hillel IT school

Lesson 40

План заняття

- FS
- Streams
- Express

File system

Модуль node:fs (https://nodejs.org/api/fs.html) дозволяє взаємодіяти з файловою системою за моделлю стандартних функцій. Це вбудований модуль, він має 2 інтерфеса:

1. Інтерфейс Promises: const fs = require('node:fs/promises');
2. Інтерфейс callbacks: const fs = require('node:fs');

Усі операції з файловою системою мають синхронні форми, форми зворотного виклику та форми на основі Promises і доступні за допомогою синтаксису CommonJS і модулів ES6 (ESM).

Документація тут https://nodejs.org/api/fs.html#file-system

Ви можете використовувати як звичайні функції з колбеками так і створювати потоки або streams

Важливо!

Запам’ятайте що усі стандартні методи з бібліотеки fs є не оптимальними, але ви можете їх використовувати. Тому що вони працюють з файлами "жорстко" напряму записуючи їх до оперативної пам’яті системи.

Streams

Потоки є однією з фундаментальних концепцій, на яких працюють програми Node.js. Вони є методом обробки даних і використовуються для послідовного читання або запису вхідних даних у вихідні. Це більше патерн ніж технологія. У Node.js потоки в основному використовуються для обробки даних.

Як ви можете побачити, коли ми використовуємо звичайну функцію з модуля fs наприклад readFile якийсь системний апі шукає дані за шляхом який ми передали, використовує опції. Зчитує весь файл до оперативної пам’яті і потім вже віддає адресу того де записані ці данні.

Якщо коротко поки всі дані не завантажаться до оперативної пам’яті ми нічого не побачимо.

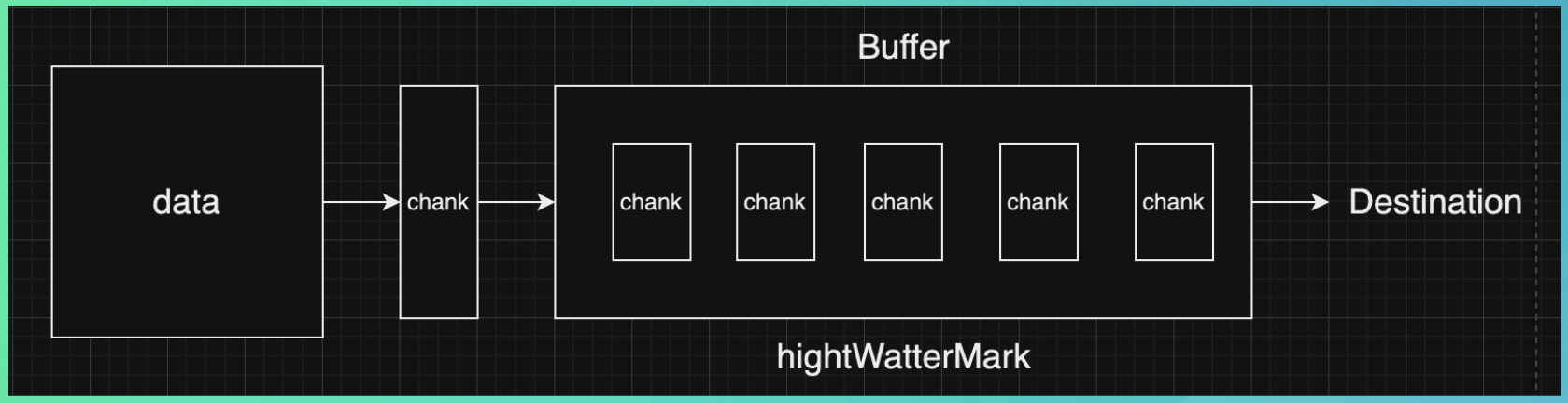
Тут народжується проблема, а якщо наш файл багато важить, скільки часу буде використано для обробки усіх даних?

На поміч нам прийде модель потоків.

У основі потоків лежить EventEmitter

Streams

Модель потоків



Ідея у тому щоб розділити один великий файл на декілька маленьких шматочків і поступово заповнювати їми оперативну пам’ять. Stream надає механізм обробки даних шматочків дати, а розділенням займається системне апі.

Саме з буфером працює Stream. Буфер це певний об’єм пам’яті у якому зберігаються тимчасові дані.

Streams

Потоки бувають:

1. Readable - поток читання
2. Writable - поток записусу
3. Duplex - поток читання та запису
4. Transform - це різновид duplex, де вихід певним чином пов’язаний із входом (він дозволяє трансформувати дані). Як і всі дуплексні потоки, потоки Transform реалізують інтерфейси для читання та запису.

Документація https://nodejs.org/api/webstreams.html#web-streams-api

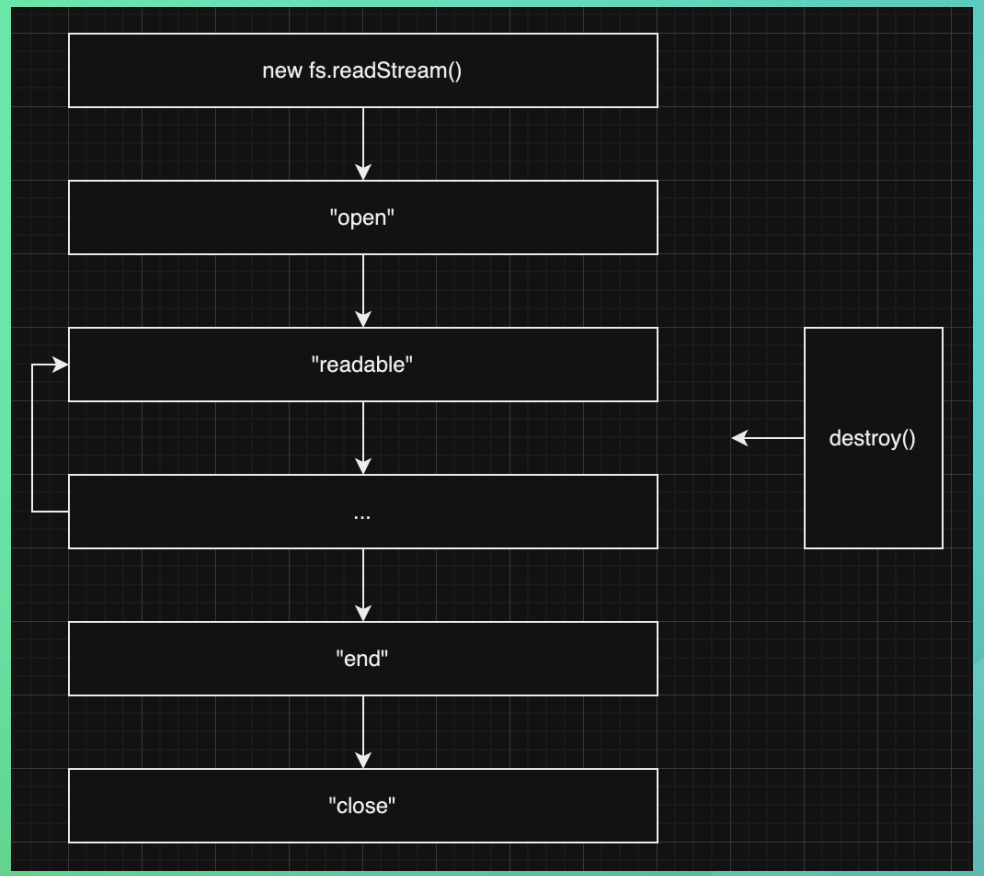
У Node.js більшість системних модулів дозволяють нам створювати потоки (http, fs …). Наприклад

1. У модулі fs - createReadableStream - це поток
2. У модулі http – createServer має request && response це потоки

Ви можете створити свої власні інстанси потоків за допомогою модуля streams

Streams

Приклад схеми потока readable



Streams

Events потока readable:

- readable
- data
- end
- error
- close
- open (only for fs)

readable.on(event, () => { … })

Документація https://nodejs.org/api/stream.html#readable-streams

Streams

Methods потока readable:

- readable.setEncoding('utf-8') - за замовченням бінарні данні.
- readable.pause()
- readable.isPaused()
- readable.resume()
- readable.read([size])
- readable.pipe()
- readable.unpipe()
- readable.destroy()
- …

Streams

Для того щоб зчитати файл створюємо новий потік за допомогою createReadStream з модуля fs

```
const readStream : ReadStream  = createReadStream(filePath, {
  encoding: "utf-8",
  highWaterMark: 128,
});
```

Передаємо туди шлях до файлу на необхідні парамери

Далі ми можемо підписатись на події цього потоку за допомогою .on(event, cb)

Документація https://nodejs.org/api/fs.html#filehandlecreatereadstreamoptions

Streams

Для того щоб зчитати файл створюємо новий потік за допомогою createReadStream з модуля fs

```
const readStream : ReadStream  = createReadStream(filePath, {
  encoding: "utf-8",
  highWaterMark: 128,
});
```

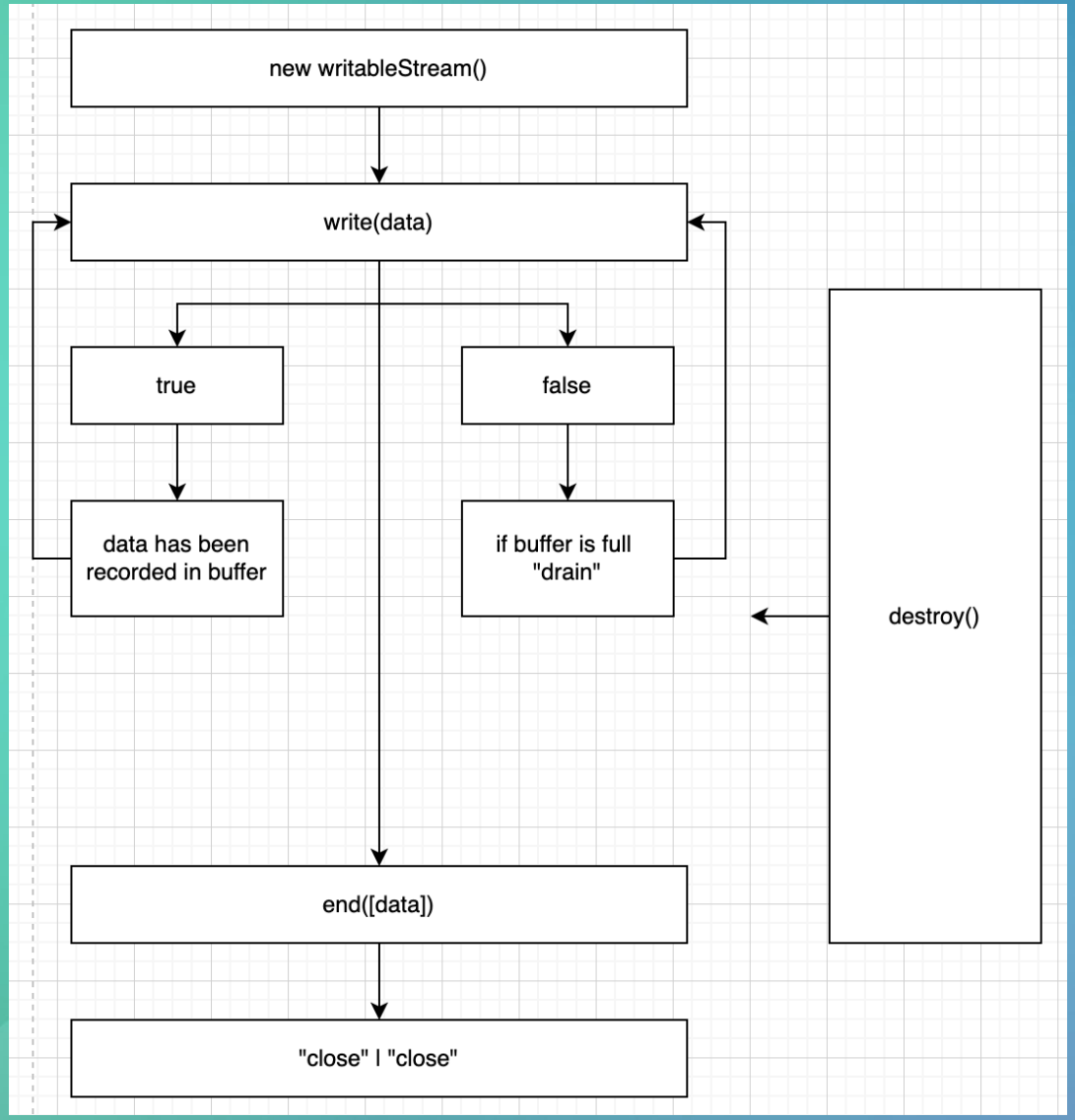
Передаємо туди шлях до файлу на необхідні парамери

Далі ми можемо підписатись на події цього потоку за допомогою .on(event, cb)

Документація https://nodejs.org/api/fs.html#filehandlecreatereadstreamoptions

Streams

Приклад схеми потока witeable



Streams

Events потока wtitable:

- drain
- finish
- pipe
- unpipe
- error
- close(only for fs)

wtitable.on(event, () => { … })

Документація https://nodejs.org/api/stream.html#writable-streams

Streams

Methods потока readable:

- writable.setDefaultEncoding(encoding)
- writable.cork()
- writable.write(chunk[, encoding][, callback])
- writable.end([chunk[, encoding]][, callback])
- writable.destroy([error])
- …

REST

RESTful API - це інтерфейс, що використовуються двома комп'ютерними системами для безпечного обміну інформацією через Інтернет.

Інтерфейс прикладного програмування (API) визначає правила, які необхідно дотримуватися для зв'язку з іншими програмними системами. Розробники впроваджують або створюють API-інтерфейси, щоб інші програми могли програмно взаємодіяти з їхніми програмами.

Таким чином, мережний API функціонує як шлюз між клієнтами та ресурсами в Інтернеті.

Representational State Transfer (REST) – це програмна архітектура, яка визначає умови роботи API. Спочатку REST створювалася як керівництво для управління взаємодіями у складній мережі, такій як Інтернет.

Принципи REST:

Єдиний інтерфейс є конструктивною основою будь-якого веб-сервісу RESTful. Це свідчить про те, що сервер передає інформацію у стандартному форматі.

Єдиний інтерфейс накладає чотири архітектурні обмеження:

1. Запити мають ідентифікувати ресурси. Це відбувається з допомогою єдиного ідентифікатора ресурсів.
2. Клієнти мають достатньо інформації у поданні ресурсу, щоб за бажання змінити чи видалити ресурс. Сервер виконує цю умову, відправляючи метадані, які додатково описують ресурс.
3. Клієнти отримують інформацію про подальшу обробку уявлень. Сервер реалізує це, відправляючи описові повідомлення, де містяться метадані у тому, як клієнт може використовувати їх оптимальним чином.
4. Клієнти отримують інформацію про всі пов'язані ресурси, необхідні виконання завдання. Сервер реалізує це, відправляючи гіперпосилання, щоб клієнти могли динамічно виявляти більше ресурсів.

Принципи REST:

Відсутність збереження стану

В архітектурі REST відсутність збереження стану відноситься до методу зв'язку, при якому сервер виконує кожен запит клієнта незалежно від усіх попередніх запитів. Клієнти можуть вимагати ресурси у будь-якому порядку, і кожен запит або ізольований від інших запитів, або його стан не зберігається. Це конструктивне обмеження REST API передбачає, що сервер може щоразу повністю зрозуміти та виконати запит.

Принципи REST:

Багаторівнева система

У багаторівневій системній архітектурі клієнт може підключатися до інших авторизованих посередників між клієнтом та сервером і, як і раніше, отримувати відповіді від сервера. Сервери також можуть надсилати запити іншим серверам. Ви можете спроектувати свою веб-службу RESTful для роботи на декількох серверах з кількома рівнями (безпекою, додатками та бізнес-логікою), які спільно виконують клієнтські запити. Ці рівні залишаються невидимими клієнта.

Принципи REST:

Місткість кешу

Веб-служби RESTful підтримують кешування, тобто процес збереження деяких відповідей на клієнта або посередника для скорочення часу відповіді сервера. Наприклад, ви заходите на веб-сайт із загальним зображенням верхнього та нижнього колонтитулів на кожній сторінці. Щоразу, коли ви відвідуєте нову сторінку веб-сайту, сервер повинен повторно надсилати ті самі зображення. Щоб уникнути цього, клієнт кешує або зберігає ці зображення після першої відповіді, а потім використовує зображення з кешу. Веб-служби RESTful керують кешуванням за допомогою відповідей API, які визначають себе як кешовані або некешовані.

Принципи REST:

Код за запитом

В архітектурному стилі сервери REST можуть тимчасово розширювати або налаштовувати функціональні можливості клієнта, передаючи код програмного забезпечення. Наприклад, коли ви заповнюєте реєстраційну форму на будь-якому веб-сайті, ваш браузер відразу ж виділяє всі допущені помилки (наприклад, неправильні номери телефонів). Це відбувається завдяки коду, надісланому сервером.

REST flow

Базовий принцип роботи RESTful API збігається із принципом роботи в Інтернеті. Клієнт зв'язується з сервером за допомогою API, коли йому потрібний ресурс. Розробники описують принцип використання REST API клієнтом у документації на API серверної програми. Нижче наведено основні етапи запиту REST API:

1. Клієнт надсилає запит на сервер. Керуючись документацією API, клієнт форматує запит так, щоб його розумів сервер.
2. Сервер автентифікує клієнта та підтверджує, що клієнт має право зробити цей запит.
3. Сервер отримує запит та внутрішньо обробляє його.
4. Сервер повертає клієнтові відповідь. Відповідь містить інформацію, яка повідомляє клієнту, чи був запит успішним. Також запит включає відомості, запрошені клієнтом.

Відомості про запит і відповідь REST API можуть відрізнятися залежно від того, як розробники проектують API.

REST clent request

API RESTful вимагає, щоб запити містили такі основні компоненти:

- Унікальний ідентифікатор ресурсу

Сервер надає кожному ресурсу унікальний ідентифікатор ресурсу. У випадку служб REST сервер ідентифікує ресурси за допомогою універсального покажчика ресурсів (URL). URL-адреса вказує шлях до ресурсу. URL-адреса аналогічна адресі веб-сайту, який ви вводите в браузері для відвідування веб-сторінки. URL-адреса також називається адресою запиту і чітко вказує серверу, що потрібно клієнту.

- Метод

Як правило, розробники реалізують RESTful API за допомогою протоколу передачі гіпертексту (HTTP). Метод HTTP повідомляє серверу, що необхідно зробити з ресурсом.

- Дані

Запити REST API можуть містити дані для успішної роботи POST, PUT та інших методів HTTP.

- Параметри

Запити RESTful API можуть включати параметри, які надають серверу докладнішу інформацію про необхідні дії. Cookie, параметри запиту, параметри шляху

Дякую за увагу