**Dasar-Dasar Node.js untuk Back-End**

Setelah Anda mengetahui teori tentang back-end, server, pola komunikasi server dengan client, serta REST API, *what’s next*? Yup! selanjutnya untuk menjadi seorang Back-End Developer tentu Anda perlu mengetahui bekal yang harus dimiliki dalam membangun sebuah aplikasi back-end atau web service.

Kita akan menggunakan Node.js dalam membangun web service. Namun sebelum menyelami materi ini lebih jauh, mari kita pelajari terlebih dahulu beberapa dasar-dasar API yang ada pada Node.js. Beberapa dasar yang akan dipelajari kali ini akan sangat membantu kita dalam pengembangan web service nantinya.

Pada akhir modul ini, diharapkan Anda dapat:

* Mengetahui apa itu Node.js
* Mengeksekusi JavaScript dengan Node.js
* Mengetahui Node.js Global Object
* Memahami Modularization
* Menggunakan Node Package Manager
* Memahami Eventing
* Memahami Filesystem
* Memahami teknik Readable Stream dan Writable Stream

Tools Belajar Dasar-Dasar Node.js untuk Back-End

Modul kali ini memiliki prasyarat sebelum Anda mengikutinya. Selain kemampuan JavaScript, terdapat dua tools yang perlu Anda siapkan, yakni Text Editor dan Node.js. Bila Anda sudah mengikuti seluruh latihan pada kelas Belajar Dasar Pemrograman JavaScript, tools ini seharusnya sudah terpasang pada komputer Anda. Bila belum, silakan unduh dan pasang dulu yah.

Text Editor

Selama mengikuti kelas ini, kami merekomendasikan Anda untuk menggunakan VSCode. Inilah text editor yang sangat populer dan gratis untuk digunakan. Selain itu, text editor ini memiliki plugin berlimpah yang dapat membuat fungsionalitas menjadi lebih kaya lagi. Visual Studio Code dapat dijalankan pada sistem operasi Windows, macOS, ataupun Linux. Untuk mengunduhnya, silakan kunjungi laman unduh visual studio code.

Node.js

Pastikan komputer Anda sudah terpasang Node.js dengan minimal versi 18 atau lebih tinggi. Untuk mengetahui versi Node.js yang terpasang, silakan tulis perintah ini pada Terminal atau CMD.

node -v

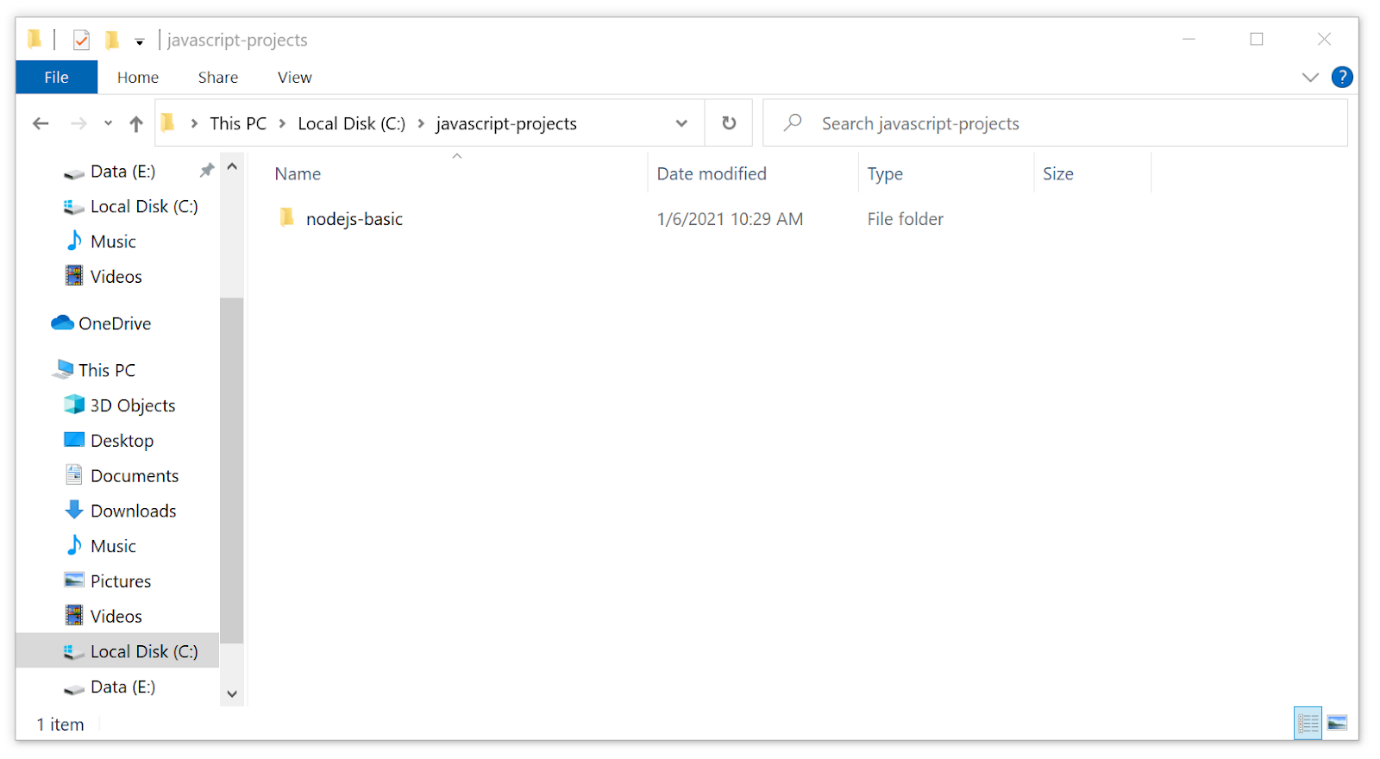
Sudah siap dengan prasyarat yang ada? Jika ya, yuk lanjut ke materi selanjutnya yah!

**Latihan: Membuat Proyek Node.js**

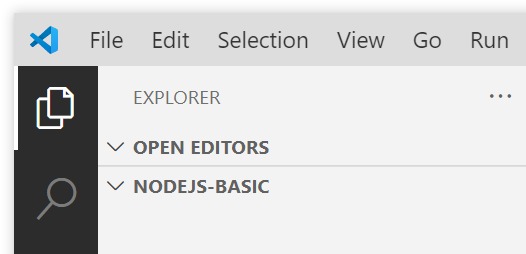
Setelah mengenal Node.js kini saatnya kita belajar cara menggunakannya. Kita akan mulai dari membuat proyek Node.js, menjalankan kode JavaScript menggunakan Node.js, hingga mempelajari berbagai API yang ada di dalamnya. Jadi sekali lagi pastikan prasyarat yang ada sudah Anda penuhi yah. Siapkan Text Editor, dan kita akan mulai menuliskan kode JavaScript.

Ikuti langkah-langkah berikut untuk membuat proyek Node.js.

1. Sebelum membuat proyek, buatlah folder baru terlebih dahulu. Folder ini akan digunakan sebagai tempat penyimpanan berkas proyek dan JavaScript yang kita tulis nanti. Kami sarankan, Anda buat folder tersebut di alamat C -> javascript-projects -> nodejs-basic bagi pengguna Windows; home -> javascript-projects -> nodejs-basic bagi pengguna Linux atau macOS.  
   * [**Windows**](https://www.dicoding.com/academies/261/tutorials/14202?from=14197#tab1-code1)
   * [Linux](https://www.dicoding.com/academies/261/tutorials/14202?from=14197#tab1-code2)
   * [macOS](https://www.dicoding.com/academies/261/tutorials/14202?from=14197#tab1-code3)

[](https://www.dicoding.com/academies/261/tutorials/14202?from=14197)

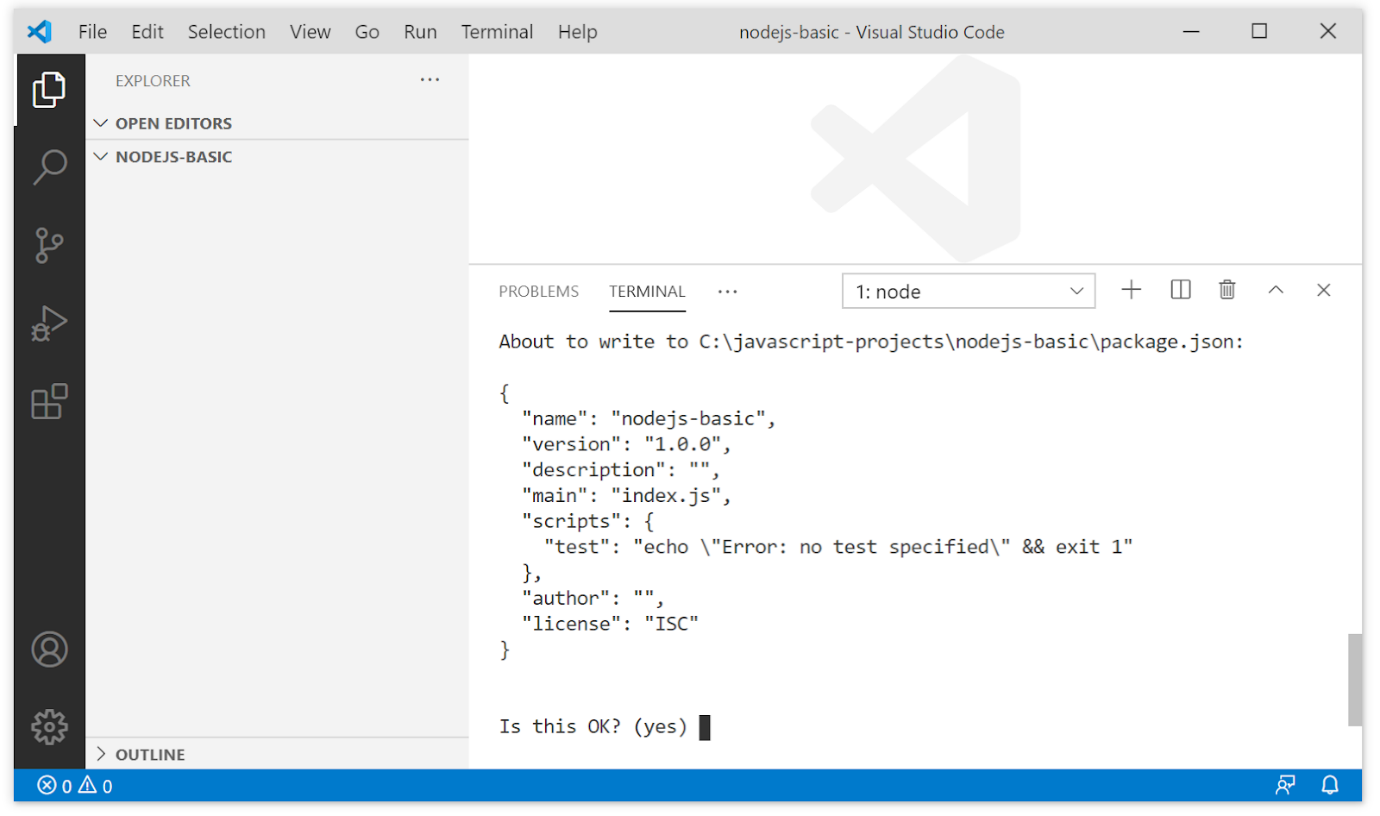
1. Selanjutnya, buka folder nodejs-basic menggunakan VSCode. Caranya, pada Visual Studio Code pilih menu File -> Open Folder -> [pilih foldernya]. Folder pun berhasil terbuka melalui VSCode.

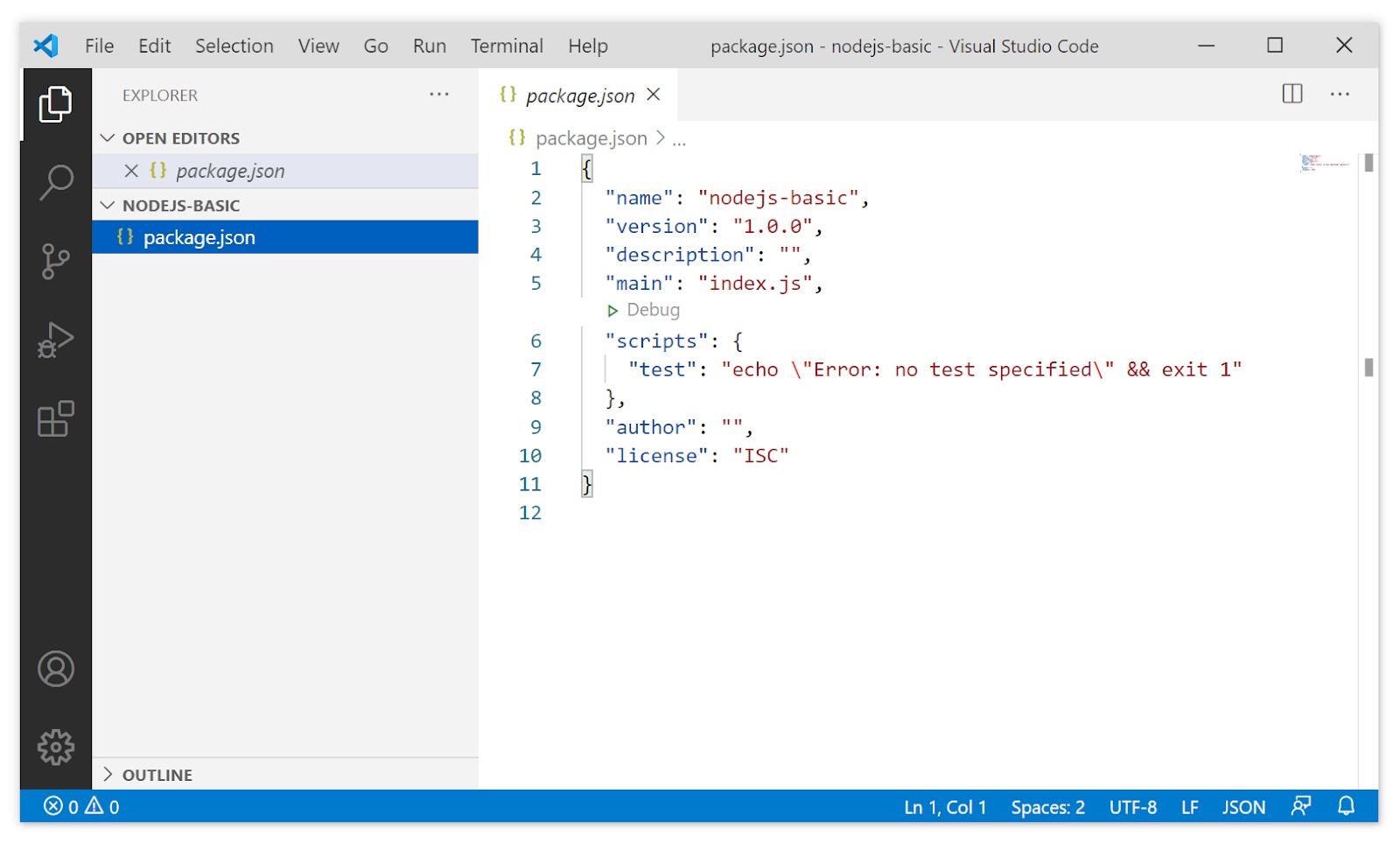
[](https://www.dicoding.com/academies/261/tutorials/14202?from=14197)

1. Untuk membuat proyek JavaScript, silakan buka Terminal pada VSCode. Pilih menu Terminal -> New Terminal, kemudian tuliskan perintah:
2. npm init

NPM alias Node Package Manager merupakan JavaScript Package Manager bawaan dari Node.js. Melalui NPM ini kita dapat membuat Node.js package (proyek) dan mengelola penggunaan package eksternal yang digunakan. Kita akan membahas NPM lebih detail nanti.

Jika Anda yang tidak menggunakan Visual Studio Code, gunakan Terminal/Command Prompt usungan OS Anda. Namun, sesuaikan lokasinya pada folder proyek ya.

1. Setelah menuliskan perintah di atas, Anda akan diberikan beberapa pertanyaan untuk mengisi nilai package name, version, description. Semua itu merupakan informasi dasar dari aplikasi yang Anda buat.  
   Nilai yang berada di dalam tanda kurung merupakan nilai default. Anda dapat menggunakan nilainya dengan langsung menekan tombol Enter. Untuk saat ini, cukup berikan semua pertanyaan dengan nilai default.
2. Setelah mengisi seluruh pertanyaan yang diberikan, Anda akan diberitahu untuk melihat hasil akhir yang dibuat pada berkas package.json.  
   [](https://www.dicoding.com/academies/261/tutorials/14202?from=14197)

Jika nilai yang ditampilkan sudah sesuai, langsung saja tekan tombol Enter. Berkas package.json pun telah berhasil dibuat pada proyek kita.        [](https://www.dicoding.com/academies/261/tutorials/14202?from=14197)

Voila! Anda berhasil membuat proyek Node.js.

**Apa itu Node.js**

Dari dulu hingga kini, browser menjadi tempat satu-satunya yang dapat mengeksekusi kode JavaScript. Karenanya, Web Developer perlu mempelajari bahasa pemrograman yang berbeda untuk mengembangkan aplikasi Front-End dan Back-End. Meskipun secanggih dan sekuat apa pun JavaScript berkembang, ia hanya akan digunakan di sisi Front-End saja.

Sebenarnya banyak developer yang mencoba membuat teknologi agar JavaScript dapat dijalankan di luar browser. Namun belum ada yang berhasil. Hingga pada tahun 2009, Ryan Dahl berhasil menciptakan [Node.js](https://nodejs.org/), teknologi yang diharapkan oleh banyak web developer. Tak disangka saat ini teknologi yang diciptakannya menuai popularitas tinggi. Node.js banyak digunakan oleh perusahaan besar sekelas Netflix, Uber, Paypal, dan eBay.

**Node.js berhasil menjadi JavaScript Runtime yang dapat mengeksekusi kode JavaScript di luar browser.** Node.js seolah-olah menjadi gerbang bagi para JavaScript Developer untuk mengembangkan sistem di luar dari browser. JavaScript menjadi bahasa multiplatform yang banyak menggiring developer untuk menggunakannya. Popularitas JavaScript pun meroket! Pada tahun 2014 hingga 2020 JavaScript menjadi bahasa pemrograman nomor satu yang banyak digunakan oleh developer. [[1]](https://octoverse.github.com/)

JavaScript menjadi salah satu pilihan tepat dalam membangun web server, terlebih bila Anda adalah seorang Front-End Web Developer. Anda tentu tidak perlu menggunakan bahasa yang berbeda dalam membangun Back-End. Anda bisa menjadi Full-Stack Developer dengan mempelajari satu bahasa pemrograman saja.

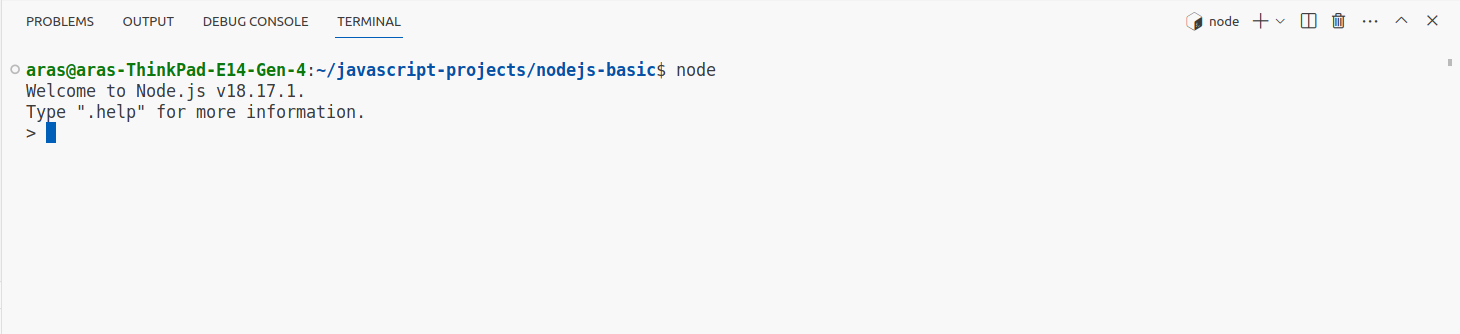
**Latihan: Menjalankan JavaScript Menggunakan Node.js**

Terdapat dua cara dalam menjalankan kode JavaScript menggunakan Node.js. Yang pertama dengan memanfaatkan Node REPL dan yang kedua dengan mengeksekusi berkas berekstensi JS. Mari kita kupas keduanya!

**The Node.js REPL**

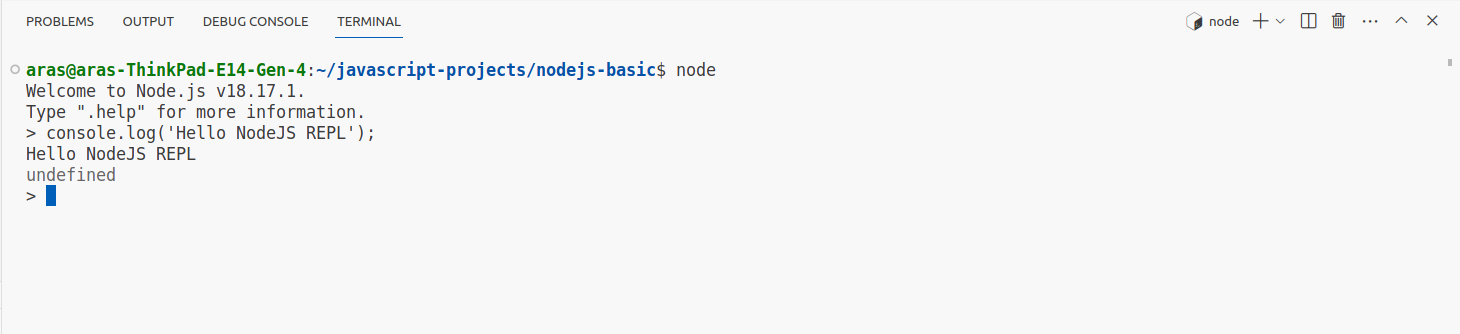
Node.js memiliki fitur REPL atau **R**ead-**E**val-**P**rint **L**oop. Sesuai namanya, fitur ini berfungsi untuk membaca kode JavaScript, mengevaluasi kode tersebut, kemudian mencetak hasil evaluasinya ke console. Nah, untuk *loop,* berarti proses tersebut selalu berulang.

1. REPL merupakan fitur bawaan dari Node.js. Anda bisa mengaksesnya menggunakan perintah **node** pada Terminal.

[](https://www.dicoding.com/academies/261/tutorials/14207?from=14202)

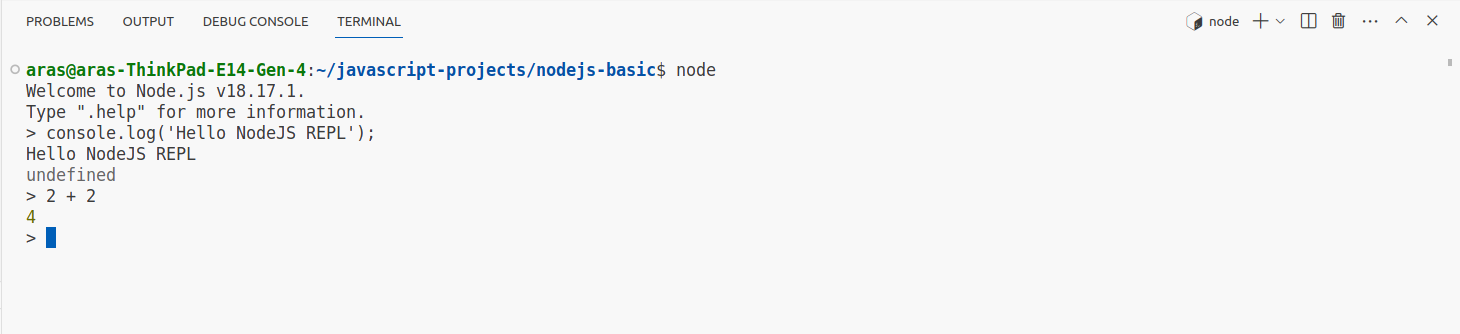
Tanda **>** pada Terminal menunjukan Anda sudah masuk ke mode Node REPL.

1. Sekarang, Anda bisa menuliskan kode JavaScript dan mengeksekusinya dengan menggunakan *enter*.

[](https://www.dicoding.com/academies/261/tutorials/14207?from=14202)

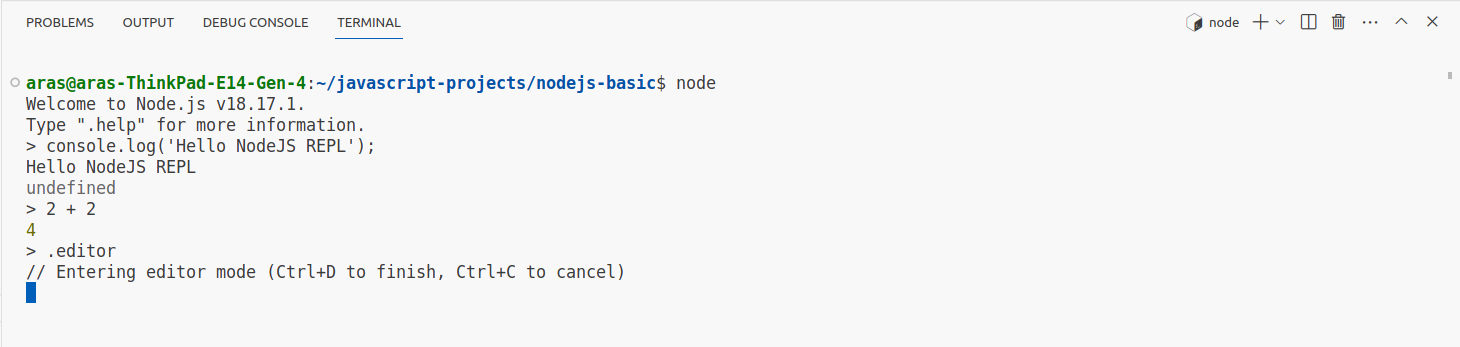
Lihat gambar di atas. Ketika mengeksekusi **console.log(‘Hello Node.js REPL’),**selain pesan “Hello Node.js REPL”, nilai **undefined** juga tercetak. Hal tersebut karena REPL selalu menampilkan nilai evaluasi pada console. Karena method **console.log()** tidak mengembalikan nilai, jadi *undefined*-lah yang tercetak pada console.

1. Untuk membuktikan hal itu, cobalah Anda tuliskan statement yang mengembalikan nilai. Contoh sederhananya **2+2**. Maka nilai **4** akan tercetak pada console.

[](https://www.dicoding.com/academies/261/tutorials/14207?from=14202)

Cukup asik kan fitur REPL? Tapi kok terkesan hanya dapat mengeksekusi kode satu baris saja ya? Bila Anda beranggapan seperti itu, sebenarnya tidak tepat karena di dalam REPL terdapat mode editor yang berfungsi untuk menuliskan kode JavaScript lebih dari satu baris.

1. Untuk menggunakan mode editor, Anda bisa tuliskan perintah **.editor**.

[](https://www.dicoding.com/academies/261/tutorials/14207?from=14202)

Ketika masuk ke mode editor, Anda bisa secara leluasa menuliskan kode JavaScript lebih dari satu baris menggunakan **enter**. Fungsi untuk mengeksekusi kode digantikan dengan kombinasi tombol **CTRL+D.**Untuk keluar dari mode editor, gunakan kombinasi **CTRL+C**.

[](https://www.dicoding.com/academies/261/tutorials/14207?from=14202)

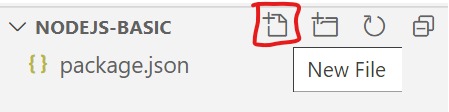
Nilai variabel yang Anda buat di REPL dapat diakses selama Anda masih berada di dalam REPL. Jika Anda menutup Terminal atau keluar dari REPL menggunakan perintah **.exit**, variabel yang sudah Anda buat sebelumnya tidak bisa diakses kembali. Itu artinya, REPL hanya menyimpan memory ketika session masih berlangsung.

Fitur REPL sangat berguna ketika Anda hendak melakukan kalkulasi sederhana, bereksperimen, atau belajar potongan kode JavaScript. Karena melalui REPL Anda bisa mengeksekusi kode JavaScript dan mendapatkan hasil dengan cepat tanpa harus membuat berkas JavaScript terlebih dahulu.

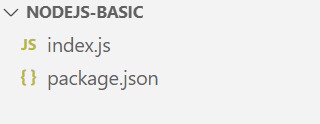
**Running JavaScript File using Node.js**

Cara lain untuk mengeksekusi kode JavaScript menggunakan Node.js adalah melalui berkas JS.

1. Silakan buat berkas JavaScript pada proyek nodejs-basic. Gunakan VSCode agar lebih mudah yah.

[](https://www.dicoding.com/academies/261/tutorials/14207?from=14202)

1. Buatlah berkas JavaScript dengan nama **“index.js”**.

[](https://www.dicoding.com/academies/261/tutorials/14207?from=14202)

1. Di dalam berkas **index.js**, Anda bisa menuliskan kode JavaScript sesuka Anda. Pastikan kode yang Anda tulis menampilkan nilai di console yah, jadi Anda bisa melihat nilai yang tampak pada console. Jika bingung, silakan tuliskan saja kode berikut.
2. **const** message = (name) => {
3. console.log(`Hello ${name}`);
4. }
6. message('JavaScript');
7. Untuk mengeksekusi kode tersebut, silakan buka kembali Terminal. Kemudian, tuliskan perintah:
8. node index.js

Node.js pun akan mengeksekusi berkas**‘index.js’**. Bila Anda menuliskan kode seperti yang dicontohkan di atas, maka akan muncul teks ‘Hello JavaScript’ pada console.

**Node.js Global Object**

JavaScript hanyalah bahasa pemrograman. Ia tidak mengetahui apakah Anda menjalankannya menggunakan browser atau Node.js. Di browser, JavaScript dapat mengontrol fungsionalitas browser seperti mengunjungi halaman, memuat ulang, menutup tabs, serta menampilkan *alert dialog*. JavaScript mampu melakukan itu karena browser menambahkan objek window pada JavaScript.

Di Node.js pun demikian, ia menambahkan objek global guna memberikan fungsionalitas lebih pada JavaScript. Hal ini bertujuan untuk mendukung pengembangan pada environment-nya. Contoh, melalui objek global kita dapat melihat berapa CPU yang digunakan pada komputer, modularisasi berkas JavaScript, menampilkan nilai pada console, dan hal lainnya.

Objek window pada browser dan objek global pada Node.js merupakan *Global Object*. Seluruh fungsi atau properti yang menjadi member dari global object dapat diakses di mana saja alias memiliki cakupan global. Pada Node.js Anda bisa melihat apa saja yang termasuk member dari global objek dengan menggunakan kode berikut:

1. Object.getOwnPropertyNames(global);

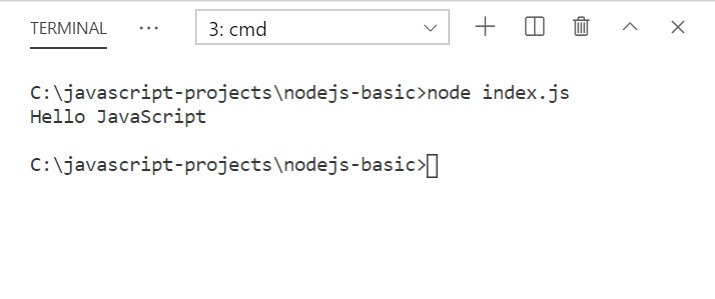
Coba jalankan pada REPL. Ia akan mengembalikan semua *member*-nya.

Banyak sekali yah member dari global objek. Namun dilansir dari website Node.js, sebenarnya mereka hanya menambahkan beberapa objek saja. Objek tersebut dinamakan dengan ‘true globals’. [[2]](https://nodejs.org/api/globals.html#:~:text=Class%3A%20WritableStreamDefaultWriter-,global%20objects)  
Berikut adalah daftarnya:

* global : Global namespace. Member apa pun di dalam object ini dapat diakses pada cakupan global.
* process : menyediakan interaksi dengan proses Node.js yang berjalan.
* console : menyediakan berbagai fungsionalitas [STDIO](http://www.cplusplus.com/reference/cstdio/).
* setTimeout, clearTimeout, setInterval, clearInterval.

Ada juga objek yang merupakan ‘pseudo-globals’ atau objek global semu. Objek ini tidak terlihat bila dicetak menggunakan Object.getOwnPropertyNames(global) sebab ia bukan member langsung dari objek global, melainkan diturunkan dari cakupan module. Karena pada Node.js semua berkas JavaScript adalah module [[3]](https://javascript.info/modules-intro), jadi pseudo-globals dapat diakses layaknya global objek.  
Berikut adalah daftarnya:

* module : digunakan untuk sistem modularisasi pada Node.js.
* \_\_filename : keyword untuk mendapatkan lokasi berkas JavaScript yang dieksekusi. Keyword ini tidak tersedia pada Node.js REPL.
* \_\_dirname : keyword untuk mendapatkan root directory dari berkas JavaScript yang dieksekusi.
* require : digunakan untuk mengimpor module JavaScript.

[](https://www.dicoding.com/academies/261/tutorials/14207?from=14202)

**Modularization**

Semakin kompleks program yang dikembangkan, semakin kompleks juga kode yang dituliskan. Jika kode dituliskan dalam satu berkas saja, maka akan sangat sulit untuk membaca serta memelihara kode tersebut. Idealnya, satu berkas JavaScript hanya memiliki satu tanggung jawab saja. Bila lebih dari satu, itu berarti Anda perlu berkenalan dengan modularisasi.

Modularisasi dalam pemrograman merupakan teknik pemisahan kode menjadi modul-modul yang bersifat independen namun bisa saling digunakan untuk membentuk suatu program yang kompleks. Pemisahan kode menjadi modul-modul terpisah inilah yang dapat membuat kode JavaScript lebih mudah diorganisir.

Pada Node.js, setiap berkas JavaScript adalah modul. Anda bisa membagikan nilai variabel, objek, class, atau apa pun itu antar modul. Untuk melakukannya, Anda perlu mengekspor nilai pada module tersebut.

Untuk mengekspornya, simpanlah nilai tersebut pada properti module.exports. Contoh seperti ini:

* [**coffee.js**](https://www.dicoding.com/academies/261/tutorials/14217#tab1-code1)

1. const coffee = {
2. name: 'Tubruk',
3. price: 15000,
4. }

7. module.exports = coffee;

Setelah itu nilai coffee dapat digunakan pada berkas JavaScript lain dengan cara mengimpor nilainya melalui fungsi global require().

* [**app.js**](https://www.dicoding.com/academies/261/tutorials/14217#tab2-code1)

1. const coffee = require('./coffee');
3. console.log(coffee);
5. /\*\*
6. \* node app.js
7. \*
8. \* output:
9. \* { name: 'Tubruk', price: 15000 }
10. \*/

Perhatikan nilai parameter yang diberikan pada require(). Parameter merupakan lokasi dari module target impor. Ingat! Jika Anda hendak mengimpor modul lokal (*local module*), selalu gunakan tanda ./ di awal alamatnya ya.

Bila berkas **coffee.js** diletakkan di folder yang berbeda dengan **app.js**, contohnya memiliki struktur seperti ini:

1. root folder:.
2. ├── app.js
3. ├── package.json
4. └── lib
5. └── coffee.js

Maka kita perlu mengimpornya dengan alamat:

* [**app.js**](https://www.dicoding.com/academies/261/tutorials/14217#tab3-code1)

1. const coffee = require('./lib/coffee');

Anda juga bisa menggunakan tanda ../ untuk keluar dari satu level folder. Ini berguna bila ingin mengimpor module yang berbeda hirarki seperti ini:

* [**app.js**](https://www.dicoding.com/academies/261/tutorials/14217#tab4-code1)

1. const coffee = require('../lib/coffee');

Bila Anda menggunakan VSCode, Anda akan terbantu dengan fitur auto import yang disediakan. Melalui fitur tersebut Anda tidak perlu repot-repot menuliskan alamat modul secara manual. Tinggal tulis saja nilai yang Anda ingin impor, VSCode akan menangani penulisan fungsi require().

Dalam melakukan impor dan ekspor nilai, kita bisa memanfaatkan object literal dan object destructuring agar dapat mengimpor dan mengekspor lebih dari satu nilai pada sebuah modul. Contoh:

* [**user.js**](https://www.dicoding.com/academies/261/tutorials/14217#tab5-code1)
* [app.js](https://www.dicoding.com/academies/261/tutorials/14217#tab5-code2)

1. const firstName = 'Harry';
2. const lastName = 'Potter';

5. /\* gunakan object literal
6. untuk mengekspor lebih dari satu nilai. \*/
7. module.exports = { firstName, lastName };

Untuk memudahkan developer dalam proses pengembangan, Node.js menyediakan beberapa modul bawaan yang dapat Anda manfaatkan guna mendukung efisiensi untuk melakukan hal-hal yang umum. Modul bawaan tersebut dikenal sebagai core modules. Anda bisa mengimpor *core modules* dengan fungsi yang sama, yakni require().

1. // Mengimpor core module http
2. const http = require('http');

Lokasi core module dituliskan tidak seperti local module. Lokasi bersifat mutlak (core module disimpan folder **lib** pada lokasi Node.js dipasang) sehingga kita cukup menuliskan nama modulnya saja.

Ada 3 jenis modul pada Node.js, Anda sudah mengetahui dua di antaranya. Berikut rinciannya:

* **local module** : module yang dibuat secara lokal berlokasi pada Node.js project Anda.
* **core module** : module bawaan Node.js berlokasi di folder **lib** di mana Node.js terpasang pada komputer Anda. Core module dapat digunakan di mana saja.
* **third party module** : module yang dipasang melalui Node Package Manager. Bila third party module dipasang secara lokal, maka modul akan disimpan pada folder **node\_modules** di Node.js project Anda. Bila dipasang secara global, ia akan disimpan pada folder **node\_modules** di lokasi Node.js dipasang.

Itulah tadi pembahasan mengenai modularisasi. Pada materi selanjutnya kita akan berkenalan dengan third party module dan Node Package Manager.

**Latihan: Modularization**

Sekarang Anda sudah tahu bagaimana cara menerapkan modularisasi pada JavaScript. Namun rasanya tidak afdal bila Anda tidak mempraktikannya sendiri. Untuk menguji pemahaman Anda tentang modularisasi, silakan lakukan latihan berikut.

Buat folder baru dengan nama **modularization** pada proyek nodejs-basic dan di dalamnya buat tiga berkas JavaScript baru yakni **Tiger.js**, **Wolf.js**, dan **index.js**.

Di dalam masing-masing berkas JavaScript, tuliskan starter code berikut:

* [**Tiger.js**](https://www.dicoding.com/academies/261/tutorials/14217#tab6-code1)
* [Wolf.js](https://www.dicoding.com/academies/261/tutorials/14217#tab6-code2)
* [index.js](https://www.dicoding.com/academies/261/tutorials/14217#tab6-code3)

1. class Tiger {
2. constructor() {
3. this.strength = Math.floor(Math.random() \* 100);
4. }
6. growl() {
7. console.log('grrrrr!')
8. }
9. }
11. // TODO 1

Selesaikan kode yang ditandai TODO dengan ketentuan berikut:

* **TODO 1** : Ekspor class Tiger agar dapat digunakan pada berkas JavaScript lain.
* **TODO 2** : Ekspor class Wolf agar dapat digunakan pada berkas JavaScript lain.
* **TODO 3** : Import class Tiger dari berkas **Tiger.js**.
* **TODO 4**: Import class Wolf dari berkas **Wolf.js**.

Setelah selesai mengerjakan TODO, eksekusi berkas index.js dengan perintah:

1. node ./modularization/index.js

Maka console akan menghasilkan output seperti ini:

Grrrr! Harimau memenangkan pertandingan!

**Mengalami kesulitan dalam menyelesaikan latihan?**  
Cobalah untuk ulas kembali materi yang diberikan atau tanyakan kesulitan yang Anda alami pada [forum diskusi](https://www.dicoding.com/academies/261/discussions). Hindari melihat atau membandingkan [kode solusi pada latihan modularization](https://github.com/dicodingacademy/a261-backend-pemula-labs/tree/201-node-js-basic/node-js-basic/modularization" \t "_blank). sebelum Anda mencobanya sendiri.

**Filesystem**

Seluruh data di komputer dikelola dan diakses melalui *filesystem*. Ketika menjalankan kode JavaScript pada browser, sangat penting untuk melimitasi JavaScript dalam mengakses filesystem. Teknik ini dinamakan dengan *sandboxing*. Sandboxing melindungi kita dari program jahat serta tindakan pencurian yang dapat merampas privasi penggunanya.

Bagaimana dengan JavaScript yang dijalankan di back-end? Limitasi tentu tetap ada, namun tidak seketat ketika JavaScript dieksekusi pada browser. Di back-end malah filesystem menjadi fitur esensial karena dalam pengembangan back-end akan sering sekali mengakses atau menulis sebuah berkas di dalam komputer.

Node.js menyediakan core modules fs yang dapat mempermudah kita dalam mengakses filesystem. Setiap method yang ada di module fs tersedia dalam dua versi, yakni versi asynchronous (default) dan versi synchronous.

Tentu Anda sudah tahu apa itu asynchronous dan synchronous kan? Jika belum, pelajari kedua hal tersebut pada kelas [Belajar Dasar Pemrograman JavaScript](https://www.dicoding.com/academies/256).

Untuk mengakses berkas pada komputer kita dapat menggunakan method fs.readFile(). Method ini menerima tiga argumen yakni: lokasi berkas, encoding, dan callback function yang akan terpanggil bila berkas berhasil/gagal diakses.

1. const fs = require('fs');
3. const fileReadCallback = (error, data) => {
4. if(error) {
5. console.log('Gagal membaca berkas');
6. return;
7. }
8. console.log(data);
9. };
11. fs.readFile('todo.txt', 'UTF-8', fileReadCallback);

Sebagai alternatif, Anda juga bisa gunakan method versi synchronous fs.readFileSync().

1. const fs = require('fs');
3. const data = fs.readFileSync('todo.txt', 'UTF-8');
4. console.log(data);

**Latihan: Filesystem**

Pada latihan kali ini, Anda akan ditugaskan untuk membuat program JavaScript yang dapat membaca teks dari berkas **.txt**. Karena Anda sudah belajar filesystem, tentu ini tidak akan sulit.

Silakan buat folder baru dengan nama filesystem. Di dalamnya buat dua berkas, yakni **index.js** dan **notes.txt**.

Di dalam masing-masing berkas, tuliskan kode/teks berikut:

* [**notes.txt**](https://www.dicoding.com/academies/261/tutorials/14642?from=14227#tab1-code1)
* [index.js](https://www.dicoding.com/academies/261/tutorials/14642?from=14227#tab1-code2)

1. Di hari minggu saya akan:
2. 1. Berolahraga pagi.
3. 2. Membersihkan halaman rumah.
4. 3. Menonton film.
5. 4. Membaca buku Laskar Pelangi.

Tugas Anda tertulis jelas seperti yang ada pada berkas index.js. Cobalah tampilkan teks pada berkas notes.txt pada console menggunakan filesystem. Jika TODO sudah selesai dikerjakan, eksekusi berkas index.js dengan perintah:

1. node ./filesystem/index.js

Bila TODO berhasil Anda kerjakan, maka outputnya tampak seperti ini:

**Tips**: Anda bisa gunakan method

1. path.resolve(\_\_dirname, 'notes.txt');

Dari core modules path dalam menetapkan alamat berkas secara lengkap dan dinamis.

**Mengalami kesulitan dalam menyelesaikan latihan?**  
Cobalah untuk ulas kembali materi yang diberikan atau tanyakan kesulitan yang Anda alami pada [forum diskusi](https://www.dicoding.com/academies/261/discussions). Hindari melihat atau membandingkan [kode solusi pada latihan filesystem](https://github.com/dicodingacademy/a261-backend-pemula-labs/tree/201-node-js-basic/node-js-basic/filesystem" \t "_blank). sebelum Anda mencobanya sendiri.

**Readable Stream**

Pada materi sebelumnya Anda sudah mengetahui cara mengakses berkas melalui fungsi fs.readFile(). Fungsi readFile baik versi asynchronous ataupun synchronous, bekerja dengan membaca berkas hingga selesai sebelum mengembalikan data. Itu berarti bila Anda menggunakannya untuk mengakses berkas yang besar, maka akan membutuhkan waktu lama dan memori yang besar untuk mendapatkan hasilnya. Hal ini sungguh tidak efektif.

Solusinya adalah dengan menggunakan teknik stream. Teknik ini tidak membaca berkas secara sekaligus, tapi dengan mengirim bagian demi bagian. Cara inilah yang digunakan oleh YouTube agar video dapat ditampilkan seketika kepada pengguna.

Teknik stream merupakan salah satu konsep fundamental yang mendukung aplikasi Node.js bekerja. Teknik ini dapat menangani kasus baca tulis berkas, komunikasi jaringan, atau beban kerja apapun agar dapat berjalan dengan lebih efisien. Sabar dulu yah, kasus yang disebutkan tadi terlalu kompleks untuk kita pelajari sekarang. Untuk memahami bagaimana stream bekerja kita gunakan kasus paling sederhana, yakni membaca teks pada berkas secara bagian-per-bagian.

Kita dapat membuat readable stream dengan menggunakan method createReadStream() dari core module fs.

1. const fs = require('fs');
3. const readableStream = fs.createReadStream('./article.txt', {
4. highWaterMark: 10
5. });
7. readableStream.on('readable', () => {
8. try {
9. process.stdout.write(`[${readableStream.read()}]`);
10. } catch(error) {
11. // catch the error when the chunk cannot be read.
12. }
13. });
15. readableStream.on('end', () => {
16. console.log('Done');
17. });

Fungsi createReadStream() menerima dua argumen. Yang pertama adalah lokasi berkas yang hendak dibaca, dan yang kedua adalah objek konfigurasi. Di dalam objek konfigurasi kita bisa menetapkan ukuran buffer melalui properti highWaterMark. Nilai default dari properti ini adalah 16384 bytes (16kb). Anda tidak perlu menetapkan properti ini bila ingin tetap memiliki nilai default. Namun karena kita hanya menggunakan berkas teks yang ukurannya sangat kecil, jadi kita buat ukuran buffer menjadi 10 bytes. Itu artinya berkas akan dibaca setiap 10 karakter (1 karakter = 1 bytes).

Buffer di dalam stream adalah memori sementara yang digunakan oleh stream dalam menyimpan data hingga data tersebut dikonsumsi.

createReadStream() mengembalikan EventEmitter, di mana kita dapat menetapkan fungsi listener setiap kali event readable dibangkitkan. Event readable akan dibangkitkan ketika buffer sudah memiliki ukuran sesuai dengan nilai yang ditetapkan pada properti highWaterMark, dalam arti buffer sudah siap dibaca. Kemudian event end akan dibangkitkan setelah proses stream selesai.

Buatlah berkas **article.txt** dan tuliskan teks berikut ini.

* [**article.txt**](https://www.dicoding.com/academies/261/tutorials/14647?from=14642#tab1-code1)

1. Stream di Node.js
3. Teknik stream merupakan salah satu konsep fundamental yang mendukung aplikasi Node.js bekerja. Teknik ini dapat menangani kasus baca tulis berkas, komunikasi jaringan, atau beban kerja apapun agar dapat berjalan dengan lebih efisien.

Bila kode di atas dijalankan maka akan menghasilkan *output* seperti ini:

* [**result**](https://www.dicoding.com/academies/261/tutorials/14647?from=14642#tab2-code1)

[Stream di ][Node.js  
  
][Teknik str][eam merupa][kan salah ][satu konse][p fundamen][tal yang m][endukung a][plikasi No][deJS beker][ja. Teknik][ ini dapat][ menangani][ kasus bac][a tulis be][rkas, komu][nikasi jar][ingan, ata][u beban ke][rja apapun][ agar dapa][t berjalan][ dengan le][bih efisie][n.][null]Done

**Rangkuman Dasar-Dasar Node.js untuk Back-End**

Anda berada di akhir dari modul Dasar-Dasar Node.js untuk Back-End. Mari kita uraikan materi yang sudah Anda pelajari untuk mempertajam pemahaman.

**Pengenalan Node.js**

Dari dulu hingga kini, browser menjadi tempat satu-satunya yang dapat mengeksekusi kode JavaScript. Karenanya, *Web Developer* perlu mempelajari bahasa pemrograman yang berbeda untuk mengembangkan aplikasi *Front-End* dan *Back-End*. Meskipun secanggih dan sekuat apa pun JavaScript berkembang, ia hanya akan digunakan di sisi *Front-End* saja.

Sebenarnya banyak *developer* yang mencoba membuat teknologi agar JavaScript dapat dijalankan di luar *browser*. Namun, belum ada yang berhasil. Hingga pada tahun 2009, Ryan Dahl berhasil menciptakan [Node.js](https://nodejs.org/), teknologi yang diharapkan oleh banyak *web developer*. Tak disangka, saat ini teknologi yang diciptakannya menuai popularitas tinggi. Node.js banyak digunakan oleh perusahaan besar sekelas Netflix, Uber, Paypal, dan eBay.

Node.js berhasil menjadi JavaScript Runtime yang dapat mengeksekusi kode JavaScript di luar *browser*.

**Menjalankan JavaScript Menggunakan Node.js**

Terdapat dua cara dalam menjalankan kode JavaScript menggunakan Node.js, yakni Node.js REPL dan berkas javascript.

* **Node.js REPL**  
  Node.js memiliki fitur REPL atau Read-Eval-Print Loop. Sesuai namanya, fitur ini berfungsi untuk membaca kode JavaScript, mengevaluasi kode tersebut, kemudian mencetak hasil evaluasinya ke console. Nah, untuk *loop*, berarti proses tersebut selalu berulang.  
    
  Jika Anda mengeksekusi console.log(‘Hello NodeJS REPL’), selain pesan “Hello NodeJS REPL”, nilai undefined juga akan tercetak. Hal tersebut terjadi karena REPL selalu menampilkan nilai evaluasi pada console. Karena method console.log() tidak mengembalikan nilai, teks undefined-lah yang tercetak pada console.
* **Berkas JavaScript**  
  Cara lain untuk mengeksekusi kode JavaScript menggunakan Node.js adalah melalui berkas berekstensi .js. Caranya mudah, tinggal buat berkas JavaScript pada proyek Node.js, tulis kode JavaScript di berkas tersebut, lalu eksekusi dengan perintah node namafile.js.

**Node.js Global Object**

Dilansir dari website Node.js, sebenarnya mereka hanya menambahkan beberapa objek saja. Objek tersebut dinamakan dengan ‘true globals’. Berikut adalah daftarnya:

* **global**: Global namespace. Member apa pun yang ada di dalam object ini dapat diakses pada cakupan global.
* **process**: menyediakan interaksi dengan proses Node.js yang berjalan.
* **console**: menyediakan berbagai fungsionalitas [STDIO](http://www.cplusplus.com/reference/cstdio/).
* **setTimeout**, **clearTimeout**, **setInterval**, **clearInterval**: berkaitan dengan waktu.

Ada juga objek yang merupakan ‘pseudo-globals’ atau objek global semu. Objek ini tidak terlihat bila dicetak menggunakan Object.getOwnPropertyName(global) sebab ia bukan member langsung dari objek global, melainkan diturunkan dari cakupan module. Karena pada Node.js semua berkas JavaScript adalah module [[3]](https://javascript.info/modules-intro), objek pseudo-globals dapat diakses layaknya global objek. Berikut adalah daftarnya:

* **module**: digunakan untuk sistem modularisasi pada Node.js.
* **\_\_filename**: keyword untuk mendapatkan lokasi berkas JavaScript yang dieksekusi. Keyword ini tidak tersedia pada Node.js REPL.
* **\_\_dirname**: keyword untuk mendapatkan root directory dari berkas JavaScript yang dieksekusi.
* **require**: digunakan untuk mengimpor module JavaScript.

**Process Object**

Pada Node.js, global objek process memiliki fungsi dan properti yang dapat memberikan informasi mengenai proses yang sedang berjalan.

Salah satu yang sering digunakan adalah properti process.env. Melalui properti ini, kita dapat menyimpan nilai atau mendapatkan informasi mengenai *environment* yang digunakan selama proses sedang berlangsung. Contoh, process.env memiliki properti process.env.PWD yang menyediakan informasi mengenai lokasi di mana proses dijalankan; properti process.env.USER menyimpan informasi nama user pada komputer Anda; dan masih banyak properti lainnya. Anda bisa lihat daftar lengkap properti yang ada pada halaman [dokumentasi Node.js mengenai process.env](https://nodejs.org/dist/latest-v8.x/docs/api/process.html" \l "process_process_env" \t "_blank).

Anda juga bisa secara manual menyimpan nilai di dalam process.env. Hal ini berguna untuk menentukan alur code seperti if-else dalam program berdasarkan *environment* yang Anda berikan. Contohnya, ketika Anda ingin nilai variabel host berbeda di kala pengembangan (*development*) dan produksi (*production*), Anda bisa membuat properti NODE\_ENV pada process.env. Jadi, Anda bisa menentukan nilai host berdasarkan kondisi NODE\_ENV.

**Modularization**

Makin kompleks program yang dikembangkan, Makin kompleks pula kode yang dituliskan. Jika kode aplikasi hanya ditulis dalam satu berkas saja, tentu itu akan menyulitkan kita atau developer lain untuk membaca dan memelihara aplikasi. Idealnya, satu berkas JavaScript hanya memiliki satu tanggung jawab saja. Bila lebih dari satu, itu berarti Anda perlu berkenalan dengan modularization atau modularisasi.

Modularisasi dalam pemrograman merupakan teknik pemisahan kode menjadi modul-modul yang bersifat independen, tetapi bisa saling digunakan untuk membentuk suatu program yang kompleks. Pemisahan kode menjadi modul-modul terpisah inilah yang dapat membuat kode JavaScript lebih mudah dikelola.

Pada Node.js, setiap berkas JavaScript adalah module. Anda bisa membagikan nilai variabel, objek, class, atau apa pun antar module. Untuk melakukannya, Anda perlu mengekspor nilai pada module tersebut.

Untuk mengekspornya, simpanlah nilai tersebut pada properti module.exports.

**Node Package Manager**

Dalam pengembangan aplikasi saat ini, industri gencar memanfaatkan module atau package luar agar proses pengembangan dapat lebih cepat. Semakin kompleks aplikasi tersebut, semakin banyak pula module/package yang digunakan. Di sinilah kita memerlukan sebuah package Manager.

Node Package Manager (NPM) merupakan pengelola package untuk JavaScript yang dapat memudahkan kita dalam mengelola package yang tersedia pada <https://www.npmjs.com/>. NPM merupakan standard package manager yang disediakan oleh Node.js dan sudah otomatis terpasang ketika memasang Node.js pada komputer kita. NPM dapat dioperasikan melalui CMD atau Terminal, Anda pun sudah mencobanya ketika membuat proyek JavaScript.

Selain untuk membuat proyek JavaScript, NPM dapat digunakan untuk memasang atau menghapus *third party module* (modul pihak ketiga). Modul yang dipasang melalui NPM akan disimpan pada folder **node\_modules**.

Terdapat dua tipe pemasangan module melalui NPM: global dan lokal.

* Bila dipasang secara global, module akan bersifat layaknya core module dan dapat digunakan di mana pun.
* Bila dipasang secara lokal, module hanya dapat digunakan pada cakupan proyek Node.js saja.

Namun, saat ini kami sangat menyarankan Anda untuk memasang modul pihak ketiga secara lokal saja. Hindari pemasangan modul secara global karena akan menyebabkan banyak masalah. Sebaiknya gunakan *npx* bila Anda ingin menjalankan Node.js package di mana pun yang Anda inginkan.

**Events**

Aplikasi Node.js biasanya dikenal memiliki pola *event-driven* atau memiliki alur berdasarkan suatu kejadian. Apa maksudnya itu? Mari kita jelajahi lebih dalam lagi.

Dunia nyata penuh dengan kejadian. Alarm berbunyi, ponsel berdering, turun hujan, ataupun kejadian lainnya. Sebagai manusia, kita membuat keputusan, lantas bertindak berdasarkan kejadian yang ada. Contohnya:

* Ketika berjalan dan tetiba turun hujan, kita bergegas menggunakan payung.
* Ketika ponsel berdering, kita bereaksi dengan mengangkat panggilan.
* Ketika merasa lapar, kita makan.

Seperti inilah pola yang terjadi di kehidupan nyata, sudah sejak lama kita bertahan hidup dengan pola seperti ini. Inilah yang dimaksud dengan pola *event-driven*.

Kita kembali ke dunia komputer. Tradisionalnya, programming dilakukan dengan cara yang imperatif. Agar komputer dapat melakukan sesuatu hal, kita perlu banyak menuliskan instruksi secara runtut beserta langkah-langkahnya. Komputer akan membaca kode dari atas ke bawah sesuai dengan urutan yang kita definisikan.

Dengan pola yang kaku seperti itu, kita akan sulit membangun program yang dapat menangani suatu kejadian. Karena kita saja tidak tahu kapan suatu kejadian akan terjadi, lantas bagaimana cara memberikan instruksi pada komputer? Lalu, bagaimana solusinya? Berkaca dari dunia nyata, program komputer juga harus bekerja dengan pola *event-driven*. Syukurlah dengan Node.js kita dapat menerapkan pola tersebut dengan mudah.

Node.js menyediakan EventEmitter class yang merupakan member dari events core module.

**Filesystem**

Ketahuilah bahwa seluruh data di komputer dikelola dan diakses melalui *filesystem*. Ketika kita menjalankan kode JavaScript pada browser, sangat penting untuk melimitasi JavaScript dalam mengakses filesystem. Teknik ini dinamakan dengan *sandboxing*. Sandboxing melindungi kita dari program jahat serta tindakan pencurian yang dapat merampas privasi penggunanya.

Bagaimana dengan JavaScript yang dijalankan di back-end? Limitasi tentu tetap ada, tetapi tidak seketat ketika JavaScript dieksekusi pada browser. Di back-end, justru filesystem menjadi fitur esensial karena dalam pengembangan back-end akan sering sekali mengakses atau menulis sebuah berkas di dalam komputer.

Node.js menyediakan core modules fs yang dapat mempermudah kita dalam mengakses filesystem. Setiap method yang ada di module fs tersedia dalam dua versi, yakni versi asynchronous (default) dan versi synchronous.

Tentu Anda sudah tahu apa itu asynchronous dan synchronous 'kan? Jika belum, pelajari kedua hal tersebut pada kelas [Belajar Dasar Pemrograman JavaScript](https://www.dicoding.com/academies/256).

Untuk mengakses berkas pada komputer, kita dapat menggunakan method fs.readFile(). Method ini menerima tiga argumen: lokasi berkas, encoding, dan callback function yang akan terpanggil bila berkas berhasil/gagal diakses.

**Readable Stream**

Fungsi readFile, baik versi asynchronous maupun synchronous, bekerja dengan membaca berkas hingga selesai sebelum mengembalikan data. Itu berarti, bila Anda menggunakannya untuk mengakses berkas yang besar, proses tersebut akan membutuhkan waktu lama dan memori yang besar untuk mendapatkan hasilnya. Hal ini sungguh tidak efektif!

Lantas, bagaimana? Solusinya adalah dengan menggunakan teknik stream. Teknik ini tidak membaca berkas secara sekaligus, melainkan dengan mengirim bagian demi bagian. Cara inilah yang digunakan oleh YouTube agar video dapat ditampilkan seketika kepada pengguna.

Teknik stream merupakan salah satu konsep fundamental yang mendukung aplikasi Node.js bekerja. Teknik ini dapat menangani kasus baca tulis berkas, komunikasi jaringan, atau beban kerja apa pun agar dapat berjalan dengan lebih efisien. Sabar dulu ya, kasus yang disebutkan tadi terlalu kompleks untuk kita pelajari sekarang. Untuk memahami bagaimana stream bekerja, kita akan gunakan kasus paling sederhana, yakni membaca teks pada berkas secara bagian-per-bagian.

Kita dapat membuat readable stream dengan menggunakan method createReadStream() dari core module fs.

**Writable Stream**

Apakah Anda tahu bahwa teknik *stream* juga dapat digunakan untuk menulis berkas? Teknik ini disebut *writable stream*. Untuk membuat *writable* *stream* dalam menulis berkas, gunakanlah method createWriteStream() dari core module fs.

Fungsi ini menerima satu argumen yakni alamat berkas untuk menyimpan hasil data yang dituliskan. Berkas *output* akan dibuat secara otomatis jika tidak ada. Namun, bila berkas tersebut sudah ada, data sebelumnya akan tertimpa!

Dengan ringkasan tersebut, diharapkan Anda dapat memahami semua materi yang telah disampaikan. Jika belum, Anda bisa ulas kembali materi yang diberikan pada modul ini. Untuk Anda yang sudah merasa mantap, yuk lanjut ke modul berikutnya!

**Video Dasar-Dasar Node.js**

Untuk memperdalam dan mempermudah pemahaman pada materi ini, Anda dapat menyimak video pembahasan berikut.

**Membangun Web Service menggunakan Node.js**

Selamat! Anda sudah belajar dasar dari Node.js. Kini, Anda telah memiliki bekal cukup untuk melangkah ke materi selanjutnya.

Seperti yang sudah Anda pahami, karena hadirnya Node.js, JavaScript tak lagi dikekang oleh browser. Node.js membuat JavaScript bebas dijalankan di mana saja. JavaScript menjadi bahasa pemrograman server side layaknya Java, PHP, C#, atau bahasa lainnya. Sekarang Anda bisa membuat apa saja menggunakan JavaScript! Termasuk aplikasi yang akan kita buat saat ini.

Yeay! Pada modul ini kita akan membuat web service menggunakan JavaScript, hal yang sulit dilakukan tanpa adanya Node.js. Karena itu, pastikan Anda mengikuti materi ini dengan baik yah! Hingga pada akhir modul ini Anda dapat:

* Membangun Web Service menggunakan Node.js secara *native*.
* Mengenal Node.js HTTP framework.
* Membangun Web Service menggunakan Hapi.js (Node.js HTTP framework).
* Membangun RESTful API sederhana.

Sudah tidak sabar ingin memulai membuat web service? yuk lanjut ke materi selanjutnya yah.

**Membuat HTTP Server**

Pengembangan back-end adalah hal prioritas untuk Node.js. Ia andal dalam membangun aplikasi back-end, salah satunya web server alias sebuah komputer yang dapat menangani dan menanggapi permintaan dari client. Node.js menyediakan core modules http untuk membangun web server.

1. const http = require('http');

HTTP module memiliki banyak member seperti objek, properti, atau method yang berguna untuk melakukan hal-hal terkait protokol HTTP. Salah satu member yang penting untuk kita ketahui sekarang adalah method http.createServer().

Sesuai namanya, method ini berfungsi untuk membuat HTTP server yang merupakan instance dari http.server. Method ini menerima satu parameter custom callback yang digunakan sebagai request listener. Di dalam request listener inilah logika untuk menangani dan menanggapi sebuah request dituliskan.

1. const http = require('http');
3. /\*\*
4. \* Logika untuk menangani dan menanggapi request dituliskan pada fungsi ini
5. \*
6. \* @param request: objek yang berisikan informasi terkait permintaan
7. \* @param response: objek yang digunakan untuk menanggapi permintaan
8. \*/
9. const requestListener = (request, response) => {
11. };
13. const server = http.createServer(requestListener);

Request listener memiliki 2 parameter, yakni request dan response. Seperti yang tertulis pada contoh kode di atas, request merupakan objek yang menyimpan informasi terkait permintaan yang dikirimkan oleh client. Di dalam objek ini kita bisa melihat alamat yang diminta, data yang dikirim, ataupun HTTP metode yang digunakan oleh client.

Sementara itu, response merupakan objek yang digunakan untuk menanggapi permintaan. Melalui objek ini kita bisa menentukan data yang diberikan, format dokumen yang digunakan, kode status, atau informasi response lainnya.

1. const requestListener = (request, response) => {
2. response.setHeader('Content-Type', 'text/html');
4. response.statusCode = 200;
5. response.end('<h1>Halo HTTP Server!</h1>');
6. };

Kode di atas merupakan contoh logika yang bisa dituliskan di dalam request listener. Request listener akan menanggapi setiap permintaan dengan dokumen HTML, kode status 200, dan menampilkan konten “Halo HTTP Server!”.

Lalu, bagaimana caranya agar server selalu sedia menangani permintaan yang masuk? Setiap instance dari http.server juga memiliki method listen(). Method inilah yang membuat http.server selalu standby untuk menangani permintaan yang masuk dari client. Setiap kali ada permintaan yang masuk, request listener akan tereksekusi.

Method listen() dapat menerima 4 parameter, berikut detailnya:

* port (number) : jalur yang digunakan untuk mengakses HTTP server.
* hostname(string) : nama domain yang digunakan oleh HTTP server.
* backlog (number) : maksimal koneksi yang dapat ditunda (pending) pada HTTP server.
* listeningListener (function) : callback yang akan terpanggil ketika HTTP server sedang bekerja (listening).

Namun, keempat parameter di atas bersifat opsional. Kita bisa memberikan nilai port saja, atau kombinasi apa pun dari keempatnya. Hal itu tergantung terhadap kebutuhan Anda. Namun lazimnya, ketika memanggil method listen() kita memberikan nilai port, hostname, dan listeningListener.

1. const port = 5000;
2. const host = 'localhost';
4. server.listen(port, host, () => {
5. console.log(`Server berjalan pada http://${host}:${port}`);
6. });

**Latihan Membuat HTTP Server**

Nah, sekarang giliran kita praktikan pada proyek **nodejs-web-server**. Silakan hapus kode yang ada pada **server.js** dan ganti dengan kode untuk membuat http server seperti ini:

1. const http = require('http');
3. const requestListener = (request, response) => {
4. response.setHeader('Content-Type', 'text/html');
6. response.statusCode = 200;
7. response.end('<h1>Halo HTTP Server!</h1>');
8. };

11. const server = http.createServer(requestListener);
13. const port = 5000;
14. const host = 'localhost';
16. server.listen(port, host, () => {
17. console.log(`Server berjalan pada http://${host}:${port}`);
18. });

Setelah itu, jalankan perintah npm run start pada Terminal. Jika server berhasil dijalankan, maka Anda akan melihat pesan ‘Server berjalan pada [http://localhost:5000](http://localhost:5000/)’.

Untuk pengguna sistem operasi Windows, bila pop-up di bawah ini muncul, pilih saja “**Allow access**”.

Selamat! Anda berhasil membuat HTTP Server pertama menggunakan Node.js. Anda bisa coba melakukan request pada server tersebut melalui cURL seperti ini:

* [**Terminal/CMD**](https://www.dicoding.com/academies/261/tutorials/14672?from=14667#tab1-code1)

1. curl -X GET http://localhost:5000/

Silakan eksekusi kode di atas pada Terminal atau CMD. Anda akan mendapatkan response sesuai dengan yang dituliskan pada *request listener*.

Anda juga bisa mencoba langsung pada browser dengan mengunjungi halaman <http://localhost:5000/>.

*Good Job!*Akhirnya Anda berhasil membuat web server pertama menggunakan Node.js! Walau masih sangat sederhana, namun hal tersebut merupakan hal pertama yang dilakukan oleh back-end developer dalam meniti karirnya.

Untuk materi selanjutnya, mari kita pelajari mengenai request dan response lebih dalam lagi.

**Method/Verb Request**

Web server yang sudah kita buat pada latihan sebelumnya sudah berhasil merespons dan menampilkan data dalam dokumen HTML. Namun, tahukah Anda bahwa web server yang kita buat belum mengenali sepenuhnya permintaan yang diberikan oleh client?

Maksudnya, meskipun client meminta dengan path atau method yang berbeda, server akan merespons dengan data yang sama. Server kita saat ini tidak peduli permintaan datang seperti apa, dia akan mengembalikan data yang sama. Anda bisa mencobanya sendiri melalui cURL dengan menggunakan HTTP method yang berbeda.

1. curl -X POST http://localhost:5000
2. // output: <h1> Halo HTTP Server!</h1>
3. curl -X PUT http://localhost:5000
4. // output: <h1> Halo HTTP Server!</h1>
5. curl -X DELETE http://localhost:5000
6. // output: <h1> Halo HTTP Server!</h1>
7. curl -X GET http://localhost:5000
8. // output: <h1> Halo HTTP Server!</h1>

Ketika mencobanya pastikan HTTP Server Anda sedang berjalan. Bila dalam keadaan terhenti, jalankan kembali server dengan perintah npm run start pada Terminal proyek nodejs-web-server.

Hal tersebut wajar karena kita memang belum menuliskan logika dalam menangani permintaan dari method yang berbeda. Lalu, bagaimana caranya agar bisa melakukan hal tersebut?

Fungsi request listener menyediakan dua parameter yakni request dan response. Fokus ke parameter request, parameter ini merupakan instance dari http.ClientRequest yang memiliki banyak properti di dalamnya.

Melalui properti-propertinya ini kita dapat mengetahui seperti apa karakteristik dari permintaan HTTP yang dilakukan oleh client. Seperti method yang digunakan, path atau alamat yang dituju, data yang dikirimkan (bila ada), dan informasi lain mengenai permintaan tersebut.

Untuk mendapatkan nilai method dari request, gunakanlah properti request.method seperti ini:

1. const requestListener = (request, response) => {
2. const method = request.method;
3. };

Atau, Anda bisa menggunakan cara yang lebih *clean* dengan menggunakan *object destructuring* seperti ini:

1. const requestListener = (request, response) => {
2. const { method } = request;
3. };

Properti method bernilai tipe method dalam bentuk string. Nilainya dapat berupa “GET”, “POST”, “PUT”, atau method lainnya sesuai dengan yang client gunakan ketika melakukan permintaan. Dengan memiliki nilai method, kita bisa memberikan respons berbeda berdasarkan tipe method-nya.

1. const requestListener = (request, response) => {
2. const { method } = request;
4. if(method === 'GET') {
5. // response ketika GET
6. }
8. if(method === 'POST') {
9. // response ketika POST
10. }
12. // Anda bisa mengevaluasi tipe method lainnya
13. };

Sekali lagi, tidak hanya properti method, objek request kaya akan properti dan fungsi lain di dalamnya. Anda dapat mengeksplorasi properti atau fungsi lainnya pada halaman dokumentasi [Node.js tentang HTTP Client Request](https://nodejs.org/api/http.html#http_class_http_clientrequest).

**Latihan Handling Request**

Sekarang Anda sudah tahu kan bagaimana cara mendapatkan nilai method dari permintaan client? Yuk terapkan pada web server yang sudah kita buat.

Pada latihan ini, kita akan melakukan konfigurasi agar server mengembalikan respons dengan kata “hello” dalam berbagai bahasa sesuai method yang digunakan client.

Silakan buka kembali berkas server.js pada proyek **nodejs-web-server**. Lalu, sesuaikan respons berdasarkan request method yang dilakukan oleh client.

* [**server.js**](https://www.dicoding.com/academies/261/tutorials/14677?from=14672#tab1-code1)

1. const http = require('http');
3. const requestListener = (request, response) => {
4. response.setHeader('Content-Type', 'text/html');
5. response.statusCode = 200;
7. const { method } = request;
9. if(method === 'GET') {
10. response.end('<h1>Hello!</h1>');
11. }
13. if(method === 'POST') {
14. response.end('<h1>Hai!</h1>');
15. }
17. if(method === 'PUT') {
18. response.end('<h1>Bonjour!</h1>');
19. }
21. if(method === 'DELETE') {
22. response.end('<h1>Salam!</h1>');
23. }
24. };
26. const server = http.createServer(requestListener);
28. const port = 5000;
29. const host = 'localhost';
31. server.listen(port, host, () => {
32. console.log(`Server berjalan pada http://${host}:${port}`);
33. });

Simpan perubahan pada berkas **server.js**; jalankan ulang server dengan perintah npm run start; dan coba lakukan permintaan ke server dengan menggunakan method yang berbeda melalui cURL.

Hasilnya akan tampak seperti ini:

1. curl -X GET http://localhost:5000
2. // output: <h1>Hello!</h1>
3. curl -X POST http://localhost:5000
4. // output: <h1>Hai!</h1>
5. curl -X PUT http://localhost:5000
6. // output: <h1>Bonjour!</h1>
7. curl -X DELETE http://localhost:5000
8. // output: <h1>Salam!</h1>

**Body Request**

Ketika client melakukan permintaan dengan method POST atau PUT, biasanya permintaan tersebut memiliki sebuah data yang disimpan pada body request. Data pada body bisa berupa format teks, JSON, berkas gambar, atau format lainnya. Data tersebut nantinya digunakan oleh server untuk diproses di database atau disimpan dalam bentuk objek utuh.

Ketahuilah bahwa http.clientRequest merupakan turunan dari readable stream, yang di mana untuk mendapatkan data body akan sedikit sulit dibandingkan dengan mendapatkan data header. Data di body didapatkan dengan teknik stream, seperti yang sudah kita ketahui, teknik ini memanfaatkan EventEmitter untuk mengirimkan bagian-bagian datanya. Dalam kasus http.clientRequest event data dan end-lah yang digunakan untuk mendapatkan data body.

Berikut adalah contoh bagaimana mendapatkan data body:

1. const requestListener = (request, response) => {
2. let body = [];
4. request.on('data', (chunk) => {
5. body.push(chunk);
6. });
8. request.on('end', () => {
9. body = Buffer.concat(body).toString();
10. });
11. };

Mari kita bedah kodenya.

* Pertama, kita deklarasikan variabel body dan inisialisasikan nilainya dengan array kosong. Ini berfungsi untuk menampung buffer pada stream.
* Lalu, ketika event data terjadi pada request, kita isi array body dengan chunk (potongan data) yang dibawa callback function pada event tersebut.
* Terakhir, ketika proses stream berakhir, maka event end akan terbangkitkan. Di sinilah kita mengubah variabel body yang sebelumnya menampung buffer menjadi data sebenarnya dalam bentuk string melalui perintah Buffer.concat(body).toString().

*Huft!*Cukup melelahkan yah untuk mendapatkan data melalui teknik stream. Guna memantapkan pemahaman, mari kita praktikan pada proyek web server sebelumnya.

**Latihan Mendapatkan Body Request**

Di latihan sebelumnya, web server yang kita buat sudah berhasil merespons sesuai request method yang dilakukan client. *That’s nice!*

Nah, di latihan kali ini kita akan coba mendapatkan data pada body request ketika client mengirimkan request menggunakan method POST.

Buatlah web server merespons permintaan method POST dengan menampilkan sapaan dan nama berdasarkan data body yang dikirim oleh client. Bila client mengirimkan nama “Dicoding”, maka respons akan menampilkan “Hai, Dicoding!”.

Client akan mengirimkan data nama tersebut menggunakan format JSON seperti ini:

1. { "name": "Dicoding" }

Namun sebelum itu, agar latihan lebih fokus terhadap bagaimana mendapatkan data pada body, kita hapus dulu logika method yang sebenarnya belum kita butuhkan, seperti PUT dan DELETE.

Jadi, silakan buka berkas server.js pada proyek **nodejs-web-server** dan hapuslah logika method PUT dan DELETE. Sehingga, berkas server.js tampak lebih ringkas seperti ini:

* [**server.js**](https://www.dicoding.com/academies/261/tutorials/14682?from=14677#tab1-code1)

1. const http = require('http');
3. const requestListener = (request, response) => {
4. response.setHeader('Content-Type', 'text/html');
5. response.statusCode = 200;
7. const { method } = request;
9. if(method === 'GET') {
10. response.end('<h1>Hello!</h1>');
11. }
13. if(method === 'POST') {
14. response.end('<h1>Hai!</h1>');
15. }
16. };
18. const server = http.createServer(requestListener);
20. const port = 5000;
21. const host = 'localhost';
23. server.listen(port, host, () => {
24. console.log(`Server berjalan pada http://${host}:${port}`);
25. });

Selanjutnya, kita bisa mulai menuliskan logika stream di dalam blok POST.

1. if(method === 'POST') {
2. let body = [];
4. request.on('data', (chunk) => {
5. body.push(chunk);
6. });
8. request.on('end', () => {
9. body = Buffer.concat(body).toString();
10. response.end(`<h1>Hai, ${body}!</h1>`);
11. });
12. }

Perhatikan kode di atas! Kita memindahkan proses respons di dalam callback event end. Hal ini diperlukan karena data body siap dikonsumsi hanya ketika event end telah dibangkitkan. Dalam arti lain, server tidak akan mengirimkan respons bila proses *stream* belum selesai.

Simpan perubahan pada berkas **server.js**; jalankan ulang server dengan perintah npm run start; dan coba lakukan permintaan ke server dengan menggunakan method POST melalui cURL seperti ini:

1. curl -X POST -H "Content-Type: application/json" http://localhost:5000 -d "{\"name\": \"Dicoding\"}"

Server akan menanggapi dengan hasil berikut:

1. <h1>Hai, {"name": "Dicoding"}!</h1>

Tunggu, ini bukan hasil yang kita harapkan. Body masih bernilai data string JSON. Data ini masih perlu kita olah lagi agar bisa mendapatkan nilai name yang sebenarnya. Gunakanlah JSON.parse() untuk mengubah JSON string menjadi JavaScript objek. Sesuaikan kembali kode pada blok POST menjadi seperti ini (lihat kode yang ditebalkan):

1. if(method === 'POST') {
2. let body = [];
4. request.on('data', (chunk) => {
5. body.push(chunk);
6. });
8. request.on('end', () => {
9. body = Buffer.concat(body).toString();
10. **const { name } = JSON.parse(body);**
11. **response.end(`<h1>Hai, ${name}!</h1>`);**
12. });
13. }

Simpan perubahan pada berkas **server.js**; jalankan ulang server dengan perintah npm run start; dan coba lagi lakukan permintaan ke server dengan menggunakan method POST.

1. curl -X POST -H "Content-Type: application/json" http://localhost:5000 -d "{\"name\": \"Dicoding\"}"

maka output-nya akan:

1. <h1>Hai, Dicoding!</h1>

*Voila!* Inilah hasil yang kita harapkan! Anda bisa kirimkan permintaan POST lain dengan data nama Anda sendiri. Cobalah, apakah hasilnya sesuai atau tidak?

1. curl -X POST -H "Content-Type: application/json" http://localhost:5000 -d "{\"name\": \"Dimas\"}"
3. // output: <h1>Hai, Dimas!</h1>

**Routing Request**

Ketika menangani request, hal yang perlu kita cek selain method adalah URL atau alamat yang dituju dari request tersebut. Sebagai contoh, ketika kita mengunjungi [dicoding.com](https://www.dicoding.com/) dan [dicoding.com/about](https://www.dicoding.com/about), tentu hasil yang kita terima dari server akan berbeda, bukan?

Request ke [dicoding.com](https://www.dicoding.com/) akan menampilkan homepage Dicoding, sedangkan [dicoding.com/about](https://www.dicoding.com/about) akan menampilkan halaman tentang Dicoding. Teknik ini dinamakan dengan routing. Routing merupakan istilah dalam menentukan respons server berdasarkan path atau url yang diminta oleh client.

Dalam http.clientRequest, untuk mendapatkan nilai url sangatlah mudah, semudah kita mendapatkan nilai request method yang digunakan.

1. const requestListener = (request, response) => {
2. const { url } = request;
3. };

Properti url akan mengembalikan nilai path secara lengkap tanpa host dan port yang digunakan server. Contohnya, bila client meminta pada alamat **http://localhost:5000/about** atau **http://localhost:5000/about/**, maka url akan bernilai **‘/about’**; bila meminta alamat**http://localhost:5000** atau **http://localhost:5000/**, maka url akan bernilai **‘/’**.

Dengan mendapatkan nilai url, kita dapat merespons client sesuai dengan path yang ia minta.

1. const requestListener = (request, response) => {
2. const { url } = request;
4. if(url === '/') {
5. // curl http://localhost:5000/
6. }
8. if(url === '/about') {
9. // curl http://localhost:5000/about
10. }
12. // curl http://localhost:5000/<any>
13. };

Kita juga bisa mengombinasikan evaluasi dengan method request. Alhasil, kita dapat menentukan respons lebih spesifik lagi.

1. const requestListener = (request, response) => {
2. const { url, method } = request;
4. if(url === '/') {
6. if(method === 'GET') {
7. // curl -X GET http://localhost:5000/
8. }
10. // curl -X <any> http://localhost:5000/
11. }
13. if(url === '/about') {
15. if(method === 'GET') {
16. // curl -X GET http://localhost:5000/about
17. }
19. if(method === 'POST') {
20. // curl -X POST http://localhost:5000/about
21. }
23. // curl -X <any> http://localhost:5000/about
24. }
26. // curl -X <any> http://localhost:5000/<any>
27. };

**Latihan Routing Request**

Saatnya kita latihan lagi yuk! Karena saat ini Anda sudah paham bagaimana cara menangani request berdasarkan URL yang diminta, mari buat web server kita agar dapat menangani request yang lebih spesifik berdasarkan URL dan method request. Berikut tugas atau ketentuan yang akan kita gunakan:

* URL: ‘/’
  + Method: GET
    - Mengembalikan “Ini adalah homepage”.
  + Method: <any> (selain GET)
    - Mengembalikan “Halaman tidak dapat diakses dengan <any> request”.
* URL: ‘/about’
  + Method: GET
    - Mengembalikan “Halo! Ini adalah halaman about”.
  + Method: POST (dengan melampirkan data name pada body)
    - Mengembalikan “Halo, <name>! Ini adalah halaman about”.
  + Method: <any> (selain GET dan POST)
    - Mengembalikan “Halaman tidak dapat diakses dengan <any> request”.
* URL: <any> (selain / dan /about)
  + Method: <any>
    - Mengembalikan “Halaman tidak ditemukan!”.

Sudah paham? *Huft*, latihan kali ini sepertinya lebih menantang. Siapkan secangkir kopi agar Anda tetap fokus dan mari kita mulai.

Pertama, agar kita dapat fokus pada hal routing. Beri komentar dulu kode logika di dalam fungsi request listener yang sebelumnya kita buat.

1. const requestListener = (request, response) => {
2. response.setHeader('Content-Type', 'text/html');
3. response.statusCode = 200;
5. const { method } = request;
7. // if(method === 'GET') {
8. //     response.end('<h1>Hello!</h1>');
9. // }
11. // if(method === 'POST') {
12. //     let body = [];
14. //     request.on('data', (chunk) => {
15. //         body.push(chunk);
16. //     });
18. //     request.on('end', () => {
19. //         body = Buffer.concat(body).toString();
20. //         const { name } = JSON.parse(body);
21. //         response.end(`<h1>Hai, ${name}!</h2>`);
22. //     });
23. // }
24. };

Selanjutnya, kita ambil properti url dari request menggunakan teknik destructuring object seperti mendapatkan nilai method. Lihat kode yang ditebalkan yah.

1. const requestListener = (request, response) => {
2. response.setHeader('Content-Type', 'text/html');
3. response.statusCode = 200;
5. const { method, **url } = request;**
7. /\*\* Kode komentar disembunyikan agar lebih ringkas \*/
8. }

*Good!* Sekarang kita sudah dapat nilai url dari request. Saatnya kita menentukan logika routing url sesuai dengan ketentuan menggunakan if else.

1. const requestListener = (request, response) => {
2. response.setHeader('Content-Type', 'text/html');
3. response.statusCode = 200;
5. const { method, url } = request;
7. **if(url === '/') {**
8. **// TODO 2: logika respons bila url bernilai '/'**
9. **} else if(url === '/about') {**
10. **// TODO 3: logika respons bila url bernilai '/about'**
11. **} else {**
12. **// TODO 1: logika respons bila url bukan '/' atau '/about'**
13. **}**
15. /\*\* Kode komentar disembunyikan agar lebih ringkas \*/
16. }

*Nice!* Coba lihat komentar TODO (yang harus dikerjakan) pada kode tersebut. Kita akan selesaikan TODO sesuai urutan yang ada yah. Urutan tersebut sengaja disusun dari yang paling mudah, lalu merangkak ke yang lebih sulit.

Blok else yang paling terakhir (TODO pertama) akan tereksekusi bila url bukan bernilai ‘/**’** atau ‘/about’. Berdasarkan ketentuan yang ada di atas, kita harus merespons dengan pesan “Halaman tidak ditemukan!”. Yuk kita langsung saja tulis responsnya.

1. const requestListener = (request, response) => {
2. response.setHeader('Content-Type', 'text/html');
3. response.statusCode = 200;
5. const { method, url } = request;
7. if(url === '/') {
8. // TODO 2: logika respons bila url bernilai '/'
9. } else if(url === '/about') {
10. // TODO 3: logika respons bila url bernilai '/about'
11. } else {
12. **response.end('<h1>Halaman tidak ditemukan!</h1>');**
13. }
15. /\*\* Kode komentar disembunyikan agar lebih ringkas \*/
16. }

*Good!* Mari kita coba dahulu perubahan yang ada. Simpan perubahan pada berkas **server.js**; jalankan ulang server dengan perintah npm run start; dan silakan lakukan permintaan ke alamat selain ‘/’ atau ‘/about’. Seharusnya, server akan merespons sesuai dengan pesan yang sudah kita tetapkan.

1. curl -X GET http://localhost:5000/home
2. // output: <h1>Halaman tidak ditemukan!</h1>
3. curl -X GET http://localhost:5000/hello
4. // output: <h1>Halaman tidak ditemukan!</h1>
5. curl -X GET http://localhost:5000/test
6. // output: <h1>Halaman tidak ditemukan!</h1>

Mantap! Satu ketentuan sudah selesai.

* URL: ‘/’
  + Method: GET
    - Mengembalikan “Ini adalah homepage”.
  + Method: <any> (selain GET)
    - Mengembalikan “Halaman tidak dapat diakses dengan <any> request”.
* URL: ‘/about’
  + Method: GET
    - Mengembalikan “Halo! Ini adalah halaman about”.
  + Method: POST (dengan melampirkan data name pada body)
    - Mengembalikan “Halo, <name>! Ini adalah halaman about”.
  + Method: <any> (selain GET dan POST)
    - Mengembalikan “Halaman tidak dapat diakses dengan <any> request”.
* URL: <any> (selain / dan /about)
  + Method: <any>
    - Mengembalikan “Halaman tidak ditemukan!”.

Sekarang, kita lanjut ke ketentuan lainnya. Mari selesaikan logika untuk url ‘/’ terlebih dahulu.

Berdasarkan ketentuan yang ada, url ‘/’ hanya dapat diakses menggunakan method GET oleh client. Jika tidak, maka server akan mengembalikan pesan “Halaman tidak dapat diakses dengan <any> request”.

<any> bernilai method yang digunakan client

1. if(url === '/') {
2. if(method === 'GET') {
3. // response bila client menggunakan GET
4. } else {
5. // response bila client tidak menggunakan GET
6. }
7. }

Lanjut, kita berikan respons sesuai ketentuan pada masing-masing blok if dan else.

1. if(url === '/') {
2. if(method === 'GET') {
3. response.end('<h1>Ini adalah homepage</h1>');
4. } else {
5. response.end(`<h1>Halaman tidak dapat diakses dengan ${method} request</h1>`);
6. }
7. }

Setelah selesai, ayo coba lagi perubahan yang kita lakukan. Simpan perubahan pada berkas **server.js**; jalankan ulang server dengan perintah npm run start; dan lakukan permintaan ke alamat ‘/’ dengan method GET dan lainnya. Harusnya sudah berjalan sesuai dengan ketentuan yah!

1. curl -X GET http://localhost:5000
2. // output: <h1>Ini adalah homepage</h1>
3. curl -X POST http://localhost:5000
4. // output: <h1>Halaman tidak dapat diakses dengan POST request</h1>
5. curl -X DELETE http://localhost:5000
6. // output: <h1>Halaman tidak dapat diakses dengan DELETE request</h1>

*Nice Work!* Satu ketentuan lain sudah selesai!

* URL: ‘/’
  + Method: GET
    - Mengembalikan “Ini adalah homepage”.
  + Method: <any> (selain GET)
    - Mengembalikan “Halaman tidak dapat diakses dengan <any> request”.
* URL: ‘/about’
  + Method: GET
    - Mengembalikan “Halo! Ini adalah halaman about”.
  + Method: POST (dengan melampirkan data name pada body)
    - Mengembalikan “Halo, <name>! Ini adalah halaman about”.
  + Method: <any> (selain GET dan POST)
    - Mengembalikan “Halaman tidak dapat diakses dengan <any> request”.
* URL: <any> (selain / dan /about)
  + Method: <any>
    - Mengembalikan “Halaman tidak ditemukan!”.

Bagaimana, mudah bukan? Kita lanjut ke ketentuan terakhir yuk!

Berdasarkan ketentuan yang ada, halaman /about dapat diakses oleh client dengan menggunakan method GET dan POST. Selain kedua method tersebut, server akan mengembalikan pesan “Halaman tidak dapat diakses menggunakan <any> request”.

Jadi, tulislah logika if else seperti ini:

1. else if(url === '/about') {
2. if(method === 'GET') {
3. // respons bila client menggunakan GET
4. } else if(method === 'POST') {
5. // respons bila client menggunakan POST
6. } else {
7. // respons bila client tidak menggunakan GET ataupun POST
8. }
9. }

Pada blok else terakhir, berikan respons sesuai dengan ketentuan.

1. else if(url === '/about') {
2. if(method === 'GET') {
3. // respons bila client menggunakan GET
4. } else if(method === 'POST') {
5. // respons bila client menggunakan POST
6. } else {
7. **response.end(`<h1>Halaman tidak dapat diakses menggunakan ${method} request</h1>`);**
8. }

Selanjutnya, lengkapi juga respons bila client menggunakan GET dan POST sesuai dengan ketentuan.

Tips: Agar tidak menulis ulang seluruh kode stream, salinlah kode blok POST yang diberikan komentar. Namun, sesuaikan nilai responsnya ya.

Setelah selesai kode pada blok ‘/about’ akan tampak seperti ini:

1. else if (url === '/about') {
2. if (method === 'GET') {
3. response.end('<h1>Halo! Ini adalah halaman about</h1>')
4. } else if (method === 'POST') {
5. let body = [];
7. request.on('data', (chunk) => {
8. body.push(chunk);
9. });
11. request.on('end', () => {
12. body = Buffer.concat(body).toString();
13. const {name} = JSON.parse(body);
14. response.end(`<h1>Halo, ${name}! Ini adalah halaman about</h1>`);
15. });
16. } else {
17. response.end(`<h1>Halaman tidak dapat diakses menggunakan ${method} request</h1>`);
18. }
19. }

Sekarang kita coba yuk! Simpan perubahan pada berkas **server.js**; jalankan ulang server dengan perintah npm run start; dan coba lakukan permintaan ke alamat ‘/about’ dengan method GET, POST, dan lainnya. Harusnya sudah berjalan sesuai dengan ketentuan yah!

1. curl -X GET http://localhost:5000/about
2. // output: <h1>Halo! Ini adalah halaman about</h1>
3. curl -X POST -H "Content-Type: application/json" http://localhost:5000/about -d "{\"name\": \"Dicoding\"}"
4. // output: <h1>Halo, Dicoding! Ini adalah halaman about</h1>
5. curl -X PUT http://localhost:5000/about
6. // output: <h1>Halaman tidak dapat diakses menggunakan PUT request</h1>
7. curl -X DELETE http://localhost:5000/about
8. // output: <h1>Halaman tidak dapat diakses menggunakan DELETE request</h1>

*Well Done!* Akhirnya seluruh ketentuan selesai kita terapkan!

* URL: ‘/’
  + Method: GET
    - Mengembalikan “Ini adalah homepage”.
  + Method: <any> (selain GET)
    - Mengembalikan “Halaman tidak dapat diakses dengan <any> request”.
* URL: ‘/about’
  + Method: GET
    - Mengembalikan “Halo! Ini adalah halaman about”.
  + Method: POST (dengan melampirkan data name pada body)
    - Mengembalikan “Halo, <name>! Ini adalah halaman about”.
  + Method: <any> (selain GET dan POST)
    - Mengembalikan “Halaman tidak dapat diakses dengan <any> request”.
* URL: <any> (selain / dan /about)
  + Method: <any>
    - Mengembalikan “Halaman tidak ditemukan!”.

Anda bisa merapikan kode saat ini dengan menghapus komentar kode yang sudah tidak digunakan lagi. Sehingga, sekarang berkas **server.js** tampak seperti ini:

* [**server.js**](https://www.dicoding.com/academies/261/tutorials/14687?from=14682#tab1-code1)

1. const http = require('http');
3. const requestListener = (request, response) => {
4. response.setHeader('Content-Type', 'text/html');
5. response.statusCode = 200;
7. const { method, url } = request;
9. if(url === '/') {
10. if(method === 'GET') {
11. response.end('<h1>Ini adalah homepage</h1>');
12. } else {
13. response.end(`<h1>Halaman tidak dapat diakses dengan ${method} request</h1>`);
14. }
15. } else if(url === '/about') {
16. if(method === 'GET') {
17. response.end('<h1>Halo! Ini adalah halaman about</h1>')
18. } else if(method === 'POST') {
19. let body = [];
21. request.on('data', (chunk) => {
22. body.push(chunk);
23. });
25. request.on('end', () => {
26. body = Buffer.concat(body).toString();
27. const { name } = JSON.parse(body);
28. response.end(`<h1>Halo, ${name}! Ini adalah halaman about</h1>`);
29. });
30. } else {
31. response.end(`<h1>Halaman tidak dapat diakses menggunakan ${method} request</h1>`);
32. }
33. } else {
34. response.end('<h1>Halaman tidak ditemukan!</h1>');
35. }
36. };
38. const server = http.createServer(requestListener);
40. const port = 5000;
41. const host = 'localhost';
43. server.listen(port, host, () => {
44. console.log(`Server berjalan pada http://${host}:${port}`);
45. });

**Daftar Modul**

34% Selesai

**[Pendahuluan](https://www.dicoding.com/academies/261/tutorials/14687?from=14682)**

**Response Status**

Sejauh ini kita telah membahas banyak tentang request. Kita sudah mengenal dan menggunakan method, url, body request, kemudian memberikan respons sesuai dengan karakteristik request yang ada.

Meskipun kita sudah bisa membuat server merespons permintaan, tapi sebenarnya kita belum belajar lebih dalam mengenai respons. Untuk itu, mari beranjak membahas lebih detail mengenai parameter kedua dari fungsi request listener ini.

Seperti yang sudah Anda ketahui pada modul pengenalan back-end, respons yang dibawa oleh server dibagi menjadi tiga bagian penting. Yang pertama status line, atau bisa kita sebut response status; yang kedua response header; dan yang ketiga response body. Kita bahas mulai dari response status dahulu yah.

Response status merupakan salah satu bagian dari respons yang berisikan tentang informasi apakah sebuah request berhasil atau gagal dilakukan. Status yang diberikan berupa kode (status code) dan pesan dari kode tersebut dalam bentuk teks (status message).

Indikasi keberhasilan request client ditentukan oleh response status code yang dikirim oleh server. Karena itu, tentu nilai status code tak bisa sembarang kita tetapkan. Status code haruslah bernilai 3 digit angka dengan ketentuan berikut:

* 100-199 : informational responses.
* **200 - 299 : successful responses.**
* 300-399 : redirect.
* **400-499 : client error.**
* **500-599 : server errors.**

Fokus terhadap poin yang ditebalkan yah karena poin itu akan sering digunakan. Silakan eksplorasi lebih detail mengenai status code pada [halaman MDN mengenai HTTP Status](https://developer.mozilla.org/id/docs/Web/HTTP/Status" \t "_blank).

Pada Node.js, penetapan nilai status code pada response dilakukan melalui properti response.statusCode.

1. const requestListener = (request, response) => {
2. // memberitahu client bahwa request resource yang diminta tidak ada.
3. response.statusCode = 404;
4. };

Oh ya! Dari halaman MDN yang diberikan di atas, kita juga bisa melihat bahwa status code selalu diiringi dengan status message. Contoh **200 Ok**, **400 Bad Request**, dan **404 Not Found**. Melalui status message ini kita dan juga client bisa paham maksud dari status kode.

Status message memiliki nilai standar sesuai dengan response code. Namun, kita bisa mengubahnya bila diperlukan. Untuk mengubah status message, Anda bisa gunakan properti response.statusMessage.

1. const requestListener = (request, response) => {
2. response.statusCode = 404;
4. // 404 defaultnya adalah 'not found'
5. response.statusMessage = 'User is not found';
6. };

Ketahuilah bahwa Anda sebaiknya tidak mengubah statusMessage dari nilai default bila tidak diperlukan. Walaupun hanya sekadar menerjemahkannya menjadi “Tidak ditemukan”.

**Latihan Mengubah Response Code**

Web server yang kita buat saat ini masih “cuek” dalam memberikan respons status pada client. Maksudnya, apa pun respons yang diberikan server statusnya selalu **200 OK**. Tidak percaya? Silakan lakukan request berikut pada curl.

1. curl -X GET http://localhost:5000/about -i
3. curl -X GET http://localhost:5000/test -i
5. curl -X DELETE http://localhost:5000/ -i

Hasilnya adalah:

Lihat informasi yang diberi warna kuning. Semua respons dari server berstatus 200 OK, ini membuat client mengira bahwa request berhasil dilakukan, tapi faktanya tidak.

Ada beberapa respons yang seharusnya bisa diberikan status yang lebih relevan. Seperti ketika client meminta resource yang tidak ditemukan (404 Not Found) atau menggunakan method request yang tidak tepat (400 Bad Request).

Karena Anda saat ini sudah mengetahui cara mengubah status code pada respons, mari kita perbaiki kesalahan ini.

Silakan buka kembali berkas server.js. Kemudian, hapus kode berikut dari fungsi request listener:

1. response.statusCode = 200;

Sebagai gantinya, kita tuliskan nilai status code satu per satu sebelum perintah response.end(). Tentu, sesuaikan nilai status code dengan kasus-kasus yang ada.

Contohnya, bila halaman tidak ditemukan, beri nilai **404** pada status code; bila halaman tidak bisa diakses menggunakan method tertentu, beri nilai **400** pada status code; sisanya, bila request berhasil dilakukan, beri nilai **200** pada status code. Yuk kita eksekusi!

Setelah semuanya selesai, fungsi request listener tampak seperti ini:

1. const requestListener = (request, response) => {
2. response.setHeader('Content-Type', 'text/html');
4. const { method, url } = request;
6. if(url === '/') {
7. if(method === 'GET') {
8. **response.statusCode = 200;**
9. response.end('<h1>Ini adalah homepage</h1>');
10. } else {
11. **response.statusCode = 400;**
12. response.end(`<h1>Halaman tidak dapat diakses dengan ${method} request</h1>`);
13. }
14. } else if(url === '/about') {
15. if(method === 'GET') {
16. **response.statusCode = 200;**
17. response.end('<h1>Halo! Ini adalah halaman about</h1>')
18. } else if(method === 'POST') {
19. let body = [];
21. request.on('data', (chunk) => {
22. body.push(chunk);
23. });
25. request.on('end', () => {
26. body = Buffer.concat(body).toString();
27. const { name } = JSON.parse(body);
28. **response.statusCode = 200;**
29. response.end(`<h1>Halo, ${name}! Ini adalah halaman about</h1>`);
30. });
31. } else {
32. **response.statusCode = 400;**
33. response.end(`<h1>Halaman tidak dapat diakses menggunakan ${method} request</h1>`);
34. }
35. } else {
36. **response.statusCode = 404;**
37. response.end('<h1>Halaman tidak ditemukan!</h1>');
38. }
39. };

Simpan perubahan pada berkas **server.js**; jalankan ulang server dengan perintah npm run start; dan coba lakukan lagi permintaan berikut menggunakan cURL:

1. curl -X GET http://localhost:5000/about -i
3. curl -X GET http://localhost:5000/test -i
5. curl -X DELETE http://localhost:5000/ -i

Sekarang server akan memberikan status respons yang sesuai.

**Response Header**

Pada web server yang sudah kita buat, ia memberikan respons dengan format dokumen HTML. Dokumen ini digunakan oleh browser dalam menampilkan website. Anda bisa melihat ini ketika mengakses web server melalui browser.

Pada url [http://localhost:5000](http://localhost:5000/) server akan mengembalikan pesan “Ini adalah homepage” atau pada url <http://localhost:5000/about> server akan mengembalikan pesan “Halo! Ini adalah halaman about”. Pesan yang ditampilkan tampak besar dan tebal karena ia dibungkus oleh elemen heading HTML.

Sebenarnya, server bisa merespons dengan memberikan data dalam tipe (MIME types) lain, seperti XML, JSON, gambar, atau sekadar teks biasa. Apa pun MIME types yang digunakan, web server wajib memberi tahu pada client.

Caranya, lampirkan property ‘Content-Type’ dengan nilai MIME types yang disisipkan pada header response. Untuk menyisipkan nilai pada header response, gunakanlah method setHeader().

1. const requestListener = (request, response) => {
2. response.setHeader('Content-Type', 'text/html');
3. };

Silakan eksplorasi apa saja MIME types yang bisa diberikan pada header Content-Type di halaman [Common Types dari MDN](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types) ini.

Anda bisa menetapkan data pada header sebanyak yang diinginkan. Method setHeader() menerima dua parameter, yakni nama properti dan nilai untuk headernya.

1. const requestListener = (request, response) => {
2. response.setHeader('Content-Type', 'text/html');
3. response.setHeader('Powered-By', 'Node.js');
4. };

Ketahuilah juga bahwa penulisan properti header dituliskan secara Proper Case atau setiap kata diawali dengan huruf kapital dan setiap katanya dipisahkan oleh tanda garis (-).

**Latihan mengubah dan menambah nilai header response**

Mari kita latihan!

Saat ini web server yang kita buat menampilkan format HTML dalam mengirimkan respons ke client. Nah, pada latihan kali ini kita akan mengubah format HTML menjadi format JSON. Selain itu, kita akan tambahkan properti Powered-By pada header untuk memberitahu client teknologi server apa yang kita gunakan.

Latihan ini akan sangat mudah. Jadi, yuk langsung saja!

Buka kembali berkas server.js dan lihat kode apa yang sekiranya perlu kita ubah untuk memberi tahu server bahwa kita akan menggunakan JSON untuk responnya?

Jika Anda mengira kode ini:

1. response.setHeader('Content-Type', 'text/html');

Tebakan Anda tepat sekali!

Seperti yang sudah Anda ketahui, properti header ‘Content-Type’ berfungsi untuk memberi tahu client seperti apa ia harus menampilkan data.

Contoh dengan nilai ‘text/html’, client khususnya browser akan menampilkan data yang dikirim oleh respons akan di-render atau ditampilkan dalam bentuk HTML. Itulah mengapa pesan respons tampak besar ketika melakukan request menggunakan browser.

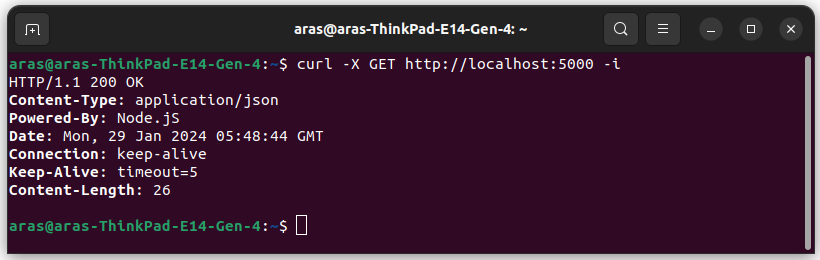
Karena kita ingin mengubah Content-Type menjadi JSON, maka ubah text/html menjadi application/json.

1. response.setHeader('Content-Type', **'application/json'**);

Selanjutnya, tetapkan juga nilai properti Powered-By dengan nilai “**Node.js**”.

1. response.setHeader('Content-Type', 'application/json');
2. **response.setHeader('Powered-By', 'Node.js');**

Simpan perubahan pada berkas **server.js**; jalankan ulang server dengan perintah npm run start; dan coba lakukan lagi permintaan ke server menggunakan cURL.

[](https://www.dicoding.com/academies/261/tutorials/14697?from=14692)

Anda bisa lihat, sekarang nilai Content-Type berubah menjadi **‘application/json’**. Selain itu, ada properti header baru yang kita tetapkan, yakni Powered-By dengan nilai ‘**Node.js**’.

Oh ya, karena server tidak lagi mengirimkan konten dalam bentuk HTML, maka browser tidak akan lagi menampilkan dalam bentuk HTML. Coba buka [http://localhost:5000](http://localhost:5000/) melalui browser. Sekarang konten HTML tidak lagi ter-render.

Dengan begitu, memberikan pesan dalam format HTML sudah tidak relevan lagi. Kita akan mengubahnya menjadi format JSON. Tapi sabar, kita tak akan lakukan itu sekarang. Kita pelajari dahulu lebih dalam bagaimana cara mengirimkan body respons pada server.

**Response Body**

Header respons menampung informasi terkait respons yang diberikan oleh server. Informasi dapat berupa status respons, MIME types, tanggal, atau informasi lainnya yang mungkin dibutuhkan oleh client.

Walaupun kita bisa memberikan informasi apa pun, namun tidak semua informasi cocok disimpan di header. Informasi pada header hanya sebagai metadata atau informasi yang menjelaskan tentang sebuah data lain (data utama).

Selain header, HTTP respons juga membawa body (Anda mengetahui ini pada materi pola [komunikasi client dan server](https://www.dicoding.com/academies/261/tutorials/14147/)). Di dalam body inilah data utama (atau bisa kita sebut konten) seharusnya disimpan.

Ketahuilah bahwa objek response yang berada pada parameter fungsi request listener adalah instance dari http.serverResponse. Di mana ia merupakan WritableStream. Masih ingat cara menulis data pada WritableStream? Nah, cara itulah yang digunakan untuk menuliskan data pada body response.

Seperti objek WritableStream lainnya, untuk menuliskan data pada respons kita bisa gunakan method response.write() dan diakhiri dengan method response.end().

1. const requestListener = (request, response) => {
2. response.write('<html>');
3. response.write('<body>');
4. response.write('<h1>Hello, World!</h1>');
5. response.write('</body>');
6. response.write('</html>');
7. response.end();
8. };

Seperti yang sudah Anda ketahui juga, method end() pada WritableStream dapat digunakan untuk menulis data terakhir sebelum proses penulisan diakhiri. Jadi, untuk kasus di atas dapat dipersingkat penulisannya menjadi seperti ini.

1. const requestListener = (request, response) => {
2. response.end('<html><body><h1>Hello, World!</h1></body></html>');
3. };

Ketahuilah bahwa penting untuk menuliskan status dan header response sebelum Anda menuliskan data pada body. Karena tidak masuk akal bila Anda sudah menuliskan body, namun belum memberikan metadata terkait data apa yang hendak dikirim.

**Latihan Mengubah Data pada Body Response**

Di latihan sebelumnya Anda sudah mengubah properti Content-Type pada header menjadi application/json. Namun untuk konten yang dikirim server pada body, masih berformat HTML. Nah, pada latihan kali ini kita akan mengubah konten pada body menjadi format JSON. Ayo kita eksekusi!

Pastikan Anda sudah tahu dan paham apa itu dan bagaimana penulisan JSON. Bila tidak, silakan ulas kembali materi [format request dan response](https://www.dicoding.com/academies/261/tutorials/14162).

Ketentuannya begini, setiap JSON yang akan kita kirimkan harus memiliki message. Nilai properti message akan diisi dengan pesan yang sebelumnya kita berikan dalam format HTML. Untuk lebih jelasnya, berikut contoh response body ketika client meminta halaman yang tidak ditemukan.

1. {
2. "message": "Halaman tidak ditemukan!"
3. }

Sudah paham? Yuk langsung saja kita buka kembali **server.js**.

Kita ubah konten yang mudah dahulu yah, lebih tepatnya ketika client mengakses halaman yang tidak ditemukan. Silakan ubah kode ini:

1. response.end('<h1>Halaman tidak ditemukan!</h1>');

Menjadi:

1. response.end(**JSON.stringify({**
2. **message: 'Halaman tidak ditemukan!',**
3. **})**);

Karena response.end() menerima string (atau buffer), maka kita perlu mengubah objek JavaScript menjadi JSON string menggunakan JSON.stringify().

Mari kita coba dulu perubahan yang ada. Simpan perubahan pada berkas server.js; jalankan ulang server dengan perintah npm run start; dan coba lakukan permintaan ke alamat selain ‘/’ atau ‘/about’. Seharusnya, server akan merespons konten dengan format JSON.

1. curl -X GET http://localhost:5000/anything
2. // output: { "message":"Halaman tidak ditemukan!"}
3. curl -X GET http://localhost:5000/test
4. // output: { "message":"Halaman tidak ditemukan!"}

Mantap! Silakan lanjut ubah format pesan untuk respons yang lain juga yah. Hingga fungsi request listener pada **server.js** tampak seperti ini:

1. const requestListener = (request, response) => {
2. response.setHeader('Content-Type', 'application/json');
3. response.setHeader('Powered-By', 'Node.js');
5. const { method, url } = request;
7. if(url === '/') {
8. if(method === 'GET') {
9. response.statusCode = 200;
10. **response.end(JSON.stringify({**
11. **message: 'Ini adalah homepage',**
12. **}));**
13. } else {
14. response.statusCode = 400;
15. **response.end(JSON.stringify({**
16. **message: `Halaman tidak dapat diakses dengan ${method} request`,**
17. **}));**
18. }
19. } else if(url === '/about') {
20. if(method === 'GET') {
21. response.statusCode = 200;
22. **response.end(JSON.stringify({**
23. **message: 'Halo! Ini adalah halaman about',**
24. **}));**
25. } else if(method === 'POST') {
26. let body = [];
28. request.on('data', (chunk) => {
29. body.push(chunk);
30. });
32. request.on('end', () => {
33. body = Buffer.concat(body).toString();
34. const { name } = JSON.parse(body);
35. response.statusCode = 200;
36. **response.end(JSON.stringify({**
37. **message: `Halo, ${name}! Ini adalah halaman about`,**
38. **}));**
39. });
40. } else {
41. response.statusCode = 400;
42. **response.end(JSON.stringify({**
43. **message: `Halaman tidak dapat diakses menggunakan ${method}, request`**
44. **}));**
45. }
46. } else {
47. response.statusCode = 404;
48. **response.end(JSON.stringify({**
49. **message: 'Halaman tidak ditemukan!',**
50. **}));**
51. }
52. };

*Well done!* Simpan perubahan pada berkas **server.js**; jalankan ulang server dengan perintah npm run start; dan coba lakukan lagi permintaan ke server menggunakan cURL. Server saat ini akan merespon dengan JSON sepenuhnya.

1. curl -X GET http://localhost:5000/
2. // output: {"message":"Ini adalah homepage"}
3. curl -X GET http://localhost:5000/about
4. // output: {"message":"Halo! ini adalah halaman about"}
5. curl -X DELETE http://localhost:5000/
6. // output: {"message":"Halaman tidak dapat diakses dengan DELETE request"}

**Node.js Web Framework**

Selamat yah, Anda sudah bisa membuat web server sederhana menggunakan Node.js. Dengan begitu, Anda diharapkan semakin mengerti dalam memahami bagaimana client dan server berkomunikasi melalui HTTP.

Ketika membuat web server menggunakan Node.js, mungkin sebagian Anda bertanya-tanya, “Apakah tidak ada cara yang lebih efektif lagi untuk membuat server menggunakan Node.js? Apakah harus sesulit itu? Haruskah fungsi request listener menampung seluruh logika? Bisakah membuat request handler secara spesifik berdasarkan url atau method? Bisakah kita mengatur kode agar lebih efektif dan mudah dibaca?” Wah, tenang, tenang!

Sejatinya memang seperti itulah dasar pembuatan web server menggunakan Node.js. Node.js tidak menyediakan cara mudah bahkan untuk melakukan hal-hal umum yang biasanya dilakukan ketika membuat web server.

Lantas, apakah cara yang sudah kita pelajari sejauh ini bisa digunakan untuk pengembangan web server yang kompleks, seperti membangun REST API?

Tentu bisa, namun akan sulit. Sulit untuk dipelihara, sulit untuk dipahami, dan juga sulit untuk dikembangkan. Tapi jangan berkecil hati dulu, bila mengalami kesulitan, kita harus cari bala bantuan. Siapa yang bisa menolong kita saat ini?

Jawabannya adalah tentu developer lain! Masalah yang kita hadapi saat ini sudah banyak dialami oleh Node.js developer lainnya.

Karena itu, baik developer, organisasi, atau bahkan instansi berlomba-lomba membuat solusi dengan membangun framework yang dapat membantu membuat web server dengan Node.js lebih cepat dan lebih mudah dikembangkan. Dengan bekal dasar yang sudah dimiliki saat ini, Anda berhak untuk mengeksplorasi dan menggunakan framework yang ada.

Tapi sabar dulu, sebelum menggunakannya, alangkah lebih baik kita pahami dahulu lebih dalam apa itu Web Framework.

**Apa itu Web Framework?**

Web Framework adalah sebuah kerangka yang dapat membantu mempermudah pengembangan web termasuk dalam membuat web server. Dengan menggunakan framework, penulisan kode akan lebih terstruktur, mudah dipelihara, dan gampang dikembangkan.

Web Framework menyediakan sekumpulan tools dan library yang dapat menyederhanakan hal-hal yang sering dilakukan dalam pengembangan web, seperti pembuatan server, routing, menangani permintaan, interaksi dengan database, otorisasi, hingga meningkatkan ketahanan web dari serangan luar.

**Web Framework di Node.js**

Di Node.js terdapat banyak Web Framework yang dapat Anda gunakan. Masing-masing framework yang ada tentu memiliki kelebihan dan kekurangan.

[Expressjs](https://expressjs.com/)merupakan web framework tertua dan terpopuler di Node.js saat ini. Framework ini sangat ringan, mudah diintegrasikan dengan aplikasi web front-end, dan penulisan kodenya tidak jauh beda dengan Node.js native.

Namun karena sifat ringannya tersebut, ia menjadi framework yang unopinionated alias tidak memiliki aturan untuk menggunakannya. Express tidak menyediakan struktur atau kerangka kerja yang baku untuk diikuti oleh developer. Sehingga, developer menjadi sulit menentukan seperti apa kode yang optimal.

Framework lainnya seperti [Hapi](https://hapi.dev/" \t "_blank) menyediakan environment yang lengkap untuk mengembangkan web server yang kompleks. Bila menggunakan Hapi, kita tak perlu tools lain untuk menerapkan layer **authentication**, **tokenize**, **cors**, dan lain sebagainya.

Kelemahan Hapi adalah abstraksinya yang terlalu jauh dari Node.js native. Kita perlu belajar secara dalam, untuk menguasai framework ini.

Penggunaan framework menjadi pilihan personal. Salah satu faktornya adalah kasus yang hendak Anda hadapi. Ketika ingin membangun server yang sederhana, katakanlah untuk mendukung aplikasi front-end di-*render*di sisi server, express adalah pilihan yang tepat.

Namun, bila Anda ingin membangun web server yang kompleks tanpa membutuhkan effort yang besar, Hapi adalah pilihan yang tepat.

Kita akan membangun web server dengan arsitektur REST yang kompleks ke depannya. Agar Anda selalu “Hapi” ketika mengikuti alur belajar, kita akan gunakan Hapi dalam membangun web server.

Ketahuilah bahwa Hapi memiliki environment yang cukup luas. Kelas ini tidak akan mengajarkan secara dalam tentang API yang ada di Hapi, melainkan hanya fitur-fitur yang menjadi dasar pembuatan REST API. Jadi, bila Anda ingin mendalami terkait framework Hapi, sempatkan waktu untuk eksplorasi di [dokumentasi Hapi](https://hapi.dev/tutorials/?lang=en_US" \t "_blank) yang disediakan yah.

Pada materi selanjutnya kita akan belajar dasar-dasar dari Hapi sambil coba membuat ulang web server dengan spesifikasi yang sama seperti yang kita lakukan pada latihan sebelumnya.

**Membangun Web Server menggunakan Hapi**

**Menyiapkan Project**

Mari kita awali dengan membuat proyek baru. Silakan buat folder di*C -> javascript-projects* (Windows) atau *home -> javascript-projects*(Linux dan macOS) dengan nama “**hapi-web-server**”.

Buka folder menggunakan VSCode, kemudian inisialisasi proyek pada Terminal dengan menggunakan perintah:

1. npm init --y

Lanjut, kita atur NPM runner pada package.json menjadi seperti ini:

1. "scripts": {
2. "start": "node server.js"
3. },

Lalu, buatlah berkas JavaScript baru dengan nama server.js. Kemudian, tuliskan kode berikut:

* [**server.js**](https://www.dicoding.com/academies/261/tutorials/14712?from=14707#tab1-code1)

1. console.log('Halo, kita akan belajar membuat server menggunakan Hapi');

Simpan perubahan pada berkas server.js dan coba jalankan perintah berikut pada Terminal:

1. npm run start

Bila Anda melihat pesan “Halo, kita akan belajar membuat server menggunakan Hapi”, maka proyek telah siap digunakan.