

О. В. ВЛАСЕНКО, Ю. М. ЗДОРЕНКО, В. В. ФЕСЬОХА

Технологія розробки програмного забезпечення

Навчальний посібник



Київ – 2021

МІНІСТЕРСТВО ОБОРОНИ УКРАЇНИ

**ВІЙСЬКОВИЙ ІНСТИТУТ ТЕЛЕКОМУНІКАЦІЙ
ТА ІНФОРМАТИЗАЦІЇ ІМЕНІ ГЕРОЇВ КРУТ**

О. В. Власенко, Ю. М. Здоренко, В. В. Фесьоха

**ТЕХНОЛОГІЯ РОЗРОБКИ
ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

Навчальний посібник

Київ – 2021

УДК 004.738.5 (075.8)

В 58

О. В. Власенко, Ю. М. Здоренко, В. В. Фесьоха. **Технологія розробки програмного забезпечення:** Навчальний посібник – К.: BITI, 2021. – 148 с.

У навчальному посібнику розглянуті поняття щодо технології створення програмного забезпечення. У якості засобів розробки пропонується використання сучасної мови програмування JavaScript. Матеріал посібника зосереджений на можливостях сучасної технології Node.js для реалізації серверної частини додатку та браузерного інтерпретатора JavaScript для виконання коду на клієнтській стороні.

Навчальний посібник призначений для курсантів (студентів), які вивчають дисципліни «Web-технології та Web-дизайн», «Технологія розробки програмного забезпечення», а також виконують курсові та кваліфікаційні роботи освітньо-кваліфікаційного рівня «бакалавр».

Рекомендовано до друку Вченого радою Військового інституту телекомунікацій та інформатизації імені Героїв Крут (протокол № 2 від 27 жовтня 2021 року).

Рецензенти:

O. Я. Сова – доктор технічних наук, начальник кафедри Військового інституту телекомунікацій та інформатизації імені Героїв Крут;

O. В. Сілко – кандидат технічних наук, заступник начальника Військового інституту телекомунікацій та інформатизації імені Героїв Крут з навчальної роботи – начальник навчальної частини.

© Військовий інститут телекомунікацій та інформатизації імені Героїв Крут, 2021

ЗМІСТ

ВСТУП	5
РОЗДІЛ 1. ОСНОВИ РОЗРОБКИ ДОДАТКІВ НА БАЗІ ТЕХНОЛОГІЙ NODE.JS.....	7
1.1. Основні підходи до розробки програмного забезпечення.....	7
1.2. Події в платформі Node.js. Цикл подій. Клас EventEmmiter.....	16
1.3. Основні вбудовані модулі Node.js.....	26
<i>Контрольні запитання до розділу 1.....</i>	55
РОЗДІЛ 2. СТВОРЕННЯ СЕРВЕРНИХ ПРОГРАМНИХ ДОДАТКІВ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЙ NODE.JS.....	56
2.1. Асинхронне програмування та багатопоточність в Node.js.....	56
2.2. Створення серверних веб-додатків на основі фреймворку Express.js.....	73
2.3. Характеристики RESTful API.....	95
<i>Контрольні запитання до розділу 2.....</i>	102
РОЗДІЛ 3. ОСНОВИ РОБОТИ З БАЗАМИ ДАНИХ В NODE.JS	103
3.1. Загальна характеристика та можливості засобів Node.js для роботи з СКБД MongoDB	103
3.2. Основні можливості та особливості роботи з СКБД MySQL	112

3.3. Кешування на рівні додатку з використанням СКБД Redis.....	119
<i>Контрольні запитання доrozділу 3.....</i>	126
РОЗДІЛ 4. РОЗШИРЕНОІ МОЖЛИВОСТІ ПЛАТФОРМИ NODE.JS ДЛЯ СТВОРЕННЯ СЕРВЕРНИХ ВЕБ-ДОДАТКІВ	127
4.1. Архітектурний патерн MVC в Node.js	127
4.2. Поняття сокету і Вебсокету. Особливості бібліотеки Socket.io.	134
4.3. Фреймворк Mocha. Тестування серверних додатків.....	139
<i>Контрольні запитання доrozділу 4.....</i>	144
БІБЛІОГРАФІЧНИЙ СПИСОК.....	146
ПРЕДМЕТНИЙ ПОКАЖЧИК.....	148

ВСТУП

Різноманітність підходів щодо створення програмних продуктів показує, що задача вибору правильного шляху розробки це складна і трудомістка робота, яка вимагає високої кваліфікації та досвіду фахівців, що беруть у ній участь. Однак до теперішнього часу створення програмних продуктів нерідко виконується на інтуїтивному рівні із застосуванням неформалізованих методів, заснованих на наявному практичному досвіді, експертних знаннях розробників та не завжди є оптимальним. Основним аргументом, що спонукав використати програмну платформу Node.js було забезпечення свободи розробника у створенні програмних продуктів, що здатні виконуватися на пристроях різних типів і в різних середовищах.

Запропонований авторами практичний підхід до викладу матеріалу не є простим повторенням документації Node.js, а дозволяє сконцентруватися на вирішенні конкретних завдань, пов'язаних з розробкою програмних додатків різного рівня складності.

У першому розділі розглянуті основні підходи розробки програмного забезпечення, показані основи проєктування інформаційних систем на базі послідовності стадій та процесів життевого циклу розробки програмного забезпечення. Наведена загальна характеристика платформи Node.js. Детально розглянута подійно-орієнтована модель обслуговування в даній платформі. Подана характеристика основних вбудованих модулів

платформи Node.js, їх взаємозв'язок та особливості практичного використання при створені власних додатків.

У другому розділі детально розглянуто особливості розробки програмного забезпечення на основі асинхронного програмування та багатопоточності, особливості їх використання в Node.js. У розділі міститься інформація про основні можливості серверного фреймворку Express.js, а також про те, як з його допомогою організувати роботу серверу.

Третій розділ присвячено основам роботи з найбільш популярними базами даних використовуючи платформу Node.js. Розглянуто основні прийоми з'єднання з такими системами керування базами даних (СКБД), як MySQL та MongoDB, порядок виконання запитів на вибірку, оновлення та видалення даних. Також розглянуті питання кешування даних з використанням СКБД Redis.

Четвертий розділ присвячено розширеним можливостям платформи Node.js. В розділі показані принципи розробки додатку з використанням архітектурного паттерну MVC, розглянуті поняття сокету і Вебсокету, особливості бібліотеки Socket.io, а також порядок тестування серверних додатків з використанням фреймворку Mocha.

Автори висловлюють подяку рецензентам доктору технічних наук О. Я. Сові та кандидату технічних наук О. В. Сілку за цінні зауваження та рекомендації, які сприяли поліпшенню цього видання і позбавленню ряду помилок.

ОСНОВИ РОЗРОБКИ ДОДАТКІВ НА БАЗІ ТЕХНОЛОГІЙ NODE.JS

1.1. Основні підходи до розробки програмного забезпечення

Типова структура повного життєвого циклу створення та впровадження програмного забезпечення передбачає ряд етапів, які потребують типового розподілу функцій між спеціалістами в ІТ-галузі [2]. Загальне керівництво здійснюється керівником проекту, який забезпечує взаємодію між замовником та безпосередніми виконавцями. Типовий розподіл функцій передбачає наявність таких виконавців, як: аналітик (для здійснення предпроектного аналізу), архітектор, розробник, тестувальник, менеджер з впровадження та підтримки. Безпосереднє створення програмного продукту включає ряд етапів, серед яких основними є: проектування, розробка та контроль якості і тестування програмного рішення. Кожен з перелічених виконавців має ряд специфічних функцій на зазначених етапах створення програмного продукту.

Технологія розробки програмного забезпечення (ТРПЗ) – це сукупність процесів і методів створення програмного продукту [1].

Життєвий цикл програмного забезпечення (software development life cycle, SDLC) – період часу, який

починається з моменту прийняття рішення про необхідність створення програмного продукту і закінчується в момент його повного виводу з експлуатації (Рис.1.1).

Життєвий цикл програмного забезпечення можна представити у вигляді моделей [1]. В даний час найбільш поширеними моделями для цього є:

- каскадна;
- поетапна модель з проміжним контролем;
- спіральна модель;
- змішана модель.

Етапи життєвого циклу програмного забезпечення для всіх зазначених моделей включають:

- аналіз вимог;
- підготовка специфікацій (технічного завдання);
- проєктування;
- розробка (безпосередня реалізація);
- тестування та налагодження;
- інтеграція (введення в експлуатацію).
- супроводження;
- вивід із експлуатації.



Рис. 1.1. Життєвий цикл програмного забезпечення

Розглянемо особливості зазначених моделей життєвого циклу програмного забезпечення.

Каскадна модель (англ. *Waterfall model*) – модель процесу розробки програмного забезпечення, життєвий цикл якої виглядає як потік, що послідовно проходить фази аналізу вимог, проєктування, реалізації, тестування, інтеграції та підтримки (Рис.1.2).

Процес розробки реалізується за допомогою впорядкованої послідовності незалежних кроків.

Дана модель передбачає, що кожен наступний крок починається після повного завершення виконання попереднього кроку.

На всіх етапах моделі виконуються допоміжні та організаційні процеси і роботи, що включають управління проектом, оцінку і управління якістю, верифікацію і атестацію, управління конфігурацією, розробкою документації.

В результаті завершення кожного етапу формуються проміжні результати, які не можуть змінюватися на наступних кроках.



Рис. 1.2. Каскадна модель розробки програмного забезпечення

Переваги каскадної моделі:

- стабільність вимог протягом усього життєвого циклу розробки;
- на кожній стадії формується закінчений набір проектної документації, який відповідає критеріям повноти і узгодженості;

- визначеність і зрозумілість кроків моделі і простота її застосування;
- виконувані в логічній послідовності етапи робіт дозволяють планувати терміни завершення всіх робіт і відповідні ресурси (грошові, матеріальні і людські).

Каскадна модель [1] добре зарекомендувала себе при побудові відносно простого програмного забезпечення, коли на самому початку розробки можна досить точно і повно сформулювати всі вимоги до продукту.

Недоліки каскадної моделі:

- складність чіткого формулювання вимог і неможливість їх динамічної зміни протягом життєвого циклу;
- низька гнучкість в управлінні проєктом;
- непридатність проміжного продукту для використання;
- пізнє виявлення проблем, пов'язаних зі збіркою проєкту, в зв'язку з одночасною інтеграцією всіх результатів в кінці розробки;
- недостатня участь користувача в створенні системи, а тільки, – на самому початку (при розробці вимог) і в кінці (під час приймальних випробувань);
- користувачі не можуть переконатися в якості продукту, що розробляється до закінчення всього процесу розробки. Вони не мають можливості оцінити якість, через те, що не можна побачити готовий продукт розробки.

Поетапна модель з проміжним контролем являє підхід з розробки програмного забезпечення з лінійної по послідовністю стадій, але в кілька інкрементів (версій), тобто із запланованим удосконаленням програмного продукту за весь час поки життєвий цикл розробки ПЗ не завершиться.

Розробка програмного забезпечення ведеться ітераціями з циклами зворотного зв'язку між етапами. Міжетапні коригування дозволяють враховувати реально існуючий взаємоплив результатів розробки на різних етапах, час життя кожного з етапів розтягується на весь період розробки.

Опис життєвого циклу в відповідності до даної моделі є характерним при розробці складних і комплексних систем, для яких є чітке бачення (як з боку замовника, так і з боку розробника) того, що собою має представляти кінцевий результат (Рис.1.3).

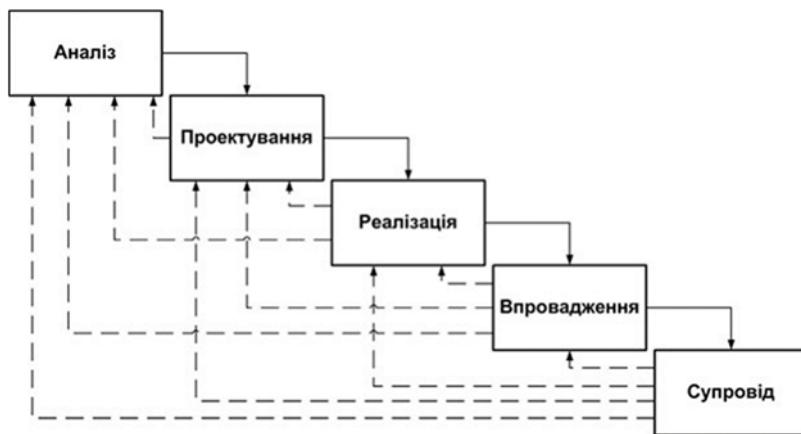


Рис. 1.3. Поетапна модель з проміжним контролем

Розробка по версіям ведеться виходячи з різних причин, наприклад:

- відсутності у замовника можливості відразу профінансувати весь дорогий проект;
- відсутності у розробника необхідних ресурсів для реалізації складного проекту в стислі терміни;
- вимог поетапного впровадження і освоєння продукту кінцевими користувачами. Впровадження всієї системи відразу може викликати у її користувачів неприйняття і тільки "загальмувати" процес переходу на нові технології.

Переваги і недоліки цієї моделі (стратегії) подібні, *каскадній* (класичної моделі життєвого циклу) моделі. Але на відміну від класичної стратегії замовник може раніше побачити результати. Уже за результатами розробки та впровадження першої версії він може дещо змінити вимоги до розробки, відмовитися від неї або запропонувати розробку більш досконалого продукту з укладенням нового договору.

Разом з тим модель не дозволяє оперативно враховувати виникаючі зміни і уточнення вимог до ПЗ. Узгодження результатів розробки з користувачами проводиться тільки в точках, планованих після завершення кожного етапу робіт, а загальні вимоги до ПЗ зафіксовані у вигляді технічного завдання на весь час її створення. Таким чином,

користувачі часто отримують програмний продукт, що не задовольняє їх реальним потребам.

Ще одна відома модель – *спіральна*. У спіральній моделі життєвий цикл розробки програмного продукту зображується у вигляді спіралі, яка, розпочавшись на етапі планування, розгортається з проходженням кожного наступного кроку.

Таким чином, на виході з чергового витка отримуємо готовий протестований прототип, який доповнює існуючу збірку. Прототип, що задовольняє всім вимогам, готовий до випуску (Рис. 1.4.)

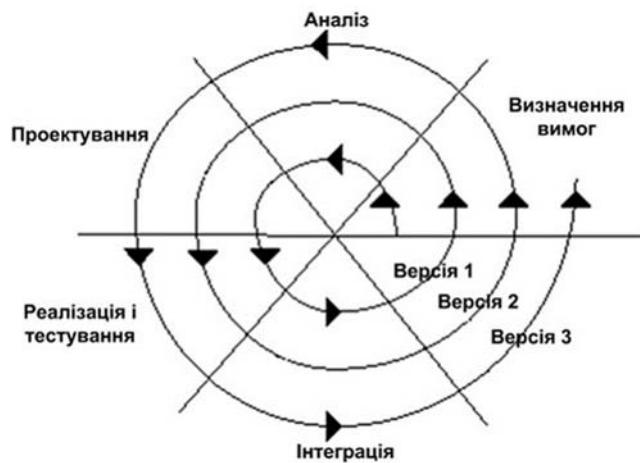


Рис. 1.4. Спіральна модель

Переваги спіральної моделі:

- приділяється особлива увага управлінню ризиками;

- додаткові функції можуть бути додані на пізніх етапах;
- є можливість гнучкого проєктування.

Недоліки спіральної моделі:

- оцінка ризиків на кожному етапі є досить витратною;
- постійні корегування замовника можуть призводити до нових ітерацій та як наслідок до збільшення термінів розробки ПЗ;
- більш підходить для великих (масштабних) проектів.

Гнучка модель (Agile model) являє собою сукупність різних підходів до розробки ПЗ. Вона включає серії підходів до розробки програмного забезпечення, орієнтованих на використання поетапної розробки, динамічне формування вимог і забезпечення їх реалізації в результаті постійної взаємодії всередині самоорганізованих робочих груп, що складаються з фахівців різного профілю. Окрема ітерація являє собою мініатюрний програмний проєкт. Однією з основних ідей *Agile* є взаємодія всередині команди та з замовником безпосередньо.

Переваги гнучкої моделі:

- швидке прийняття рішень за рахунок постійних комунікацій;
- мінімізація ризиків;
- полегшена робота з документацією.

Недоліки гнучкої моделі:

- велика кількість уточнень, що може збільшити час розробки продукту;
- складність довгострокового планування процесів, чере постійну зміну вимог.

Таким чином існує велика кількість різноманітних підходів для розробки програмного забезпечення. Але вибір конкретної з вищезазначених моделей визначається особливостями окремо взятого проекту. Тому на етапі проєктування дуже важливим є всебічний аналіз всіх таких особливостей майбутнього проєкту та можливостей щодо його виконання.

1.2. Основні характеристики Node.js. Події в платформі Node.js. Клас EventEmmiter

Node.js – це серверна платформа JavaScript, призначена для створення масштабних розрізнених мережевих додатків. В основу якої покладена подійно-орієнтована архітектура та асинхронне взаємодія з неблокуючим вводом/виводом [4]. Побудована Node.js на ядрі JavaScript V8 (перетворює JavaScript в машинний код). Є продуктом від компанії Google.

Node.js надає можливість JavaScript взаємодіяти з пристроями вводу-виводу через свій API (написаний на C++), підключаючи інші вбудовані бібліотеки (Рис.1.3), створені на різних мовах та гарантуючи виклики до них з JavaScript-коду.

Розробка Node.js фінансиється компанією Joyent, заснованою Райаном Далом.

V8 – ядро JavaScript з відкритим вихідним кодом, що розробляється данським відділенням компанії Google. Воно написано на C++ і використовується в Google Chrome. Ядро може працювати автономно або бути встановлено в будь-який C++ додаток.

Незважаючи на динамічну природу JavaScript, розробникам вдалося застосувати методи, характерні для реалізації класичних об'єктно-орієнтованих мов, такі як компіляція коду «на льоту», внутрішнє кешування, точний процес складання сміття, снепшотінг при створенні контекстів.

V8 відрізняється від інших «рушіїв» (JScript, SpiderMonkey, JavaScriptCore, Nitro) високою продуктивністю.

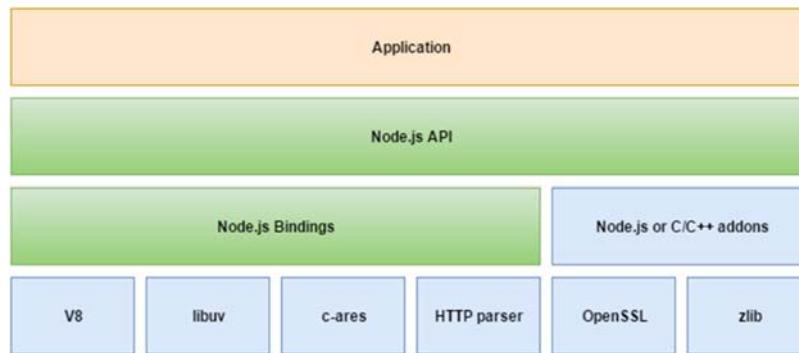


Рис. 1.3. Архітектура Node.js

Платформа Node.js була створена у 2009 в ході досліджень по створенню подійно-орієнтованих серверних систем.

Метою її розробки була необхідність запропонувати «простий спосіб побудови масштабованих мережніх серверів». Сильною стороною Node.js є можливість створення мережевих додатків, що забезпечують швидку роботу з мережевими з'єднаннями і здатність обробляти сотні тисяч одночасних підключень до сервера при невеликому споживанні ресурсів [7].

Типи додатків, які розробляються на Node.js:

- чати і системи обміну миттєвими повідомленнями;
- багатокористувацькі ігри в реальному часі;
- системи одночасної роботи над проєктами;
- мережеві сервіси для збору і відправки інформації.

Так, за допомогою Node.js створена популярна система збирання веб-проєктів Gulp.js.

В ході своїх досліджень Райан Дал прийшов до висновку, що замість традиційної моделі паралелізму на основі потоків слід використати *подійно-орієнтовані системи*.

Подійно-орієнтоване програмування (event-driven programming) – парадигма програмування, в якій виконання програми визначається подіями: подіями користувача (натиски клавіш клавіатури, рухи мишою), повідомленнями інших програм і потоків, подіями

операційної системи (наприклад, надходженням мережевого пакета) та ін [4].

Подійно-орієнтоване програмування можна також визначити як спосіб побудови комп'ютерної програми, при якому в коді (як правило, в головній функції програми) явним чином виділяється основний цикл програми, тіло якого складається з двох частин: отримання повідомлення про подію і обробка події.

Подійно-орієнтоване програмування, як правило, застосовується в таких випадках:

- при побудові користувацьких інтерфейсів (в тому числі графічних);
- при створенні серверних додатків у разі, якщо з них чи інших причин небажаним є породження обслуговуючих процесів;
- при програмуванні ігор, в яких здійснюється управління значною кількістю об'єктів.

Події – це певні дії або випадки, що виникають в програмованій системі, про які здійснюється повідомлення, щоб розробники могли з ними взаємодіяти і на них реагувати.

Так, якщо створюється файл, розробник може передбачити реагування на цю подію, наприклад, вивести в консоль інформаційне повідомлення.

При виникненні події система генерує певний сигнал, а також надає механізм, за допомогою якого можна автоматично робити які-небудь інші дії (наприклад,

виконати певний код), коли відбувається подія. З клієнтської сторони події виникають всередині вікна браузера і, як правило, обробляються в контексті конкретного елементу, який в ньому знаходиться.

Модель подій у Node.js заснована на тому, що існують слухачі, які відстежують події, і емітери (*передавачі*), які періодично генерують події. Кожна подія може мати обробник подій – блок коду (зазвичай це функція зворотного виклику JavaScript), який буде запускатися при настанні події. Коли блок коду визначено на виконання у відповідь на виникнення події, цей процес називається реєстрація обробника події. Обробники подій іноді називають в загальному випадку – слухачі подій (*event listeners*).

Слухач відстежує подію, а обробник – це код, виконання якого здійснюється у відповідь на подію.

В основі Node.js лежить бібліотека *libuv*, що реалізує цикл подій (*event loop*) [7]. Цей механізм буде розглянутий далі.

Libuv – написана на мові C бібліотека подієво-орієнтованої обробки даних, яка призначена для спрощення асинхронного неблокуючого вводу/виводу.

Асинхронна модель була обрана через простоту, низькі накладні витрати (в порівнянні з ідеологією «один потік на кожне з'єднання») і високу швидкодію.

Обробка подій – основа роботи з Node.js. Події генерують практично всі об'єкти. За події в Node.js відповідає спеціальний модуль – *events*.

Призначати об'єкту обробник події можна методом *addListener (event, listener)*. Аргументами для нього служить ім'я події (рядок, зазвичай в camelCase-стилі: *connect*, *messages*, *messageBegin*) та функція зворотного виклику – обробник події.

```
let events = require('events');
let emitter = new events.EventEmitter()
```

EventEmitter – це основний клас, який реалізує роботу обробників подій і містить методи для роботи з подіями.

Всі об'єкти, які генерують події повинні бути екземпляром класу *EventEmitter*.

Коли відбувається генерація події, всі функції-обробники прикріплені до даної події викликаються синхронно.

Щоб встановити обробник на подію, необхідно використовувати один з наступних методів [4]:

- *on ('им'я_події', функція-обробник);*
- *addListener ('им'я_події', функція-обробник);*
- *once ('им'я_події', функція-обробник) – обробник спрацює тільки один раз, і буде видалений після спрацювання.*

Для генерації події використовується метод *emit ('event' [, args])*.

Наприклад:

```
emitter.on('click', function(){})
```

```
console.log('My first event'))};  
emitter.emit('click');
```

Коли відбувається підписка на подію, генерується системна подія *newListener*.

Для видалення обробника події використовується метод *removeListener* (*'ім'я_події'*, функція-обробник) і метод *off('event', handler)*.

Метод приймає аргументи:

event – ім'я події;

handler – функція-обробник події.

Одній події може бути призначено декілька обробників. Коли відбувається видалення обробника з якої-небудь події відбувається системна подія *removeListener*.

Emitter-об'єкти генерують іменовані події, які призводять до виклику раніше зареєстрованих функцій обробників подій. Таким чином у емітера є дві основні функції:

- генерування іменованих подій.
- реєстрація та видалення функцій обробників подій.

Для роботи з *EventEmitter* потрібно створити клас наслідуваний від класу *EventEmitter*:

```
class MyEmitter extends EventEmitter {}
```

Емітери – це об'єкти, які ініціюються з класів на основі *EventEmitter*:

```
const emitter = new MyEmitter();
```

У будь-який момент життєвого циклу емітерів можна скористатися функцією *emit* і згенерувати подію [7].

```
emitter.emit('Щось трапилося!!!');
```

Генерування подій – це сигнал, що виконана якась умова, яка відповідає настанню певної події. Зазвичай мова йде про зміну стану генеруючого об'єкта. За допомогою методу *on* можна додати функції-слухачі, які будуть виконуватися кожного разу, коли емітери генерують власні асоційовані події.

Якщо для однієї події зареєструвати декілька слухачів, то вони стануть викликатися в певному порядку. Перший зареєстрований буде і першим викликаним.

Наприклад:

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();
myEmitter.on('sendMess', () => {console.log('Подія
настало!');});
myEmitter.on('sendMess', () => {console.log('Подія
настало - 2!');});
myEmitter.on('sendMess', () => {console.log('Подія
настало - 3!');});
myEmitter.emit('sendMess');
```

Результат виконання показаний на Рис.1.4.

```
MacBook-Air-Oleksandr:gz_2_4 oleksvlas$ node event.js
Подія настало!
Подія настало - 2!
Подія настало - 3!
```

Рис. 1.4. Результат обробки подій в Node.js

Якщо потрібно визначити нового слухача, але щоб він викликався першим, можна скористатися методом *prependListener*. Наприклад:

```
myEmitter.prependListener('sendMess', () => {
  console.log('Подія настало - 2!');
});
```

У методі *emit* після вказання першим аргументом імені події можна вказувати будь-яку кількість аргументів і всі вони будуть доступні всередині функцій-слухачів, які були зареєстровані для цих подій.

Наприклад:

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();
myEmitter.on('event', function firstListener() {
    console.log('Helloooo! first listener');
});
myEmitter.on('event', function secondListener(arg1,
arg2) {
    console.log(`event with parameters ${arg1}, ${arg2}
in second listener`);
});
myEmitter.on('event', function thirdListener(...args) {
    const parameters = args.join(', ');
    console.log(`event with parameters ${parameters} in
third listener`);
});
console.log(myEmitter.listeners('event'));
myEmitter.emit('event', 1, 2, 3, 4, 5);
```

Результат виконання показаний на Рис.1.5.

```
MacBook-Air-Oleksandr:gz_2_4 oleksvlas$ node event_1.js
[
  [Function: firstListener],
  [Function: secondListener],
  [Function: thirdListener]
]
Helloooo! first listener
event with parameters 1, 2 in second listener
event with parameters 1, 2, 3, 4, 5 in third listener
```

Рис. 1.5. Результат обробки подій з передаванням аргументів в Node.js

В об'єкті, який створений функцією-конструктором (класом) *EventEmitter*, є додаткові методи для роботи з подіями [4]:

listenerCount('им'я_події') – метод повертає кількість обробників для вказаної події;

eventNames() повертає масив з іменами подій;

getMaxListeners() – метод повертає максимально допустиму кількість слухачів;

listeners('им'я_події') повертає копію масиву з функціями обробниками для вказаної події;

prependListener('им'я_події', функція-обробник) додає обробник в початок масиву обробників;

setMaxListeners(n) встановлює максимально допустиму кількість обробників.

Теоретично з одним об'єктом можна пов'язати безліч обробників, але за замовчуванням їх кількість обмежена до 10. Це зроблено для запобігання витоків пам'яті.

Обмеження встановлюється методом *setMaxListeners(n)*, де *n* – необхідна максимальна допустима кількість обробників.

Для видалення функції-обробника використовується метод *removeListener()*:

```
const EventEmitter = require('events').EventEmitter;
const myEmitter = new EventEmitter;
let customer = function(name){
    console.log('customer name: ' + name);
};
let message = function(msg){
    console.log('message: ' + msg);
};
myEmitter.on('customer', customer);
```

```
myEmitter.on('message', message);
myEmitter.emit('customer', 'Gary');
myEmitter.removeListener('customer', customer);
myEmitter.emit('customer', 'John');
myEmitter.emit('message', 'welcome to nodejs');
```

Результат виконання даного коду показаний на Рис.1.4.

```
MacBook-Air-Oleksandr:gz_2_4 oleksvlas$ node event_1.js
customer name: Gary
message: welcome to nodejs
```

Рис. 1.4. Результат обробки подій в Node.js після видалення одного з обробників подій

Таким чином одним з перспективних напрямків в програмуванні є використання подійно-орієнтованого підходу при написанні коду. Платформа Node.js надає широкий спектр можливостей для реалізації програмного забезпечення за подійно-орієнтованою парадигмою.

1.3. Основні вбудовані модулі Node.js

Модулі в платформі Node.js – це базові будівельні блоки для додатків. Будь-який JavaScript файл, який використовується в Node.js є модулем.

Переваги модульної системи:

- структурованість програмних частин;
- модулі схожі на простір імен;
- взаємозамінність;
- спрощується взаємодія розробника з різними частинами додатку;

Разом з Node.js інсталяється декілька вбудованих модулів.

Підключення інстальованого модуля відбувається за допомогою виклику функції *require*, якій в якості аргументу потрібно передати шлях до файлу [4]. При повторному підключенні Node.js перевіряє, чи підключався модуль раніше, якщо це так, то повертається посилання на вже існуючий модуль.

Перед виконанням коду модуля Node.js обгортає його оборткою функції, яка виглядає наступним чином:

```
(function(exports, require, module, __filename,  
__dirname) {  
    // Безпосередньо коду модулю  
});
```

Завдяки цьому Node.js дозволяє забезпечити збереження змінних верхнього рівня (оголошених з використанням *var*, *const* або *let*) в області видимості модуля, а не в глобальному об'єкті; надає додаткові глобальні змінні: об'єкти *module* та *exports*, які реалізуються експорт значень із одного модуля в інший; змінні *__filename* та *__dirname*, що містять абсолютну назву файла та шлях до каталогу.

Приклад підключення модуля:

```
let module = require('filename');
```

Формат CommonJS застосовується в Node.js і використовує для визначення залежностей модулів *require* і *module.exports*:

```
let dep1 = require('./dep1');  
let dep2 = require('./dep2');  
module.exports = function(){//      // ...}
```

Для створення нового модуля необхідно створити файл з розширенням **.js* і реалізувати в ньому потрібну логіку.

Щоб функціонал модуля був доступний в інших модулях, потрібні складові модуля потрібно помістити в об'єкт *exports*.

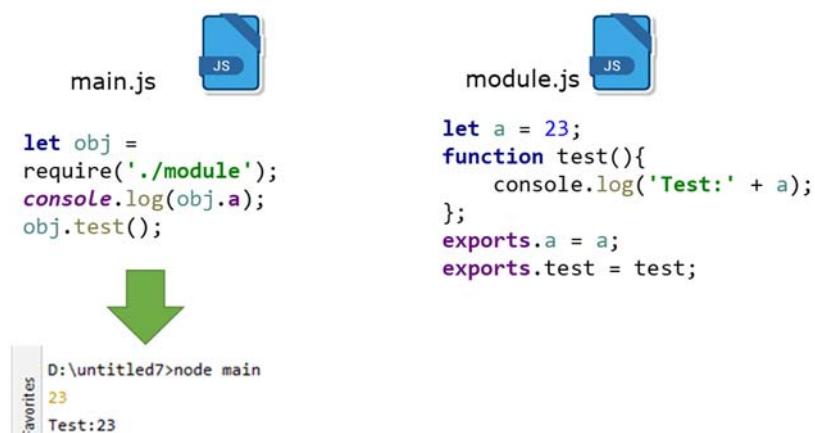


Рис. 1.4. Експорт модулів в Node.js

Об'єкт *module* зберігає інформацію про поточний модуль. *Exports* є посиланням на властивість об'єкта *module.exports*.

Властивості об'єкта *module*:

- id* - шлях до виконуваного файлу;
- exports* - об'єкт який повертається функцією *require*;
- parent* - посилання на батьківський модуль;
- filename* - абсолютний шлях до файлу;
- loaded* - статус обробки файлу;
- children* - масив дочірніх модулів;
- paths* - масив шляхів, за якими відбувається пошук модулів.

Для отримання значень цих властивостей скористаємося способом представленим на Рис 1.5.

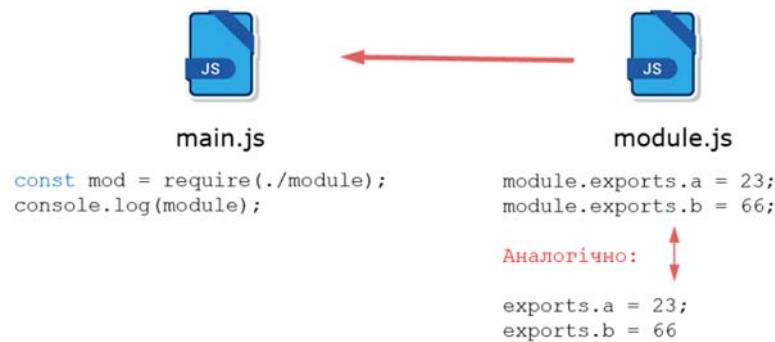


Рис. 1.5. Експорт модулів в Node.js

В результаті отримаємо вивід значень властивостей в консоль, представлений на Рис.1.6.

```

D:\untitled7>node main
Module {
  id: '.',
  path: 'D:\\untitled7',
  exports: {},
  parent: null,
  filename: 'D:\\untitled7\\main.js',
  loaded: false,
  children: [
    Module {
      id: 'D:\\untitled7\\module.js',
      path: 'D:\\untitled7',
      exports: [Object],
      parent: [Circular],
      filename: 'D:\\untitled7\\module.js',
      loaded: true,
      children: [],
      paths: [Array]
    }
  ],
  paths: [ 'D:\\untitled7\\node_modules', 'D:\\node_modules' ]
}
D:\untitled7>

```

Рис. 1.6. Властивості об'єкта *module* в Node.js

Для професійної розробки та тестування або участі в декількох проектах може знадобитися можливість перемикання між різними версіями Node.js на одному комп'ютері.

Для цього рекомендується використовувати менеджери версій Node.js замість того, щоб видаляти старі версії і встановлювати нові.

Для Windows – це *nodist* або *nvwmt*, для **nix* і *Mac OS* це *nvm*. Всі ці менеджери можна встановити за допомогою системи управління пакетами *NPM*.

Для більш зручного запуску програм за допомогою Node.js існують менеджери процесів. Це програми, які дозволяють легко налаштувати автозапуск тих чи інших програм на Node.js при завантаженні системи і їх автоматичний перезапуск при несподіваному завершення роботи через критичних помилок.

Також менеджери процесів дозволяють зручно налаштувати логування стандартних потоків виводу у файли, надають більш зручний інтерфейс для оперативного перегляду логів і загального стану запущених через Node.js скриптів.

Популярні менеджери процесів *pm2* і *forever* розроблені для **nix* систем і також встановлюються з використанням менеджера *NPM*. Щоб переглянути список команд менеджерів досить набрати в консолі *pm2* або *forever*.

NPM – це менеджер пакетів, що використовується Node.js-додатками [7]. З допомогою нього можна

інсталювати багато готових модулів, для спрощення роботи веб-розробника. Аналогом в інших мовах програмування є Maven для Java або Composer для PHP.

Перевірка поточної версії node.js:

```
npm -v
```

Отримання основної інформації про встановлене середовище Node.js:

```
npm config list
```

Для глобального встановлення пакетів використовується команда *install* і опція *--global*, чи скорочений запис *-g*.

```
npm install package_name --global
```

Вивід списку глобально встановлених пакетів:

```
npm list --global
```

Видалення пакетів локально:

```
npm uninstall package_name
```

Встановлення визначеної версії пакету:

```
npm install package_name@1.1.2.
```

Оновлення версії пакету:

```
npm update package_name
```

Ініціалізація пакетного менеджера npm в робочий каталог з проєктом:

```
npm init
```

В результаті буде створено файл *package.json*.

В файлі *package.json* вказується список модулів, які використовуються для розробки проекту і безпосередньо в самому проекті:

devDependencies – це модулі, які необхідні лише під час розробки;

Dependencies – це модулі, які потрібні під час виконання програми.

Атрибути (властивості) файлу *package.json*:

name – назва проекту (модулю);

version – версія проекту (модулю);

description – опис проекту (модулю);

homepage – домашня сторінка проекту (модулю);

author – автор;

contributors – ім'я учасників проекту (модулю);

dependencies – список залежностей, які необхідні для роботи модулю;

repository – тип репозиторію та адреса до нього;

main – точка входу проекту (основний файл);

keywords – ключові слова.

Наприклад:

```
{
  "name": "gulp_start",
  "version": "1.0.0",
  "description": "my first node.js project",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit
  },
  "1"
```

```
        "author": "oleksvlas",
        "license": "ISC"
    }
```

Для встановлення модулів, які потрібні під час виконання програми необхідно виконати команду:

```
npm install package_name --save
```

Запис інформації про встановлений модуль до файлу *package.json*:

```
"dependencies" {
    "express": "^4.17.1"}
```

Для встановлення модулів, які потрібні під час розробки програми необхідно виконати команду [4]:

```
npm install package_name --save-dev
```

Запис інформації про встановлений модуль, який потрібен для розробки програм до файлу *package.json*:

```
"devDependencies" {
    "gulp": "^4.0.2"
}
```

Так, для прикладу, встановимо до робочого проєкту модуль *colors* (Рис. 1.7).

Далі створимо файл *main.js*, підключимо модуль *colors* та викличемо метод *rainbow*:

```
const colors = require('colors');
let text = 'Літери цього тексту відобразяться кольорами радуги!';
console.log(text.rainbow);
```

Результат виконання даного коду представлено на Рис. 1.8.



Рис. 1.7. Встановлення модуля в проект з використанням *npm*

```
C:\Users\oleksvlas\WebstormProject\js>node main
Літери цього тексту відобразяться кольорами радуги!
```

Рис. 1.8. Результат використання можливостей модуля *colors* в проекті

В Node.js існують глобальні об'єкти доступні для всіх модулів. Їх не потрібно окремо включати в додаток, їх можна використовувати безпосередньо без імпортuvання.

Наприклад до них відносять розглянуті вище об'єкти [7]:

__dirname представляє ім'я директорії, в якій розташовується і виконується в даний момент скрипт.

__filename є ім'ям файлу з виконуваним кодом. Це абсолютний шлях до цього файлу. Для основної програми

не обов'язково використовувати те саме ім'я файлу, яке використовується в командному рядку. Значення всередині модуля – це шлях до файлу модуля.

Об'єкт *console* забезпечує налагодження додатків і підтримку наступних можливостей:

Методи *console.log()*, *console.error()*, *console.warn()* використовуються для виведення інформації в потік.

Глобальний об'єкт *console*, сконфігуртований на запис даних в стандартний потік помилок.

Він також не потребує імпортuvання модуля.

Об'єкт *console*, містить методи, *console.time()*, *console.timeEnd()*, що дозволяють вимірюти продуктивність виконання скриптів. Метод *console.dir()* дозволяє вивести інформацію про об'єкт.

Процес запущеного Node.js-додатка представлений глобальним об'єктом *process*, який є екземпляром класу *EventEmmiter* і надає дані про систему, параметри запуску, змінні середовища і ресурси процесора, які використовуються.

Найчастіше використовуємі властивості об'єкта *process*:
arch – інформація про архітектуру процесора, на якому запущений додаток Node.js;

argv – масив з параметрами запуску процесу;

env – об'єкт з середовищем користувача;

pid – унікальний ідентифікатор процесу;

platform – платформа операційної системи;

version – поточна версія Node.js.

Наприклад:

```
console.log("Платформа ОС: " + process.platform);
console.log("Архітектура процесора: " + process.arch);
console.log("Ідентифікатор процесу: " + process.pid);
```

Результат виводу зображенено на Рис.1.9.

```
MacBook-Air-Oleksandr:gz_1_12 oleksvlas$ node process
Платформа ОС: darwin
Архітектура процесора: x64
Ідентифікатор процесу: 3819
```

Рис. 1.9. Використання об'єкту *process* в Node.js

Найчастіше використовуємі методи об'єкта *process*:

cputime() – повертає об'єкт з системним і користувальницьким часом використання процесу в мілісекундах, на основі якого можливо розрахувати завантаженість процесора в процентному значенні;

hrtime() – повертає високоточний час у вигляді масиву з двома елементами: секунди і наносекунди;

cwd() – повертає робочу директорію Node.js процесу;

memoryUsage() – повертає об'єкт з даним про використання процесом Node.js оперативної пам'яті;

uptime() – повертає кількість мілісекунд з моменту запуску процесу.

on() – метод для обробки подій.

exit() – завершує процес з заданим кодом (за замовчуванням код дорівнює 0)

В даний час в Node.js є 34 вбудованих модуля (Рис.1.10):

```

assert,           fs,           stream,
buffer,          http,         string_decoder,
c/c++_addons,    https,        timers,
child_process,   module,       tls_(ssl),
cluster,         net,          tracing,
console,         os,           tty,
crypto,          path,         dgram,
deprecated_apis, punycode,    url,
dns,             querystring, util,
domain, Events,  readline,    repl,        v8, vm, zlib

```

Рис. 1.10. Вбудовані модулі в Node.js

Розглянемо основні можливості деяких з представлених модулів.

Так, модуль *path* – це вбудований модуль платформи Node.js, який використовується для обробки і перетворення шляхів до файлів [4].

Підключення модулю здійснюється командою:

```
let path = require("path");
```

Властивості модулю *path*:

path.sep – окремий роздільник файлів ('\\' або '/').

path.delimiter – покажчик шляху для конкретної платформи ; або '!'.

path posix – надає доступ до методів *path*.

path win32 – надає доступ до описаних вище методів *path*, завжди взаємодіє сумісним чином з *win32* платформою.

Методи модулю *path*:

path.normalize(p) – нормалізує строковий шлях, обробляючи «..» і «.».

path.join ([path1] [, path2] [...]) – об'єднує всі аргументи і нормалізує отриманий шлях.

path.resolve ([from ...], to) – обробляє абсолютний шлях.

path.isAbsolute (path) – визначає, чи є шлях абсолютноним. Абсолютний шлях завжди буде оброблятися в тому ж місці, незалежно від робочого каталогу.

path.relative (from, to) – обробляє відносний шлях від *from* до *to*.

path.dirname (p) – повертає ім'я директорії для заданого шляху. Подібно команді *dirname* в Unix.

path.basename (p [, ext]) – повертає останню частину шляху. Подібно команді *basename* в Unix.

path.basename (p) – повертає з шляху розширення останньої частини, починаючи від останньої '!' до кінця рядка. Якщо '!' не зустрічається в останній частині шляху або на його початку, повертає порожній рядок.

path.parse (pathString) – повертає об'єкт з рядка шляху.

path.format (pathObject) – повертає рядок шляху з об'єкта, протилежно *path.parse*.

Модуль *url* – розділяє URL-адресу на логічні частини.

Підключення модулю *url*:

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

Методи модулю *url*:

url.format(urlObject) – повертає відформатовану URL-адресу, отриману з об'єкта *urlObject*.

url.parse(urlString) – повертає відформатовану URL-адресу, отриману з об'єкта *urlObject*.

url.resolve(from, to) – повертає відформатовану URL-адресу, отриману з об'єкта *urlObject*.

Для роботи з файловою системою в Node.js використовується вбудований модуль *fs*.

Node.js реалізує запис/читування файлів, використовуючи прості обгортки навколо стандартних функцій POSIX.

Кожен метод у модулі *fs* має як синхронну, так і асинхронну форму.

Асинхронні методи приймають останнім параметром функцію зворотного виклику, яка першим параметром завжди приймає об'єкт помилки.

Краще використовувати асинхронний метод замість синхронного методу, оскільки перший ніколи не блокує програму під час її виконання.

Синтаксис методу відкриття файлу в асинхронному режимі виглядає так:

fs.open(path, flags[, mode], callback)

Параметри:

path – це строка, в якій вказується ім'я файлу та шлях до нього.

flags – прапори вказують на поведінку файлу, який потрібно відкрити.

mode – встановлює права доступу до файлу, але лише в тому випадку, коли файл був створений. Значення за замовчуванням – *0666*.

callback – це функція зворотного виклику, яка отримує два аргументи (*error, fd*).

The diagram illustrates the creation of a file named 'testFile.txt' using Node.js. On the left, a code editor shows a file named 'main_1.js' with the following content:

```
let fs = require("fs");
fs.open('testFile.txt',
  'w', (err) => {
    if(err) throw err;
    console.log('File created');
});
```

An arrow points from the code editor to a terminal window on the right, which displays the command: "C:\Users\oleksvlas\WebstormProject\js>node main_1". Below the command, the output "File created" is shown. A red box highlights the terminal output. Another arrow points from the terminal window to a file explorer window on the right. The file explorer shows a folder structure under "js C:\Users\oleksvlas\WebstormProject\js": "node_modules library root", "main.js", "main_1.js", "package.json", "package-lock.json", "some.txt", and "testFile.txt". The "testFile.txt" file is highlighted with a red box.

Рис. 1.9. Створення файлу в Node.js

В прикладі наведеному на попередньому слайді метод *fs.open()* використовується для створення нового файлу. Як перший аргумент він приймає ім'я новоствореного файлу. Його другий аргумент являє собою пропор, який вказує системі на те, що саме необхідно зробити з файлом. В даному випадку це пропор *w* (скорочення від *writing*), який вказує на те, що ми хочемо відкрити файл для запису.

Метод *open()* може приймати різні пропори. Ось деякі з них:

- r*: відкрити файл для читання;
- r+*: відкрити файл для читання і запису;
- rs*: відкрити файл для читання в синхронному режимі;
- w*: відкрити файл для запису;

- a*: відкрити файл для запису даних в кінець файлу;
a+: відкрити файл для читання і для запису даних в кінець файлу.

Синтаксис методу отримання інформації про файл:

fs.stat(path, callback)

Параметри методу *stat*:

path – це строка, в якій вказується ім'я файлу та шлях до нього;

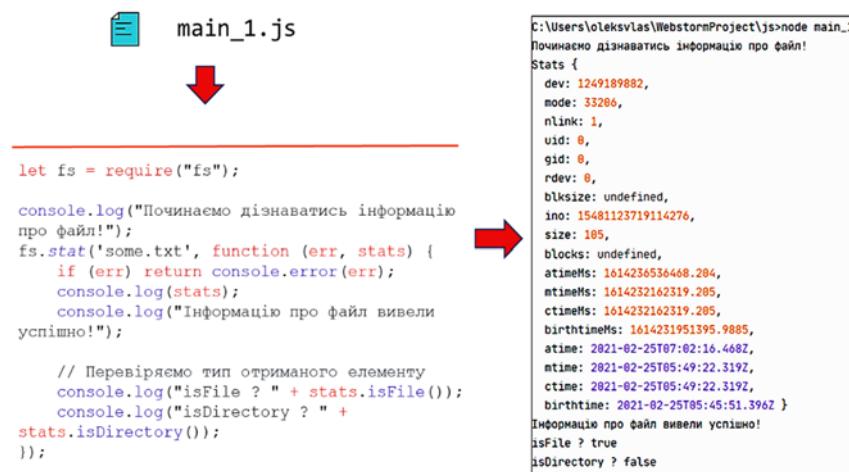
callback – це функція зворотного виклику, яка отримує два аргументи (*error, stats*), де *stats* є об'єктом класу *fs.Stats*.

Методи об'єкту *stats*:

stats.isFile() – повертає *true*, якщо тип просто файл;

stats.isDirectory() – повертає *true*, якщо тип файлу каталог;

stats.isSymbolicLink() – повертає *true*, якщо тип файлу символічне посилання.



The screenshot shows a terminal window with the command `C:\Users\oleksylas\WebstormProject\js>node main_1.js`. Below the command, the output is displayed in two columns. On the left, the code for `main_1.js` is shown, which includes code to log file information and check if it's a file or directory. On the right, the detailed file statistics object is printed to the console.

```

let fs = require("fs");

console.log("Починаємо дізнатись інформацію про файл!");
fs.stat('some.txt', function (err, stats) {
  if (err) return console.error(err);
  console.log(stats);
  console.log("Інформацію про файл вивели успішно!");

  // Перевіряємо тип отриманого елементу
  console.log("isFile ? " + stats.isFile());
  console.log("isDirectory ? " +
  stats.isDirectory());
});

```

```

Stats {
  dev: 1249189882,
  mode: 33286,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  blksize: undefined,
  ino: 15481123719114276,
  size: 105,
  blocks: undefined,
  atimeMs: 1614236536468.204,
  mtimeMs: 1614232162319.205,
  ctimeMs: 1614232162319.205,
  birthtimeMs: 1614231951395.9885,
  atime: 2021-02-25T07:02:16.468Z,
  mtime: 2021-02-25T05:49:22.319Z,
  ctime: 2021-02-25T05:49:22.319Z,
  birthtime: 2021-02-25T05:45:51.396Z }
}

Інформація про файл виведена успішно!
isFile ? true
isDirectory ? false

```

Рис. 1.10. Отримання даних про файл в Node.js

Синтаксис методу запису інформації у файл буде наступним:

fs.writeFile(filename, data[, options], callback)

Параметри методу:

path – це строка, в які вказується ім'я файлу та шлях до нього;

data – це строка або буфер, який слід записати у файл;

options – це об'єкт, який містить *{encoding, mode, flag}*;

callback – це функція зворотного виклику, яка отримує в якості аргумента об'єкт помилки.

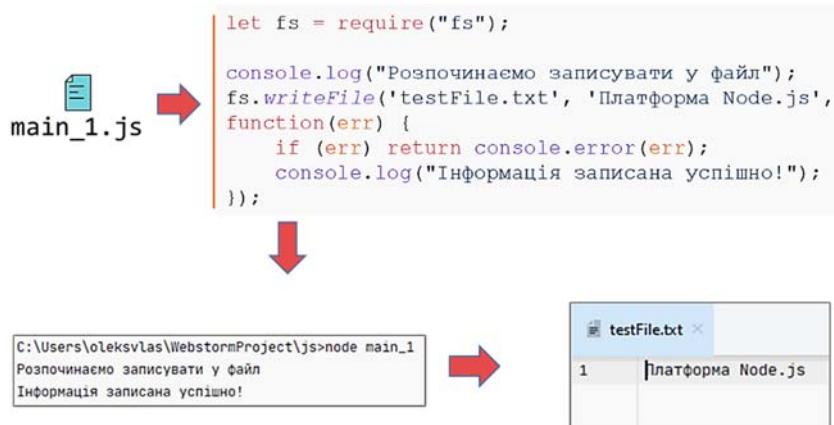


Рис. 1.11. Запис даних в файл

Синтаксис методу запису інформації у кінець файлу буде наступним:

fs.appendFile(filename, data[, options], callback)

Параметри:

path – це строка, в якій вказується ім'я файлу та шлях до нього;

data – це строка або буфер, який слід записати у файл;

options – це об'єкт, який містить *{encoding, mode, flag}*;

callback – це функція зворотного виклику, яка отримує в якості аргумента об'єкт помилки.

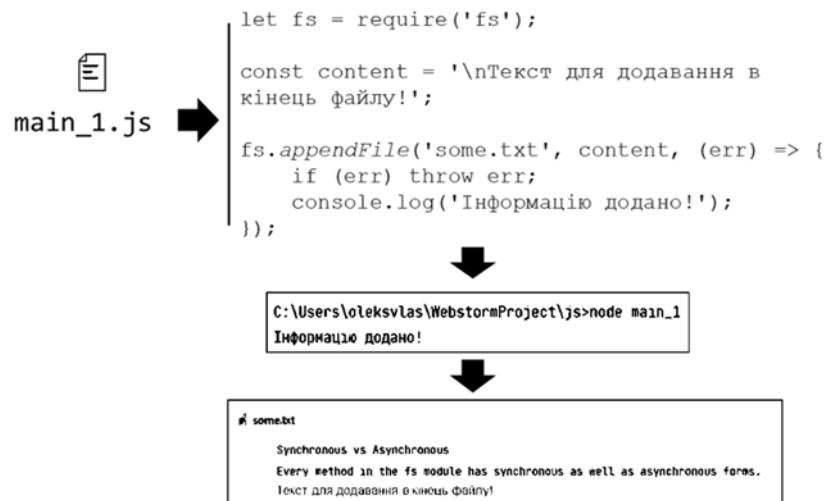


Рис. 1.12. Запис даних в кінець файлу

Синтаксис методу зчитування інформації з файлу:

fs.readFile(filename, encoding, callback)

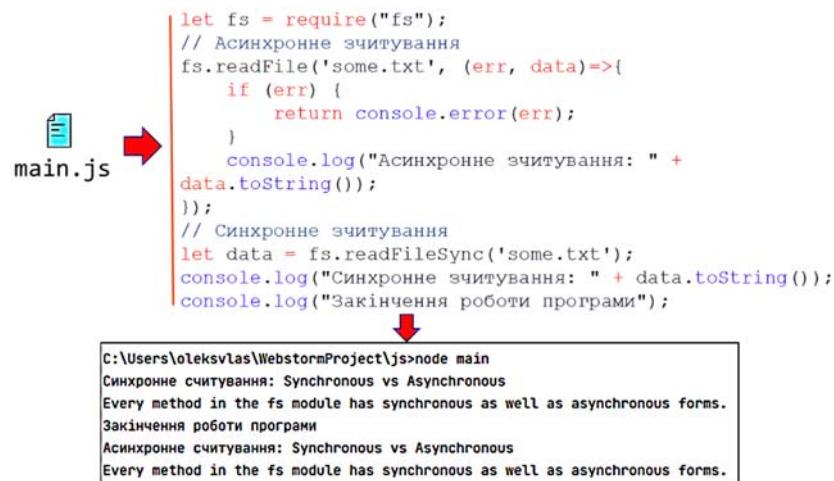
Параметри:

path – це строка, в якій вказується ім'я файлу та шлях до нього;

encoding – це строка, в якій вказується тип кодування файлу. Значення за замовчуванням – ‘*utf8*’;

callback – це функція зворотного виклику, яка отримує два аргументи - об'єкт помилки та контент (дані, інформацію) з файлу.

Для синхронного зчитування файлу використовується синхронний метод *readFileSync* (Рис.1.13.).



```
main.js →
let fs = require("fs");
// Асинхронне зчитування
fs.readFile('some.txt', (err, data)=>{
  if (err) {
    return console.error(err);
  }
  console.log("Асинхронне зчитування: " +
  data.toString());
});
// Синхронне зчитування
let data = fs.readFileSync('some.txt');
console.log("Синхронне зчитування: " + data.toString());
console.log("Закінчення роботи програми");

C:\Users\oleksvlas\WebstormProject\js>node main
Синхронне считування: Synchronous vs Asynchronous
Every method in the fs module has synchronous as well as asynchronous forms.
Закінчення роботи програми
Асинхронне считування: Synchronous vs Asynchronous
Every method in the fs module has synchronous as well as asynchronous forms.
```

Рис. 1.13. Асинхронне та синхронне зчитування файлу

Синтаксис методу видалення файлу:

fs.unlink (path, callback)

Параметри:

path – це строка, в якій вказується ім'я файлу та шлях до нього;

callback – це функція зворотного виклику, яка отримує в якості аргументу об'єкт помилки.

The screenshot shows a code editor with a file named `main_1.js`. The code uses the `fs.unlink` method to delete a file named `testFile.txt`. A red arrow points from the file icon to the code. Another red arrow points from the code to a terminal window below. The terminal window displays the command `C:\Users\oleksvlas\WebstormProject\js>node main_1` and the output: "Починаємо процес видалення файлу" and "Файл видалено!". A third red arrow points from the terminal window to a file explorer window on the right. The file explorer shows a directory structure with files: `main.js`, `main_1.js`, `package.json`, `package-lock.json`, and `some.txt`.

```

let fs = require("fs");
console.log("Починаємо процес видалення файлу");
fs.unlink('testFile.txt', function(err) {
  if (err) return console.error(err);
  console.log("Файл видалено!");
});

```

Рис. 1.14. Видалення файлу

Синтаксис методу перегляду вмісту каталогу:

`fs.readdir(path, callback)`

Параметри:

`path` – це строка, в якій вказується ім'я файлу та шлях до нього;

`callback` – це функція зворотного виклику, яка отримує два аргументи: об'єкт помилки та масив імен файлів у каталозі.

The screenshot shows a code editor with a file named `main_1.js`. The code uses the `fs.readdir` method to list the contents of a directory. A red arrow points from the file icon to the code. Another red arrow points from the code to a terminal window below. The terminal window displays the command `C:\Users\oleksvlas\WebstormProject\js>node main_1` and the output: ".idea", "main.js", and "main_1.js".

```

let fs = require("fs");
fs.readdir(__dirname, function(err, items) {
  for (let i=0; i<items.length; i++) {
    console.log(items[i]);
  }
});

```

Рис. 1.15. Перегляд вмісту каталогу

Синтаксис методу перейменування файлу:

`fs.rename(path, newname, callback)`

Параметри:

path – це строка, в якій вказується ім'я файлу та шлях до нього;

newname – це строка, в якій вказується нове ім'я файлу;

callback – це функція зворотного виклику, яка отримує в якості аргументу об'єкт помилки.



Рис. 1.16. Перейменування файлу

Для отримання доступу до веб-сторінок будь-якого додатку потрібен веб-сервер. Node.js надає можливості створити власний веб-сервер, який буде асинхронно обробляти запити HTTP.

Для використання клієнта і сервера HTTP необхідно підключити відповідний модуль за допомогою `require('http')`.

Для створення сервера використовується метод `http.createServer([requestListener])`. Метод повертає новий екземпляр об'єкта `http.Server`.

Метод `server.listen(port,[hostname],[callback])` починає прийом з'єднань на зазначеному порту і імені хоста. Якщо ім'я хоста не вказано, сервер буде приймати з'єднання на будь-яку IPv4-адресу машини.

Наприклад:

```
let http = require('http');
let server = http.createServer(function(req, res) {
    res.end('Hi,guys!'); }).listen(3000);
```

Далі, для прикладу наведено створення серверу з декількома маршрутами (Рис.1.17).

```
const http = require('http');
let server = http.createServer(function (req, res) {
    if (req.url == '/') {
        res.writeHead(200, {'Content-Type':
'text/html'});
        res.write('<html><body><p>This is home
Page.</p></body></html>');
        res.end();
    } else if (req.url == "/cadet") {
        res.writeHead(200, {'Content-Type':
'text/html'});
        res.write('<html><body><p>This is cadet
page.</p></body></html>');
        res.end();
    } else res.end('Invalid Request!');
});
server.listen(5000);
console.log('Node.js web server at port 5000 is
running..')
```



Рис. 1.17. Результат виконання запитів за різними маршрутами

У Node.js для спрощення реалізації серверної частини використовується модуль *Express*, який оснований на використанні модулю *http*. Детально модуль *Express* буде розглянуто в Розділі 2. На даний момент розглянемо варіант рішення задачі завантаження файлів на сервер на основі цього модулю. Рішення задачі завантаження файлів на сервер можливе з використанням наступних бібліотек:

multer – відмінне рішення, якщо використовується фреймворк Express;

busboy – більш низькорівневе рішення, ніж *multer*.
Можливо пристосувати до будь-якого фреймворку;

formidable – універсальний модуль для завантаження файлів.

В подальшому скористаємося можливостями модулю *multer*. Встановлення і підключення модулю *multer* здійснюється командами [11]:

```
npm install multer --save
let multer = require('multer');
let upload = multer();
```

Multer - це *middleware* (функція проміжної обробки) для фреймворку *Express.js*, що призначена для обробки форм типу *multipart/form-data*, та використовується при завантаженні файлів. Створена вона, як обгортка над модулем *busboy* для забезпечення його максимально ефективного використання.

Multer не оброблює нікий інший вид форм, крім *multipart/form-data*.

Multer додає об'єкт *body* і об'єк файлу *file* (або *files*) всередину об'єкта *request*. Об'єкт *body* містить значення текстових полей форм, об'єкт *file* (або *files*) містить файл або файли, які завантажуються через форму.

Опис і інформація про модуль знаходиться за ресурсом:

<https://github.com/expressjs/multer>.

Кожен об'єкт *file* містить таку інформацію (Рис.1.18):

Властивість	Опис	Примітка
<code>fieldname</code>	Назва поля, вказана у формі	
<code>originalname</code>	Ім'я файлу на комп'ютері користувача	
<code>encoding</code>	Тип кодування файлу	
<code>mimetype</code>	Mime-тип файлу	
<code>size</code>	Розмір файлу в байтах	
<code>destination</code>	Папка, в яку збережено файл	DiskStorage
<code>filename</code>	Ім'я файлу в межах destination	DiskStorage
<code>path</code>	Повний шлях до завантаженого файлу	DiskStorage
<code>buffer</code>	Буфер всього файлу	MemoryStorage

Рис. 1.18. Властивості об'єкта *file*

Multer приймає об'єкт параметрів, найголовнішим з яких є *dest* властивість, яка повідомляє *Multer*, в яке місце необхідно завантажити файли [11]. У випадку, якщо не вказати об'єкт параметрів, файли зберігатимуться в пам'яті та ніколи не запишуться на диск.

За замовчуванням *multer* буде перейменовувати файли, щоб уникнути конфліктів імен. Функцію перейменування можна налаштувати відповідно до власних потреб. Параметри налаштування *multer* вказані на Рис.1.19.

Ключ	Опис
<i>dest or storage</i>	Місце, де зберігати файли
<i>fileFilter</i>	Функція контролю, які файли приймаються для завантаження
<i>limits</i>	Обмеження завантаження даних
<i>preservePath</i>	Зберігати повний шлях до файлів, а не лише базове ім'я

Рис. 1.19. Параметри налаштування *multer*

Наприклад:

```
let upload = multer({dest: 'uploads/'});
```

Методи об'єкта *multer*:

single(fieldname) – приймає один файл з ім'ям *fieldname*.

Файл буде збережений в *req.file*.

array(fieldname[, maxCount]) – приймає масив файлів з ім'ям *fieldname*. Можливо задати помилку при спробі завантаження більш *maxCount* файлів. Масив файлів буде збережений в *req.files*.

fields(fields) – приймає набір файлів, визначених у *fields*. Об'єкт з масивом файлів буде збережений в *req.files*. *Fields* повинен бути масивом об'єктів з полями *name* та *maxCount*. Наприклад:

```
[  
  { name: 'avtor', maxCount: 1 },  
  { name: 'gallery', maxCount: 8 }  
]
```

none() приймає тільки текстові поля форми. При спробі завантаження файлу видає помилку *"Limit_Unexpected_File"*.

any() приймає всі передані файли. Масив файлів буде збережений в *req.files*.

Обовязково необхідно переконатися в коректній обробці завантаження файлів додатком. Не слід використовувати *multer* для всіх роутів, тому що користувач може завантажити шкідливі файли, і тим самим порушити роботу додатку.

Для прикладу створимо простий додаток для завантаження файлів на сервер. Для початку створимо файл *load.html* з формою:

```
<!DOCTYPE html>  
<html lang="ua">  
  <head>  
    <title>Модуль multer</title>  
  <meta charset="utf-8" />  
  </head>  
  <body>  
    <h1>Додаток завантаження файлу</h1>  
  <form action="/load" method="post"
```

```

enctype="multipart/form-data">
<label>Додаток завантаження файлу</label><br>
<input type="file" name="image" /><br>
    <button type="submit">Send</button>
</form>
</body>
</html>

```

Для реалізації серверної сторони створимо файл *load_app.js*:

```

let express = require('express');
let multer = require('multer');
let app = express();
const storageConfig = multer.diskStorage({
    destination: (req, file, callback) =>{
        callback(null, "uploads");
    },
    filename: (req, file, callback) =>{
        callback(null, file.originalname);
    }
});
let upload = multer({ storage:
storageConfig });
app.get(req, res)=>{
    res.sendFile(__dirname +"/load.html");
};
app.post('/load', upload.single('image'), (req,
res)=>{
    console.log(req.file);
    res.send("OK!")
});
app.listen(3030, ()=>{
    console.log('Server starting...')
});

```

Так при звернені за головною адресою ресурсу за портом 3030 користувачу буде відправлено сторінку *load.html*:

```

app.get(req, res)=>{
    res.sendFile(__dirname +"/load.html");});

```

Користувач заповнить дані форми, вказавши файл який необхідно завантажити на сервер та натисне на кнопку *Send*. В результаті спрацює обробник за маршрутом '/load' в якому другим параметром визначено об'єкт *multer*:

```
upload.single('image')
```

Конфігурація налаштувань, а саме: каталог для збереження завантажених файлів – "uploads" та кінцеве ім'я завантажуваного файлу – *file.originalname*, визначені в об'єкті *StorageConfig*:

```
const storageConfig = multer.diskStorage({
  destination: (req, file, callback) => {
    callback(null, 'uploads');
  },
  filename: (req, file, callback) => {
    callback(null, file.originalname);
  }
});
let upload = multer({ storage: storageConfig });
```

В результаті файл обраний користувачем буде завантажений на сервер в каталог "uploads" з збереженням початкової назви файлу.

Контрольні запитання до розділу 1

1. У чому різниця між *module.exports* і *exports*?
2. Чому в модулях змінні верхнього рівня не є глобальними?
3. Поясніть чи допустимо використовувати синхронні методи для роботи з файловою системою ?
4. В чому полягає різниця між методами *get* і *post*, що використовуються при роботі протоколу *http*?
5. На основі якого протоколу забезпечується функціонування *Web*-додатків?
6. Як перевірити яка версія Node.js встановлена?
7. Як створити модуль в Node.js?
8. Яким чином здійснюється управління залежностями проекту?
9. Скільки існує потоків в Node.js по замовчуванню?
10. Що таке *pm2*?
11. Як встановити додаткову бібліотеку (модуль) в Node.js?
12. Назвіть основні характеристики модулю Express.js
13. Які переваги надає модуль Express.js?
14. Що таке рушій *V8*? Де він використовується?
15. Яким чином можна завантажити файл на сервер Node.js?

**СТВОРЕННЯ СЕРВЕРНИХ ПРОГРАМНИХ
ДОДАТКІВ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЙ
NODE.JS**

**2.1. Асинхронне програмування та багатопоточність в
Node.js**

Асинхронність представляє можливість одночасно виконувати відразу декілька завдань. Асинхронність грає велику роль в Node.js та основана на декількох важливих аспектах, таких як: цикл подій, неблокуючий ввід-вивід, функції зворотного виклику, проміси та ін.

Розглянемо основні складові, на яких ґрунтуються асинхронність в Node.js.

Таймер у Node.js – це внутрішня конструкція, яка викликає задану функцію через певний проміжок часу. Виклик функції таймера, залежить від того, який метод був використаний для створення таймера, і яку іншу роботу виконує цикл подій Node.js [4].

В JavaScript є методи, які дозволяють викликати функцію не відразу, а через деякий проміжок часу (в асинхронному режимі).

Існують два таких основних методи [7]:

- *setTimeout* дозволяє викликати функцію один раз через певний інтервал часу;
- *setInterval* дозволяє викликати функцію та регулярно, повторювати виклик через певний інтервал часу.

Ці методи не є частиною специфікації JavaScript. Але більшість середовищ виконання js-коду мають внутрішній планувальник і надають доступ до цих методів. Зокрема, вони підтримуються у всіх браузерах і платформі Node.js.

Синтаксис:

```
let timer=setTimeout(func, [delay],[arg1], [arg2], ...)
```

Параметри:

func – функція, яка буде виконана через певний період часу;

delay – затримка перед запуском в мілісекундах (1000 мс = 1 с). Значення за замовчуванням - 0.

arg1, arg2, ... – аргументи, що передаються в функцію.

Наприклад, Рис.2.1:

```
let str = "!!!";
setTimeout((arg1)=>{
    console.log("Hi! setTimeout"
+ arg1);
}, 1000, str);               ➡ C:\Users\oleksvlas\WebstormProject\gz_2_4>node app
((arg1)=>{
    console.log("Hi! function" +
arg1);
}) (str);
```

Рис. 2.1. Результат виклику функції *setTimeout*

Виклик *setTimeout* повертає «ідентифікатор таймера» *timerName*, який можна використовувати методом *clearTimeout* для скасування подальшого виконання.

Наприклад:

```
let timerName = setTimeout(...);
clearTimeout(timerName);
```

Метод *setInterval* має наступний синтаксис:

```
let timerName = setInterval(func, [delay], [arg1],  
[arg2], ...)
```

Параметри методу:

func – функція, яка буде виконана через певний період часу.

delay – затримка перед виконанням в мілісекундах (1000 мс = 1 с). Значення за замовчуванням – 0.

arg1, arg2, ... – аргументи, що передаються в функцію.

Метод *setImmediate()* є спеціальним таймером, який працює в окремій фазі циклу подій. *setImmediate()* призначений для виконання сценарію після завершення поточної фази *poll* (опитування).

Синтаксис методу:

```
setImmediate(() => {  
    //run something  
});
```

Порядок (послідовність) виконання таймерів буде змінюватися залежно від контексту, в якому вони викликаються. Якщо вони викликаються з головного модуля, тоді синхронізація буде обмежена продуктивністю процесу (на ней можуть вплинути інші працюючі програми).

setImmediate() не завжди буде виконуватися до вищерозглянутих таймерів.

Завдяки неблокуючому вводу/виводу Node.js може бути асинхронним. Механізм неблокуючого вводу/виводу надає операційна система.

Для роботи з блокуючими операціями призначена бібліотека *libuv*.

Особливості бібліотеки *libuv*:

- у операційній системі Linux операції над локальними файлами є блокуючими і для їх обробки використовується бібліотека *libuv*;
- для емуляції неблокуючого вводу/виводу використовуються потоки;
- за замовчуванням створюється пул з чотирьох потоків;
- більше чотирьох складних операцій над локальними файлами блокують додаток.

Також в *libuv* реалізований цикл подій (*Event loop*) (Рис.2.2), деталі якого будуть зараз розглянуті.

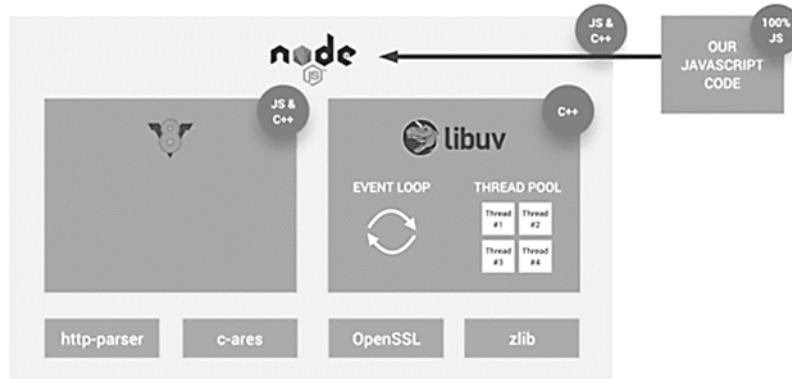


Рис. 2.2. Узагальнена архітектура Node.js

Цикл подій передбачає декілка фаз почергового опитування та виконання різних категорій задач (Рис.2.3), а саме таких:

- *таймери*: у цій фазі виконуються функції зворотного

- виклику, заплановані у таймерах `setTimeout()` і `setInterval()`;
- *pending callback*: виконуються майже всі функції зворотного виклику, за винятком подій *close*, таймерів і `setImmediate()`;
 - *idle*, *prepare* (очікування, підготовка): використовується тільки для внутрішніх потреб;
 - *poll* (опитування): отримання нових подій введення/виведення. Node.js може блокуватися на цьому етапі;
 - *check* (перевірка): функції зворотного виклику, викликані `setImmediate()`, викликаються на цьому етапі;
 - *callback close* (функції зворотного виклику події *close*): наприклад, `socket.on ('close', ...)`.

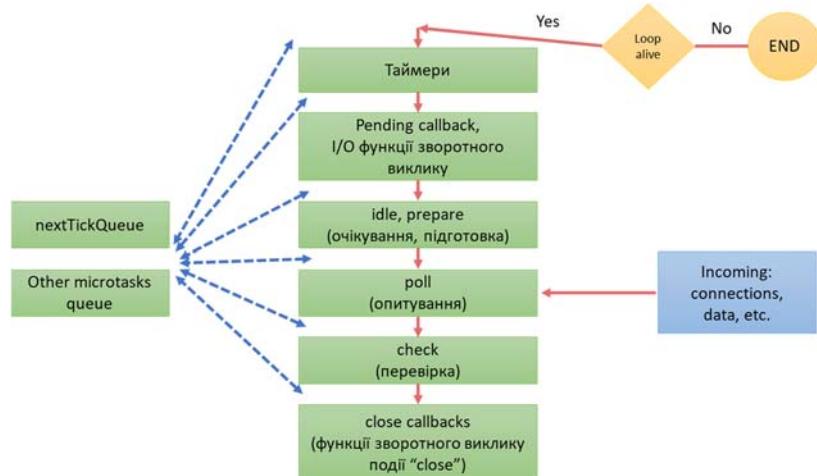


Рис. 2.3. Цикл подій (*event loop*) в Node.js

Так, наступний код, відповідності до зазначених правил

обробки, буде мати представлений на Рис.2.4 вивід.

```
const fs = require('fs');
console.log('ПОЧАТОК!');
setTimeout(()=>console.log('setTimeout 1'), 0);
setImmediate(()=>console.log('setImmediate'));
fs.readFile(__filename, ()=>{
    setTimeout(()=>console.log('readFile setTimeout'),
0);
    setImmediate(()=>console.log('readFile
setImmediate'));
    process.nextTick(()=> console.log('readFile Next
Tick'));
});
Promise.resolve().then(()=>{
    console.log('Promise');
    process.nextTick(()=> console.log('Promise Next
Tick'));
});
process.nextTick(()=> console.log('Next
Tick'));
setTimeout(()=>console.log('setTimeout 2'), 0);
setImmediate(()=>console.log('setImmediate
2'));
console.log('КІНЕЦЬ!');

> ПОЧАТОК!
КІНЕЦЬ!
Next Tick
Promise
Promise Next Tick
	setTimeout 1
	setTimeout 2
	setImmediate
	setimmdiate 2
	readFile Next Tick
	readFile setImmediate
	readFile setTimeout
```

Рис. 2.4. Черговість виконання задач в циклі подій Node.js

Операції в циклі подій виконуються в такій послідовності

[4]: спочатку виконаються всі синхронні операції, потім операції з мікроскопами (*nextTick*), проміси (*other microtasks*), таймери, функції зворотного виклику.

Розміщення власного коду в різних фазах циклу подій, з врахуванням визначених правил обслуговування, впливає на продуктивність його виконання.

У асинхронному програмуванні існують правила написання функцій зворотного виклику:

- функції зворотного викладку завжди передаються останнім аргументом;
- перший аргумент функції зворотного виклику повинен бути об'єкт помилки.

Наприклад:

```
function greeting(name) {  
    console.log(`Доброго дня ${name}`);  
}  
function introduction(firstName, lastName, callback)  
{  
    const fullName = `${firstName} ${lastName}`;  
    callback(fullName);  
}  
introduction('Влад', 'Бублик', greeting);
```

В асинхронному програмуванні зручною альтернативою функціям зворотного виклику є використання *промісів* (*Promises*).

Проміс (*promise*) – це об'єкт, що відображає кінцеве завершення або невдачу асинхронної операції [7].

Проміси мають переваги перед використанням функцій зворотного виклику, а саме дозволяють в деяких випадках уникнути сильної вкладеності та пов'язаної з нею низькою

читабельністю коду.

Об'єкт *Promise* може знаходитись в одному з таких станів:

pending (у стані очікування): початковий стан, і не виконаний, і не відхилений;

fulfilled (виконаний): означає, що операція завершилася вдало;

rejected (відхилений): означає, що операція була неуспішною.

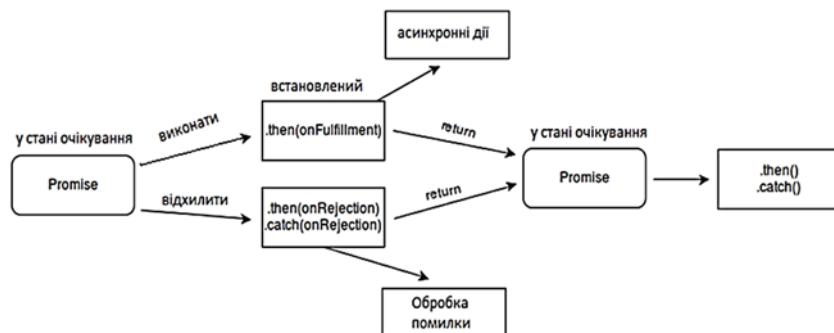


Рис. 2.5. Стани об'єкта *promise*

На відміну від *callback*-функцій, проміс постачається з певними гарантіями:

- функції зворотного виклику ніколи не будуть викликані до завершення поточного виконання циклу подій Javascript;
- функції зворотного виклику, додані за допомогою *then()*, навіть після успіху або невдачі асинхронної операції, будуть викликані;
- можливо додати більше одного зворотного виклику,

викликавши метод `then()` декілька разів. Кожен зворотний виклик виконується один за одним, у тому порядку, в якому вони були додані.

Припустимо, курсант обіцяє самостійно завершити вивчення JavaScript до наступного місяця.

Він точно не знає, чи зможе використати свій вільний час на вивчення JavaScript до наступного місяця, але має такий намір. Він може або вивчити JavaScript, або ні.

Проміс, в такому випадку, має три стани:

- *на розгляді*: курсант не знає точно, чи завершить вивчення JavaScript до наступного місяця.
- *виконано*: ви вивчаєте JavaScript до наступного місяця.
- *відхилено*: ви взагалі не вивчаєте JavaScript.

Обіцянка (проміс) починається з стану очікування, що вказує на те, що обіцянка (проміс) «зареєстрована», але не виконана. Вона закінчується або виконаним (успішним), або відхиленим (невдалим) станом (Рис. 2.6).

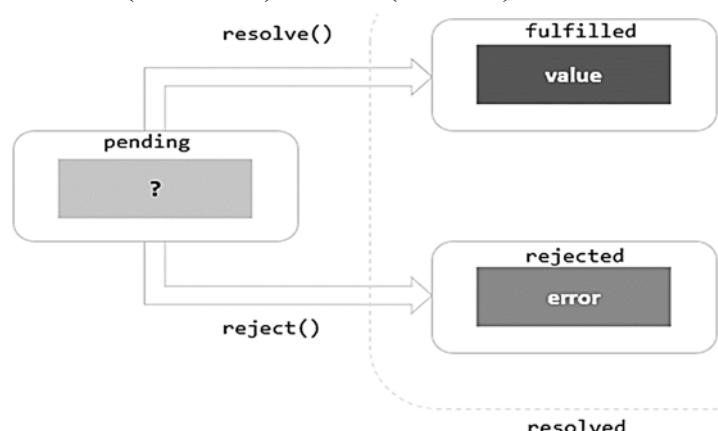


Рис. 2.6. Стани об'єкта *promise*

Усередині функції (*executor*) необхідно викликати *resolve()* функцію, якщо результат виконання функції успішний, і викликати *reject()* функцію у разі виникнення помилки. Наприклад:

```
let completed = true;
let learnJS = new Promise(function (resolve, reject) {
    if (completed) {
        resolve("I have completed learning JS.");
    } else {
        reject("I haven't completed learning JS yet.");
    }
});
```

Метод *then()* – використовується для планування зворотного виклику і буде виконуватися тоді, коли проміс успішно виконається.

Метод *then()* приймає дві функції зворотного виклику :

```
promiseObject.then(onFulfilled, onRejected);
```

Якщо необхідно запланувати виконання зворотного виклику, коли обіцянку відхилено, ви можете скористатися методом *catch()*. Внутрішньо метод *catch()* викликає метод *then(undefined, onRejected)*:

```
promiseObject.catch(
    reason => console.log(reason)
);
```

Метод *then()* використовується, щоб запланувати зворотний виклик, який буде виконаний, коли результат виконання промісу завершився успішно, а метод *catch()* – щоб запланувати зворотний виклик, який буде викликаний, коли проміс буде відхилено.

Метод *finally()* застосовується коли потрібно виконати один і той же фрагмент коду незалежно від того, виконано

проміс чи відхилено. У метод `finally()` розміщують код, який потрібно виконати, незалежно від того, виконано проміс чи відхилено.

Наприклад:

```
const promise = new Promise((resolve, reject) => {
    resolve('We did it!');
});
promise.then((response) => {
    console.log(response);
});
console.dir(promise);
```

Результат виконання промісу зображенено на Рис.2.7:

```
MacBook-Air-Oleksandr:gz_2_6 oleksvlas$ node app
Promise { 'We did it!' }
We did it!
```

Рис. 2.7. Результат виконання промісу

Для забезпечення ланцюгового передавання результатів виконання промісу необхідно обов'язково використовувати оператор `return` (Рис.2.8). Наприклад:

```
const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve('Resolving an asynchronous
request!'), 2000)
});
promise.then((firstResponse) => {
    return firstResponse + ' And chaining!'
        .then((secondResponse) => {
            console.log(secondResponse)
        })
})
```

```
MacBook-Air-Oleksandr:gz_2_6 oleksvlas$ node app
Resolving an asynchronous request!
Resolving an asynchronous request! And chaining!
```

Рис. 2.8. Створення ланцюгу промісів

Об'єкт *Promise* має власні методи.

Метод *Promise.all()* повертає єдиний проміс, який виконується, коли усі проміси *o1*, *o2*, *o3*, ..., передані у вигляді ітерабельного об'єкта, були виконані, або коли ітерабельний об'єкт не містить жодного проміса.

Наприклад:

```
var o1 = Promise.resolve(7);
var o2 = 1982;
var o3 = new Promise((resolve, reject) => {
    setTimeout(resolve, 200, "kaf_22");
});

Promise.all([o1, o2, o3]).then(val=> {
    console.log(val);
});
//Буде виведено:
// [7, 1982, "kaf_22"]
```

Метод *Promise.race()* повертає проміс, який виконується чи відхиляється, як тільки один з промісів *promise1*, *promise2*, ..., ітерабельного об'єкта буде виконаний чи відхилений, зі значенням отриманим з цього проміса.

Наприклад:

```
const promise1 = new Promise((resolve, reject) => {
    setTimeout(resolve, 700, 'first');
});

const promise2 = new Promise((resolve, reject) => {
    setTimeout(resolve, 400, 'second');
});

Promise.race([promise1, promise2]).then((val) => {
    console.log(val);
    // Both resolve, but promise2 is faster
```

```
});  
//Буде виведено: "two"
```

Метод `Promise.reject()` повертає об'єкт `Promise`, відхищений з наданоючию причиною. Нариклад:

```
.then(val => {  
    if (val != 10) {  
        return Promise.reject('Not Well');  
    }  
})  
.catch(error => console.log(error)) // Not Well
```

Метод `Promise.resolve()` повертає об'єкт `Promise`, виконаний з наданим значенням. Нариклад:

```
Promise.resolve('Success').then(function(value) {  
    console.log(value); // "Success"  
}, function(value) {  
    // не викликається  
});
```

Ще одним зручним способом написання асинхронного коду є використання функцій асинхронізації. Функція асинхронізації – це модифікація синтаксису, який використовується для написання промісів. Це полегшує їх синтаксис.

Функція асинхронізації завжди повертає проміс, якщо функція повертає значення, то це буде проміс зі значенням, але якщо функція асинхронізації видає помилку, проміс відхиляється з цим значенням [4]. Нариклад:

```
async function inc(a) {  
    return a + 1;  
}  
const sum = async function(a, b) {  
    return a + b;  
};  
const max = async (a, b) => (a > b ? a : b);
```

```

const avg = async (a, b) => {
    const s = await sum(a, b);
    return s / 2;
};
class Person {constructor(name) {
    this.name = name;
}
async split(sep = ' ') {
    return this.name.split(sep);
}}
const person = new Person('Marcus Aurelius');
(async () => {
    console.log('await inc(5) =', await inc(5));
    console.log('await sum(1, 3) =', await sum(1, 3));
    console.log('await max(8, 6) =', await max(8, 6));
    console.log('await avg(8, 6) =', await avg(8, 6));
    console.log('await obj.split() =', await
obj.split());
    console.log('await person.split() =', await
person.split());
})();

```

Результат виводу показаний на Рис.2.9.

```

C:\Users\oleksvlas\WebstormProject\gz_2_6>node app
await inc(5) = 6
await sum(1, 3) = 4
await max(8, 6) = 8
await avg(8, 6) = 7
await obj.split() = [ 'Marcus', 'Aurelius' ]
await person.split() = [ 'Marcus', 'Aurelius' ]

```

Рис. 2.9. Виконання *async/await* функцій

Оголошення *async function* визначає асинхронну функцію – функцію, яка є об'єктом *AsyncFunction*. Асинхронні функції мають окремий від решти функцій порядок виконання, через цикл подій, повертуючи неявний проміс в якості результату. Але синтаксис та структура коду, який

використовують асинхронні функції, виглядають, як стандартні синхронні функції. *Await* функції використовуються *тільки* в середині *async* функцій [7].

Іншим напрямком досягнення високої продуктивності програмних додатків є використання багатопоточності.

Багатопоточність – властивість платформи (наприклад, операційної системи, віртуальної машини тощо) або додатку, яка полягає в тому, що процес, породжений в операційній системі, може складатися з декількох потоків, що виконуються "паралельно", тобто без послідовного порядку в часу.

Обмеження багатопоточних систем:

- організація взаємодії потоків;
- конкуренція за ресурси та взаємне блокування;
- складність реалізації.

Багатопоточність – це принцип побудови програми, при якому декілька блоків коду можуть виконуватися одночасно і не заважати один одному.

Модуль *worker_threads* – це пакет, який дозволяє створювати повнофункціональні багатопотокові програми Node.js [4].

Потоковий Воркер (thread worker) – фрагмент коду (зазвичай отримують із файлу), створений в окремому потоці.

Для використання потокових Воркерів потрібно імпортувати модуль *worker_threads*.

Для створення потокового Воркер необхідно створити

екземпляр класу *Worker*. У першому аргументі вказуємо шлях до файлу, який містить код Воркер; у другому надаємо об'єкт, що містить властивість з ім'ям *workerData*. Це ті дані, до яких потік буде мати доступ при запуску, за бажанням розробника. Наприклад:

```
const worker = new Worker(path, { workerData });
```

Подія *online* генерується, коли Воркер припиняє парсинг коду JavaScript і починає його виконання. Ця подія використовується нечасто, але в певних випадках воно може бути інформативним.

```
worker.on('online', () => {});
```

Подія *message* генерується, коли Воркер відправляє дані в батьківський потік.

```
worker.on('message', (data) => {});
```

Для відправки даних іншому потоку використовується метод *port.postMessage()*. Він має наступну сигнатуру:

```
port.postMessage(data[, transferList]);
```

Об'єкт *port* може бути або екземпляром *parentPort* або *MessagePort*. Приклад виводу характеристик потоків та передавання даних з дочірнього потоку в батьківський показаний далі:

```
const threads = require('worker_threads');
const { Worker } = threads;
if (threads.isMainThread) {
    console.dir({ master: threads });
    const workerData = { text: 'Data from Master to
Worker' };
    const worker = new Worker(__filename, { workerData
});
    worker.on('message', (...args) => {
        console.log({ args });
    });
}
```

```

    worker.on('error', err => {
      console.log(err.stack);
    });
    worker.on('exit', code => {
      console.dir({ code });
    });
  } else {
    console.dir({ worker: threads });
    threads.parentPort.postMessage('Hello there!');
    setTimeout(() => {
      const data = { text: 'Message from Worker to Master' };
      threads.parentPort.postMessage(data);
    }, 1000);
  }

```

Результат виконання наведений на Рис. 2.10:

```

C:\Users\oleksvlas\WebstormProject\gz_2_6>node app
{
  master: {
    isMainThread: true,
    MessagePort: [Function: MessagePort],
    MessageChannel: [Function: MessageChannel],
    markAsUntransferable: [Function: markAsUntransferable],
    moveMessagePortToContext: [Function: moveMessagePortToContext],
    receiveMessageOnPort: [Function: receiveMessageOnPort],
    resourceLimits: {},
    threadId: 0,
    SHARE_ENV: Symbol(nodejs.worker_threads.SHARE_ENV),
    Worker: [class Worker extends EventEmitter],
    parentPort: null,
    workerData: null
  }
}
{
  worker: {
    isMainThread: false,
    MessagePort: [Function: MessagePort],
    MessageChannel: [Function: MessageChannel],
    markAsUntransferable: [Function: markAsUntransferable],
    moveMessagePortToContext: [Function: moveMessagePortToContext],
    receiveMessageOnPort: [Function: receiveMessageOnPort],
    resourceLimits: {
      maxYoungGenerationSizeMb: 48,
      maxOldGenerationSizeMb: 2048,
      codeRangeSizeMb: 0,
      stackSizeMb: 4
    },
    threadId: 1,
    SHARE_ENV: Symbol(nodejs.worker_threads.SHARE_ENV),
    Worker: [class Worker extends EventEmitter],
    parentPort: MessagePort [EventTarget] {
      [Symbol(kEvents)]: [Map],
      [Symbol(kMaxListeners)]: 10,
      [Symbol(kMaxListenersWarned)]: false,
      [Symbol(kNewListener)]: [Function (anonymous)],
      [Symbol(kRemoveListener)]: [Function (anonymous)]
    },
    workerData: { text: 'Data from Master to Worker' }
  }
}
{ args: [ 'Hello there!' ] }
{ args: [ { text: 'Message from Worker to Master' } ] }
{ code: 0 }

```

Рис. 2.10. Передавання даних між потоками

2.2.Створення серверних веб-додатків на основі фреймворку Express.js

Express – це швидкий, гнучкий, мінімалістичний фреймворк для створення веб-додатків, побудований на базі фреймворку *connect*.

Express надає широкий набір функціональних можливостей. Маючи в своєму розпорядженні безліч допоміжних HTTP-методів та проміжних обробників, дозволяє створювати легко і швидко надійні API. HTTP – це протокол прикладного рівня для передачі даних від браузера до сервера і назад. HTTP-повідомлення зазвичай передаються між сервером і браузером через порт 80 або 443 при використанні HTTPS.

Express забезпечує тонкий прошарок базової функціональності для веб-додатків, що не спотворює звичну та зручну функціональність Node.js.

На *Express.js* ґрунтуються багато популярних фреймворків.

Для встановлення фреймворку *Express.js* необхідно [4]:

- ініціалізувати проект Node.js для створення файлу *package.json* командою:

npm init

- встановити *Express.js* в каталозі програмного додатку і додати його до списку залежностей:

npm install express --save

Модулі Node.js, встановлені з опцією *--save*, додаються в список *dependencies* в файлі *package.json*. Надалі, при

виконанні команди *npm install* в каталозі програмного додатку, встановлення модулів зі списку залежностей буде виконуватися автоматично. Для використання фреймворку, необхідно підключити в проект модуль *express*, і створити веб-сервер додатку. Наприклад:

```
let express = require('express');
let app = express();
```

Робота з *Express* основана на викладеній далі понятійній базі.

Request (*запит*) – запит HTTP. Клієнт надсилає повідомлення із запитом HTTP на сервер, який повертає відповідь. Запит повинен використовувати один із декількох методів запиту, таких як GET, POST тощо.

Response (*відповідь*) – відповідь HTTP. Сервер повертає клієнту повідомлення відповіді HTTP. Відповідь містить інформацію про стан завершення запиту, а також може містити запитуваний вміст у тілі повідомлення.

Route (*роут, маршрут*) – частина URL-адреси, яка ідентифікує ресурс. Наприклад, у *http://viti.edu.ua/news/id* “/news/id” – це маршрут.

Router (*маршрутізатор*) – набір однотипних маршрутів, для певної кореневої URL-адреси.

Middleware (*проміжне програмне забезпечення*) – функція, яка викликається на проміжному шарі маршрутизації перед кінцевою функцією-обробником запиту у *Express.js*. Таким чином, вона знаходиться посередині між необробленим запитом та кінцевим запланованим маршрутом.

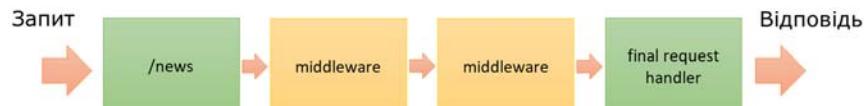


Рис. 2.11. Використання *middleware* в *Express.js*

Маршрутизація визначає, як серверний додаток відповідає на клієнтський запит до конкретної адреси і залежить від певного методу HTTP-запиту.

Кожен маршрут може мати кілька обробників, які виконуються при співпадінні маршруту.

Для визначення маршруту використовують наступну структуру (Рис.2.12) [4]:

```
app.method(path, handler);
app – екземпляр express-додатку;
method – метод HTTP-запиту;
path – шлях на сервері;
handler – функція обробник на вказаний шлях.
```

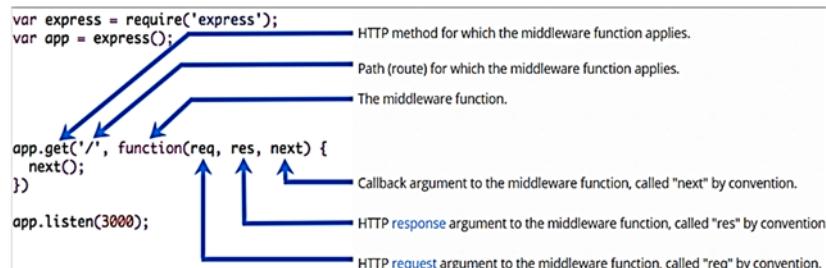


Рис. 2.12. Використання *middleware* в *Express.js*

Метод маршрутизації (*route*) є похідним від одного з методів HTTP і приєднується до екземпляру класу *express*.

```
let express = require('express');
let app = express();
app.get('/', function(req, res) {
    res.send('Hello World!');
});
app.post('/', function(req, res) {
    res.send('Got a POST request');
});
app.put('/user', function(req, res) {
    res.send('Got a PUT request at /user');
});
```

Express.js підтримує такі перераховані методи маршрутизації, які відповідають методам HTTP: *get*, *post*, *put*, *head*, *delete*, *options*, *trace*, *copy*, *lock* тощо. Найпростіший додаток в *Express.js* представлений на Рис.2.13.

The screenshot shows a terminal window titled 'app.js' containing the code for a simple Express.js application. The code defines a GET route for the root path ('/') that sends the response 'Hello cadet!'. It also includes code to start the server on port 3000 and log the listening port. Below the terminal is a browser window showing the result of visiting 'localhost:3000', which displays the message 'Hello cadet!'.

```
const express = require('express');
const app = express();
const port = 3000;
app.get('/', (req, res) => {
    res.send('Hello cadet!');
});
app.listen(port, () => {
    console.log(`Example app listening at ${port}`);
});
```

C:\Users\oleksvlas\WebstormProject\gz_3_1>node app
Example app listening at 3000

localhost:3000
← → ⌂ ⓘ localhost:3000
Hello cadet!

Рис. 2.13. Найпростіший додаток в *Express.js*

Функції проміжної обробки (*middleware*) мають доступ до об'єкта запиту (*req*), об'єкту відповіді (*res*) і до наступної функції *middleware* в циклі "запит-відповідь" додатка (*next*).

Функції *middleware* можуть використовуватися для виконання таких завдань:

- виконання будь-якого проміжного коду;
- внесення змін до об'єктів запитів і відповідей;
- завершення циклу "запит-відповідь";
- виклик наступного проміжного обробника з стека.

Для роботи з *middleware* використовується метод *use* [4].

Наприклад (Рис.2.14):

The screenshot shows the execution of an Express.js application. On the left, a terminal window displays the command `C:\Users\oleksvlas\WebstormProject\gz_3_1>node app`, followed by the output: `Example app listening at 3000` and `LOGGED`. An orange arrow points from this terminal to a browser window on the right. The browser window has the address `localhost:3000` and displays the message `Hello cadet!`.

```
app.js
const express = require('express');
const app = express();
const port = 3000;
let myLogger = function (req, res, next) {
    console.log('LOGGED')
    next()
}
app.use(myLogger);
app.get('/', (req, res) => {
    res.send('Hello cadet!');
});
app.listen(port, () => {
    console.log(`Example app listening at ${port}`);
});
```

Рис. 2.14. Використання *middleware* в *Express.js*

Об'єкт запиту (*request*) передається через перший аргумент функції обробника маршруту і його прийнято називати – *req* (*request*), але можна використовувати будь-яку назву (Рис. 2.15).



Рис. 2.15. Аргументи функції обробника маршруту

Об'єкт запиту (request) в Node.js має такі властивості:
query – об'єкт, що містить всі GET-параметри запиту.

Наприклад:

```
Запит: /api/books?page=3&limit=15  
{  
    query: {  
        page: 3,  
        limit: 15  
    },  
    ...  
}
```

body – об'єкт, який зберігає дані, що передаються POST або PUT-запитом;

cookies – значення *cookies*-файлів;

headers – об'єкт з усіма HTTP-заголовками запиту.

Наприклад:

```

{
  headers: {
    host: '127.0.0.1:7000',
    connection: 'keep-alive',
    'cache-control': 'max-age=0',
    ...
  },
}

```

url – містить маршрут з GET-параметрами, за яким визначається кінцевий обробник. Наприклад:

```

const express = require("express"), router =
express.Router(),
router.get('/books', (req, res) => {
/*
  Запит: /api/books?page=3&limit=15
  Результат: /books?page=3&limit=15
*/
  console.log(req.url);
});
app.use('/api', router)

```

route – об'єкт маршруту Node.js, екземпляр класу *Route*, який обробляє запит (зазвичай використовується для налагодження);

path – URL кінцевого маршруту, не містить дані протоколу, хоста, порту і GET-параметри. Наприклад:

```

const express = require("express"),
router = express.Router(),
router.get('/books', (req, res) => {
/*
  Запит: /api/books?page=3&limit=15
  Результат: /books
*/
  console.log(req.path);
});
app.use('/api', router);

```

ip – IP-адреса ініціатора запиту (зазвичай клієнта);
hostname – хост ініціатора запиту;
protocol – протокол, з використанням якого був відправлений запит (*http* або *https*);
secure – логічне значення, *true*, якщо запит був відправлений по протоколу *https*;
xhr – логічне значення, *true*, якщо запит був відправлений AJAX-запитом.

Об'єкт *відповіді* (*response*) в Node.js використовується для відправки повідомлення про результати опрацювання отриманого запиту.

Властивості об'єкта *відповіді* (*response*) Node.js:

headersSent – логічне значення, *true*, якщо заголовки відповіді вже були відправлені. Наприклад:

```
app.get('/', (req, res) => {
    console.log(res.headersSent); //false
    res.status(200).send({message: 'ok'});
    console.log(res.headersSent); //true
    return;
});
```

cookie (*key, value*) – встановлює значення *cookie*-файлів;

clearCookie() – здійснює очищення файлів *cookie*;

download(filename) – пропонує у відповідь на запит завантажити файл. Наприклад:

```
app.get('/user-profile', (req, res) => {
    return res.status(200).download('profile.png');
});
```

redirect(code ?, url) – перенаправляє запит на заданий

URL з 302 статусом за замовчуванням, в якості першого параметра можна вказати інший код відповіді. Наприклад:

```
app.get('/', (req, res) => {
    return res.redirect(301, '/login');
});
```

render() – використовується для генерації представлень використаного шаблонізатора.

status() – використовується для установки коду відповіді, але відповідь не відправляється. Наприклад:

```
app.get('/', (req, res) => {
    res.status(200); // встановлює код відповіді 200
    return res.send({ message: 'ok' }); // відповідь
    відправлений
});
```

set() – використовується для вказівки заголовків відповіді. Наприклад:

```
app.get('/', (req, res) => {
    res.set('Content-Type', 'text/plain');
    ...
});
```

type() – задає тип заголовків відповіді, що відправляються разом з даними. Наприклад:

```
app.get('/', (req, res) => {
    res.type('text/plain');
});
```

send() – відправляє відповідь на запит, в якості параметра приймає дані для відправки, при цьому рекомендується явно вказувати для даних їх тип. Наприклад:

```
app.get('/', (req, res) => {
    res.type('text/plain');
```

```
    return res.send('Any data');
});
```

sendStatus() – задає статус і відразу відправляє порожню відповідь;

json() – приймає дані в форматі JSON і відправляє їх в якості відповіді з правильно зазначенім заголовком *Content-Type*. Наприклад:

```
app.get('/', (req, res) => {
  return res.json({message: 'ok'});
})
```

end() – відправляє відповідь з усіма зазначеними до її виклику параметрами, не приймаючи ніяких параметрів. Наприклад:

```
app.get('/', (req, res) => {
  res.type('text/plain');
  res.status(201);
  res.end();
});
```

Для того щоб сервер з Node.js міг передавати на вимогу клієнта статичні файли (зображення, аудіо, HTML, CSS, JS), використовується функція фреймворка *Express* – *static()*.

Наприклад:

```
app.use(express.static(__dirname + '/public'));
```

Функція повинна викликатися як *middleware* з використанням методу *use()*. Як параметр *static()* приймає ім'я директорії, в якій знаходяться всі статичні файли.

При запиті самих файлів вказувати в URL директорію не потрібно. Пошук буде здійснюватися щодо зазначененої директорії. Наприклад:

```

app.use('/photos',
express.static(`__dirname/public/img`));
app.use('/styles',
express.static(`__dirname/public/css`));

```

Наприклад, маємо статичні файли (Рис.2.16):

The screenshot shows two code files side-by-side.

index.html:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Приклад роботи статичних файлів</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
<h1>Статичний файл відправлено!</h1>
</body>
</html>

```

style.css:

```

body {
    background-color: tomato;
}

```

Рис. 2.16. Статичні файли для відправки по запиту

Використаємо метод *static()*, що приймає ім'я директорії, в якій знаходяться всі статичні файли для відображення.

The screenshot shows the *app.js* file and its directory structure.

app.js:

```

const express = require('express');
const app = express();
const port = 3000;
app.use(express.static(__dirname + '/public'))
app.get('/', (req, res) => {
    res.sendFile(__dirname + '/public/index.html');
});
app.listen(port, () => {
    console.log(`Example app listening at ${port}`);
});

```

Directory Structure:

```

gt_3.1 C:\Users\aleksaf\WebsiterProject\gt_3.1
  node_modules
  public
    index.html
    style.css
  app.js
  package.json
  package-lock.json

```

A red arrow points from the *public* folder in the directory structure to the *static()* call in the *app.js* file.

Browser Preview:

Приклад роботи статичних файлів
localhost:3000
Статичний файл відправлено!

Рис. 2.17. Результат використання методу *static()*

Екземпляр фреймворка *Express* має методи, що збігаються за назвою з основними методами HTTP для передачі даних (*get*, *post*, *put*, *delete*) і за допомогою яких формується Node.js маршрутизація. Кожен з цих методів приймає два параметри:

- URL-адресу, за якою буде здійснюватися запит;
- функцію-обробник запиту (яка приймає в якості аргументів об'єкт запиту та об'єкт відповіді).

Передані дані та інформація про відправника запиту міститься в об'єкті запиту, а за допомогою об'єкта відповіді формується відповідь на запит.

При описі маршрутів в Node.js допускається використання спеціальних символів. Так, символ «?», вказує на необов'язковість символу, після якого він іде. Наприклад:

```
app.get('/api/users?', (req, res) => {...});
```

Якщо необхідно вказати необов'язковість групи символів, необхідно взяти їх в круглі дужки «()».

Наприклад:

```
// /api/user ma /api/userrs  
app.get('/api/user(rs)?', (req, res) => {...});
```

Символ «+» означає, що попередній йому символ може повторюватися в цьому місці необмежену кількість разів.

Наприклад:

```
// /api/users, /api/userss, /api/usersss, ...  
app.get('/api/users+', (req, res) => {...});
```

Символ «*» означає, що на його місці може бути абсолютно будь-яка послідовність символів, що не обмежена по довжині. Наприклад:

```
// /api/users, /api/clients, /api/cart, ...
app.get('/api/*', (req, res) => {...});
```

Також в Node.js для забезпечення маршрутизації підтримуються регулярні вирази при визначені URL. Наприклад:

```
// /api/users/1, /api/clients/2, ...
app.get('/api/.*s\[0-9]+$', (req, res) => {...});
```

Метод *all()* дозволяє визначити один обробник на всі типи HTTP запитів по заданому URL.

```
// GET | POST | PUT | DELETE /api/users
app.all('/api/users', (req, res) => {...});
```

При описі маршрутизації важливий порядок визначення маршрутів. Запит потрапляє в перший відповідний обробник і далі пошук вже не здійснюється.

Параметри маршруту називаються сегментами URL-адрес, які використовуються для фіксації значень, зазначених у їхній позиції в URL-адресі. Отримані значення зберігаються в об'єкті *req.params*, іменем параметра маршруту, зазначенім у шляху, є відповідні ключі.

Наприклад:

```
//Route path: /users/:userId/books/:bookId
//Request URL:
http://localhost:3000/users/34/books/8989
// req.params: { "userId": "34", "bookId": "8989" }
app.get('/users/:userId/books/:bookId', function (req,
res){
```

```
res.send(req.params);
});
```

Модуль *body-parser* надає різні функції-фабрики для створення проміжних оброблювачів *middleware*. Всі такі посередники заповнюють властивість *req.body* обробленою інформацією, якщо заголовок запиту *Content-Type* відповідає встановленому типу.

Модуль забезпечує наступні типи парсерів:

- json body parser;
- raw body parser;
- text body parser;
- URL-encoded from body parser.

Встановлення модуля у проект здійснюється командою [4]:

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

Розглянемо приклад використання данного модуля для отримання даних з форми. Для цього створимо найпростішу сторінку з формою для відправки за маршрутом */login* (Рис.2.18). Форма складається з елемента вводу даних та кнопки для їх відправки. До форми введемо дані про ім'я користувача і натиснувши на кнопку, відправимо до сервера. Для обробки даних форми на серверній стороні буде наступний код (Рис 2.19):

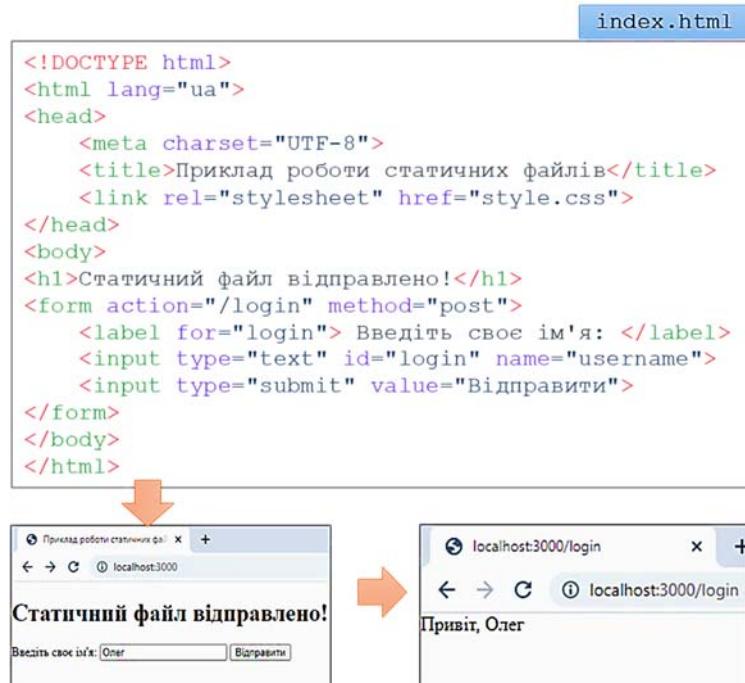


Рис. 2.18. Результат відправки запиту методом *post*

```

const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const port = 3000;
app.use(express.static(__dirname + '/public'));
let urlencodedParser = bodyParser.urlencoded({ extended: false });
app.get('/', (req, res) => {
    res.sendFile(__dirname + '/public/index.html');
});
app.post('/login', urlencodedParser, (req, res) =>{
    console.dir(req.body);
    res.send('Привіт, ' + req.body.username);
});
app.listen(port, () => {
    console.log(`Example app listening at ${port}`);
});

```

Рис. 2.19. Сервер з обробкою даних форми

Дані з форми можна відправити на сервер альтернативним способом з використанням AJAX. Розглянемо цей спосіб детальніше. Для цього створимо просту сторінку з формою (Рис.2.20). Дані форма буде відправлятися на сервер Node.js при натисканні на кнопку «Відправити» та мати вигляд як на Рис. 2.18. Обробка натискання на кнопку буде здійснена з використанням скрипта «*main.js*» на основі AJAX-запиту.

```
ajax.html
<!DOCTYPE html>
<html lang="ua">
<head>
    <meta charset="UTF-8">
    <title>Приклад роботи статичних файлів</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>Статичний файл відправлено!</h1>
    <form name="actionForm">
        <label for="username"> Введіть своє ім'я: </label>
        <input type="text" id="username" name="username">
        <button type="submit"
id="submit">Відправити</button>
    </form>
    <script src="main.js"></script>
</body>
</html>
```

Рис. 2.20. Код простої сторінки з формою

Для виконання AJAX-запиту необхідно на клієтській стороні використати об'єкт *XMLHttpRequest* (Рис.2.21):

```

main.js
document.getElementById("submit").addEventListener("click",
function (e) {
    e.preventDefault();

    let actionForm = document.forms["actionForm"];
    let userName = actionForm.elements["username"].value;

    let user = JSON.stringify({userName: userName});
    let request = new XMLHttpRequest();

    request.open("POST", "/login", true);
    request.setRequestHeader("Content-Type",
"application/json");
    request.addEventListener("load", function () {
        console.log(request.response);
    });
    request.send(user);
});

```

Рис. 2.21. Код виконання AJAX запиту

Серверна сторона буде мати наступний код (Рис.2.22):

```

app.js
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const port = 3000;
app.use(express.static(__dirname +'/public'));
let jsonParser = bodyParser.json();
app.get('/', (req, res) => {
    res.sendFile(__dirname +'/public/ajax.html');
});
app.post('/login', jsonParser, (req, res)=>{
    console.log(req.body);
    res.send('Привіт, ' + req.body.username);
});
app.listen(port, () => {
    console.log(`Example app listening at ${port}`);
});

```



C:\Users\oleksvlas\WebstormProject\gz_3_1>node app
Example app listening at 3000
{ userName: 'Oleg' }

Рис. 2.22. Код серверної сторони та результат його виконання

Як бачимо результат такий же. Перед відправкою дані серилізуються з використанням методу *stringify* для JSON обєкта:

```
let user = JSON.stringify({userName: userName});
```

Потім відправляються з використанням AJAX при натисненні на кнопку «Відправити». На серверній стороні для їх коректної інтерпретації використовується метод *json()* об'єкта *bodyParser*.

```
let jsonParser = bodyParser.json();
app.post('/login', jsonParser, (req, res) => {
  console.log(req.body);
  res.send('Привіт, ' + req.body.username);
});
```

Отримані дані відправляються в якості відповіді.

Також у Node.js для генерації і формуванні контенту HTML-сторінки використовується 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.

Налаштування Node.js шаблонізатора здійснюється завданням двох параметрів [7]:

views – шлях до директорії, в якій знаходяться шаблони;

view engine – вказівник ядра шаблонізатора.

Для завдання цих параметрів використовується метод *Express set()*.

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

Шаблонізатор *EJS* – це проста мова шаблонів, яка дозволяє генерувати розмітку HTML на основі JavaScript

[4].

Особливості шаблонізатора *EJS*:

- швидка компіляція та рендеринг;
- прості теги шаблону: `<% %>`;
- спеціальні роздільники (наприклад, використовуйте `[??]` замість `<% %>`);
- можливість розділення на підшаблони;
- підтримка javascript як на стороні сервера, так і браузера;
- статичне кешування проміжного javascript;
- статичне кешування шаблонів;
- відповідає системі *express view*.

Встановлення шаблонізатор *Ejs* до проєкту:

```
npm install ejs --save
```

За допомогою методу `engine()` здійснюється налаштування *Ejs*, вказується шаблон за замовчуванням, в який будуть довантажуватися шаблони сторінок.

Генерація і віддача представлення здійснюється за допомогою методу `render()`, який приймає два параметри:

- файл шаблону;
- дані для шаблону у вигляді об'єкта (якщо необхідно).

Шаблони *Ejs* представляють собою звичайні файли HTML в форматі *ejs*, в яких за допомогою спеціального синтаксису виводяться дані, що передаються.

Для відображення значення властивості переданого об'єкта використовується запис: `<%= назва_властивості %>`

Приклади роботи з шаблонізатором *EJS* показано на Рис.

2.23 - 2.26:

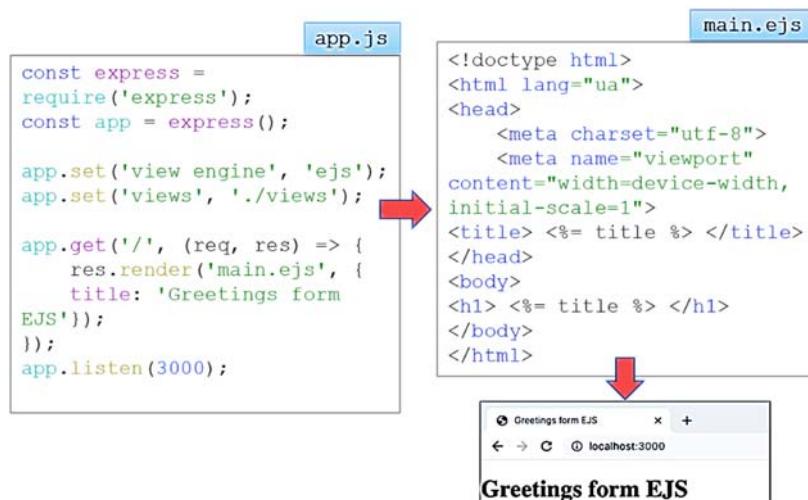


Рис. 2.23. Використання шаблонізатора *Ejs*

При наявності декілької маршрутів:

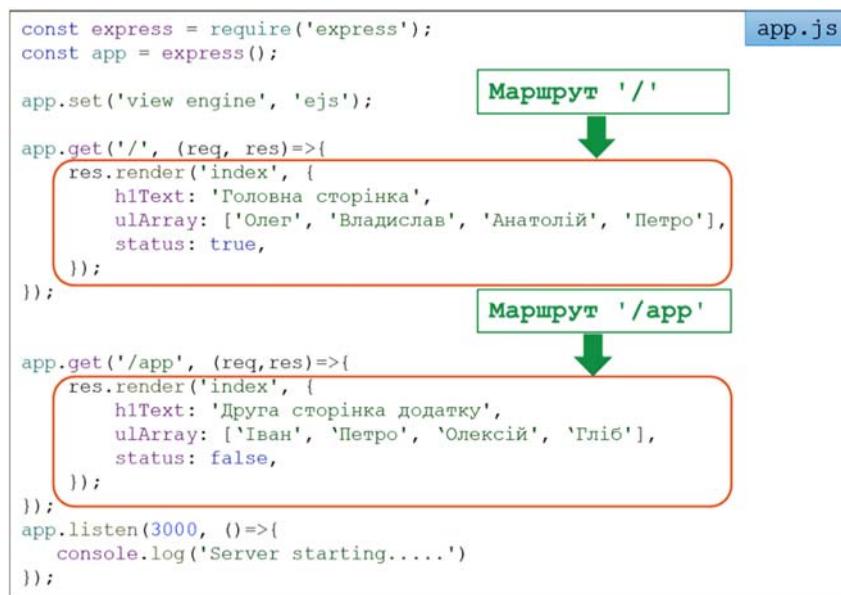


Рис. 2.24. Маршрутизація з використанням шаблонізатора

The screenshot shows a code editor with an EJS template file named `index.ejs`. The file contains the following code:

```
<!doctype html>
<html lang="ua">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, user-scalable=no,
        initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>EJS</title>
</head>
<body>
    <h1><%= h1Text %></h1> 1
    <ul>
        <% ulArray.forEach((item)=>{ %>
            <li><%=item %></li>
        <% }); %>
    </ul>
    <br>
    <% if (status) { %>
        <h2>Гарного Вам дня!!!!</h2>
    <% } %>
    </body>
</html>
```

Three numbered callouts point to specific parts of the code:

- Callout 1 points to the `<h1><%= h1Text %></h1>` line.
- Callout 2 points to the `<% ulArray.forEach((item)=>{ %>` line.
- Callout 3 points to the `<% if (status) { %>` line.

To the right of the code editor is a file tree window showing the project structure:

- trpz_3_1 (~/Documents/03_projects/trpz_3_1)
- node_modules (library root)
- views
 - index.ejs
 - app.js
 - package.json
 - package-lock.json

Рис. 2.25. Використання шаблонізатора *Ejs*

Результат рендерингу зображенено на Рис. 2.26:

Маршрут '/'

Маршрут '/app'

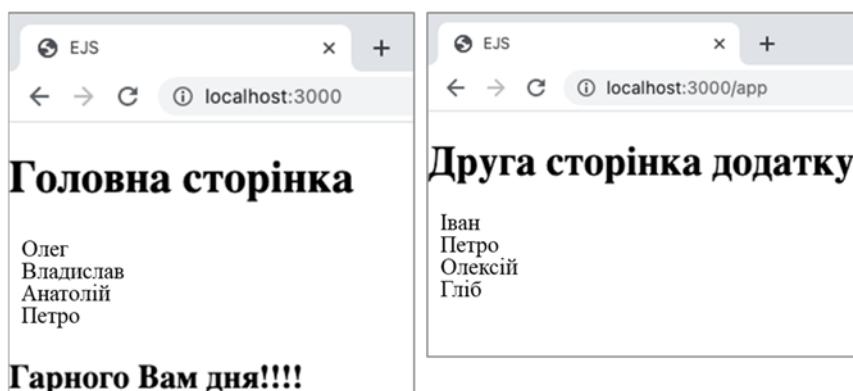


Рис. 2.26. Рендерінг сторінок на основі шаблонізатора *Ejs*

Робота з шаблонізаторами також передбачає використання шаблонів, які знаходяться в зовнішніх файлах. Для цього використовується оператор `-include()`. Наприклад, Рис.2.27:

The diagram shows the rendering process of an EJS template. On the left, the main template file `./views/index.ejs` contains code including an `<%- include('header', h1Text); %>` directive. This directive points to the `./views/header.ejs` file on the right, which contains the rendered content `<h1><%=h1Text %></h1>`. A red arrow indicates the flow of data from the header template back to the main template.

```
<!doctype html>
<html lang="ua">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, user-scalable=no,
        initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>EJS</title>
</head>
<body>
<%- include('header', h1Text); %>
<ul>
<% ulArray.forEach((item)=>{ %>
    <li><%=item %></li>
<% }); %>
</ul>
<br>
<% if (status) { %>
    <h2>Гарного Вам дня!!!!</h2>
<% } %>
</body>
</html>
```

Рис. 2.27. Рендерінг на основі зовнішніх файлів з шаблонами *Ejs*

Результат рендерингу, після підключення файлу `header.ejs`, буде аналогічний тому, як зображено на рис. 2.26.

2.3.Характеристики RESTful API

REST – архітектурний стиль взаємодії компонентів розподіленого додатка в мережі. Він є узгодженим набором обмежень, що враховуються при проєктуванні розподіленої інформаційної системи.

API (Application Programming Interface – інтерфейс

програмних додатків) – це набір функцій і правил, що дозволяє взаємодіяти між програмним забезпеченням, яке надає API і іншими програмними компонентами [10]. У Веб-розробці, під API зазвичай розуміють набір стандартних методів, властивостей, подій і URL-посилань для взаємодії з Веб-контентом.

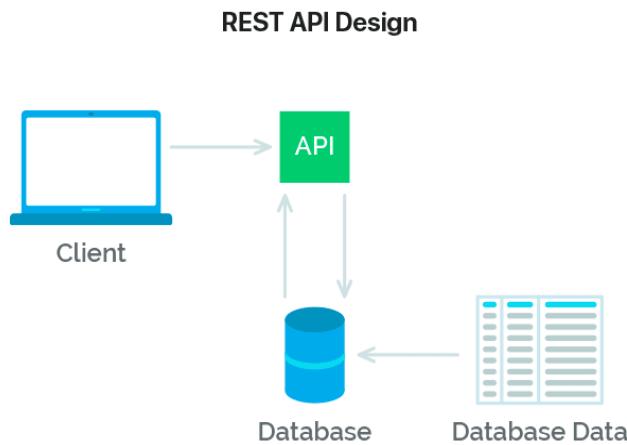


Рис. 2.28. Архітектура REST

Основні характеристики RESTful API:

- має високий ступінь масштабованості, оскільки різні компоненти можуть бути розгорнуті незалежно один від одного на різних серверах;
- використовує HTTP-методи, такі як: *get, post, delete, put, options* тощо;
- підтримує різні формати передавання даних, включаючи JSON (є найпопулярнішим).

Особливості використання методів http в REST:

- метод *get* повинен бути використаний тільки для

отримання (*retrieval*). Він ніколи не повинен використовуватися для створення, оновлення та ін.;

– метод *post* повинен бути використаний для оновлення або отримання даних. Якщо URI до цього не було і є необхідність його створити, то використовується метод *post*;

– метод *put* повинен використовуватися для оновлення (*updating*), що означає заміну (*replacing*). URI повинен вже існувати;

– метод *delete* повинен використовуватися для видалення (*deleting*).

Співставлення основних методів http, SQL та CRUD-операцій, наведено на Рис.2.29:

HTTP	POST	GET	PUT	DELETE
SQL	INSERT	SELECT	UPDATE	DELETE
CRUD	CREATE	READ	UPDATE	DELETE

Рис.2.29. Співставлення основних методів http, SQL та CRUD-операції

Протокол HTTP повертає коди за результатами виконання запитів [8]. Далі наведено стандартні коди, які повинні повертати сервіси:

200: "Done, it was okay." Зазвичай *get* повертає цей код.

201: "Done, and created." Зазвичай *post*, *put* повертає цей код.

204: "Done, and no body." Зазвичай *delete* повертає цей

код.

400: "Client sent me junk, and I'm not going to mess with it."

401: "Unauthorized, the client should authenticate first."

403: "Not allowed. You can not have it because you logged in but do not have permission to this thing or to delete this thing."

404: "Can not find it."

410: "Marked as deleted."

451: "The government made me not show it."

Обмеження RESTful API:

– єдиний інтерфейс (*Uniform Interface*) – ресурс в системі повинен мати тільки один логічний URI, а потім повинен надавати спосіб отримання додаткових даних.

– відсутність станів (*Stateless*) – необхідний стан для обробки запиту міститься в самому запиті, або в рамках URI, параметрах рядка запиту, тіла або заголовках. URI унікально ідентифікує ресурс і тіло містить стан (або зміну стану) цього ресурсу.

– кешування відповіді (*Cacheable*) – в REST кешування має застосовуватися до ресурсів, коли це можливо зробити, і тоді ці ресурси повинні оголошувати себе як кешовані. Кешування може бути реалізовано на стороні сервера або клієнта.

– клієнтська і серверна частини додатку повинні мати можливість розвиватися окремо, незалежно один від одного. Клієнт повинен знати тільки URI ресурсів.

– багаторівнева система (*Layered System*) – REST дозволяє використовувати багаторівневу систему

архітектуру, де є можливість розгорнути API на одному сервері, і, наприклад, зберігати дані на іншому сервері, а аутентифікувати запити на третьому. Клієнт не повинен зазвичай знати, підключений він безпосередньо до кінцевого сервера чи до посередника.

– код на вимогу (*Code on Demand*) – більшу частину часу дані будуть відправлятися у форматі XML або JSON. Але за необхідністю, можна повернути виконуваний код для забезпечення підтримки певної частини додатку.

Для роботи зі зовнішніми API використовуються модулі *request* або *superagent*. Модул *request* є аналогом модулю *request-promise*, який базується на промісах.

Інсталяція модулю *request* або *request-promise* в проект здійснюється командою [4]:

npm install --save request

або

npm install --save request-promise

Підключення модулю *request* до проекту:

const request = require('request');

Підключення модулю *request-promise* до проекту :

const request = require('request-promise');

Наприклад, скористаємося даними, які надають статистику про кількість захворювань на Covid19 в Україні з зовнішнього ресурсу для демонстрації можливостей модулю *request*:

```
let request = require("request");
let options = {
  'method': 'GET',
```

```

'url':
'https://api.covid19api.com/davone/country/ukraine/stat
us/confirmed',
'headers': {
},
};

request(options, function (error, response, body) {
    if (error) throw new Error(error);
}
    let obj = JSON.parse(body)
console.log(obj);
});

```

З використанням модулю *request-promise* рішення буде наступним:

```

let request = require("request-promise");
const options = {
    method: 'GET',
url:
'https://aoi.covid19api.com/davone/country/ukraine/stat
us/confirmed',
    headers: {
        },
    json: true
};
request(options)
    .then(function (response) {
        console.log(response);
    })
    .catch(function (err) {

// обробка помилки
});

```

Отримаємо однаковий результат:

```
[{Country: 'Ukraine',
  CountryCode: 'UA',
  Province: '',
  City: '',
  CityCode: '',
  Lat: '48.38',
```

```
    Lon: '31.17',
    Cases: 12697,
    Status: 'confirmed',
    Date: '2021-05-05T00:00:00Z' },
{ Country: 'Ukraine',
  CountryCode: 'UA',
  Province: '',
  City: '',
  CityCode: '',
  Lat: '48.38',
  Lon: '31.17',
  Cases: 13184,
  Status: 'confirmed',
  Date: '2021-05-06T00:00:00Z'}]
```

Таким чином показана можливість отримувати дані з зовнішніх ресурсів при наявності на них відповідних API. Так отримані дані можуть бути використані у власному додатку.

Контрольні запитання до розділу 2

1. Що таке цикл подій?
2. Чи є цикл подій частиною рушія *V8*?
3. Крім движка *V8* і бібліотеки *libuv*, які ще зовнішні залежності є у Node?
4. Яким чином здійснюється взаємодія між браузером і сервером?
5. В чому полягає різниця між методами *get* і *post*, що використовуються при роботі протоколу *http*?
6. Яким чином використовувати *async/await* в Node.js??
7. Чому використання блокуючих синхронних операцій є поганою практикою в Node.js?
8. У чому різниця між *setTimeout*, *process.nexttick* і *setImmediate*?
9. Що таке цикл подій в Node.js та для чого він потрібен?
10. Чи виконуються асинхронні функції паралельно?
11. Поясніть чи є Node.js багатопотоковим?
12. Що таке *Callback Hell*?
13. Яким чином можна уникнути *Callback Hell* в Node.js?
14. Які шаблонизатори підтримує Express?
15. Як відобразити простий HTML-файл в браузері з використанням Node.js?
16. Для чого використовуються функції *middleware* в додатках на Node.js?

ОСНОВИ РОБОТИ З БАЗАМИ ДАНИХ В NODE.JS

3.1. Загальна характеристика та можливості засобів Node.js для роботи з СКБД MongoDB

MongoDB – це крос-платформна, документо-орієнтована база даних, яка забезпечує високу продуктивність і легку масштабованість [5]. В основі даної БД лежить концепція колекцій і документів.

Основні переваги СКБД MongoDB:

- відсутність схеми. БД заснована на колекціях різних документів. Кількість полів, зміст і розмір цих документів може відрізнятися. Тобто різні сутності не обов'язково повинні бути ідентичні за структурою;
- проста та зрозуміла структура кожного об'єкта;
- легке масштабування;
- для зберігання даних використовуваних внутрішня пам'ять, що дозволяє отримувати більш швидкий доступ;
- дані зберігаються у вигляді JSON документів;
- MongoDB підтримує динамічні запити документів (document-based query);
- відсутність складних JOIN-запитів.

База даних в MongoDB представлена у вигляді фізичного сховища колекцій. Кожна БД має свій власний набір файлів в файловій системі. Зазвичай, один MongoDB сервер має декілька БД (Рис. 3.1).

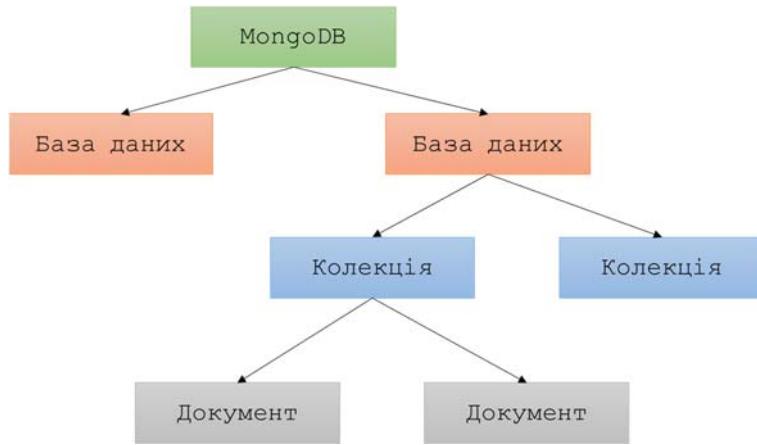


Рис. 3.1. Структура даних у MongoDB

Колекція – це група документів MongoDB [5]. Є еквівалентом простої таблиці в реляційній базі даних. Колекція знаходиться всередині однієї БД. Документ в колекції може мати різні поля. Найчастіше, всі документи в колекції створені для однієї сутності предметної області (Рис. 3.2).

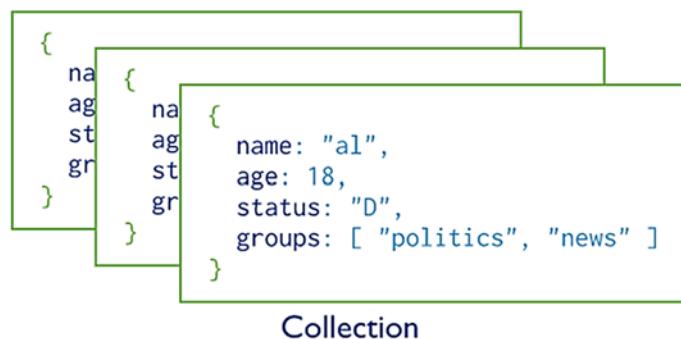


Рис. 3.2. Колекція документів у СКБД MongoDB

Документ – це набір пар "ключ-значення". Документ має динамічну схему. Це означає, що документ в одній і тій же колекції не обов'язково повинен мати один одинаковий набір полів або структуру, а загальні поля в колекції можуть мати різні типи даних.

Документи MongoDB складаються з пар ключ-значення та мають структуру:

```
{  
    field1: value1,  
    field2: value2,  
    field3: value3,  
    ...  
    fieldN: valueN  
}
```

Значення поля може бути будь-яким із типів даних BSON, включаючи інші документи, масиви та масиви документів.

При підключені і подальшій взаємодії з БД СКБД MongoDB в Node.js можна окреслити наступні етапи:

- підключення до СКБД;
- отримання об'єкта бази даних;
- отримання об'єкта колекції в базі даних;
- робота з документами поточної колекції (додавання, видалення, вибірка, оновлення даних).

Для інсталяції драйверу підключення до СКБД MongoDB в Node.js необхідно виконати команду [4]:

```
npm install mongodb
```

Для підключення до бази даних MongoDB використовується клас *MongoClient*, що входить до складу

модуля *mongodb*. На основі його використання забезпечується взаємодія з базою даних. Тому для подальшої роботи необхідно імпортувати клас *MongoClient*:

```
const MongoCli = require("mongodb").MongoClient;
```

Для з'єднання з СКБД MongoDB застосовується метод *connect()*:

```
const MongoCli = require("mongodb").MongoClient;
// створюється об'єкт MongoClient
const mongoClient = new
MongoCli("mongodb://localhost:27017/", {
useUnifiedTopology: true });
mongoClient.connect(function(error, client){
    if(error){
        return console.log(error);
    }
    // операції взаємодії з базою даних
    client.close();
});
```

На основі функції конструктора створюється об'єкт *MongoClient*. В конструктор передається два параметри: адреса сервера, що включає в себе протокол адреси "*mongodb://*", IP-адресу, та номер порту (за замовчуванням номер порту який прослуховує СКБД MongoDB – 27017); *useUnifiedTopology: true* – це об'єкт конфігурації MongoDB.

Метод *connect()* забезпечує підключення до сервера. Даний метод приймає функцію зворотного виклику, яка виконується при встановлені з'єднання з СКБД. Ця функція приймає два параметри: *error* (об'єкт помилки, що може виникнути під час активного з'єднання) і *client* (посилання на підключений до сервера клієнт).

При відсутності помилок, можна взаємодіяти з сервером через об'єкт *client*.

Після здійснення операцій взаємодії з базою даних необхідно завершити з'єднання за допомогою методу *client.close()*.

Операції взаємодії передбачають отримання об'єкту бази даних. Для цього використовується метод:

```
client.db("назва_бд");
```

Наприклад:

```
const db = client.db("name_db");
```

Після підключення до бази даних необхідно отримати об'єкт колекції "name_collection":

```
const collection = db.collection("collection_name");
```

Якщо на сервері MongoDB немає такої бази даних та колекції, то при першому до неї зверненні СКБД автоматично їх створить [4]. Розглянемо реалізацію чотирьох основних операцій маніпулювання даними в MongoDB (Рис.3.3):



Рис. 3.3. Основні операції управління даними

Щоб додати в колекцію документ використовують такі її методи:

- insertOne()* додає один документ;
- insertMany()* додає кілька документів;
- insert()* може додавати як один, так і декілька документів.

Синтаксис:

`db.name_collection.insert(document).`

Наприклад:

```
db.kursants.insert({name: Denis, age: 20,  
surname: Manchenko, rank: senior soldier})
```

Для запиту елемента колекції в MongoDB нам необхідно використовувати метод *find()*:

```
db.name_collection.find()
```

Для структурованого виведення даних колекції використовується метод *pretty()*:

```
db.name_collection.find().pretty()
```

Для отримання документа також передбачено метод *findOne()*, який повертає тільки один документ:

```
db.name_collection.findOne()
```

В MongoDB передбачена можливість визначати умови пошуку документа в колекції. Для цих цілей використовуються наступні операції:

Операція	Синтаксис	Приклад
Рівність	{<key>:<value>}	db.kursants.find({"name":"Oleg"})
Менше	{<key>:{\$lt:<value>}}	db.kursants.find({"salary":{\$lt:700}})
Менше, або дорівнює	{<key>:{\$lte:<value>}}	db.kursants.find({"salary":{\$lte:700}})
Більше	{<key>:{\$gt:<value>}}	db.kursants.find({"salary":{\$gt:700}})
Більше, або дорівнює	{<key>:{\$gte:<value>}}	db.kursants.find({"salary":{\$gte:700}})
Не дорівнює	{<key>:{\$ne:<value>}}	db.kursants.find({"salary":{\$ne:700}})

Рис. 3.4. Основні умовні операції для вибірки даних

Для комбінування декількох умов в запиті використовується оператор AND (\$and).

Наприклад:

```
db.name_collection.find(
  {
    $and: [
      {key1: value1}, {key2: value2}
    ]
  }
).pretty()
```

Метод *find()* повертає спеціальний об'єкт – *cursor*, і щоб отримати всі дані у цього об'єкта необхідно викликати метод *toArray()*. У цей метод передається функція зворотного вивиклику з аргументами: *error* (об'єкт помилки при її наявності) і *result* (об'єкт вибірки, як результат).

Наприклад:

```
collection.find().toArray(function(error, results){
    console.log(results);
})
```

Для оновлення даних використовується метод *update()*, який приймає такі параметри:

query приймає запит на вибірку документа, який треба оновити;

objNew представляє документ з новою інформацією, який буде замість старого при оновленні;

options визначає додаткові параметри при оновленні документів та може приймати два аргументи: *upsert* і *multi*.

Якщо параметр *upsert* має значення *true*, MongoDB буде оновлювати документ, якщо він знайдений, і створювати новий, якщо такого документа немає. Якщо ж він має значення *false*, то MongoDB не створюватиме новий документ, якщо запит на вибірку не знайде жодного документа.

Параметр *multi* вказує, чи повинен оновлюватися перший елемент у вибірці (використовується за умовчанням, якщо цей параметр не вказано) або ж повинні оновлюватися всі документи в вибірці.

У випадку, коли не потрібно оновлювати весь документ, а тільки значення одного з його ключів застосовується оператор *\$set*. Якщо документ не містить оновлюється поля, то воно створюється.

Наприклад:

```
db.kursants.update({name: "Denis", age: 21}, {$set:
{age: 22}})
```

Для простого збільшення значення числового поля на певну кількість одиниць застосовується оператор `$inc`.

```
db.kursants.update({name: "Denis"}, {$inc: {age: 2}})
```

Для видалення окремого ключа використовується оператор `$unset`:

```
db.kursants.update({name: "Igor"}, {$unset: {age: 1}})
```

Метод `remove()` в MongoDB використовується для видалення документа з колекції. Метод `remove()` приймає два параметри. Одним з них є критерій видалення, а другим – прапор `justOne`.

Критерії видалення – це умова для видалення відповідного документами з колекції.

`justOne` – це прапор, значення якого впливає на кількість видалених документів. Якщо він має значення `true` видаляться тільки один документ.

Основний синтаксис методу `remove()` представлений далі:

```
db.collection_name.remove(deletion_criteria, justOne).
```

Для видалення тільки одного документа:

```
db.collection_name.remove(deletion_criteria, 1)
```

Якщо не вказати критерій видалення, MongoDB видалить всі документи з колекції.

3.2. Основні можливості та особливості роботи з СКБД MySQL

Express-додаток може використовувати будь-які сучасні бази даних, які підтримуються платформою Node.js (сам по собі *Express* не має конкретних додаткових ресурсів і вимог для управління базами даних).

При виборі бази даних слід враховувати такі фактори як [3]:

- термін розробки;
- термін навчання розробника;
- простота реплікації і копіювання;
- швидкодія;
- захищеність, цілісність;
- підтримка і документація.

Окрім того, при виборі СКБД, необхідно керуватися так званою CAP-теоремою (де до абревіатури CAP входять: *Consistency* (Узгодженість, Цілісність), *Availability* (Доступність) і *Partition tolerance* (Стійкість до поділу)) (Рис. 3.5), згідно якої в розподілених системах можливо реалізувати лише дві властивості із зазначених в теоремі трьох [9]. Так, вважається що нереляційні бази даних не забезпечують вимогу по узгодженості даних на користь доступності і стійкості до поділу. В таких системах поділ розподіленої системи на декілька ізольованих частин зберігає коректність даних від кожної з них.

Тому при виборі СКБД необхідно чітко визначитися з вимогами до інформаційної системи та обрати ту СКБД, яка забезпечить їх виконання.

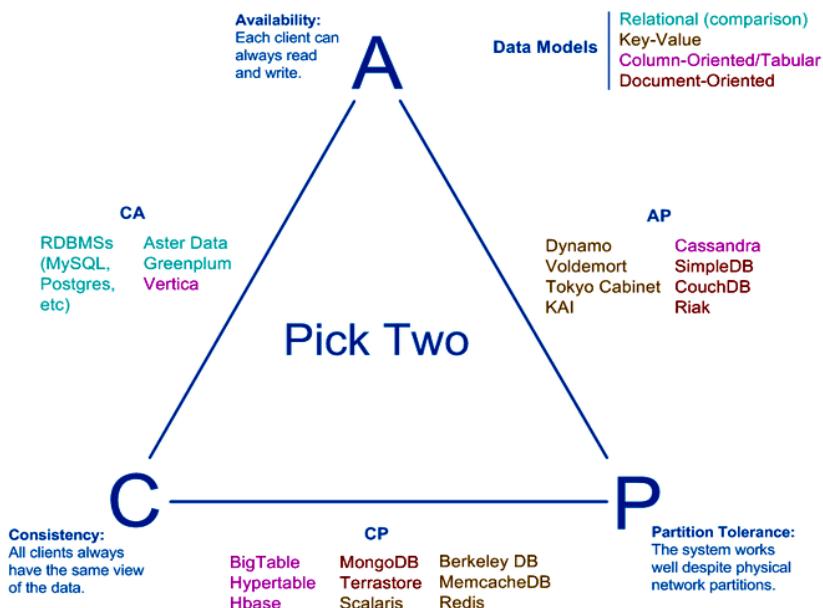


Рис.3.5. Вибір СКБД на основі САР-теореми

Розглянемо основні можливості по роботі з СКБД MySQL [6] в середовищі Node.js.

Для забезпечення роботи з MySQL необхідно інсталювати один з доступних драйверів (Рис.3.6).

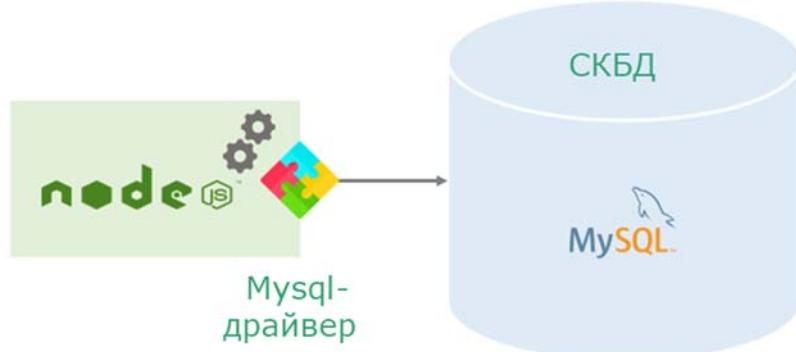


Рис.3.6. Підключення до СКБД MySQL в додатку Node.js

Інсталяція та підключення відповідних модулів здійснюється з використанням наступних команд [4]:

`npm install mysql --save`

або:

`npm install mysql2 --save`

Однією з відмінностей драйверу на основі модулю `mysql2` є те, що він дозволяє працювати з промісами та створювати пули підключень.

Далі розглянемо роботу з даною СКБД з використанням модулю `mysql`. Для створення з'єднання з СКБД MySQL використовується метод `createConnection()`, який в якості аргумента приймає об'єкт з інформацією про сервер MySQL (до якого потрібно підключитись) (Рис.3.7):

```

const mysql = require("mysql")
let connection = mysql.createConnection({
    host: 'localhost', ← IP-адреса сервера MySQL
    user: 'root', ← Логін користувача для доступу до БД
    password: '12345',
    database: 'test_bd' ← Назва бази даних
});

```

Рис. 3.7. Основні налаштування методу *createConnection()*

Для підключення до СКБД використовується метод модулю *connect()*:

```

connection.connect(function(err) {
    if (err) {
        return console.error('error: ' + err.message);
    }
    console.log('Connected to the MySQL server...');
});

```

Для закриття підключення до бази даних використовується метод *end()*:

```

connection.end(function(err) {
    if (err) {
        return console.log('error: ' + err.message);
    }
    console.log('Close the database connection.');
});

```

Параметри підключення до бази даних, зазвичай розміщують в окремому файлі, який імпортується, як окремий модуль.

Наприклад, створимо файл конфігурації *config.js* з наступним вмістом:

```

let config = {
    host : 'localhost',

```

```
    user : 'root',
    password: '',
    database: 'todoapp'
};

module.exports = config;
```

Даний файл може бути використаний в якості такого модулю.

Щоб забезпечити роботу в Node.js з основними операціями по маніпулюванню даними в таблицях бази даних, потрібно виконати наступні дії:

- підключитись до бази даних MySQL;
- виконати SQL-запит, з допомогою методу *query()* об'єкту з'єднання;
- закрити підключення до бази даних, з допомогою методу *end()*.

Розглянемо реалізацію основних операцій по маніпулюванню даними.

Так, наприклад, для додавання даних в таблицю необхідно створити відповідний запит:

```
let mysql = require('mysql');
let config = require('./config.js');
let connection = mysql.createConnection(config);

// SQL запит
let sql = `INSERT INTO todos(title, completed)
            VALUES('Learn how to insert a new
row',true)`;

// виконання запиту до бази даних
connection.query(sql);
connection.end();
```

Для визначення вказівника для додавання даних в SQL-запиті використовується знак запитання (?). Це робиться коли в запит необхідно ввести дані, які надходять зовні, з метою уникнення SQL-ін'екцій. Щоб передати дані в запит використовується масив, який передається другим параметром методу `query()` об'єкта з'єднання. Після виконання запиту є можливість отримати ідентифікатор запису із властивості `insertId` об'єкту `results`. Наприклад:

```
let mysql = require('mysql');
let config = require('./config.js');
let connection = mysql.createConnection(config);
let sql = `INSERT INTO todos(title,completed)
VALUES(?,?)`;
let todo = ['Insert a new row with placeholders',
false];
// execute the insert statement
connection.query(sql, todo, (err, results, fields) => {
  if (err) {
    return console.error(err.message);
  }
  // get inserted id
  console.log('Todo Id:' + results.insertId);
});
connection.end();
```

Виконання запиту на вибірку з передаванням параметрів:

```
let sql = `SELECT * FROM todos`;
connection.query(sql, (error, results, fields) => {
  if (error) {
    return console.error(error.message);
  }
  console.log(results);
});
```

Дані повертаються в спеціальному масиві об'єктів `RowDataPacket`:

```
[ RowDataPacket { id: 1, title: 'Learn how to insert a  
new row', completed: 1 },  
  RowDataPacket { id: 2, title: 'Insert a new row  
with placeholders', completed: 0 },  
  RowDataPacket { id: 3, title: 'Insert multiple rows  
at a time', completed: 0 },  
  RowDataPacket { id: 4, title: 'It should work  
perfectly', completed: 1 } ]
```

Виконання запиту на вибірку з передаванням параметрів:

```
let sql = `SELECT * FROM todos WHERE completed=?`;  
connection.query(sql, [true], (error, results, fields)  
=> {  
  if (error) {  
    return console.error(error.message);  
  }  
  console.log(results);  
});
```

Виконання запиту на оновлення даних з передаванням параметрів:

```
let sql = `UPDATE todos SET completed = ? WHERE id =  
?`;  
let data = [false, 1];  
// execute the UPDATE statement  
connection.query(sql, data, (error, results, fields) =>  
{  
  if (error){  
    return console.error(error.message);  
  }  
  console.log('Rows affected:',  
results.affectedRows);  
});
```

Виконання запиту на видалення даних з передаванням параметрів:

```
// DELETE statement  
let sql = `DELETE FROM todos WHERE id = ?`;  
  
// delete a row with id 1
```

```
connection.query(sql, 1, (error, results, fields) => {
  if (error)
    return console.error(error.message);
  console.log('Deleted Row(s):',
  results.affectedRows);
});
```

3.3. Кешування на рівні додатку з використанням

СКБД Redis

Один з найпопулярніших варіантів використання СКБД Redis – кешування даних. Кешування використовується для того, щоб знизити навантаження на БД та мати можливість здійснювати запити до даних, що часто використовуються, максимально швидко. Redis – це *key-value* база даних, в якій дані зберігаються в оперативній пам'яті, тому дані ми отримуються дуже швидко [12].

Взагалі існує декілька причин, для чого здійснюється кешування:

- для економії витрат, таких як оплата пропускної здатності або обсягу завантажених даних, що надсилаються по мережі.
- для зменшення часу відгуку додатка на запити.

Тому кешування, налаштоване коректно, покращить продуктивність додатку.

Для цього розглянемо проект з використанням СКБД *Redis*, *axios* та *Express Framework* для *Node.js*. Для цього необхідно встановити відповідні модулі, виконавши команду [4]:

```
npm install express redis axios
```

Створимо найпростіший сервер:

```
const express = require('express');
const app = express();
const port = 3000;
app.listen(port, () => {
```

```

        console.log(`Server running on port ${port}`);
    });
module.exports = app;

```

Надішлемо запит до загальнодоступного API. Зробимо це для до розглянутий загальнодоступний API для з статистикою про захворюваність на Covid'19 в світі, наданих ресурсом: ['https://api.covid19api.com/world?from=2021-03-01T00:00:00Z&to=2021-04-01T00:00:00Z'](https://api.covid19api.com/world?from=2021-03-01T00:00:00Z&to=2021-04-01T00:00:00Z).

У файл *index.js* додамо такий код:

```

const express = require('express');
const app = express();
const port = 3000;
const axios = require('axios');
app.get('/statistic', async (req, res) => {
    try {
        const result = await
        axios.get('https://api.covid19api.com/world?from=2021-
        03-01T00:00:00Z&to=2021-04-01T00:00:00Z');
        return res.status(200).send({
            error: false,
            data: result.data.results
        });
    } catch (error) {
        console.log(error)
    }
});
app.listen(port, () => {
    console.log(`Server running on port ${port}`);
});
module.exports = app;

```

Щоб мати можливість у повній мірі скористатися можливостями Redis, необхідно інсталювати дану СКБД завантаживши необхідні файли з офіційного джерела. Для цього запустимо у терміналі такі команди [12]:

```
wget http://download.redis.io/redis-stable.tar.gz
```

```
tar xvzf redis-stable.tar.gz
```

```
cd redis-stable
```

```
make.
```

Далі, запустимо сервер СКБД Redis, командою:

```
redis-server
```

Для перевірки запустимо клієнт з командою перевірки:

```
redis-cli ping
```

В якості відповіді повинні отримати: *PONG*. Якщо так, все готове для подальшої роботи.

Запустимо сервер, і відкриємо браузер, щоб зробити запит до зовнішнього API з статистикою про захворювання на Covid19 в світі.



The screenshot shows a browser window with the URL api.covid19api.com/world?from=2021-03-01T00:00:00Z&to=2021-04-01T00:00:00Z. The page displays a large JSON array of data points representing daily COVID-19 statistics. Each object in the array contains fields for NewConfirmed, TotalConfirmed, NewDeaths, TotalDeaths, and NewRecovered, along with a timestamp. The data spans from March 1, 2021, to April 1, 2021.

```
[{"NewConfirmed":199403,"TotalConfirmed":120062685,"NewDeaths":4122,"TotalDeaths":2658360,"NewRecovered":03-16T23:48:04.208Z}, {"NewConfirmed":231292,"TotalConfirmed":123517783,"NewDeaths":4561,"TotalDeaths":2720855,"NewRecovered":23T23:46:23.253Z}, {"NewConfirmed":169849,"TotalConfirmed":116990710,"NewDeaths":4064,"TotalDeaths":2597998,"NewRecovered":09T23:56:57.22"}, {"NewConfirmed":267505,"TotalConfirmed":116351396,"NewDeaths":5598,"TotalDeaths":2585919,"NewRecovered":07T23:57:56.551Z}, {"NewConfirmed":325641,"TotalConfirmed":124041154,"NewDeaths":7762,"TotalDeaths":2731975,"NewRecovered":24T23:55:49.657Z}, {"NewConfirmed":241687,"TotalConfirmed":117405407,"NewDeaths":7188,"TotalDeaths":2608113,"NewRecovered":10T23:56:11.172Z}, {"NewConfirmed":194487,"TotalConfirmed":119720249,"NewDeaths":3480,"TotalDeaths":2651562,"NewRecovered":15T23:55:06.557Z}, {"NewConfirmed":189346,"TotalConfirmed":114293209,"NewDeaths":5000,"TotalDeaths":2536778,"NewRecovered":02T23:44:35.982Z}, {"NewConfirmed":322311,"TotalConfirmed":121632076,"NewDeaths":7388,"TotalDeaths":2688928,"NewRecovered":19T21:14:04.244Z}, {"NewConfirmed":266794,"TotalConfirmed":115467401,"NewDeaths":6978,"TotalDeaths":2567360,"NewRecovered":05T23:08:44.362Z}, {"NewConfirmed":220131,"TotalConfirmed":116692141,"NewDeaths":3396,"TotalDeaths":2591163,"NewRecovered":08T23:51:09.018Z}, {"NewConfirmed":168368,"TotalConfirmed":113918854,"NewDeaths":3586,"TotalDeaths":2528689,"NewRecovered":01T23:56:49Z}],
```

Рис. 3.8. Фрагмент даних отриманих за запитом з зовнішнього API

Час на виконання такого запиту показаний нижче (Рис. 3.9):

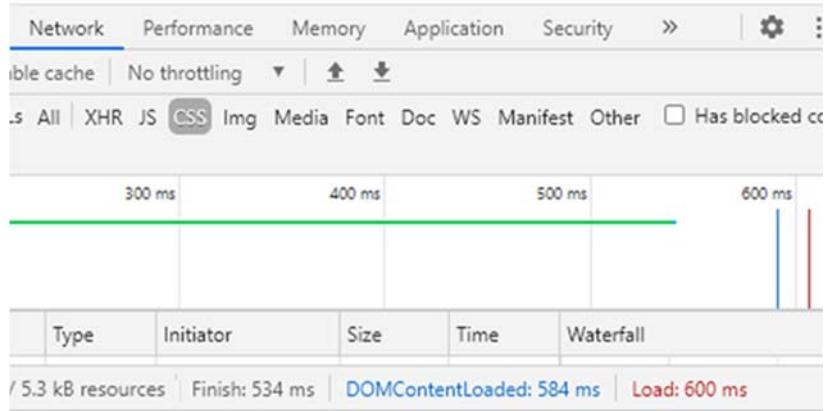


Рис. 3.9. Час на виконання запиту до зовнішнього API

Як бачимо, запит виконано за *600 ms*, це час для отримання даних, які не часто змінюються. Спробуємо покращити його, впровадивши кешування за допомогою Redis [12].

Доповнимо код з кешуванням попередньо отриманих даних:

```
const express = require('express');
const axios = require('axios');
const redis = require('redis');
const app = express();
const port = 3000;
// make a connection to the local instance of redis
const client = redis.createClient(6379);
client.on("error", (error) => {
    console.error(error);
});
app.get('/statistic/', (req, res) => {
    try {
        const SearchPeriod = req.query.from;
        //Спочатку перевіряємо наявність даних в кеші
        Redis
            client.get(SearchPeriod, async (err, result) => {
                if (result) {
                    res.json(result);
                } else {
                    const response = await axios.get(`https://api.somesite.com/statistics?from=${SearchPeriod}`);
                    result = response.data;
                    client.set(SearchPeriod, result);
                    res.json(result);
                }
            });
    } catch (error) {
        console.error(error);
    }
});
```

```

        if (result) {
            return res.status(200).send({
                error: false,
                message: `Statistics for ${ SearchPeriod} from the
cache`,
                data: JSON.parse(result)
            })
        } else {
            // Коли дані відсутні в кеші робимо запит до сервера
            const result = await axios.get(
                `https://api.covid19api.com/world?from=${
                    SearchPeriod}`);
            //Зберігаємо запис в кеш для наступних запитів з часом
            //збереження 1500с
            client.setex(SearchPeriod, 1500,
                JSON.stringify(result.data));
            // повертає результат клієнту
            return res.status(200).send({
                error: false,
                message: `Statistics for ${
                    SearchPeriod } from the server`,
                data: result.data
            });
        }
    }
} catch (error) {
    console.log(error)
}
);
app.listen(port, () => {
    console.log(`Server running on port ${port}`);
});
module.exports = app;

```

По-перше, необхідно підключитися до СКБД *Redis* з додатку. Для цього використано встановлений модуль *redis*.

СКБД *Redis* прослуховує порт 6379 за замовчуванням. Отже, створеному клієнту передано цей номер порту для підключення до СКБД *Redis*.

Потім реалізована логіка для зберігання та отримання даних із кешу. Коли отримується запит клієнта до маршруту `/statistic`, спочатку фіксується пошуковий параметр, який був відправлений в запиті:

```
const SearchPeriod = req.query.from;
```

Потім запитуються дані з кешу, з передаванням пошукового параметру, який використовується, як ключ під час зберігання даних у кеші. Оскільки модуль `redis` не має власної підтримки промісів, необхідно також передати функцію зворотного виклику для обробки отриманих даних. Якщо значення, які повертаються з запиту до СКБД `Redis`, не є нульовими, це означає, що відповідні дані існують у кеші, тому їх можна повернути у відповіді.

Якщо повернуте значення було нульовим, необхідно надіслати запит до зовнішнього API для отримання відповідних даних. Коли дані з API отримані, їх необхідно зберегти у СКБД `Redis`. Це робиться для того, щоб наступного разу, коли той самий запит буде відправлений на сервер `Node`, відповідь сформуватина основі даних, що зберігаються в кеші, замість того, щоб запитувати їх із API.

Зверніть увагу, як використана `setex`-функція для зберігання даних у кеші. Використання цієї функції дозволяє встановити час закінчення для збереженої в кеші пари ключ-значення. `Redis` автоматично видалив ці дані з кешу, коли закінчиться встановлений час збереження даних.

Для надсилання запитів на сервер та вимірювання часу їх завершення був використаний *Postman*.

Спочатку, при надсиланні запиту на сервер з новим пошуковим параметром, програма займає більше часу (74 секунди, див. Рис.3.9), оскільки дані потрібно отримувати із зовнішнього API. Коли такий самий запит робиться повторно,

сервер реагує швидше, оскільки результати вже існують у кеші. Так, на Рис.3.10 показано результат вимірювання часу для повторного запиту:

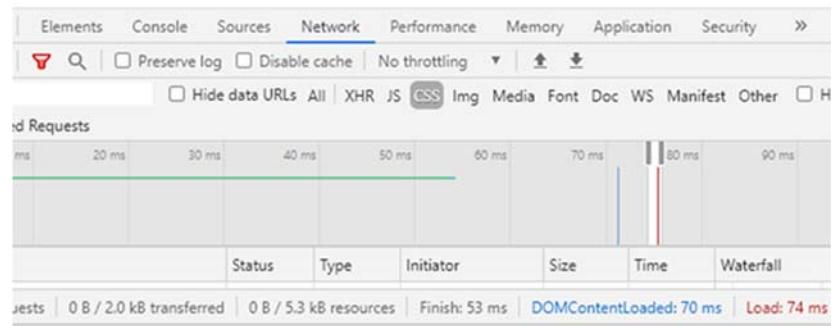


Рис. 3.10. Час на виконання запиту з використанням кешу

Таким чином використання СКБД Redis для кешування даних запитів дозволяє значно покращити продуктивність додатку.

Контрольні запитання до розділу 3

1. Яким чином забезпечується робота з СКБД MySQL з додатку на Node.js?
2. Яким чином забезпечується робота з СКБД MongoDB з додатку на Node.js?
3. Які переваги використання СКБД MongoDB для зберігання даних?
4. Які переваги використання СКБД MySQL для зберігання даних?
5. Яким чином отримується об'єкт *колекції* з СКБД MongoDB в додатку на Node.js?
6. Який драйвер СКБД MongoDB використовується в Node.js для забезпечення розробки на основі ODM підходу?
7. Назвіть переваги ODM (Object Data Modelling) підходу для роботи з базами даних?
8. Назвіть основні параметри необхідні для підключення до бази даних в СКБД MySQL?
9. Визначіть основні методи для маніпуляції даними (вибірка, додавання, оновлення, видалення) в базі даних СКБД MongoDB?
10. Яким чином додати один запис в базу даних СКБД MongoDB?
11. Яким чином забезпечується фільтрація даних при здійсненні запиту на вибірку даних в базі даних СКБД MongoDB з додатку Node.js?
12. Яким чином отримується об'єкт *бази даних* з СКБД MongoDB в додатку на Node.js?

**РОЗШИРЕНІ МОЖЛИВОСТІ ПЛАТФОРМИ
NODE.JS ДЛЯ СТВОРЕННЯ СЕРВЕРНИХ ВЕБ-
ДОДАТКІВ**

4.1. Архітектурний патерн MVC в Node.js

Патерн проєктування – це типовий спосіб вирішення певної задачі, що часто зустрічається при проєктуванні архітектури програмного забезпечення [10].

На відміну від готових функцій чи бібліотек, патерн не можна просто взяти й скопіювати в програму.

Патерн являє собою не якийсь конкретний код, а загальний принцип вирішення певної проблеми, який майже завжди треба підлаштовувати для потреб тієї чи іншої програми.

Патерни часто плутають з алгоритмами, адже обидва поняття описують типові рішення відомих задач. Але якщо алгоритм – це чіткий набір дій, то патерн – це високорівневий опис рішення, реалізація якого може відрізнятися у двох різних програмах.

Опис патернів зазвичай дуже формальний й найчастіше складається з таких складових:

- задача, яку вирішує патерн;
- обґрутування, щодо вирішення задачі способом, який визначений в патерні;
- структура класів та інших складових рішення;
- приклад реалізації однією з мов програмування;

- особливості реалізації в різних контекстах;
- зв’язки з іншими патернами.

Патерни відрізняються за рівнем складності, деталізації та охоплення проєктованої системи.

Найбільш низькорівневі та прості патерни – *ідіоми*. Вони не є універсальними, тому що мають сенс лише в рамках однієї мови програмування.

Найбільш універсальні – *архітектурні* патерни, які можливо реалізувати практично будь-якою мовою. Вони потрібні для проєктування всього додатку, а не окремих його елементів.

Патерни також розрізняються за призначенням [1]:

Породжуючі патерни – піклуються про гнучке створення об’єктів без внесення в програму зовнішніх залежностей.

Структурні патерни – показують різні способи побудови зв’язків між об’єктами.

Поведінкові патерни – піклуються про ефективну комунікацію між об’єктами.

MVC (Model-View-Controller) – архітектурний патерн, який використовується під час проєктування та розробки програмного забезпечення (Рис.3.11).

Патерн *MVC* складається з наступних компонентів:

1. *Модель (Model)* – власне дані, методи для роботи з даними, зміни та оновлення даних.
2. *Представлення (View)* – відображення даних, оформлення та інші аспекти презентації моделі.

3. Контролер (*Controller*) – реагує на дії користувача, інтерпретує дані, введені користувачем, інформує модель і проводить необхідні маніпуляції з моделлю і представленням.

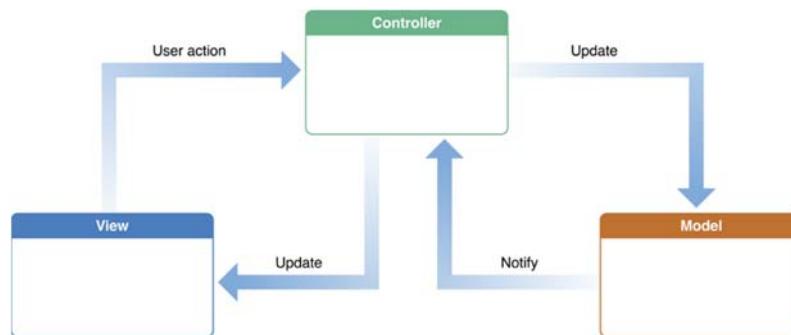


Рис.3.11. Патерн MVC

У веб-розробці *Model* і *View* взаємодіють один з одним через *Controller*.

Так, для прикладу, процес навчання в навчальному закладі можна описати з використанням патерну *MVC* (Рис.3.12):

Model: навчальний процес, в якому викладачі здійснюють навчання.

View: отримані знання, диплом.

Controller: навчальний заклад, який приймає курсантів для навчання та забезпечує навчання.

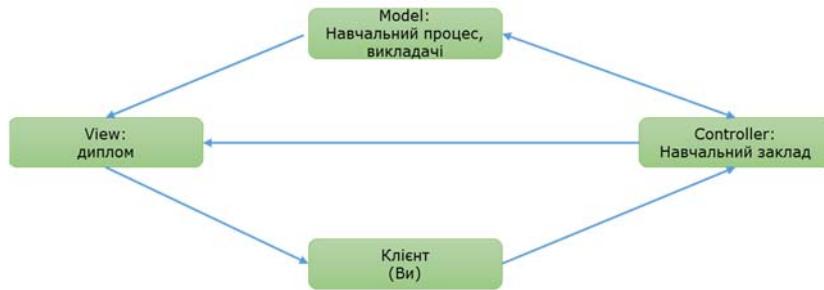


Рис.3.12. Використання патерну MVC

Схематично реалізація MVC у Node.js представлена на Рис.3.13. Патерн MVC включає додатковий компонент, такий як *система маршрутизації*. Він зіставляє запити з маршрутами і вибирає для обробки запитів необхідний контролер [4]. Моделі визначають структуру і логіку організації даних. Представлення визначають як дані будуть відображатися.

Контролери здійснюють обробку вхідних http-запитів, використовуючи для обробки моделі та представлення, і відправляють у відповідь клієнту результат обробки (у вигляді html-коду). Так, при надходженні запиту, система маршрутизації обирає потрібний контролер для обробки запиту. Контролер здійснює обробку запиту. В процесі обробки він звертається до даних через моделі і для рендеринга відповіді використовує представлення. Результат обробки контролером відправляється у відповідь клієнту. Нерідко відповідь представлено html-сторінкою, зміст якої відображається користувачу в браузері.

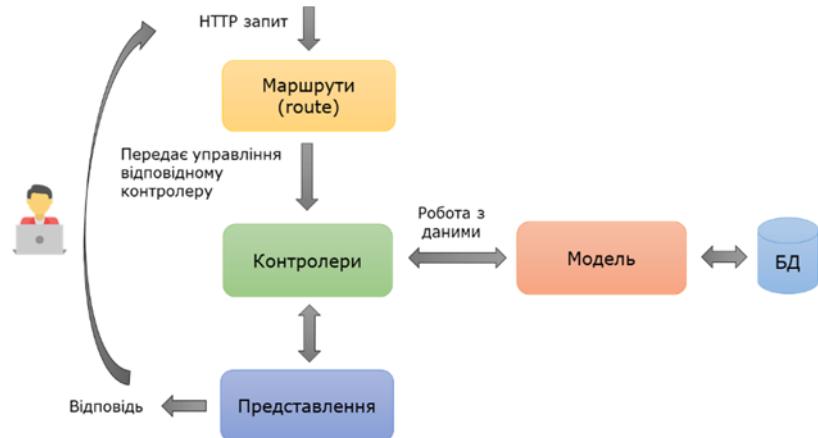


Рис.3.13. MVC у Node.js

Таким чином контролери в патерні MVC дозволяють пов'язати представлення і моделі, а також виконують деяку логіку по обробці запиту.

В основу проєкту, покладено фреймворк Express, який необхідно підключити.

Головний файл додатку *app.js* буде знаходитись в корні файлової системи (Рис. 3.14) та містити наступний код:

```

let express = require('express');
let app = express();
let indexRoute = require('./routers/indexRoute');
app.use(express.static('public'));
app.use('/', indexRoute);
app.set('view engine', 'ejs'); app.set('views',
'./views');
app.listen(3000, ()=>{
    console.log('Server Starting')
}

```

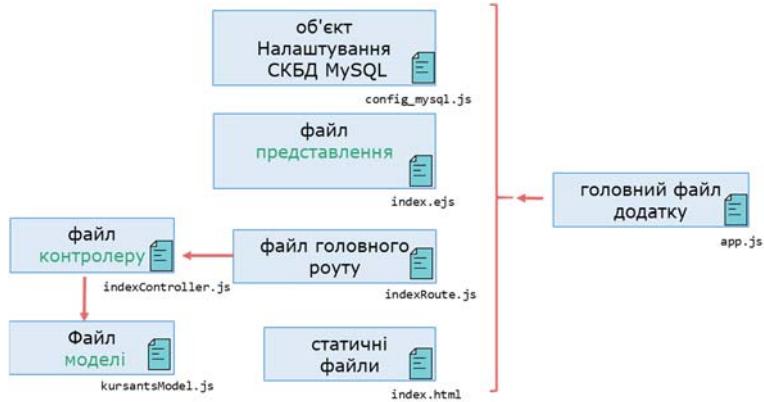


Рис.3.14. Основні файли додатку відповідності до патерну
MVC у Node.js

Файл, в якому описуються маршрути *indexRoute.js*, буде мати наступний вміст:

```
let express = require("express");
let router = express.Router();
let indexController =
require("../controllers/indexController")
router.get('/', indexController);
module.exports = router;
```

Файл, в якому описаний контролер *indexController.js* буде мати наступний вміст:

```
let Kursant = require('../models/kursantsModel');
exports.getKursants = (req, res)=>{
    let kursant = new Kursant();
    kursant.getAllKursants().then((data)=>{
        res.render('index', { allKursants: data
    });
});
};
```

Файл в якому описана модель *kursantsModel.js* буде мати наступний вміст:

```
let mysql = require('mysql');
let config = require('../config/config_mysql');
let connection = mysql.createConnection(config);
class Kursant {
constructor(id)
{
    this.id = id
}
getAllKursants() {
    let query = "SELECT * FROM kursants";
    return new Promise( (resolve)=>{
        connection.query(query, (error, results)=>{
            if(error) throw error;
            resolve(results);
        });
    });
};
module.exports = Kursant;
```

Для збереження даних, в якості прикладу використаємо СКБД MySQL [6]. Файл конфігурації *config_mysql.js* для підключення до СКБД буде містити код:

```
let config_mysql = {
    host: 'localhost',
    user: 'root',
    password: '12345',
    database: 'kaf_22'
};
module.exports = config_mysql
```

Файл представлення *index.ejs* буде містити код:

```
<!DOCTYPE html>
<html>
<head>
```

```

<title>Список курсантів</title>
<meta charset="utf-8" />
</head>
<body>
<h1>Список курсантів</h1>
<table>
    <tr>
        <td>Ідентифікатор</td>
        <td>Ім'я курсанта</td>
    </tr>
    <% data.forEach((results)=>{ %>
        <tr>
            <td><%=results.id%></td>
            <td><%=results.name%></td>
        </tr>
    <% }); %>
</table>
</body>
</html>

```

Таким чином при запиті за основним маршрутом додатку ‘/’ в вікні браузера будуть відображені дані про всіх курсантів з бази даних в табличній формі.

4.2.Поняття сокету і Вебсокету. Особливості бібліотеки Socket.io.

Сокети (Socket) – назва програмного інтерфейсу для забезпечення обміну даними між процесами. Процеси при такому обміні можуть виконуватися як на одній ЕОМ, так і на різних ЕОМ, пов'язаних між собою мережею [10]. *Сокети* – абстрактний об'єкт, що представляє кінцеву точку з'єднання.

Розрізняють *клієнтські* і *серверні* сокети. Клієнтські сокети можливо порівняти з кінцевими апаратами телефонної мережі, а серверні – з комутаторами. Клієнтський додаток (наприклад, браузер) використовує лише клієнтські сокети, а серверний (наприклад, веб-сервер, якому браузер надсилає запити) – як клієнтські, так і серверні сокети.

Вебсокети (WebSockets) – це технологія, що дозволяє відкрити постійне двонаправлене мережеве з'єднання між клієнтом та сервером. За допомогою його API є можливість відправити повідомлення на сервер і отримати відповідь без виконання http-запиту, причому цей процес буде управлятися на основі подій [4]. *WebSockets* стандартизовані протоколом, що описує процес обміну інформацією між браузером та веб-сервером в режимі реального часу. Прикладний програмний інтерфейс *WebSocket* був стандартизований W3C, крім того протокол *WebSocket* стандартизований IETF як RFC 645 [10].

Socket.io – JavaScript бібліотека для веб-додатків і обміну даними в реальному часі з використанням протоколу *WebSockets*. Вона складається з двох частин: клієнтської, код якої виконується в браузері і серверної яка виконується на платформі Node.js [4]. Обидва компоненти мають ідентичний API.

Подібно Node.js, *Socket.io* подієво-орієнтована бібліотека. *Socket.io* використовується для:

- швидкого обміну даними для онлайн ігор, чатів тощо;

- розробки веб-додатків з інтенсивним обміном даними, що вимогливі до швидкості обміну і каналу;
- push-повідомлень.

Схема організації з'єднань на основі *Socket.io* показана на Рис.3.15:

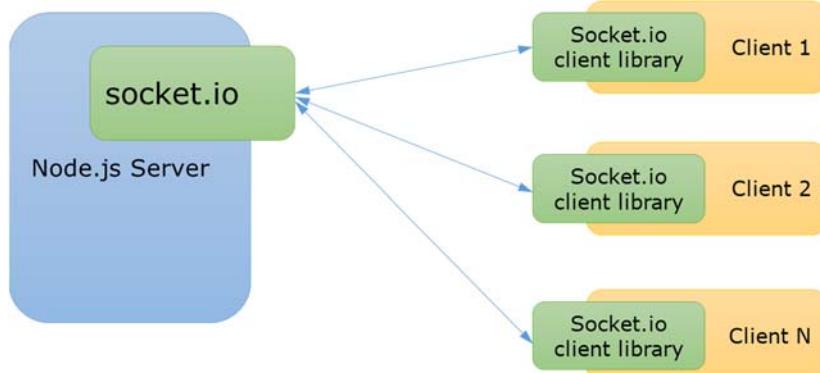


Рис. 3.15. Схема роботи *Socket.io* у *Node.js*

Для установки бібліотеки *socket.io* необхідно виконати команду:

npm install socket.io.

Підключення бібліотеки до проекту з використанням фреймворку *Express.js* здійснюється наступним чином:

```

let express = require('express');
let app = express();
let http = require('http').createServer(app);
let io = require('socket.io')(http);
  
```

Підключення бібліотеки до проекту з використанням модулю *http*:

```
const app = require('http').createServer(handler);
const io = require('socket.io')(app);
```

Для забезпечення комунікації з клієнтами на серверній стороні необхідно прослухати подію підключення та створити обробники для інших очікуваних подій:

```
io.on('connection', (socket)=>{
    socket.emit('news', { hello: 'world' });
    socket.on('my other event', (data)=>{
        console.log(data);
    });
});
```

Так, для прикладу, розглянемо реалізацію простого додатку на основі сокетів. На клієнтській стороні маємо файл *index.html* з таким кодом:

```
<!doctype html>
<html lang="ua">
    <head>
        <title>Робота з сокетами</title>
    </head>
    <body>
        <ul id="messages"> </ul>
        <label for="mess_text">Повідомлення</label>
        <input id="mess_text" autocomplete="off">
        <button id="button">Відправити</button>
        <script src="/socket.io/socket.io.js"></script>
        <script>
            let socket = io.connect();
            document.getElementById("button").
            addEventListener("click", ()=>{
                socket.emit('send message',
                document.getElementById('mess_text').value)
            });
            socket.on('add message', (data)=>{
                document.getElementById( 'messages'
                ).innerHTML+="<li>" +data.msg + "</li>";
            });
        </script>
```

```
</body>
</html>
```

Головний файл додатку *app.js* має наступний лістинг:

```
let express = require('express');
let app = express();
let http = require('http').createServer(app);
let io = require('socket.io')(http);
app.use(express.static(__dirname + "/public"));
app.set("view engine", "ejs"); app.set("views",
"views");
app.get('/', (req, res)->{
    res.sendFile(__dirname + '/index.html');
});
io.sockets.on('connection', (socket) =>
    console.log('a user connected');
    socket.on('send message', (data)=>{
        io.sockets.emit('add message', {msg: data})
    });
});
http.listen( port 3000, listeningListener ()=>{
    console.log("Server starting...");
});
```

При запиті по головному маршруту додатку буде відображенено файл *index.html* з формою для введення даних. При виникненні події **'click'** для елемента кнопки генерується подія **'send message'** на основі методу **emit** об'єкту **socket**. На основі механізму вебсокетів реалізовано з'єднання з сервером. На сервері здійснюється обробка події **'send message'**, яка згенерована на клієнтській стороні. Її обробник, генерує подію **'add message'**, яка, в свою чергу, обробляється на клієнтській стороні. В результаті введені користувачем в форму дані відображаються на сторінці, як елемент списку:

```
socket.on('add message', (data)=>{
```

```
document.getElementById('messages').innerHTML += "<li>"  
+data.msg + "</li>");});
```

4.3. Фреймворк Mocha. Тестування серверних додатків.

При розробці програмного забезпечення важливим етапом є його *тестування*. У Node.js для рішення цієї задачі є фреймворки, які спрощують процес тестування. Одним з найбільш відомих серед таких є фреймворк *mocha*. Детальніше про нього можна дізнатися на офіційній сторінці [13]. Далі буде розглянуто деякі базові можливості роботи з ним.

Створимо новий проект та додамо модуль *mocha* за допомогою наступної команди:

```
npm install mocha --save-dev
```

Ключ «*--save-dev*» призначений для додавання в проект модулів необхідних тільки на етапі розробки [4]. Оскільки фреймворк *mocha* необхідний тільки для тестування додатку, то запис про нього додається у файлі *package.json* в секцію *devDependencies*.

Для тестування визначимо тестовий модуль. Для цього додамо в проект файл *add.js* з наступним змістом:

```
const add = (a, b) => a + b;  
module.exports.add = add();
```

Функція визначає операцію додавання двох чисел. Для тестування цього модуля додамо в проект новий файл *tests.js* з наступним вмістом:

```

let add = require("./add");
it("Сума двох чисел", function(){
    let expectedResult = 8;
    let result = add.sum(3, 5);
    if(result!==expectedResult){
        throw new Error(`Очікується ${expectedResult}, але вийшло ${result}`);
    }
});

```

Розглянемо цей тест. Для тестування результату виконання функції додавання використана функція *it()*, яка надається фреймворком *mocha*.

Ця функція приймає два параметри: текстовий опис тестованого дії "сума двох чисел", та саму тестуючу функцію.

Так, в наведеному прикладі необхідно перевірити коректність функції додавання двох чисел *add()*. Для цього в ній необхідно передати два числа і порівняти її результат з очікуваним. Якщо результат не співпаде з очікуваним значенням, то згенерується помилка.

Для запуску тестів в файл *package.json* додамо запис:

```

"scripts":{
  "test": "mocha *js"
}

```

Запустимо створений тест на виконання (Рис.3.16) командою *mocha tests*:

```
Terminal: Local +  
D:\untitled7>mocha tests  
  ✓ suma двох чисел  
  1 passing (6ms)  
D:\untitled7>
```

Рис. 3.16. Результат тестування функції у Node.js

Далі перевизначимо очікуваний результат тестування функції:

```
let expectedResult = 7.
```

У випадку помилки будемо мати наступний результат

Рис.3.17:

```
Terminal: Local +  
  1) suma двох чисел:  
    Error: Очікується 7, але вийшло 8  
        at Context.<anonymous> (tests.js:6:15)  
        at processImmediate (internal/timers.js:456:21)  
D:\untitled7>
```

Рис. 3.17. Помилка в функції при тестуванні

Аналогічним чином можна визначати й інші тести в файлі *tests.js*. Функції, які тестиються, при цьому можуть знаходитись в різних модулях.

Для тестування асинхронних функцій використовується функція *done()*, яка передається в тестуючу функцію і дозволяє дочекатися завершення асинхронної функції. Після чого завершується тест. В іншому випадку тест завершиться раніше, ніж завершиться асинхронна функція [13]. Наприклад, змінимо зміст файлу *add.js* та створимо асинхронну функцію для додавання двох чисел:

```
const asyncSum = (a, b, func) => {
    setTimeout(function(){
        func(a+b);
    }, 1000)
}
module.exports.asyncSum = asyncSum;
```

Також створимо тест для тестування цієї асинхронної функції з використанням вищезазначененої функції *done()*:

```
let add = require("./add");
it("сума двох чисел через 1 секунду", function(done) {
    let expectedResult = 7;
    add.asyncSum(2, 5, function (result) {
        if (result !== expectedResult) {
            throw new Error(`Очикується ${expectedResult}, але вийшло ${result}`);
        }
        done();
    });
});
```

Результат тестування асинхронної функції показаний на Рис.3.18:

The screenshot shows a terminal window with the following output:

```
D:\untitled7>mocha tests

  ✓ suma двох чисел через 1 секунду (1015ms)

  1 passing (1s)

D:\untitled7>
```

The terminal has tabs at the bottom: Find, TODO, Terminal.

Рис. 3.18. Результат успішного тестування асинхронної функції

Для спрощення верифікації результатів при тестуванні використовується модуль *assert*. Для його інсталяції необхідно виконати команду:

```
npm install assert --save-dev
```

При використанні даного модуля застосовують метод *equal()*, який дозволяє порівняти два значення, очікуваний результат з визначенім еталоном.

Наприклад, для попередньо визначеної функції додавання двох чисел будемо мати:

```
let add = require("./add");
it("Сума двох чисел", function(){
  let expectedResult = 8;
  let result = add.sum(3, 5);
  assert.equal(result, expectedResult);
});
```

Якщо обидва значення однакові, тест завершиться успішно. В іншому випадку буде згенерована помилка. Також в модулі є функція *notEqual()*, при використанні якої

генерується помилка у випадку рівності значень, які порінюються:

```
assert.notEqual(result, expectedResult).
```

В процесі розробки в проекті може бути велика кількість модулів, і для кожного може бути визначено декілька тестів. За допомогою методу *describe()*, який визначений в *mocha*, можна оформити тести в пов'язані групи тестів. Наприклад, тести для перевірки функцій одного модуля складатимуть одну групу, а тести для перевірки функціоналу іншого модуля будуть відноситись до іншої. Створення груп тестів надає можливість полегшити процес тестування шляхом швидкої ідентифікації, який модуль не пройшов тест, особливо якщо тестів велика кількість. Наприклад, створимо для попередних тестів групу:

```
let add = require("./add");
describe("Тести для операції додавання", function(){
    it("Сума двох чисел (синхронний метод)",
    function(){
        let expectedResult = 8;
        let result = add.sum(3, 5);
        assert.equal(result, expectedResult);
    });
    it("сума двох чисел через 1 секунду (асинхронний
    метод)",
        function(done) {
    let expectedResult = 7;
    add.asyncSum(2, 5, function (result) {
        if (result !== expectedResult) {
            throw new Error(`Очікується
${expectedResult}, але вийшло ${result}`);
        }
        done();
    }); assert.equal(result, expected);}));
});
```

Метод *describe()* в якості першого параметру приймає строку з описом групи тестів, а в якості другого функцію з множиною тестів.

Завершальним етапом перед впровадження програмного продукту є його тестування. Цей етап є дуже важливим оскільки дозволяє виявити всі існуючі невідповідності та їх усунути. Розглянута платформа Node.js має потужні можливості для проведення тестування на основі сучасних фреймворків.

Контрольні запитання до розділу 4

1. Який модуль можна використати для відкриття двостороннього з'єднання з клієнтом на основі протоколу http в реальному часі?
2. Поясніть особливості реалізації додатку на основі патерну MVC в Node.js.?
3. Поясніть основні функції покладені на елемент *Controller* патерну MVC в додатку Node.js?
4. Поясніть основні функції покладені на елемент *Model* патерну MVC в додатку Node.js?
5. Поясніть основні функції покладені на елемент *View* патерну MVC в додатку Node.js?
6. Дайте визначення *Вебсокету (WebSocket)*?
7. Поясніть призначення роутерів в моделі MVC в Node.js?
8. Поясніть, яким чином здійснити обробку запитів з використанням моделі MVC?
9. Для чого використовується бібліотека «*socket.io*»?
10. Яким чином здійснюється тестування додатків в платформі Node.js?
11. Назвіть призначення та основні можливості модулю *mocha*.
12. Поясніть порядок створення тесту для перевірки довільної функції з використанням фреймворку *mocha*.
13. Назвіть призначення та основні можливості модулю *assert*.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Любарський С. В., Хусаїнов П. В., Нестеренко М.М. Технологія створення програмних продуктів. Частина 1: Навчальний посібник. – К: ВІТІ НТУУ „КПІ”, 2012.
2. Власенко О. В., Здоренко Ю. М., Фесьоха В. В. Web-технології та Web-дизайн: Навчальний посібник – К.: ВІТІ, 2020. – 190 с.
3. Власенко О. В., Любарський С. В., Здоренко Ю. М. Організація баз даних та знань: Навчальний посібник. Частина 1,– К.: ВІТІ, 2020.– 148 с.
4. Офіційний сайт Node.js [Електронний ресурс]. – Режим доступу: <https://nodejs.org/uk/>.
5. Офіційний сайт Mongodb [Електронний ресурс]. – Режим доступу: <https://www.mongodb.com/>
6. Офіційний сайт Mysql [Електронний ресурс]. – Режим доступу: <https://www.mysql.com/>
7. Node.js Tutorial [Electronic resource]. – Access mode: <https://www.w3schools.com/nodejs/>.
8. RFC7540. Hypertext Transfer Protocol. Version 2 (HTTP/2) [Electronic resource]. – Access mode: <https://tools.ietf.org/html/rfc7540>
9. Brewer's CAP Theorem. [Electronic resource]. – Access mode: <https://www.julianbrowne.com/article/brewers-cap-theorem>

10. RFC 645. Network Standard Data Specification Syntax. [Electronic resource]. – Access mode: <https://www.rfc-editor.org/rfc/rfc645.html>
11. Modul Multer. User Guide [Electronic resource]. – Access mode: <https://github.com/expressjs/multer>.
12. Redis Quick Start [Electronic resource]. – Access mode: <https://redis.io/topics/quickstart>
13. Mocha framework [Electronic resource]. – Access mode: <https://mochajs.org/>

ПРЕДМЕТНИЙ ПОКАЖЧИК

A	M
Адаптивність <i>37, 40, 122</i>	Метод <i>7, 133, 135</i>
Атрибут <i>53, 61-63</i>	Мова <i>8, 51, 123</i>
Б	Модель <i>7, 10, 54-56</i>
База даних	Модуль <i>222</i>
Бібліотека	O
Бінарний оператор <i>136</i>	Об'єкт <i>10, 27, 55-57</i>
Браузер <i>8-9, 16-19, 29, 33</i>	Обробка <i>10, 103, 166-168</i>
V	Оператор <i>104-105, 129, 135-139</i>
Варіант <i>6, 32-36, 72</i>	P
Веб <i>6, 19</i>	Патерн
Вебсокет	Перезавантаження сторінки <i>19, 26</i>
Верстка <i>38, 50, 71</i>	Платформа Node.js <i>175</i>
Властивість <i>74, 86, 92-94</i>	Подія <i>162-165, 172-176</i>
D	Програма <i>51, 52</i>
Дані <i>8, 13, 16, 33</i>	Програмування <i>58-59, 120, 123</i>
Додаток <i>6, 20, 23, 25</i>	Проект <i>41-48, 112-116</i>
Домен <i>8, 11, 16</i>	Програмне забезпечення
Драйвер	Протокол
J	P
Життєвий цикл <i>19, 21</i>	Ресурс <i>8, 10-14, 18-20</i>
Z	Розробка <i>26, 29, 34, 37</i>
Забезпечення <i>6, 8, 25-26</i>	Розширення <i>8, 35, 51, 58, 72</i>
Змінна <i>138, 132, 144</i>	C
Знак <i>21, 80, 117, 141</i>	Селектор <i>7, 74-77, 80-83</i>
I	Сервер <i>6-11, 13-22</i>
Ідентифікатор <i>10, 11, 63-65</i>	Система <i>37-43, 111</i>
Інтерфейс <i>6, 22, 27-29</i>	Система керування базами даних <i>107, 110</i>
Інформаційна система <i>18</i>	Сокет
Інформаційна технологія <i>18</i>	Структура <i>6, 10, 38, 47, 53-54</i>
Інформаційний ресурс <i>22</i>	T
Інформація <i>8, 11, 50, 69</i>	Технологія <i>7, 22, 100</i>
Y	U
K	Управління <i>18, 42 66-67</i>
Контролер	F
Клас <i>7, 18, 69, 73</i>	Файл
Клієнт <i>6-14, 24-27</i>	Формат <i>12, 14, 33, 37</i>
L	Фреймворк <i>7, 23, 28, 33</i>
Логічний вираз <i>139</i>	Функція <i>23, 133-134, 145-146</i>

X	Ч
Хост 11, 14, 36	Час 22, 159, 172
Ц	Частина 8, 19, 24, 50, 110
Цілі числа 130	Черга 32-33, 76
Цільовий елемент 172	