкінці файлу вставляємо код:

```
const mapStateToProps = (state) => {
  return {
    destinations: state.destinations
  };
};
const mapDispatchToProps = dispatch => {
  return {
    fetchData: url => dispatch(destinationsFetchData(url))
  };
};
export default connect (mapStateToProps, mapDispatchToProps) (Header);
```

Дія 3.3.9. Тестуємо виведення компонент.

- у консолі в корні проекту вводимо

```
npm run dev
```

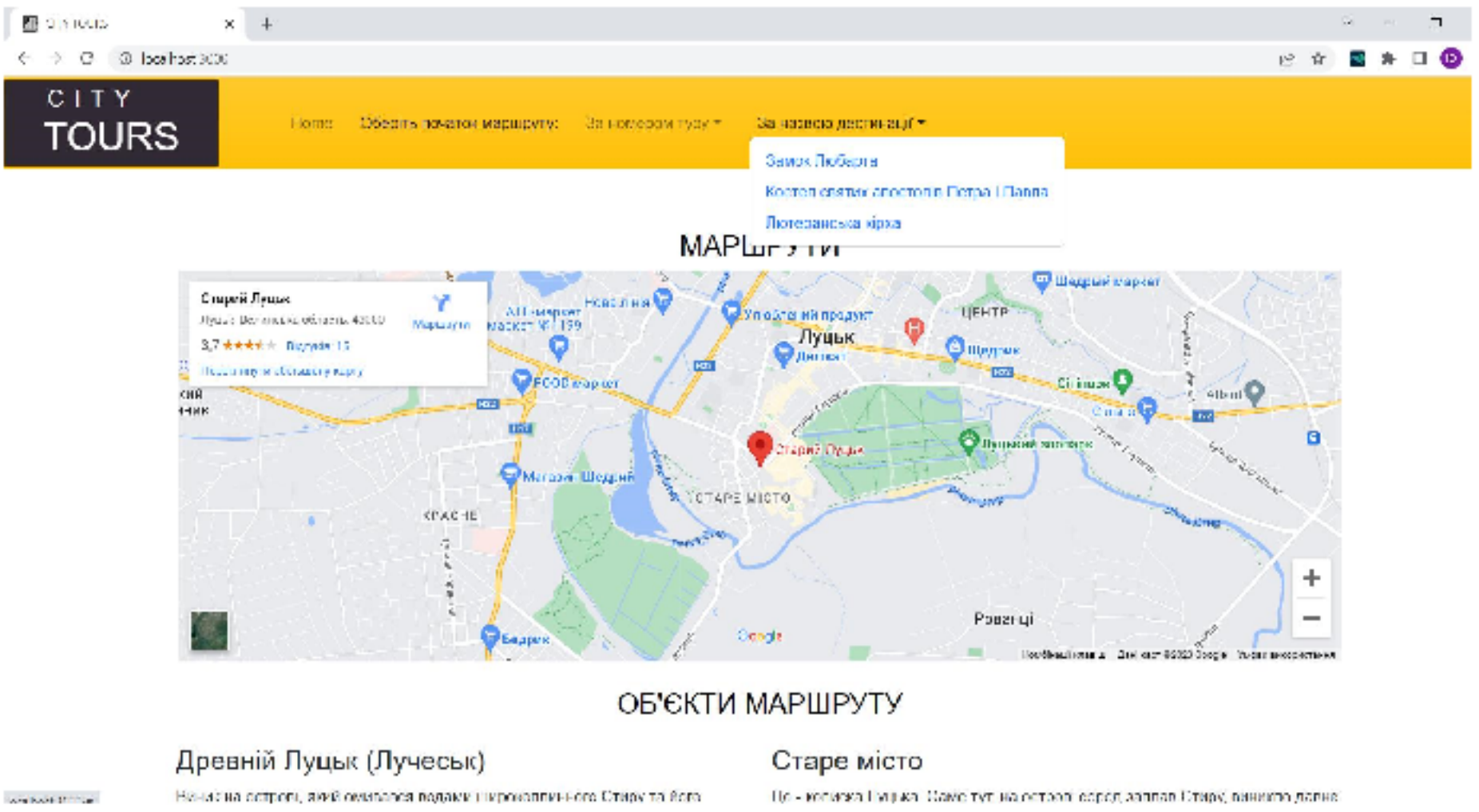


Рис. 3.10. Встановлений зв’язок з REST API

Етап 3.4. Підключення на головній сторінці викликів на дочірні сторінки

Дія 3.4.1. У папці components створюємо файл DestinationPage.js. До файлу записуємо код:

Лістинг 3.8 : Код файлу DestinationPage.js

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { Link } from 'react-router-dom';
import { destinationsFetchData } from "../actions/destination";
import Row from 'react-bootstrap/Row';
import Col from 'react-bootstrap/Col';
        var turNumber = null;
class DestinationPage extends Component {
 render() {
   turNumber = this.props.turNumber.split(":")[1];
   return (
    <div className="wrap">
     <div className="destinationItems">
<Row>
 <h3 className="blog_head">
  Перелік дестинацій туру № {turNumber}{" "}
 </h3>
 {this.props.destinations.map((destination) => {
  if (destination.turNumber == turNumber) {
   let imgName = "/static/files/" + destination.image;
   return (
    <Col>
     <div className="destinationItem">
      <h5>{destination.destName}</h5>
      <img src={imgName} width="250" />
      <p>
       <Link to={`/detail/:${destination._id}`}>
        Більше інформації...
       </Link>
      </p>
```

Продовження лістингу 3.8

```
                </div>
              </Col>
            );
          }
        })}
      </Row>
    </div>
  </div>
 );
 }
}
const mapStateToProps = (state) => {
 return {
  destinations: state.destinations,
 };
};
const mapDispatchToProps = dispatch => {
 return {
    fetchData: url => dispatch(destinationsFetchData(url))
  };
};
export default connect (mapStateToProps, mapDispatchToProps) (DestinationPage);
```

кінець лістингу 3.8

Дія 3.4.2. У папці components створюємо файл DetailPage.js. До файлу записуємо код:

Лістинг 3.9 : Код файлу DetailPage.js

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { destinationsFetchData } from "../actions/destination";   // Уточнюємо
назву файла
        var streetid = null;
```

Продовження лістингу 3.9

```
            class DetailPage extends Component {
render() {
 return (
  <div>
    {this.props.destinations.map((destination)=> {
         streetid = this.props.id.split(':')[1];
     if (destination._id == streetid) {
     let imgName = '/static/files/' + destination.image;
                return  (
                <div>
                 <h4  className='destName'>{destination.destName}</h4>
                 <img src={imgName}   width='250'/>
                 <p>  {destination.destShort} </p>
                 <p> {destination.destLong} </p>
                </div>
                 )
                 }
   })
          }
         </div>
       );
     }
}
const mapStateToProps = (state) => {
   return {
     destinations: state.destinations
   };
 };
 const mapDispatchToProps = dispatch => {
   return {
     fetchData: url => dispatch(destinationsFetchData(url))
   };
 };
 export default connect (mapStateToProps, mapDispatchToProps) (DetailPage);
```

кінець лістингу 3.9

Дія 3.4.3. У файлі Header.js в частині навігаційної панелі змінюємо код:

```
<NavDropdown title="За номером туру" id="basic-nav-dropdown">
  <NavDropdown.Item href="/links/:1">Тур 1 </NavDropdown.Item>
  <NavDropdown.Item href="/links/:2">Тур 2 </NavDropdown.Item>
  <NavDropdown.Item href="/links/:3">Тур 3 </NavDropdown.Item>
</NavDropdown>
```

Дія 3.4.4. Файл routes.js, що відповідає за маршрутизацію внутрішньої структури сторінок web-додатку на основі URL, змінюємо відповідно коду як на лістингу 3.10.

Лістинг 3.10 : Код файлу routes.js

```
import React from 'react'
import {Switch, Route, Redirect} from 'react-router-dom'
import MainPage from './components/MainPage';
import DestinationPage from './components/DestinationPage';
import DetailPage from './components/DetailPage';
export const routes = () => {
 return (
    <Switch>
      <Route path="/" exact>
          <MainPage />
      </Route>
<Route path="/links/:turNumber" render={(props) => (<DestinationPage
turNumber={props.match.params.turNumber} /> )} />
<Route path="/detail/:id" render={(props) => (<DetailPage
id={props.match.params.id} /> )} />
          <Redirect to="/" />
    </Switch>
     )
}
```

кінець лістингу 3.10

Дія 3.4.5. Тестуємо виведення компонент.

- у консолі в корні проекту вводимо

```
npm run dev
```

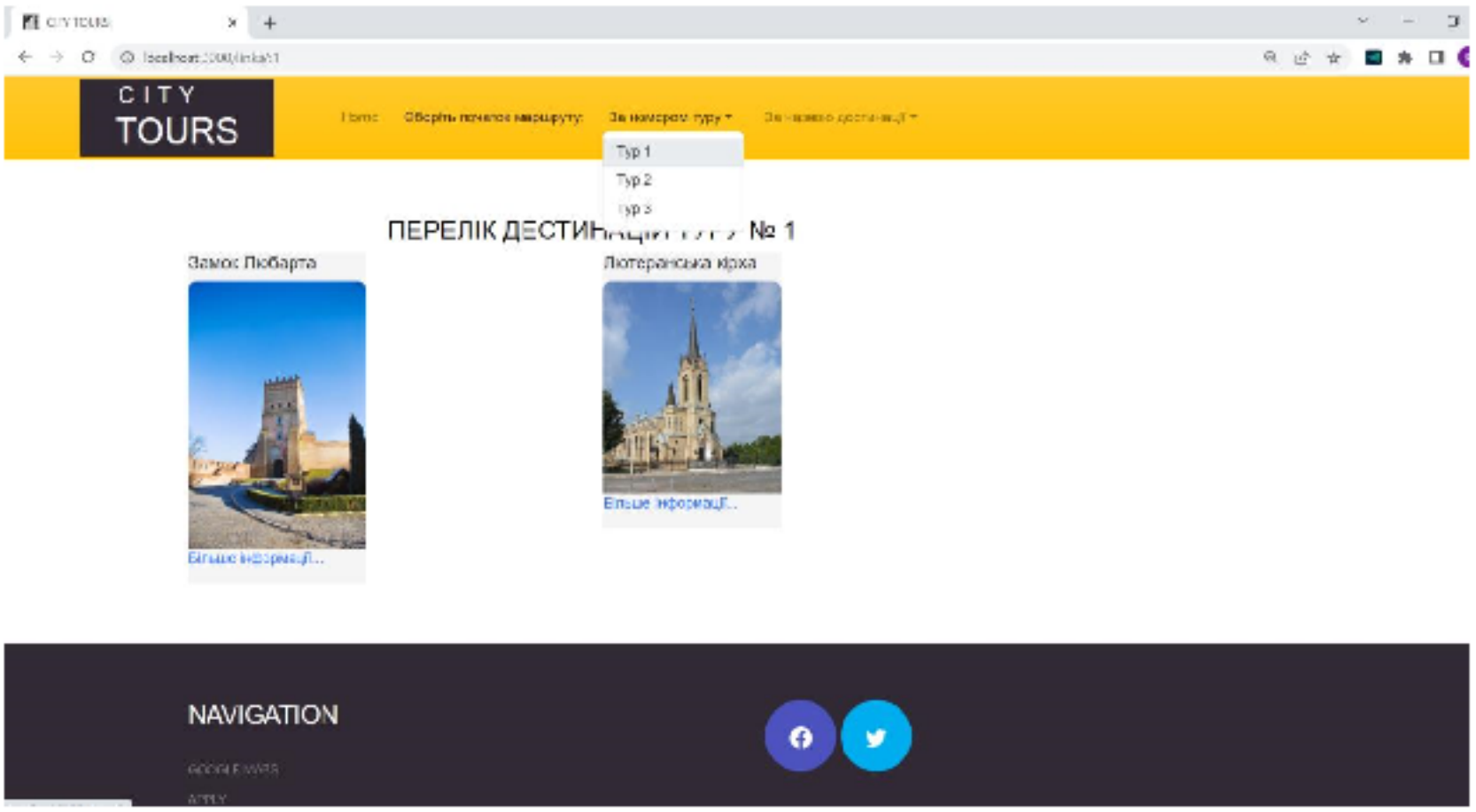


Рис. 3.11. Сторінка дестинацій за туром 1

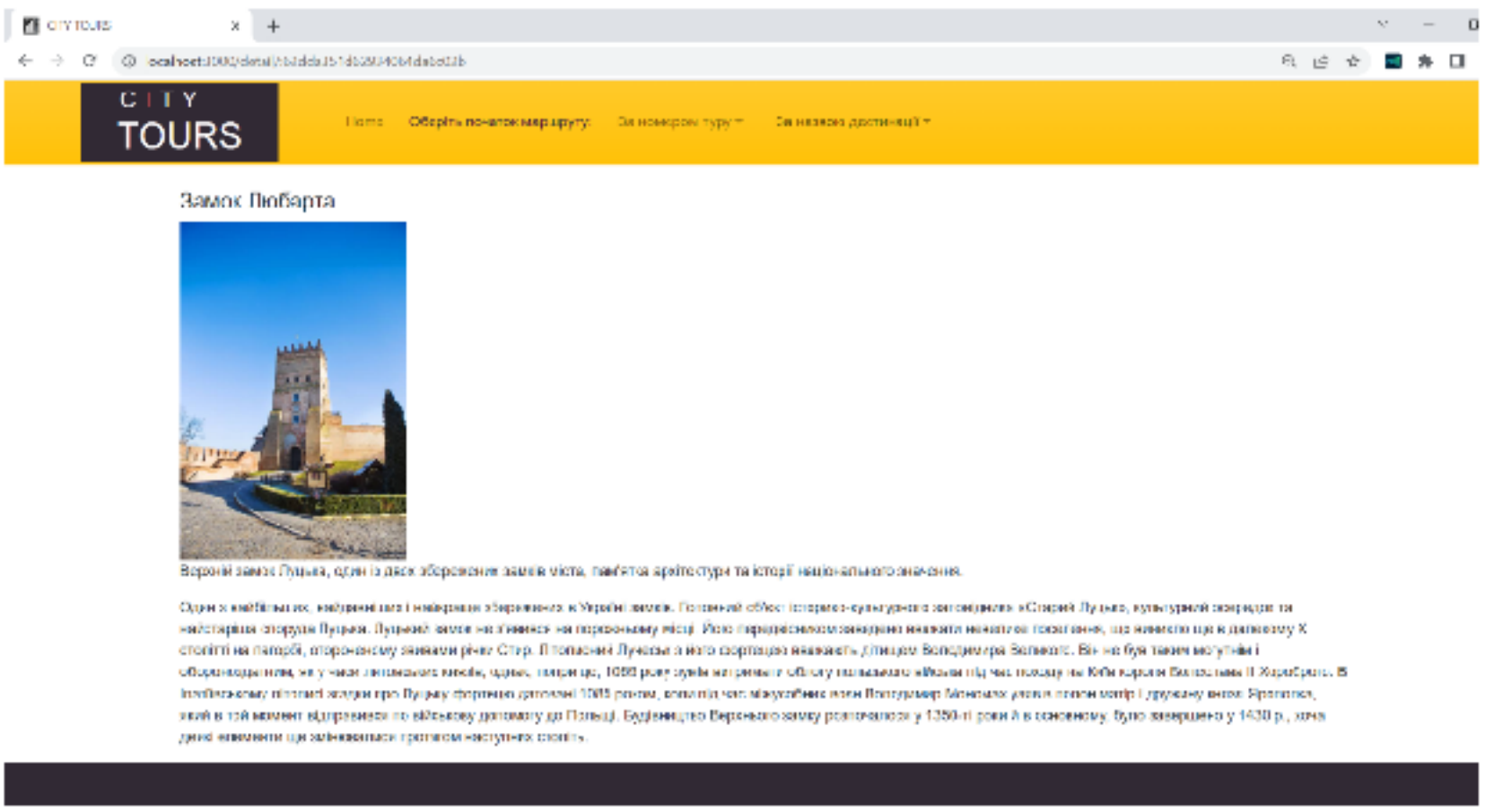


Рис. 3.12. Сторінка деталізації інформації про дестинацію

3.4. ТЕСТОВІ ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ

1. React – це
   a) JavaScript-бібліотека
   b) JavaScript-фреймворк
   c) JSX JavaScript-код
   d) скриптинг XSS
   e) компілятор
2. React призначений
   a) для роботи з користувацьким інтерфейсом
   b) для передачі API даних
   c) для отримання API даних
   d) для взаємодії з сервером
   e) для розбиття web сторінки на компоненти
3. Вперше React почали використовувати на сайті соціальної мережі
   a) Facebook
   b) Google
   c) MicroSoft
   d) Instagram
   e) YouTube
4. Вихідний код React став доступним у відкритому доступі з
   a) 2013 р.
   b) 2011 р.
   c) 2010 р.
   d) 2012 р.
   e) 2014 р.
5. Відмітьте базовий принцип React
   a) застосування компонентів для створення користувацького інтерфейсу
   b) розширення синтаксису JavaScript
   c) зчитування об'єктів для використання при побудови DOM
   d) управління фронтендом

6. Який код на JSX позначає компонент
   a) <Hello />
   b) {hello}
   c) <h1> Hello </h1>
   d) ReactDOM.render
7. Який код на JSX позначає елемент React
   a) <Hello />
   b) {hello}
   c) <h1> Hello </h1>
   d) const name =
8. Який код на JSX позначає оголошення змінної
   a) {hello}
   b) <h1> Hello </h1>
   c) ReactDOM.render
   d) const name =
9. Який код на JSX позначає об'єкт, в якому зберігається інформація про стан інтерфейсу
   a) <Hello />
   b) {hello}
   c) <h1> Hello </h1>
   d) ReactDOM.render
   e) const name =
10. Код на JSX : const header = text ? <h1>{text}</h1> : null; - це
   a) вираз
   b) об'єкт
   c) елемент
   d) компонент
   e) змінна
11. Що таке компонент у React
   a) самостійна частина додатку, що реалізує бізнес логіку та відображення сторінки
   b) функція додатку, що реалізує бізнес логіку
   c) клас додатку, що реалізує бізнес логіку
   d) будь-яка частина додатку, що реалізує дизайн відображення сторінки
   e) будь-який блок додатку, що реалізує дизайн відображення сторінки

12. Компонент в React на JS є
    a) функцією або класом JS
    b) функцією JS
    c) класом JS
    d) частиною View
    e) частиною коду JS
13. Основна мета React – це (дві відповіді)
    a) створення View
    b) програмування Front-end
    c) створення Controller
    d) створення Model
14. Що таке Віртуальна DOM? (дві відповіді)
    a) відображенням реальної DOM в пам'яті
    b) кеш структури даних в пам'яті
    c) API
    d) дерево компонент
    e) теги в пам'яті
15. Прості об'єкти JavaScript. Містять інформацію, яка впливає на результати рендеринга. Якщо передається дані компоненту аналогічно параметрам функції, то це
    a) властивість
    b) стан
    c) пропс
    d) немає відповіді
16. Прості об'єкти JavaScript. Містять інформацію, яка впливає на результати рендеринга. Якщо управляється всередині компонента аналогічно змінним, оголошеним всередині функції, то це
    a) властивість
    b) стан
    c) пропс
    d) немає відповіді

17. Прості об'єкти JavaScript. Містять інформацію, яка впливає на результати рендеринга. Якщо функції приймають довільні дані і повертають React-елементи, які описують те, що повинно з'явитися на екрані, то це

a) властивість
b) стан
c) пропс
d) немає відповіді

18. Особливість компонентів React у тому, що вони завжди починаються з

a) з великої (прописної) літери
b) з малої (рядкової) літери
c) з символу тега « < »
d) з символів « = < »
e) з « const element = < »

19. Етап життєвого циклу коли компонент React готує установку початкового стану і параметрів за замовчуванням

a) Ініціалізація
b) Монтування
c) Оновлення
d) Розмонтування

20. Етап життєвого циклу коли компонент React готовий для запису в DOM браузера

a) Ініціалізація
b) Монтування
c) Оновлення
d) Розмонтування

21. Етап життєвого циклу коли компонент відправляє нові властивості і значення стану

a) Ініціалізація
b) Монтування
c) Оновлення
d) Розмонтування

22. Етап життєвого циклу коли етап охоплює методи життєвого циклу componentWillMount і componentDidMount

a) Ініціалізація
b) Монтування
c) Оновлення
d) Розмонтування

23. Метод, що відправлятємережеві запити на сервер і довантажує нову інформацію по мірі необхідності

a) hooks
b) fetch
c) response
d) setState

24. Функція, що приймає нове значення стану і ставить в чергу повторний рендер компонента

a) hooks
b) fetch
c) response
d) setState

25. Клас, що надає метод декодування відповіді в форматі JSON, який заснований на promise

a) hooks
b) fetch
c) response
d) setState

3.5. ЗАВДАННЯ ДО САМОСТІЙНОЇ РОБОТИ

1. Ознайомитись із принципами та концепціями побудови front-end.
2. Ознайомитись з основними пакетами Node JS які є основними функціонування React.js.
3. Ознайомитись з ключовими особливостями React.js.
4. Ознайомитись з синтаксисом мови шаблонів JSX.
5. Ознайомитись з основами віртуальної DOM у React.js.
6. Ознайомитись з процес синхронізації віртуальної DOM з реальною DOM за допомогою бібліотеки.
7. Ознайомитись з алгоритмами етапів життєвого циклу компоненту у React.js.
8. Ознайомитись з основними засобами та підходами до формування елементів дизайну UI у React.js.
9. Ознайомитись з основними засобами побудови навігаційних елементів UI у React.js.
10. Ознайомитись з методами використання менеджера станів Redux.

ВИКОРИСТАНІ ДЖЕРЕЛА

1. Components and Props. URL: https://reactjs.org/docs/components-and-props.html (дата звернення: 28.04.2023).
2. Introducing JSX. URL: https://ru.react.js.org/docs/introducing-jsx.html (дата звернення: 28.04.2023).
3. Virtual DOM and Internals. URL: https://reactjs.org/docs/faq-internals.html (дата звернення: 28.04.2023).
4. Diagram lifecycle. URL: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram (дата звернення: 28.04.2023).
5. React.Component. URL: https://reactjs.org/docs/react-component.html. (дата звернення: 28.04.2023).
6. Understanding React — Component life-cycle. URL: https://habr.com/ru/post/358090 (дата звернення: 28.04.2023).
7. Using Bootstrap with React: Tutorial with examples. URL: https://blog.logrocket.com/using-bootstrap-with-react-tutorial-with-examples (дата звернення: 28.04.2023).
8. React Router Tutorial. URL: https://reactrouter.com/en/main/start/tutorial (дата звернення: 28.04.2023).
9. React Bootstrap. Navbars. URL: https://react-bootstrap.github.io/components/navbar (дата звернення: 28.04.2023).
10. React Router - W3Schools. URL: https://www.w3schools.com/react/react_router.asp (дата звернення: 28.04.2023).
11. Redux Fundamentals. URL: https://redux.js.org/tutorials/fundamentals/part-1-overview (дата звернення: 28.04.2023).
12. Flavio Copes. A quick guide to Redux for beginners. URL: https://www.freecodecamp.org/news/a-quick-guide-to-redux-for-beginners-971d18c0509b (дата звернення: 28.04.2023).
13. Sophie DeBenedetto. Async Redux: Connecting React to an External API. URL: https://www.thegreatcodeadventure.com/react-redux-tutorial-part-iii-async-redux (дата звернення: 28.04.2023).
14. Mdn web docs. Using the Fetch API. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch (дата звернення: 28.04.2023).
15. A story about React, Redux and server-side rendering. URL: https://krasimirtsonev.com/blog/article/story-about-react-redux-server-side-rendering-ssr (дата звернення: 28.04.2023).
16. Redux Thunk. URL: https://www.npmjs.com/package/redux-thunk (дата звернення: 28.04.2023).

Розділ 4. РЕЛІЗ САЙТУ. SEO-ПРОСУВАННЯ САЙТУ

4.1. HTTP – ПРОТОКОЛ КЛІЄНТ-СЕРВЕРНОЇ ВЗАЄМОДІЇ

4.1.1. *Теоретичний модуль*

Абревіатура HTTP розшифровується як HyperText Transfer Protocol - це міжнародне позначення протоколу передавання гіпертексту. HTTP – це протокол, що дозволяє отримувати різні ресурси, наприклад HTML-документи [1]. Протокол HTTP є основою обміну даними в Internet. HTTP є протоколом клієнт-серверної взаємодії, що означає ініціювання запитів до сервера самим одержувачем, зазвичай web-браузером (web-browser).

Клієнти та сервери взаємодіють, обмінюючись одиночними повідомленнями (а не потоком даних). Повідомлення, надіслані клієнтом, зазвичай web-браузером, називаються запитами, а повідомлення, надіслані сервером, називаються відповідями.

Отриманий підсумковий документ буде складатися з різних піддокументів, що є частиною підсумкового документа: наприклад, з окремо отриманого тексту, опису структури документа, зображень, відео-файлів, скриптів та багато іншого (дивись рис. 4.1).

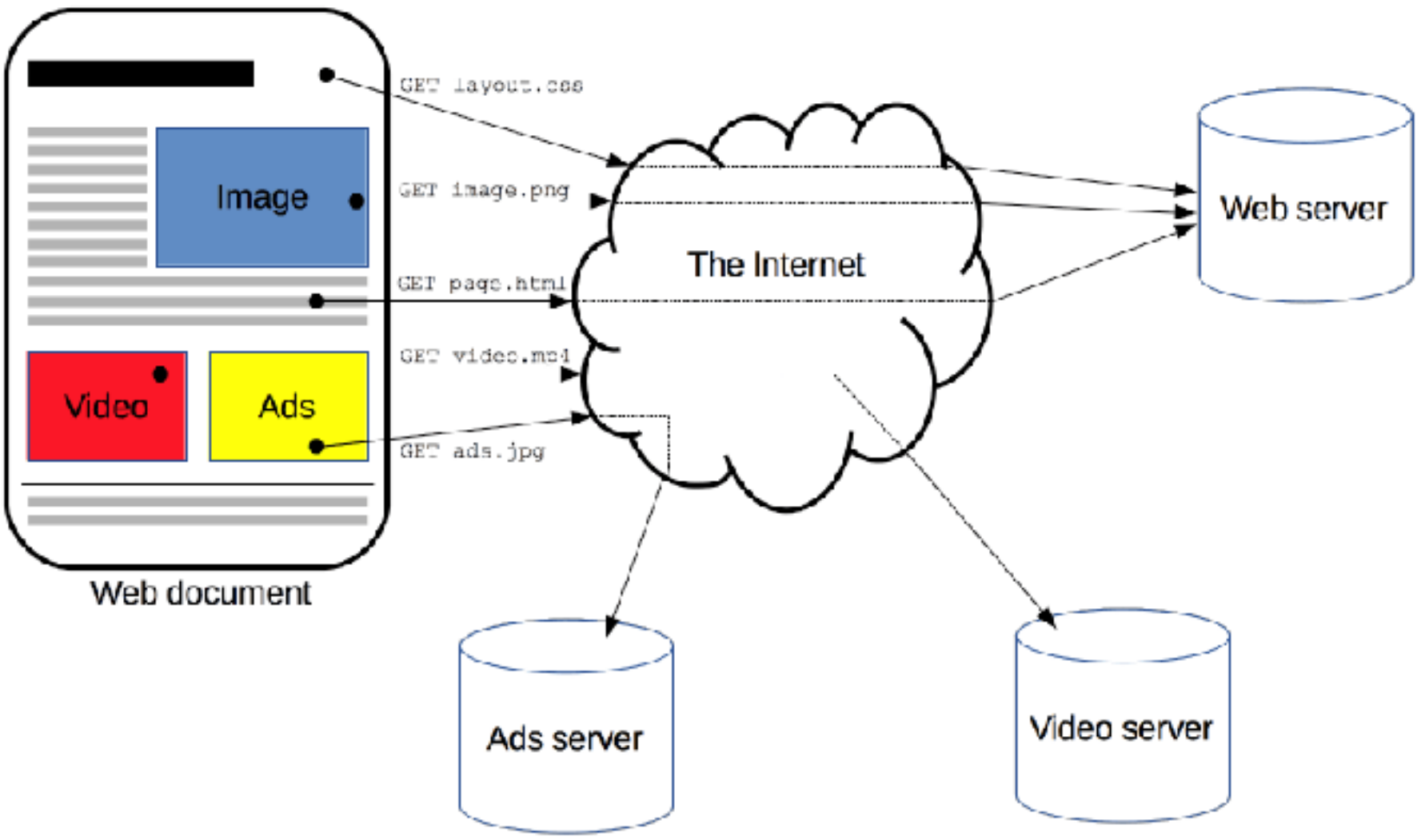


Рис. 4.1. Ілюстрація обміну повідомленнями [1]

HTTP є протоколом прикладного рівня, який найчастіше використовує можливості іншого протоколу – TCP (або TLS – захищений TCP) – для пересилання своїх повідомлень, проте будь-який інший надійний транспортний протокол теоретично може бути використаний для доставки таких повідомлень. HTTP використовується не тільки для отримання клієнтом гіпертекстових документів, зображень та відео, але й для передачі вмісту серверам, наприклад, за допомогою HTML-форм. HTTP також може бути використаний для отримання лише частин документа з метою оновлення web-сторінки на запит (наприклад, за допомогою AJAX запиту).

Складові системи, засновані на HTTP

HTTP - це клієнт-серверний протокол, тобто запити надсилаються якоюсь однією стороною - учасником обміну (user-agent) (або проксі замість нього). Найчастіше в якості учасника виступає web-браузер, але ним може бути будь-хто, наприклад, робот, що подорожує по мережі для поповнення та оновлення даних індексації web-сторінок для пошукових систем.

Кожен запит (англ. request) відправляється серверу, який обробляє його і повертає відповідь (англ. response). Між цими запитами і відповідями зазвичай існують численні посередники, звані проксі, які виконують різні операції і працюють як шлюзи або кеш (рис. 4.2).

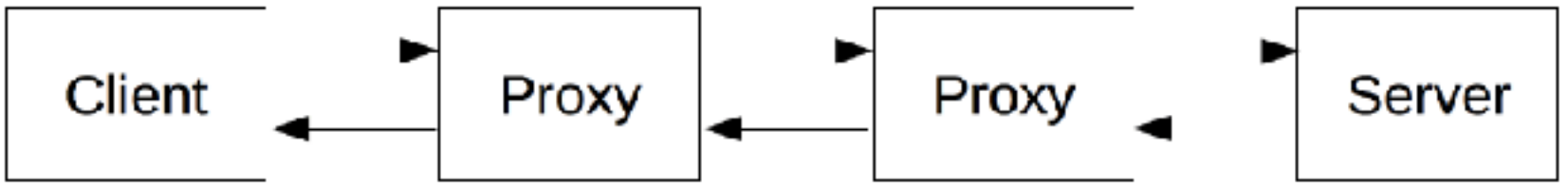


Рис. 4.2. Ілюстрація обміну повідомленнями [1]

Зазвичай, між браузером і сервером набагато більше різних пристроїв-посередників, які відіграють будь-яку роль в обробці запиту: маршрутизатори, модеми і так далі. Завдяки тому, що мережа побудована на основі системи рівнів (шарів) взаємодії, ці посередники "заховані" на мережному та транспортному рівнях. У цій системі рівнів HTTP займає найвищий рівень, який називається "прикладним" (або "рівнем додатків"). Знання про рівні мережі, такі як представницький, сеансовий, транспортний, мережевий, канальний та фізичний, мають важливе значення для розуміння роботи мережі та діагностики можливих проблем, але не потрібні для опису та розуміння HTTP.

Клієнт: учасник обміну

Учасник обміну (user agent) – це будь-який інструмент або пристрій, що діють від імені користувача. Це завдання переважно виконує web-браузер; у

деяких випадках учасниками виступають програми, які використовуються інженерами та web-розробниками для налагодження своїх програм.

Браузер завжди є суттю, яка створює запит. Сервер зазвичай цього не робить, хоча за багато років існування мережі були придумані способи, які можуть дозволити запити з боку сервера.

Щоб відобразити web-сторінку, браузер надсилає початковий запит для отримання HTML-документа цієї сторінки. Після цього браузер вивчає цей документ і запитує додаткові файли, необхідні для відображення змісту web-сторінки (скрипти, інформацію про макет сторінки - CSS таблиці стилів, додаткові ресурси у вигляді зображень і відео-файлів), які безпосередньо є частиною вихідного документа, але розташовані в інших місцях мережі. Далі браузер з'єднує всі ці ресурси для відображення їх користувачу як єдиного документа - web-сторінки. Скрипти, що виконуються самим браузером, можуть отримувати додаткові ресурси по мережі на наступних етапах обробки web-сторінки, і браузер відповідним чином оновлює відображення цієї сторінки для користувача.

Web-сторінка є гіпертекстовим документом. Це означає, що деякі частини тексту, що відображається, є посиланнями, які можуть бути активовані (зазвичай натисканням кнопки миші) з метою отримання і відповідно відображення нової web-сторінки (перехід за посиланням). Це дозволяє користувачеві "переміщатися" сторінками мережі Internet. Браузер перетворює ці гіперпосилання на HTTP-запити і надалі отримані HTTP-відповіді відображає у зрозумілому для користувача вигляді.

Web-сервер

З іншого боку комунікаційного каналу розташований сервер, який обслуговує (англ. server) користувача, надаючи йому документи на запит. З точки зору кінцевого користувача, сервер завжди є якоюсь однією віртуальною машиною, що повністю або частково генерує документ, хоча фактично він може бути групою серверів, між якими балансується навантаження, тобто перерозподіляються запити різних користувачів, або складним програмним забезпеченням, що опитує інші комп'ютери (такі як кешують сервери, сервери баз даних, сервери додатків електронної комерції та інші).

Сервер не обов'язково розташований на одній машині, і навпаки - кілька серверів можуть бути розташовані (хоститися) на одній і тій же машині. Відповідно до версії HTTP/1.1 і маючи Host заголовок, вони навіть можуть ділити ту саму IP-адресу.

Проксі

Між web-браузером та сервером знаходяться велика кількість мережних вузлів, що надсилають повідомлення HTTP. Через шарувату структуру більшість з них оперують також на транспортному мережевому або фізичному рівнях, стаючи прозорим на шарі HTTP і потенційно знижуючи продуктивність. Ці операції лише на рівні додатків називаються проксі. Вони можуть бути прозорими чи ні (змінюючи запити не пройдуть через них), і здатні виконувати безліч функцій:

• caching (кеш може бути публічним або приватним, як кеш браузера)

• фільтрація (як сканування антивірусу, батьківський контроль, …)

• вирівнювання навантаження (дозволити кільком серверам обслуговувати різні запити)

• автентифікація (контролювати доступом до різних ресурсів)

• протоколювання (дозвіл на зберігання історії операцій)

HTTP не має стану, але має сесію

HTTP не має стану: немає зв'язку між двома запитами, які послідовно виконуються по одному з'єднанню. З цього негайно випливає можливість проблем для користувача, який намагається взаємодіяти з певною сторінкою послідовно, наприклад, при використанні кошика в електронному магазині. Але хоча ядро HTTP не має стану, куки дозволяють використовувати сесії зі збереженням стану. Використовуючи розширюваність заголовків, куки додаються до робочого потоку, дозволяючи сесії кожному HTTP-запросе ділитися деяким контекстом чи станом.

HTTP та з'єднання

З'єднання керується на транспортному рівні, і тому виходить за межі HTTP. Хоча HTTP не вимагає, щоб базовий транспортний протокол був заснований на з'єднаннях, вимагаючи лише надійність або відсутність втрачених повідомлень (тобто як мінімум подання помилки). Серед двох найбільш поширених транспортних протоколів Internet TCP надійний, а UDP — ні. HTTP згодом покладається на стандарт TCP, заснований на з'єднаннях, незважаючи на те, що з'єднання не завжди потрібне.

HTTP/1.0 відкривав TCP-з'єднання для кожного обміну запитом/відповіддю, маючи дві важливі недоліки: відкриття з'єднання вимагає декількох обмінів повідомленнями, і тому повільно, хоча стає більш ефективним при надсиланні декількох повідомлень, або при регулярній відправці повідомлень: теплі з'єднання більш ефективні , ніж холодні .

Для пом'якшення цих недоліків, HTTP/1.1 надав конвеєрну обробку (яку виявилося важко реалізувати) і стійкі з'єднання: з'єднання, що лежить в основі

TCP, можна частково контролювати через заголовок Connection. HTTP/2 зробив наступний крок, додавши мультиплексування повідомлень через просте з'єднання, що допомагає тримати з'єднання теплим та ефективнішим.

Проводяться експерименти з розробки кращого транспортного протоколу, що підходить для HTTP. Наприклад, Google експериментує з QUIC (яка заснована на UDP) для надання більш надійного та ефективного транспортного протоколу.

Чим можна керувати через HTTP

Природна розширюваність HTTP згодом дозволила більше управління та функціональність Мережі. Кеш та методи аутентифікації були ранніми функціями в історії HTTP. Здатність послабити початкові обмеження, навпаки, було додано у 2010-ті.

Загальні функції, керовані з HTTP.

• Кеш. Сервер може інструктувати проксі, визначаючи, що і як довго кешувати. Клієнт може інструктувати проксі у проміжних кешах, визначаючи пріоритети документів, що зберігаються.

• Обмеження джерела. Для запобігання вторгнень шпигунським програмам та іншим порушникам, web-браузер забезпечує суворий поділ між web-сайтами. Тільки сторінки того ж web-джерела можуть отримати доступ до інформації на web-сторінці. Такі обмеження навантажують сервер, хоча заголовки HTTP можуть послабити строгий поділ на стороні сервера, дозволяючи документу стати частиною інформації з різних доменів (з причин безпеки).

• Автентифікація. Деякі web-сторінки доступні лише для спеціальних користувачів. Базова автентифікація може надаватися через HTTP або через використання заголовка WWW-Authenticate (en-US) і подібних до нього, або за допомогою налаштування спецсесії, використовуючи куки.

• Проксі та тунелювання (en-US). Сервери та/або клієнти часто розміщуються в Internet та приховують свої справжні IP-адреси від інших. HTTP запити йдуть через проксі для перетину цього бар'єру. Тому, не всі проксі – HTTP проксі. SOCKS-протокол, наприклад, оперує на нижчому рівні. Інші, наприклад, ftp, можуть бути оброблені цими проксі.

• Сесії. Використання HTTP кук дозволяє пов'язати запит зі станом на сервері. Це створює сесію, хоча ядро HTTP – протокол без стану. Це корисно не тільки для кошиків в Internet-магазинах, але також для будь-яких сайтів, що дозволяють налаштувати користувачеві вихід.

HTTP потік

Коли клієнт хоче взаємодіяти з сервером, що є кінцевим сервером або проміжним проксі, він виконує такі кроки:

1. Відкриття TCP з'єднання: TCP-з'єднання буде використовуватися для надсилання запиту (або запитів) та отримання відповіді. Клієнт може відкрити нове з'єднання, перевикористовувати існуюче або відкрити кілька з'єднань TCP до сервера.

2. Надсилання HTTP-повідомлення: HTTP-повідомлення (до HTTP/2) є людиночитаними. Починаючи з HTTP/2, прості повідомлення інкапсулюються у кадри, унеможливлюючи їх читання безпосередньо, але принципово залишаються такими ж. Наприклад:

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

3. Читає відповідь від сервера:

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html
<!DOCTYPE html... (here comes the 29769 bytes of the requested web page)
```

4. Закриває або перевикористовує з'єднання для подальших запитів.

Якщо активовано HTTP-конвеєр, кілька запитів можуть бути надіслані без очікування отримання першої відповіді. HTTP-конвеєр важко впроваджується в існуючі мережі, де старі шматки ПЗ співіснують із сучасними версіями. HTTP-конвеєр був замінений в HTTP/2 на більш надійні мультиплексні запити у кадрі.

HTTP повідомлення

HTTP/1.1 і раніше HTTP повідомлення людиноадаптовані. У версії HTTP/2 ці повідомлення вбудовані в нову бінарну структуру, кадр, що дозволяє оптимізації, такі як компресія заголовків та мультиплексування. Навіть якщо частина оригінального повідомлення HTTP відправлена в цій версії HTTP, семантика кожного повідомлення не змінюється і клієнт відтворює (віртуально) оригінальний запит HTTP. Це також корисно для розуміння HTTP/2 повідомлень у форматі HTTP/1.1. Існує два типи HTTP повідомлень, запити та відповіді, кожен у своєму форматі.

Запити

Приклад HTTP запитів:

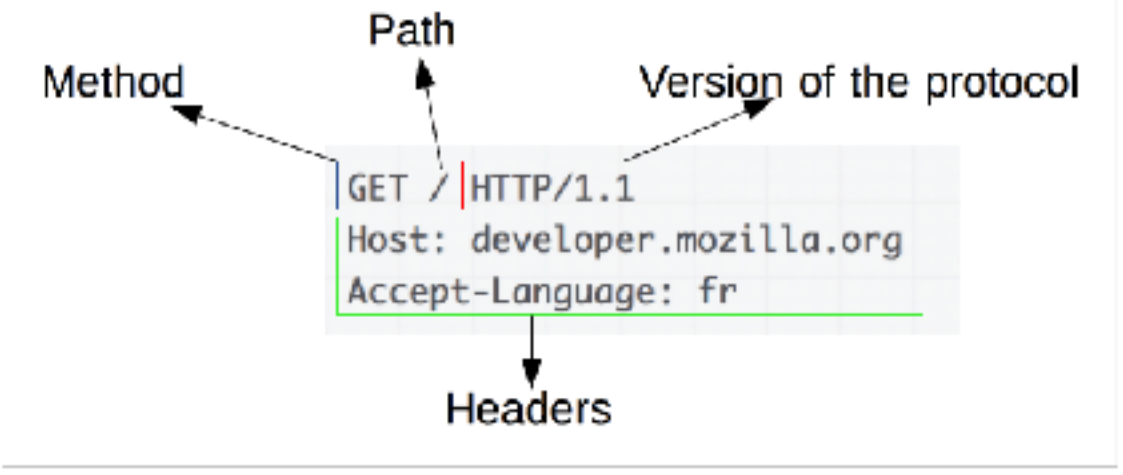


Рис. 4.3. Ілюстрація запиту [1]

Запити містять такі елементи:

• HTTP - методи GET, POST або іменник, такий як OPTIONS або HEAD, що визначає операції, які клієнт хоче виконати. Зазвичай клієнт хоче отримати ресурс (використовуючи GET) або передати значення HTML-форми (використовуючи POST), хоча інші операції можуть бути необхідні в інших випадках.

• Шлях до ресурсу: URL ресурси позбавлені елементів, які є очевидними з контексту, наприклад без протоколу (http://), домену (тут developer.mozilla.org), або TCP порту.

• Версію HTTP-протоколу.

• Заголовки (опційно), які надають додаткову інформацію для сервера.

• Або тіло для деяких методів, таких як POST, Що містить відправлений ресурс.

Відповіді

Приклад відповідей:

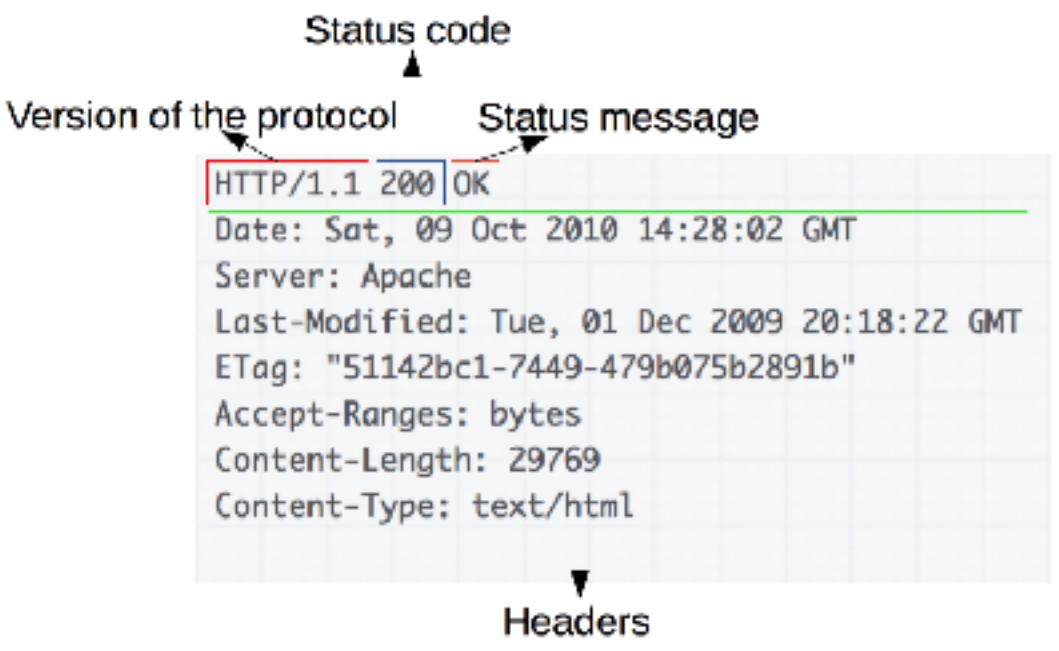


Рис. 4.4. Ілюстрація відповіді [1]

Відповіді містять такі елементи:

• Версію HTTP-протоколу.

• HTTP код стану, що повідомляє про успішність запиту або причину невдачі.

• Повідомлення стану – короткий опис коду стану.

• HTTP заголовки, подібно до заголовків у запитах.

• Опціонально: тіло, що містить ресурс, що пересилається.

Різниця між HTTP та HTTPS [2]

Будь-яка web-сторінка зберігається на сервері - комп'ютері, який постійно підключено до Internet. Коли ви переходите за посиланням або вводите доменне ім'я в адресному рядку, браузер знаходить потрібний сервер і завантажує вміст сторінки. Те саме відбувається і у зворотному напрямку. Ви вводите на сайті логін та пароль від свого облікового запису, натискаєте «Увійти», і браузер відправляє цю інформацію на сервер. Сервер перевіряє деталі та відправляє браузеру сторінку з відкритим обліковим записом.

За HTTP інформація передається у звичайному вигляді, а за HTTPS - у зашифрованому. Шифрувати дані потрібно, щоб хакери не змогли нічого прочитати, якщо їх перехоплять.

Допустимо, ви проходите опитування на сайті, який працює за HTTP-протоколом. Ось ви заповнили порожні поля та натиснули кнопку «Надіслати». Браузер надсилає ваші відповіді серверу. У цей момент хакер може перехопити інформацію та прочитати, що ви там відповідали. Ви навіть цього не помітите.

Імовірно, хакерів не цікавлять ваші відповіді на опитування. Але перехопити можна будь-яку інформацію. Наприклад, ваші паролі або номер банківської картки.

Щоб цього не сталося, HTTP-протокол вирішили вдосконалити. До існуючої технології додали шифрування і вийшов HTTPS — безпечний протокол передачі даних.

Коли ви вводите щось на сайті, що працює за HTTPS, перед відправкою даних на сервер браузер зашифровує інформацію. Щоб розшифрувати та прочитати її, потрібен спеціальний ключ, який зберігається лише на сервері. Таке шифрування називається криптографічним. Якщо навіть шахрай перехопить інформацію, він не зможе її прочитати. На те, щоб підібрати ключ до шифру, підуть роки безперервного перебору.

Різниця між HTTP та HTTPS при передачі даних проілюстрована на рис. 4.5.

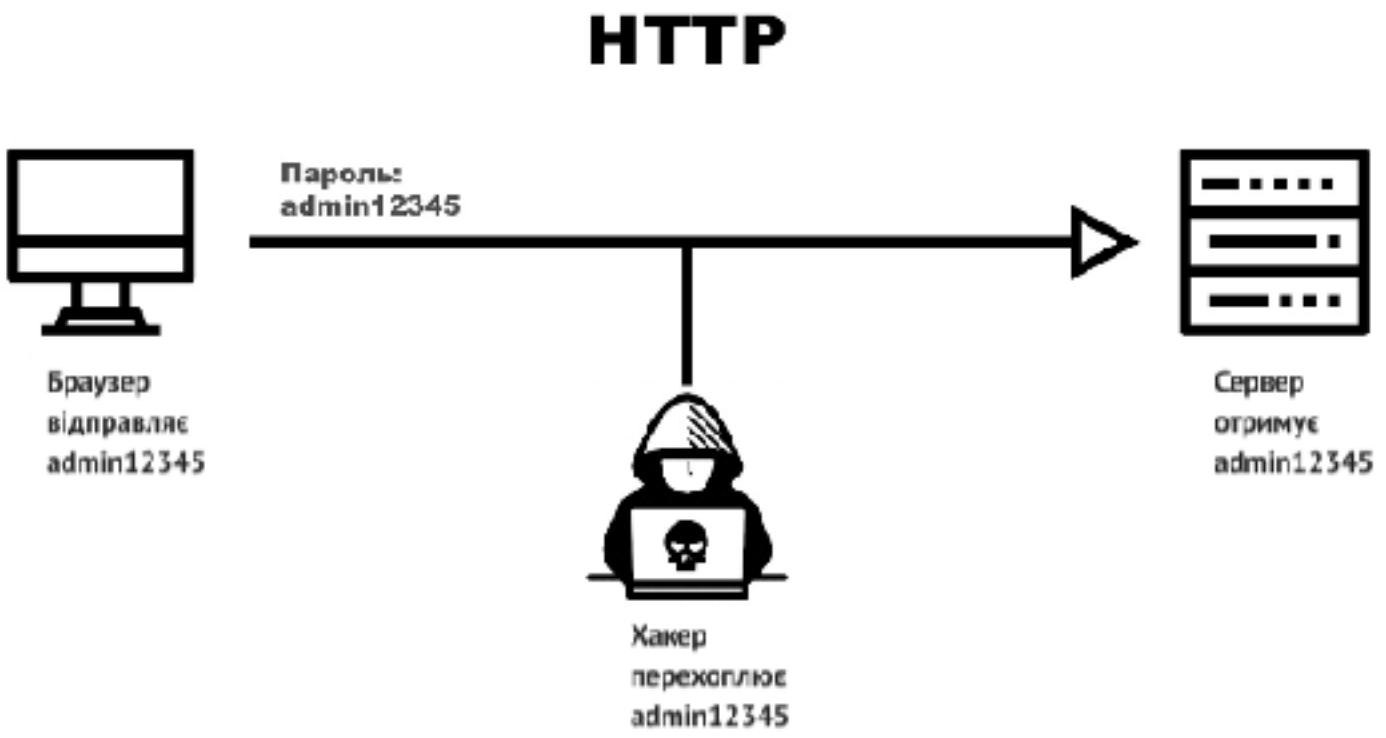


Рис. 4.5. Різниця між HTTP та HTTPS [2]

Як підключити HTTPS на сайті

За замовуванням браузери спілкуються з сервером по HTTP. Щоб з'єднання стало безпечним, власник сайту має встановити SSL-сертифікат на хостинг. Коли власник сайту встановлює SSL-сертифікат, в адресному рядку браузера з'являється піктограма замка і HTTP змінюється на HTTPS. Це означає, що на сайті безпечно вводити особисту інформацію.

Рис. 4.6. Відмінність HTTP від HTTPS в адресному рядку браузера [2]

4.2. РЕЛІЗ САЙТУ. SEO-ПРОСУВАННЯ САЙТУ

4.2.1. *Теоретичний модуль*

Після того як web-додаток розроблений настає час у його випуску у Internet. Цей етап отримав назву реліз (англ. Release - випуск) - випуск остаточної версії програми - готового для використання продукту. У релізі зазвичай випускають кінцевий web-додаток з усіма виправленнями, після яких його не потрібно оновлювати, так як він є найновішою версією [3].

Дії з виконання релізу складаються з двох частин:

1. Обрання та замовлення доменного імені (домену). Домен необхідний для того, щоб користувач міг швидко зорієнтуватися у Internet і знайти web-додаток. Реєстрація доменних імен - це процедура ідентифікації конкретного web-ресурсу. Для реєстрації доменних імен існують платні Internet сервіси, наприклад, популярні в Україні: freehost.com.ua та thehost.com.ua.

2. Розміщення останньої версії web-додатку на хостингу. Завдання хостингу – надавати цілодобовий доступ користувачам до вашого web-додатку на Internet-сервері. Тому, такий доступ (англ. hosting - приймати гостей) називається хостингом. Ресурси на серверах, клієнтські файли, бази даних, інформацію та інше зберігає хостингова компанія - провайдер. Прикладами хостингових провайдерів є thehost.com.ua та heroku.com.

Особливі дії релізу полягають у підготовці web-додатку до просування у Internet. Під просуванням розуміється комплекс дій, метою яких є підвищення показників відвідуваності web-ресурсу цільовими користувачами. Просування охоплює збір семантики та SEO-оптимізацію.

Підготовка семантичного ядра web-додатку є важливим з огляду на те, що пошукові системи, такі як Google постійно вдосконалюють алгоритми та подають лише ті сторінки, що максимально відповідають запитам користувача. Створення широкого семантичного ядра потрібно для потрапляння web-додатку до топу за результатами пошукової видачі Google. Наприклад, якщо користувач

розробленого нами web-додатку шукає «церкви та костьоли», то на сайті обов'язково повинна бути така окрема сторінка. Тобто, сторінки можуть створюватись спеціально відповідно до кожної групи ключових запитів. Систематизація ключових слів та запитів, що пов'язані між собою пропонуються об'єднувати в групи (кластери) семантичного ядра, що дозволяє сформувати структуру, яка міститиме сотні та тисячі пошукових запитів користувачів мережі та буде впорядкованою. Структура ієрархічного взаємозв'язку всіх сторінок може будуватись у вигляді піраміди, на вершині якої розташована головна сторінка.

Важливо ще з самого початку проектування приділити увагу SEO-оптимізації сайту. До оптимізації web-додатку входить оптимізація всіх сторінок згідно його структури, написання контенту, прописання всіх необхідних мета-тегів, атрибутів alt в зображеннях, а також підбір зображень й збереження релевантності на сторінках [4].

SEO-просування сайту [5]

SEO – абревіатура, що розшифровується як Search Engine Optimization (укр. - пошуковий рушій оптимізації). SEO - це комплекс дій щодо внутрішньої та зовнішньої оптимізації, спрямованих на підвищення позицій сайту в пошукових системах, наприклад Google.

Насамперед визначте, з якою метою проводиться SEO-оптимізація та яким чином вимірюватиметься її ефективність. Наприклад, SEO-просування може мати на меті:

- виведення сайту у топ певним пошуковим запитам;
- збільшення трафіку ресурсу;
- покращення позицій сайту у пошуку щодо проектів конкурентів тощо.

Для SEO-просування сайту дуже важливим є який контент повинно просувати. Тому, слід визначити той контент та ті web-сторінки, які є найбільш актуальними та важливими. Завдяки їхній першочерговій розкрутці web-додаток почне приносити прибуток швидше.

Для успішного SEO-просування потрібно скласти план дій зі списком усіх необхідних робіт та зазначенням їхнього пріоритету:

- черговості оптимізації сторінок сайту;
- порядок додавання контенту на сайт;
- ресурси, у яких можна розмістити посилання на свій проект тощо.

Етапи пошукового просування сайтів

1. Аналіз ніші та аудит конкурентів. Необхідно вивчити сайти подібної тематики на високих позиціях. Зверніть увагу на їх структуру, юзабіліті, кількість донорів, відвідуваність, вік, просування у соцмережах.

Паралельно необхідно проаналізувати нішу загалом. Звідки варто чекати на максимальний трафік, які додаткові канали задіяти? Для аналізу конкурентів можна використовувати Serpstat, Seolib, Similarweb та подібні сервіси.

Щоб швидко знайти конкурентів у пошуковій видачі, можна використовувати інструмент Serpstat «Конкуренти». Просто введіть адресу свого домену в пошуковий рядок, і сервіс покаже релевантних суперників у видачі Google на основі семантики.

2. Технічний аудит сайту. Якщо web-додаток вже існує, потрібно перевірити його поточний стан та оцінити поступ за минулий період. Нового web-додатку немає історії оптимізації, тому потрібно його підключити до інструментів web-аналітики (Google Analytics, Google Search Console).

На цьому етапі слід проаналізувати поточні позиції сайту, кількість ключових фраз, що використовуються, джерела трафіку. Також необхідно перевірити сайт на технічні помилки, Визначити швидкість завантаження сторінок, дізнатися, з яких пошуковиків приходить трафік, чи є грубі порушення, які можуть гальмувати просування. За потреби проводиться правильне настроювання технічних файлів та інших параметрів, що впливають на індексацію та ранжування сайту.

Щоб швидко перевірити сайт на наявність помилок, використовуйте модуль Serpstat "Аудит сайту". Він знайде недоліки та запропонує рекомендації щодо їх виправлення.

3. Складання семантичного ядра. Підбір максимальної кількості релевантних ключових фраз допомагає збільшити надходження відвідувачів (трафік). Немає оптимальної кількості ключових запитів, для кожного проекту вони підбираються індивідуально.

Зібрати семантичне ядро можна за допомогою Serpstat, Key Collector та подібних сервісів.

4. Опрацювання структури сайту. Залежно від зібраного ядра, необхідно створити або оновити існуючу структуру. У зв'язку з цим можуть бути додані нові розділи та сторінки каталогів. Для створення структури сайту може допомогти інструмент «Кластеризація». Він розіб'є зібрані фрази на кластери під окремі сторінки, категорії та розділи. Вам залишиться лише швидке редагування, щоб підчистити результат.

5. Наповнення сторінок контентом. Він має бути унікальним та максимально корисним для відвідувачів. Кожна сторінка повинна бути зорієнтоване на конкретний ключовий запит або групу запитів.

6. Внутрішня перелінковка. За наявності текстового та графічного контенту необхідно додати посилання, що ведуть на внутрішні сторінки сайту, щоб збільшити час перебування на сайті та глибину перегляду.

7. Зовнішня оптимізація. На цьому етапі необхідно скласти план поступового отримання посилань із сайтів-донорів, які попередньо перевіряються на авторитетність. Проаналізувати стратегії посилань конкурентів можна за допомогою «Аналізу посилань» Serpstat. Знаходимо релевантні майданчики та намагаємося отримати посилання з них.

8. Просування у соціальних мережах. Створення груп та профілів, відео-каналу підвищує впізнаваність вашого сайту, бренду, товару та опосередковано допомагає у пошуковому просуванні.

Для досягнення SEO можна використовувати обидва протоколи (HTTP і HTTPS) мають свої переваги та недоліки для пошукового просування [6].

Якщо до HTTP додати літеру “S”, ми отримаємо HTTPS. Протоколи передачі гіпертексту використовуються обмінюватись інформацією в мережі. Обидва позначення можна визначити у пошуковому рядку браузера та на початку url-адреси сайту.

Розглянемо переваги HTTPS над HTTP та їх відмінності [6].

Одна з основних відмінностей HTTP від HTTPS - це його простота та зручність застосування. Для коректного функціонування HTTP не потрібний потужний процесор і великий обсяг пам'яті, тому що йому потрібно менше одноразових з'єднань порівняно з безпечнішими протоколами. Запити та відповіді передаються без очікування, що дозволяє обробляти великий обсяг даних. HTTP знижує навантаження на мережу, оскільки йому потрібно менше TCP пакетів (TCP - це протокол керування передачею даних). Обмін сигналами між комп'ютерами (також званий TCP-рукостисканням) відбувається без затримок, щойно встановлено з'єднання. HTTP не перериває TCP-з'єднання - він лише повідомляє про помилки у процесі. У деяких обставинах це зручніше.

Відмінності проколів HTTP та HTTPS не завжди говорять на користь першого: він програє у безпеці та можливостях.

Щодо безпечності з'єднання за протоколами. HTTP допускає двоточкове з'єднання, внаслідок чого важко забезпечити безпеку даних на сайті.

HTTPS розроблявся для забезпечення безпеки під час здійснення онлайн-платежів та передачі конфіденційної інформації через Internet. Цей протокол використовувався банками та великими корпораціями, однак з 2016 року він набув ширшого поширення. Тепер він застосовується повсюдно і ми все частіше можемо бачити його в URL сайтів.

4.2.2. ***Практичний модуль***

У практичній частині продовжуємо роботу з web-додатком Tours City (екскурсії містом), що був проектований та створений у попередніх розділах посібника. Виконаємо реліз web-додатку за методикою поданою в [6, с.110].

Етап 4.1. Розміщення релізу web-додатку на хостингу Heroku (https://www.heroku.com)

Дія 4.1.1. Перш ніж використовувати Heroku, вам потрібно зареєструвати безкоштовний обліковий запис і встановити Heroku CLI на комп'ютері, який використовували для розробки.

- платформа Heroku може виконувати програми, засновані на різних типах баз вихідних текстів, тому потрібно повідомити їй, який саме додаток ми хочемо виконати. Крім інформації про те, що ми запускаємо програму Node, що використовує npm як систему управління пакетами, потрібно повідомити їй, яку версію ми запускаємо, щоб гарантувати відповідність налаштувань для версії релізу та версії, призначеної для розробки. На сучасному етапі починаючи з 28 листопада 2022 року платформа Heroku відмінила безкоштовний тариф.

Визначаємо версії розробки:

```
$ node --version
$ npm --version
```

Дія 4.1.2. Для зручності роботи встановлюємо (якщо ще не встановлена) систему контролю версій Git

- якщо працюєте у Windows, то знадобиться завантажити сумісний з GitHub термінал bash, що постачається в якості частини програми GitHub для настільних комп'ютерів. Як тільки ви все встановите та налаштуєте, ми зможемо продовжити та підготувати додаток до викладання в Internet.

Дія 4.1.3. Прописуємо завантаження потрібних версій Node та npm, що були визначені попередньо. Вам потрібно додати ці версії до нового розділу engines у файлі package.json, використовуючи вже зустрічався вам префікс ~, що додає джокерний символ для патч-версії.

У файл package.json вводимо розділ engines. Код повинен виглядати так:

```
"scripts": {
  "start": "node app.js",
  "server": "nodemon app.js",
  "client": "npm run start --prefix client",
  "dev": "concurrently \"npm run server\" \"npm run client\""
 },
" engines ": {
  "node": " ~20.2.0 ",
  "npm": "~9.6.6"
 },
```

При надсиланні на Heroku ці рядки повідомлять йому, що програма використовує останню патч-версію Node 20.2 та останню патч-версію npm 9.6.6.

Дія 4.1.4. Створення Procfile.

- файл package.json повідомляє Heroku, що йдеться про додаток Node, але не повідомляє, як його запустити. Для цього нам знадобиться Procfile. Procfile використовується для оголошення типів процесів, що використовуються додатком, і застосовуються для їх запуску команд. Web-додаток потребує процесу, який буде запускати програму Node.

- у кореневому каталозі програми файл під назвою Procfile - з урахуванням регістру, розширення файлу відсутнє. Введіть до Procfile наступний рядок:

web: npm run dev

При надсиланні на Heroku цей файл повідомить йому, що програма потрібна web-процес і що необхідно виконати команду npm run dev.

Дія 4.1.5. Виконуємо локальне тестування за допомогою Heroku CLI.

- Heroku CLI поставляється як утиліта. Її можна використовувати для перевірки наших налаштувань та локального запуску програми до розміщення на Heroku. Якщо програма вже запущена, зупиніть її, натиснувши Ctrl+C у вікні терміналу, у якому запущено процес. Потім введіть у вікні термінала наступну команду:

foreman start

Якщо з налаштуваннями все нормально, вона знову запустить додаток на локальній машині.

Тепер, коли ми знаємо, що наші налаштування діють, настає час відредагувати програми на Heroku.

Дія 4.1.6. Запускаємо сайт в Internet за допомогою Git.

Як метод розгортання Heroku використовує Git.

Зберігаємо web-додаток в Git на локальній машині. Цей процес складається із трьох кроків.

1. Задати каталог програми як репозиторій Git.
2. Повідомити Git, які файли потрібно додати до репозиторію.
3. Зафіксувати ці зміни у репозиторії.

Для кожного кроку вам знадобиться всього одна проста команда терміналу. Якщо програма запущена локально, зупиніть її в терміналі, натиснувши Ctrl+C. Потім, переконавшись, що ви ще знаходитесь у кореневому каталозі програми, виконайте в терміналі такі команди:

```
git init
git add .
git commit –m ‘web-app add’
```

Ці три дії разом утворять локальний репозиторій Git, що містить web-додаток. Коли ми у процесі супроводу модернізуватимемо додаток і будемо викладати зміни в Internet, то будемо використовувати дві останні команди з іншим повідомленням для оновлення репозиторію.

Локальний репозиторій готовий. Створюємо програму Heroku.

Дія 4.1.7. Наступним кроком ми створимо додаток на Heroku як віддалений репозиторію для нашого локального репозиторію. Все це виконується за допомогою команди терміналу:

$ heroku create

Після її виконання ви побачите в терміналі підтвердження URL, за яким буде розташовуватися додаток, адреса репозиторію Git і найменування віддаленого репозиторію, наприклад:

http://shrouded-tor-1673.herokuapp.com/ |

git@heroku.com:shrouded-tor-1673.git

Git remote heroku added

Якщо ви увійдете до свого облікового запису Heroku у браузері, то побачите також, що там з'явився додаток. Отже, у вас тепер є контейнер Heroku для програми, і наступним кроком буде відправлення туди коду програми.

Дія 4.1.8. Розгортання програми на Heroku.

На даний момент є web-додаток, що зберігається в локальному репозиторії Git, і ви створили новий віддалений репозиторій на Heroku. Віддалений репозиторій поки що порожній, так що вам потрібно відправити вміст вашого локального репозиторію у віддалений репозиторій Heroku.

У Git є окрема команда для виконання цієї дії з наступною логічною структурою:

git push heroku master

Ця команда відправить вміст локального репозиторію Git у віддалений репозиторій Heroku. Поки що у вашій репозиторії є лише одна гілка, а саме гілка master, яку ви і відправляєте на Heroku.

Коли ви виконуєте наведену команду, термінал по ходу процесу відображає масу журнальних повідомлень, завершуючи все підтвердженням того факту, що програма була розгорнута на Heroku. Це буде виглядати приблизно таким чином, за винятком того, що URL виявиться іншим:

http://shrouded-tor-1673.herokuapp.com deployed to Heroku

Дія 4.1.9. Запуск динамічного web-контейнера на Heroku

Heroku використовує поняття динамічних контейнерів для запуску та масштабування програми. Чим більше у вас динамічних контейнерів, тим більше системних ресурсів та процесів доступно вашому додатку. Додавати нові динамічні контейнери при збільшенні вашої програми та зростанні її популярності дуже просто.

У Heroku є чудовий безкоштовний пакет, що ідеально підходить для створення прототипу програми та її пробної версії. Ви отримуєте один динамічний web-контейнер безкоштовно, чого більш ніж достатньо для наших цілей.

Перш ніж ви зможете побачити програму онлайн, необхідно додати один динамічний web-контейнер. Це легко можна зробити за допомогою команди термінала:

heroku ps:scale web=1

При її виконанні термінал виведе підтвердження:

caling web dynos... done, now running 1

Тепер перевіримо URL-адресу в Internet.

Дія 4.1.10. Перегляд програми URL у Internet.

Все вже на своїх місцях і програма викладена в Internet. Ви можете переконатися в цьому, ввівши отриманий підтвердження URL через ваш обліковий запис в Heroku або за допомогою наступної команди терміналу:

heroku open

Дія 4.1.11. Звичайно, ваш URL буде іншим, і в Heroku ви можете змінити його, щоб використовувати замість отриманої від Heroku адреси своє доменне ім'я.

У налаштуваннях програми на сайті Heroku ви також можете поміняти його на більш обдуманий піддомен сайту, наприклад, herokuapp.com.

4.3. ТЕСТОВІ ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ

1. HTTP – це
   a) протокол передавання гіпертексту
   b) протокол передавання тексту
   c) протокол клієнт-серверної взаємодії
   d) локальний протокол
2. HTTP є протоколом
   a) прикладного рівня
   b) мережевого рівня
   c) канального рівня
   d) транспортного рівня
3. HTTP посередники, які виконують різні операції і працюють як шлюзи або кеш
   a) proxy
   b) request
   c) response
   d) user-agent
4. HTTP сторона - учасник обміну
   a) proxy
   b) request
   c) response
   d) user-agent
5. HTTP запит
   a) proxy
   b) request
   c) response
   d) user-agent
6. HTTP відповідь
   a) proxy
   b) request
   c) response
   d) user-agent

7. Основна різниця між HTTP та HTTPS
   a) інформація передається у зашифрованому вигляді
   b) інформація передається у звичайному вигляді
   c) інформація передається у вигляді JSON
   d) інформація передається у текстовому вигляді
8. HTTPS – щоб з'єднання стало безпечним, власник сайту має на хостингу
   a) встановити SSL-сертифікат
   b) визначити метод шифрування
   c) обрати протокол клієнт-серверної взаємодії
   d) немає відповіді
9. Випуск остаточної версії програми – це
   a) друк
   b) реліз
   c) монтування
   d) просування у Internet
10. Просування у Internet охоплює дві діяльності
   a) SEO-оптимізація
   b) обрання протоколу
   c) збір семантики
   d) обрання прикладного рівня
11. Для SEO-просування сайту є найбільш важливим
   a) який контент повинно просувати
   b) скільки переглядів
   c) обрання канального рівня
   d) обрання транспортного рівня
12. Аудит конкурентів сайту можна виконати з використанням
   a) Word
   b) Engines
   c) Serpstat
   d) Browser

13. Технічний аудит сайту для SEO передбачає
    a) аналіз поточні позиції сайту
    b) підбір максимальної кількості релевантних ключових фраз
    c) необхідно додати посилання на внутрішні сторінки сайту
    d) необхідно отримати посилання від сайтів-донорів
14. Складання семантичного ядра сайту для SEO передбачає
    a) аналіз поточні позиції сайту
    b) підбір максимальної кількості релевантних ключових фраз
    c) необхідно додати посилання на внутрішні сторінки сайту
    d) необхідно отримати посилання від сайтів-донорів
15. Внутрішня перелінковка сайту для SEO передбачає
    a) аналіз поточні позиції сайту
    b) підбір максимальної кількості релевантних ключових фраз
    c) необхідно додати посилання на внутрішні сторінки сайту
    d) необхідно отримати посилання від сайтів-донорів

4.4. ЗАВДАННЯ ДО САМОСТІЙНОЇ РОБОТИ

1. Ознайомитись із принципами та концепціями HTTP.
2. Ознайомитись із принципами та концепціями HTTPS.
3. Ознайомитись з прийомами використання сервісу аналізу SEO-оптимізації web-додатків Serpstat.
4. Ознайомитись з прийомами використання сервісу аналізу SEO-оптимізації web-додатків Seolib.
5. Ознайомитись з прийомами використання сервісу аналізу SEO-оптимізації web-додатків Similarweb.

ВИКОРИСТАНІ ДЖЕРЕЛА

1. An overview of HTTP. Mdn web docs. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (дата звернення: 23.05.2023).
2. Чим відрізняється HTTP від HTTPS. *Блог про безпеку в інтернеті і не тільки*. https://ssl.com.ua/blog/ukr/http-vs-https (дата звернення: 23.05.2023).
3. Реліз (програмне забезпечення) - це. URL: https://jak.koshachek.com/articles/reliz-programne-zabezpechennja-ce.html (дата звернення: 23.05.2023).
4. Як організувати реліз. HABR. URL: https://habr.com/ru/articles/481964 (дата звернення: 23.05.2023).
5. Що таке SEO-просування сайту: основи SEO. SERPSTAT. URL: https://serpstat.com/ru/blog/chto-takoe-seo-prodvizhenie-sajta (дата звернення: 23.05.2023).
6. У чому різниця між HTTP та HTTPS протоколами. DIRECTLINE. URL: https://directline.digital/blog/raznica-mezhdu-http-i-https-protokolami/ (дата звернення: 23.05.2023).
7. Simon Holmes. Getting MEAN with Mongo, Express, Angular, and Node. Manning Publications, 2016 - 416p.

КЛЮЧІ ДО ТЕСТОВИХ ЗАВДАНЬ

Розділ 1.
1 (a); 2 (d); 3 (a); 4 (c); 5 (a); 6 (a); 7 (a); 8 (c); 9 (a); 10 (c); 11 (a);
12 (d); 13 (b); 14 (a, b); 15 (c, d); 16 (npm); 17 (a); 18 (package.json);
19 (b); 20 (dependencies); 21 (c); 22 (d); 23 (c); 24 (d); 25 (b); 26 (a);
27 (d); 28 (a); 29 (a); 30 (mongoose).

Розділ 2.
1 (a); 2 (b); 3 (c); 4 (d); 5 (b); 6 (a); 7 (a); 8 (b); 9 (c); 10 (a); 11 (b);
12 (c); 13 (d); 14 (a); 15 (b); 16 (c); 17 (d); 18 (a); 19 (b); 20 (c); 21 (d);
22 (a); 23 (b); 24 (c); 25 (d); 26 (d); 27 (b); 28 (a); 29 (b); 30 (c).

Розділ 3.
1 (a); 2 (a); 3 (a); 4 (a); 5 (a); 6 (a); 7 (b); 8 (d); 9 (d); 10 (a); 11 (a);
12 (a); 13 (a, b); 14 (a, b); 15 (a); 16 (b); 17 (c); 18 (a); 19 (a); 20 (b);
21 (c); 22 (b); 23 (b); 24 (d); 25 (c).

Розділ 4.
1 (a, c); 2 (a); 3 (a); 4 (d); 5 (b); 6 (c); 7 (a); 8 (a); 9 (b); 10 (a, c);
11 (a); 12 (c); 13 (a); 14 (b); 15 (c).

НАВЧАЛЬНЕ ВИДАННЯ

Тулашвілі Юрій Йосипович

WEB-ПРОГРАМУВАННЯ

MERN fullstack development

Навчальний посібник
для студентів ІТ-спеціальностей

Комп'ютерний набір
та верстка: Юрій Йосипович Тулашвілі

Коректор: Оксана Володимирівна Семенюк

Дизайнер обкладинки: Олександра Ігорівна Тулашвілі

Підп. до друку 15.06.2023. Формат 60x84/16. Папір офсетний.
Гарн. Таймс. Ум. друк. арк. . Обл.-вид. арк. 19.
Тираж 300 прим. Зам. №