

Mastering Node.js: A Comprehensive Guide

This book provides an in-depth exploration of Node.js, guiding readers from foundational concepts to advanced practices. It covers everything from getting started with Node.js, handling asynchronous operations, using TypeScript, to ensuring security best practices. Ideal for developers looking to leverage Node.js for building scalable and efficient applications.

Table of Contents

Getting Started with Node.js

- [Introduction to Node.js](#)
- [How much JavaScript do you need to know to use Node.js?](#)
- [Differences between Node.js and the Browser](#)
- [The V8 JavaScript Engine](#)
- [An introduction to the npm package manager](#)
- [ECMAScript 2015 \(ES6\) and beyond](#)
- [Node.js, the difference between development and production](#)
- [Node.js with WebAssembly](#)
- [Debugging Node.js](#)
- [Profiling Node.js Applications](#)
- [Fetching data with Node.js](#)
- [WebSocket client with Node.js](#)
- [Security Best Practices](#)

TypeScript

- [Introduction to TypeScript](#)
- [Running TypeScript Natively](#)
- [Running TypeScript with a runner](#)
- [Running TypeScript code using transpilation](#)
- [Publishing a TypeScript package](#)

Asynchronous Work

- [Asynchronous flow control](#)
- [Overview of Blocking vs Non-Blocking](#)
- [JavaScript Asynchronous Programming and Callbacks](#)
- [Discover Promises in Node.js](#)
- [Discover JavaScript Timers](#)
- [The Node.js Event Loop](#)
- [The Node.js Event Emitter](#)
- [Understanding process.nextTick\(\)](#)

Node.js — Introduction to Node.js

[Learn About Download Blog Docs Contribute Certification](#)

[Change pageIntroduction to Node.js](#)

[Getting Started](#)

[Introduction to Node.js](#) [How much JavaScript do you need to know to use Node.js?](#) [Differences between Node.js and the Browser](#) [The V8 JavaScript Engine](#) [An introduction to the npm package manager](#) [ECMAScript 2015 \(ES6\) and beyond](#) [Node.js, the difference between development and production](#) [Node.js with WebAssembly](#) [Debugging Node.js](#) [Profiling Node.js](#) [Applications](#) [Fetching data with Node.js](#) [WebSocket client with Node.js](#) [Security](#) [Best Practices](#)

[TypeScript](#)

[Introduction to TypeScript](#) [Running TypeScript Natively](#) [Running TypeScript with a runner](#) [Running TypeScript code using transpilation](#) [Publishing a TypeScript package](#)

[Asynchronous Work](#)

[Asynchronous flow control](#) [Overview of Blocking vs Non-Blocking](#) [JavaScript Asynchronous Programming and Callbacks](#) [Discover Promises in Node.js](#) [Discover JavaScript Timers](#) [The Node.js Event Loop](#) [The Node.js Event Emitter](#) [Understanding process.nextTick\(\)](#) [Understanding setImmediate\(\)](#) [Don't Block the Event Loop](#)

[Manipulating Files](#)

[Node.js file stats](#) [Node.js File Paths](#) [Working with file descriptors in Node.js](#) [Reading files with Node.js](#) [Writing files with Node.js](#) [Working with folders in Node.js](#) [How to work with Different Filesystems](#)

[Command Line](#)

[Run Node.js scripts from the command line](#) [How to read environment variables from Node.js](#) [How to use the Node.js REPL](#) [Output to the command line using Node.js](#) [Accept input from the command line in Node.js](#)

Userland Migrations

[Introduction to Userland Migrations](#)

Modules

[Publishing a package](#) [How to publish a Node-API package](#) [Anatomy of an HTTP Transaction](#) [ABI Stability](#) [How to use streams](#) [Backpressuring in Streams](#)

Diagnostics

[User Journey](#) [Memory](#) [Live Debugging](#) [Poor Performance](#) [Flame Graphs](#)

Test Runner

[Discovering Node.js's test runner](#) [Using Node.js's test runner](#) [Mocking in tests](#) [Collecting code coverage in Node.js](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in

addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

CJS/MJS

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';  
const port = 3000;
```

```
const server = createServer((req, res) => {  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Hello World');  
});
```

```
server.listen(port, hostname, () => {  
  console.log(`Server running at http://${hostname}:${port}/`);  
});
```

JavaScript

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal. If you use mjs version of the code, you should save it as a `server.mjs` file and run `node server.mjs` in your terminal.

This code first includes the Node.js `http` module.

Node.js has a fantastic `standard library`, including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the `request event` is called, providing two objects: a request (an `http.IncomingMessage` object) and a response (an `http.ServerResponse` object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200;
res.setHeader('Content-Type', 'text/plain');
res.end('Hello World\n');
```

JavaScript

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

JavaScript

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

JavaScript

If you haven't already done so, [download](#) Node.js.

[Next](#)How much JavaScript do you need to know to use Node.js?

Reading Time

3 min

Authors

[F P M B R V T O O M](#) +10

Contribute

[Edit this page](#)

Table of Contents

1. [An Example Node.js Application](#)

1. [Navigate to Home](#)

2. [Getting Started](#)

3. [Introduction to Node.js](#)

[Trademark Policy Privacy Policy Version Support Code of Conduct Security](#)

[Policy](#)

[© OpenJS Foundation](#)

Node.js — Introduction to Node.js

Navigation

[Learn](#), [About](#), [Download](#), [Blog](#), [Docs](#), [Contribute](#), [Certification](#)

Change page

Introduction to Node.js

Getting Started

- [Introduction to Node.js](#)
- [How much JavaScript do you need to know to use Node.js?](#)
- [Differences between Node.js and the Browser](#)
- [The V8 JavaScript Engine](#)
- [An introduction to the npm package manager](#)
- [ECMAScript 2015 \(ES6\) and beyond](#)
- [Node.js, the difference between development and production](#)
- [Node.js with WebAssembly](#)
- [Debugging Node.js](#)
- [Profiling Node.js Applications](#)
- [Fetching data with Node.js](#)
- [WebSocket client with Node.js](#)
- [Security Best Practices](#)

TypeScript

- [Introduction to TypeScript](#)
- [Running TypeScript Natively](#)
- [Running TypeScript with a runner](#)
- [Running TypeScript code using transpilation](#)
- [Publishing a TypeScript package](#)

Asynchronous Work

- [Asynchronous flow control](#)
- [Overview of Blocking vs Non-Blocking](#)
- [JavaScript Asynchronous Programming and Callbacks](#)
- [Discover Promises in Node.js](#)
- [Discover JavaScript Timers](#)
- [The Node.js Event Loop](#)
- [The Node.js Event Emitter](#)
- [Understanding process.nextTick\(\)](#)
- [Understanding setImmediate\(\)](#)
- [Don't Block the Event Loop](#)

Manipulating Files

- [Node.js file stats](#)
- [Node.js File Paths](#)
- [Working with file descriptors in Node.js](#)
- [Reading files with Node.js](#)
- [Writing files with Node.js](#)
- [Working with folders in Node.js](#)
- [How to work with Different Filesystems](#)

Command Line

- [Run Node.js scripts from the command line](#)
- [How to read environment variables from Node.js](#)
- [How to use the Node.js REPL](#)
- [Output to the command line using Node.js](#)
- [Accept input from the command line in Node.js](#)

Userland Migrations

- [Introduction to Userland Migrations](#)

Modules

- [Publishing a package](#)
- [How to publish a Node-API package](#)
- [Anatomy of an HTTP Transaction](#)

- [ABI Stability](#)
- [How to use streams](#)
- [Backpressuring in Streams](#)

Diagnostics

- [User Journey](#)
- [Memory](#)
- [Live Debugging](#)
- [Poor Performance](#)
- [Flame Graphs](#)

Test Runner

- [Discovering Node.js's test runner](#)
- [Using Node.js's test runner](#)
- [Mocking in tests](#)
- [Collecting code coverage in Node.js](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js, the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers — you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal. If you use mjs version of the code, you should save it as a `server.mjs` file and run `node server.mjs` in your terminal.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200;
res.setHeader('Content-Type', 'text/plain');
res.end('Hello World\n');
```

We set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next, see [How much JavaScript do you need to know to use Node.js?](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js, the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers — you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use the `.mjs` version of the code, save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is triggered, providing two objects: a request (`http.IncomingMessage`) and a response (`http.ServerResponse`).

Those two objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case, with:

```
res.statusCode = 200;  
res.setHeader('Content-Type', 'text/plain');  
res.end('Hello World\n');
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already, [download](#) Node.js.

Next: [How much JavaScript do you need to know to use Node.js?](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking, and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database, or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js, the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```



```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use the `.mjs` version of the code, save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those two objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case, with:

```
res.statusCode = 200;
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download Node.js](#).

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use the `.mjs` version of the code, save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those two objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case, with:

```
res.statusCode = 200;
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next: [How much JavaScript do you need to know to use Node.js?](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use an `.mjs` version of the code, save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http](#) module.

Node.js has a fantastic [standard library](#), including support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200;
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking, and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database, or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers who write JavaScript for the browser can now also write server-side code in addition to client-side code without needing to learn a completely different language.

In Node.js, the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers — you are in charge of choosing which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common "Hello World" example of Node.js is a web server:

```
const { createServer } = require('node:http');  
  
const hostname = '127.0.0.1';
```

```
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use the `.mjs` version of the code, save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (`http.IncomingMessage`) and a response (`http.ServerResponse`).

Those two objects are essential to handle the HTTP call.

The first provides details about the request. In this simple example, it's not used, but you could access request headers and data.

The second is used to return data to the caller, as shown:

```
res.statusCode = 200;
```

```
res.setHeader('Content-Type', 'text/plain');
```

```
res.end('Hello World\n');
```

indicating a successful response with a 200 status, setting the content type, and ending the response with the message.

If you haven't already, [download Node.js](#).

Next

[How much JavaScript do you need to know to use Node.js?](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers — you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use an `.mjs` version of the code, you should save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js `http` module.

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an `http.IncomingMessage` object) and a response (an `http.ServerResponse` object).

Those two objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case, with:

```
res.statusCode = 200;
```

We set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

And we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js, the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers — you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal. If you use an `.mjs` version of the code, you should save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those two objects are essential to handle the HTTP call.

The request object provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The response object is used to return data to the caller:

```
res.statusCode = 200;
```

and set the `Content-Type` header:


```
res.setHeader('Content-Type', 'text/plain');
```

Then, we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already, [download](#) Node.js.

Next, learn more about [how much JavaScript you need to know to use Node.js](#).

Node.js — Introduction to Node.js

Navigation

- [Learn](#)
- [About](#)
- [Download](#)
- [Blog](#)
- [Docs](#)
- [Contribute](#)
- [Certification](#)

Change page

Introduction to Node.js

Getting Started

- [Introduction to Node.js](#)
- [How much JavaScript do you need to know to use Node.js?](#)
- [Differences between Node.js and the Browser](#)
- [The V8 JavaScript Engine](#)
- [An introduction to the npm package manager](#)
- [ECMAScript 2015 \(ES6\) and beyond](#)
- [Node.js, the difference between development and production](#)
- [Node.js with WebAssembly](#)
- [Debugging Node.js](#)
- [Profiling Node.js Applications](#)
- [Fetching data with Node.js](#)
- [WebSocket client with Node.js](#)
- [Security Best Practices](#)

TypeScript

- [Introduction to TypeScript](#)
- [Running TypeScript Natively](#)
- [Running TypeScript with a runner](#)
- [Running TypeScript code using transpilation](#)
- [Publishing a TypeScript package](#)

Asynchronous Work

- [Asynchronous flow control](#)
- [Overview of Blocking vs Non-Blocking](#)
- [JavaScript Asynchronous Programming and Callbacks](#)
- [Discover Promises in Node.js](#)
- [Discover JavaScript Timers](#)
- [The Node.js Event Loop](#)
- [The Node.js Event Emitter](#)
- [Understanding process.nextTick\(\)](#)
- [Understanding setImmediate\(\)](#)
- [Don't Block the Event Loop](#)

Manipulating Files

- [Node.js file stats](#)
- [Node.js File Paths](#)
- [Working with file descriptors in Node.js](#)
- [Reading files with Node.js](#)
- [Writing files with Node.js](#)
- [Working with folders in Node.js](#)
- [How to work with Different Filesystems](#)

Command Line

- [Run Node.js scripts from the command line](#)
- [How to read environment variables from Node.js](#)
- [How to use the Node.js REPL](#)
- [Output to the command line using Node.js](#)
- [Accept input from the command line in Node.js](#)

Userland Migrations

- [Introduction to Userland Migrations](#)

Modules

- [Publishing a package](#)
- [How to publish a Node-API package](#)
- [Anatomy of an HTTP Transaction](#)
- [ABI Stability](#)
- [How to use streams](#)
- [Backpressuring in Streams](#)

Diagnostics

- [User Journey](#)
- [Memory](#)
- [Live Debugging](#)
- [Poor Performance](#)
- [Flame Graphs](#)

Test Runner

- [Discovering Node.js's test runner](#)
- [Using Node.js's test runner](#)
- [Mocking in tests](#)
- [Collecting code coverage in Node.js](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and

wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use the `.mjs` version of the code, you should save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http](#) module.

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200;
res.setHeader('Content-Type', 'text/plain');
res.end('Hello World\n');
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Node.js — Introduction to Node.js



Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers — you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use the `.mjs` version of the code, save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those two objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and data.

The second is used to return data to the caller.

In this case, with:

```
res.statusCode = 200;
```

we set the `statusCode` property to `200`, indicating a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next, learn more about [How much JavaScript do you need to know to use Node.js?](#)

Node.js — Introduction to Node.js

Getting Started

- [Introduction to Node.js](#)
- [How much JavaScript do you need to know to use Node.js?](#)
- [Differences between Node.js and the Browser](#)
- [The V8 JavaScript Engine](#)
- [An introduction to the npm package manager](#)
- [ECMAScript 2015 \(ES6\) and beyond](#)
- [Node.js, the difference between development and production](#)
- [Node.js with WebAssembly](#)
- [Debugging Node.js](#)
- [Profiling Node.js Applications](#)
- [Fetching data with Node.js](#)
- [WebSocket client with Node.js](#)
- [Security Best Practices](#)

TypeScript

- [Introduction to TypeScript](#)
- [Running TypeScript Natively](#)
- [Running TypeScript with a runner](#)
- [Running TypeScript code using transpilation](#)
- [Publishing a TypeScript package](#)

Asynchronous Work

- [Asynchronous flow control](#)
- [Overview of Blocking vs Non-Blocking](#)
- [JavaScript Asynchronous Programming and Callbacks](#)
- [Discover Promises in Node.js](#)
- [Discover JavaScript Timers](#)
- [The Node.js Event Loop](#)

- [The Node.js Event Emitter](#)
- [Understanding process.nextTick\(\)](#)
- [Understanding setImmediate\(\)](#)
- [Don't Block the Event Loop](#)

Manipulating Files

- [Node.js file stats](#)
- [Node.js File Paths](#)
- [Working with file descriptors in Node.js](#)
- [Reading files with Node.js](#)
- [Writing files with Node.js](#)
- [Working with folders in Node.js](#)
- [How to work with Different Filesystems](#)

Command Line

- [Run Node.js scripts from the command line](#)
- [How to read environment variables from Node.js](#)
- [How to use the Node.js REPL](#)
- [Output to the command line using Node.js](#)
- [Accept input from the command line in Node.js](#)

Userland Migrations

- [Introduction to Userland Migrations](#)

Modules

- [Publishing a package](#)
- [How to publish a Node-API package](#)
- [Anatomy of an HTTP Transaction](#)
- [ABI Stability](#)
- [How to use streams](#)
- [Backpressuring in Streams](#)

Diagnostics

- [User Journey](#)
- [Memory](#)
- [Live Debugging](#)
- [Poor Performance](#)
- [Flame Graphs](#)

Test Runner

- [Discovering Node.js's test runner](#)
- [Using Node.js's test runner](#)
- [Mocking in tests](#)
- [Collecting code coverage in Node.js](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal. If you use mjs version of the code, you should save it as a `server.mjs` file and run `node server.mjs` in your terminal.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200;
```

We set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use the `.mjs` version of the code, save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http](#) module.

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#)) object and a response (an [http.ServerResponse](#)) object.

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200;
```


we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next

[How much JavaScript do you need to know to use Node.js?](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use the mjs version of the code, you should save it as a `server.mjs` file and run `node server.mjs` in your terminal.

This code first includes the Node.js [http](#) module.

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those two objects are essential to handle the HTTP call.

The request object provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The response object is used to return data to the caller.

In this case, with:

```
res.statusCode = 200;  
res.setHeader('Content-Type', 'text/plain');  
res.end('Hello World\n');
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next: [How much JavaScript do you need to know to use Node.js?](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use mjs version of the code, you should save it as a `server.mjs` file and run `node server.mjs` in your terminal.

This code first includes the Node.js [http](#) module.

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200;
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next, learn [How much JavaScript do you need to know to use Node.js?](#)

Node.js — Introduction to Node.js

 [Navigation Home](#)

[Learn](#) | [About](#) | [Download](#) | [Blog](#) |
[Docs](#) | [Contribute](#) | [Certification](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers — you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use the `.mjs` version of the code, save it as a `server.mjs` file and run `node server.mjs` in your terminal.

This code first includes the Node.js [http](#) module.

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (`http.IncomingMessage`) and a response (`http.ServerResponse`).

Those two objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case, with:

```
res.statusCode = 200;
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next Steps

[How much JavaScript do you need to know to use Node.js?](#)

Reading Time

3 min

Authors

+10 others

Contribute

[Edit this page](#)

Table of Contents

1. [An Example Node.js Application](#)
-

Navigation

[Navigate to Home](#) | [Getting Started](#) | [Introduction to Node.js](#)

Footer

[Trademark Policy](#)

[Privacy Policy](#)

[Version Support](#)

[Code of Conduct](#)

[Security Policy](#)

© [OpenJS Foundation](#)

[GitHub](#) | [Discord](#) | [LFX Social](#) | [Bluesky](#) | [Twitter](#) | [Slack Invite](#) | [LinkedIn](#)

Introduction to Node.js

Node.js is an *open-source* and *cross-platform* JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the *V8 JavaScript engine*, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a *unique advantage* because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example **Hello World** of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal. If you use an `.mjs` version of the code, you should save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those two objects are essential to handle the HTTP call.

The request object provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The response object is used to return data to the caller.

In this case, we set the `statusCode` property to `200`, to indicate a successful response:

```
res.statusCode = 200;
```

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

And we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next, learn [How much JavaScript do you need to know to use Node.js?](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the `V8` JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js, the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal. If you use the `mjs` version of the code, you should save it as a `server.mjs` file and run `node server.mjs` in your terminal.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200;
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next: [How much JavaScript do you need to know to use Node.js?](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers who write JavaScript for the browser can now write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js, the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers—you control the ECMAScript version by changing the Node.js version, and you can enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common "Hello World" example of Node.js is a web server:

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;
```

```
const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use an `.mjs` version of the code, save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When it's ready, the callback function is invoked, indicating the server is running.

Whenever a new request is received, the [request event](#) is triggered, providing two objects: the request (`http.IncomingMessage`) and the response (`http.ServerResponse`).

These objects are essential for handling the HTTP call.

The first contains request details; in this simple example, it is not used, but you can access headers and request data.

The second is used to send data back to the client.

For example:

```
res.statusCode = 200;
res.setHeader('Content-Type', 'text/plain');
res.end('Hello World\n');
```

Set the status code to `200`, indicate the response is plain text via headers, and close the response with `end()`, passing the content.

If you haven't already, [download Node.js](#).

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal. If you use mjs version of the code, you should save it as a `server.mjs` file and run `node server.mjs` in your terminal.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200;
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next: [How much JavaScript do you need to know to use Node.js?](#)

Reading Time: 3 min

Authors:

[flaviocopes](#) | [potch](#) | [mylesborins](#) | [RomainLanz](#) | [virkt25](#) | [Trott](#) | [onel0p3z](#) |
[OlleLauribostrom](#) | [M](#) +10

Contribute: [Edit this page](#)

Table of Contents

1. [An Example Node.js Application](#)
-

Navigate to Home: [Getting Started](#), [Introduction to Node.js](#)

Node.js — Introduction to Node.js

Navigation

- [Learn](#)
 - [About](#)
 - [Download](#)
 - [Blog](#)
 - [Docs](#)
 - [Contribute](#)
 - [Certification](#)
-

Change page

Introduction to Node.js

Getting Started

- [Introduction to Node.js](#)
- [How much JavaScript do you need to know to use Node.js?](#)
- [Differences between Node.js and the Browser](#)
- [The V8 JavaScript Engine](#)
- [An introduction to the npm package manager](#)
- [ECMAScript 2015 \(ES6\) and beyond](#)
- [Node.js, the difference between development and production](#)
- [Node.js with WebAssembly](#)
- [Debugging Node.js](#)
- [Profiling Node.js Applications](#)
- [Fetching data with Node.js](#)
- [WebSocket client with Node.js](#)
- [Security Best Practices](#)

TypeScript

- [Introduction to TypeScript](#)
- [Running TypeScript Natively](#)
- [Running TypeScript with a runner](#)
- [Running TypeScript code using transpilation](#)
- [Publishing a TypeScript package](#)

Asynchronous Work

- [Asynchronous flow control](#)
- [Overview of Blocking vs Non-Blocking](#)
- [JavaScript Asynchronous Programming and Callbacks](#)
- [Discover Promises in Node.js](#)
- [Discover JavaScript Timers](#)
- [The Node.js Event Loop](#)
- [The Node.js Event Emitter](#)
- [Understanding process.nextTick\(\)](#)
- [Understanding setImmediate\(\)](#)
- [Don't Block the Event Loop](#)

Manipulating Files

- [Node.js file stats](#)
- [Node.js File Paths](#)
- [Working with file descriptors in Node.js](#)
- [Reading files with Node.js](#)
- [Writing files with Node.js](#)
- [Working with folders in Node.js](#)
- [How to work with Different Filesystems](#)

Command Line

- [Run Node.js scripts from the command line](#)
- [How to read environment variables from Node.js](#)
- [How to use the Node.js REPL](#)
- [Output to the command line using Node.js](#)
- [Accept input from the command line in Node.js](#)

Userland Migrations

- [Introduction to Userland Migrations](#)

Modules

- [Publishing a package](#)
- [How to publish a Node-API package](#)
- [Anatomy of an HTTP Transaction](#)
- [ABI Stability](#)
- [How to use streams](#)
- [Backpressuring in Streams](#)

Diagnostics

- [User Journey](#)
- [Memory](#)
- [Live Debugging](#)
- [Poor Performance](#)
- [Flame Graphs](#)

Test Runner

- [Discovering Node.js's test runner](#)
- [Using Node.js's test runner](#)
- [Mocking in tests](#)
- [Collecting code coverage in Node.js](#)

Introduction

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking, and libraries are generally written using non-blocking paradigms.

When Node.js performs an I/O operation like reading from the network, database, or filesystem, it resumes operations upon response rather than blocking.

This enables handling thousands of concurrent connections efficiently without managing thread concurrency.

Many frontend developers who write JavaScript can now also write server-side code in Node.js without learning a different language.

In addition, the latest ECMAScript standards can be used in Node.js, with control over the version and experimental features via runtime flags.

An Example Node.js Application

The classic Hello World example is a simple web server:

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run, save as `server.js` and execute:

```
node server.js
```

If using an `.mjs` file, save as `server.mjs` and run:

```
node server.mjs
```

This code requires the Node.js `http` module, creates a server that responds with 'Hello World' to incoming requests, and listens on the specified port and

hostname.

The server callback handles incoming requests using request and response objects; the response object is used to send data back to the client.

Ensure Node.js is installed from [here](#).

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js, the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers — you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```

```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and execute `node server.js` in your terminal.

If you use the `.mjs` version of the code, save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http module](#).

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is triggered, providing two objects: a request (`http.IncomingMessage`) and a response (`http.ServerResponse`).

Those two objects are essential for handling the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access request headers and data.

The second is used to return data to the caller.

In this case, with:

```
res.statusCode = 200;
```

We set the `statusCode` property to `200`, indicating a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

And then close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already, [download Node.js](#).

For more details, see the ["How much JavaScript do you need to know to use Node.js?"](#) guide.

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers — you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');
```



```
const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal. If you use the `mjs` version of the code, you should save it as a `server.mjs` file and run `node server.mjs` in your terminal.

This code first includes the Node.js [http](#) module.

Node.js has a fantastic [standard library](#), including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those two objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200;
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Node.js — Introduction to Node.js

[Link to Home](#)

Navigation Menu

- [Learn](#)
- [About](#)
- [Download](#)
- [Blog](#)
- [Docs](#)
- [Contribute](#)
- [Certification](#)

Change Page

Introduction to Node.js

Getting Started

- [Introduction to Node.js](#)
- [How much JavaScript do you need to know to use Node.js?](#)
- [Differences between Node.js and the Browser](#)
- [The V8 JavaScript Engine](#)
- [An introduction to the npm package manager](#)
- [ECMAScript 2015 \(ES6\) and beyond](#)
- [Node.js, the difference between development and production](#)
- [Node.js with WebAssembly](#)
- [Debugging Node.js](#)
- [Profiling Node.js Applications](#)
- [Fetching data with Node.js](#)
- [WebSocket client with Node.js](#)
- [Security Best Practices](#)

TypeScript

- [Introduction to TypeScript](#)
- [Running TypeScript Natively](#)
- [Running TypeScript with a runner](#)
- [Running TypeScript code using transpilation](#)
- [Publishing a TypeScript package](#)

Asynchronous Work

- [Asynchronous flow control](#)
- [Overview of Blocking vs Non-Blocking](#)
- [JavaScript Asynchronous Programming and Callbacks](#)
- [Discover Promises in Node.js](#)
- [Discover JavaScript Timers](#)
- [The Node.js Event Loop](#)
- [The Node.js Event Emitter](#)
- [Understanding process.nextTick\(\)](#)
- [Understanding setImmediate\(\)](#)
- [Don't Block the Event Loop](#)

Manipulating Files

- [Node.js file stats](#)
- [Node.js File Paths](#)
- [Working with file descriptors in Node.js](#)
- [Reading files with Node.js](#)
- [Writing files with Node.js](#)
- [Working with folders in Node.js](#)
- [How to work with Different Filesystems](#)

Command Line

- [Run Node.js scripts from the command line](#)
- [How to read environment variables from Node.js](#)
- [How to use the Node.js REPL](#)
- [Output to the command line using Node.js](#)
- [Accept input from the command line in Node.js](#)

Userland Migrations

- [Introduction to Userland Migrations](#)

Modules

- [Publishing a package](#)
- [How to publish a Node-API package](#)
- [Anatomy of an HTTP Transaction](#)
- [ABI Stability](#)
- [How to use streams](#)
- [Backpressuring in Streams](#)

Diagnostics

- [User Journey](#)
- [Memory](#)
- [Live Debugging](#)
- [Poor Performance](#)
- [Flame Graphs](#)

Test Runner

- [Discovering Node.js's test runner](#)
- [Using Node.js's test runner](#)
- [Mocking in tests](#)
- [Collecting code coverage in Node.js](#)

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking, and libraries in Node.js are generally written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database, or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers who write JavaScript for the browser can now also write server-side code without learning a completely different language.

In Node.js, the new ECMAScript standards can be used without issues because you don't need to wait for all users to update their browsers — you control which ECMAScript version to use by changing the Node.js version, and you can enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common "Hello World" example in Node.js is a web server:

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as `server.js` and execute `node server.js` in your terminal. If using the `.mjs` version, save it as `server.mjs` and run `node server.mjs`.

This code first includes the Node.js [http module](#).

Node.js has a comprehensive [standard library](#), including robust networking support.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host. When ready, it calls the callback, indicating the server is running.

When a request arrives, the `request` event triggers, providing `request` (an `http.IncomingMessage`) and `response` (an `http.ServerResponse`) objects.

The `response` object is used to send data back to the client. For example:

```
res.statusCode = 200;
res.setHeader('Content-Type', 'text/plain');
res.end('Hello World\n');
```

Make sure you have Node.js installed by downloading it from [here](#).

Next Steps

[How much JavaScript do you need to know to use Node.js?](#)

Reading Time: 3 min

Authors:

[flaviocopes](#), [potch](#), [mylesborins](#), [RomainLanz](#), [virkt25](#), [Trott](#), [onel0p3z](#), [ollelauribostrom](#), [M](#), and +10 more.

Contribute:

[Edit this page](#)

Navigation

- [Navigate to Home](#)
- [Getting Started](#)
- [Introduction to Node.js](#)

Node.js — Introduction to Node.js

[Navigation & Menu](#)

[Change page](#)

Introduction to Node.js

Getting Started

- [Introduction to Node.js](#)
- [How much JavaScript do you need to know to use Node.js?](#)
- [Differences between Node.js and the Browser](#)
- [The V8 JavaScript Engine](#)
- [An introduction to the npm package manager](#)
- [ECMAScript 2015 \(ES6\) and beyond](#)
- [Node.js, the difference between development and production](#)
- [Node.js with WebAssembly](#)
- [Debugging Node.js](#)
- [Profiling Node.js Applications](#)
- [Fetching data with Node.js](#)
- [WebSocket client with Node.js](#)
- [Security Best Practices](#)

TypeScript

- [Introduction to TypeScript](#)
- [Running TypeScript Natively](#)
- [Running TypeScript with a runner](#)
- [Running TypeScript code using transpilation](#)
- [Publishing a TypeScript package](#)

Asynchronous Work

- [Asynchronous flow control](#)
- [Overview of Blocking vs Non-Blocking](#)
- [JavaScript Asynchronous Programming and Callbacks](#)

- [Discover Promises in Node.js](#)
- [Discover JavaScript Timers](#)
- [The Node.js Event Loop](#)
- [The Node.js Event Emitter](#)
- [Understanding process.nextTick\(\)](#)
- [Understanding setImmediate\(\)](#)
- [Don't Block the Event Loop](#)

Manipulating Files

- [Node.js file stats](#)
- [Node.js File Paths](#)
- [Working with file descriptors in Node.js](#)
- [Reading files with Node.js](#)
- [Writing files with Node.js](#)
- [Working with folders in Node.js](#)
- [How to work with Different Filesystems](#)

Command Line

- [Run Node.js scripts from the command line](#)
- [How to read environment variables from Node.js](#)
- [How to use the Node.js REPL](#)
- [Output to the command line using Node.js](#)
- [Accept input from the command line in Node.js](#)

Userland Migrations

- [Introduction to Userland Migrations](#)

Modules

- [Publishing a package](#)
- [How to publish a Node-API package](#)
- [Anatomy of an HTTP Transaction](#)
- [ABI Stability](#)
- [How to use streams](#)
- [Backpressuring in Streams](#)

Diagnostics

- [User Journey](#)
- [Memory](#)
- [Live Debugging](#)
- [Poor Performance](#)
- [Flame Graphs](#)

Test Runner

- [Discovering Node.js's test runner](#)
 - [Using Node.js's test runner](#)
 - [Mocking in tests](#)
 - [Collecting code coverage in Node.js](#)
-

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking, and libraries in Node.js are generally written with non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database, or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js resumes the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers writing JavaScript for the browser can now also write server-side code without learning

a different language.

In Node.js, the new ECMAScript standards can be used freely, as you don't have to wait for all users to update their browsers — you control the ECMAScript version via the Node.js version and can enable specific experimental features with flags.

An Example Node.js Application

The most common "Hello World" example for Node.js is a web server:

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this code, save it as `server.js` and execute `node server.js` in the terminal. For the `.mjs` version, save as `server.mjs` and run `node server.mjs`.

This code includes Node.js's `http` module, which provides networking support.

`createServer()` creates an HTTP server that is set to listen on the specified port and hostname. When the server is ready, the callback outputs a confirmation message.

Each incoming request triggers the `request` event, providing a request object and a response object. These are used to handle the HTTP call: inspecting request details or sending a response.

In this example, the response status code is set to 200 (success), the Content-Type header to plain text, and the response is closed with "Hello World".

Reading Time

3 min

Authors

- [flaviocopes](#)
- [potch](#)
- [mylesborins](#)
- [RomainLanz](#)
- [virkt25](#)
- [Trott](#)
- [onel0p3z](#)
- [olleauribostrom](#)
- [MarkPieszak](#)
- +10 more

Contribute

[Edit this page](#)

Navigation

- [Navigate to Home](#)
 - [Getting Started](#)
 - [Introduction to Node.js](#)
-

Footer

- [Trademark Policy](#)
- [Privacy Policy](#)
- [Version Support](#)

- [Code of Conduct](#)
- [Security Policy](#)

© [OpenJS Foundation](#)

[Discord](#)

[Social](#)

[Bsky](#)

[Twitter](#)

[Slack Invite](#)

[LinkedIn](#)

Node.js — Introduction to Node.js

 [Back to Home](#)

Getting Started

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version,

and you can also enable specific experimental features by running Node.js with flags.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

If you use an `.mjs` version of the code, save it as a `server.mjs` file and run `node server.mjs` in your terminal.

This code first includes the Node.js [http](#) module.

Node.js has a fantastic [standard library](#), including support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects: a request (an [http.IncomingMessage](#) object) and a response (an [http.ServerResponse](#) object).

Those two objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case, with:

```
res.statusCode = 200;
```

we set the `statusCode` property to `200`, to indicate a successful response.

We set the `Content-Type` header:

```
res.setHeader('Content-Type', 'text/plain');
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n');
```

If you haven't already done so, [download](#) Node.js.

Next

[How much JavaScript do you need to know to use Node.js?](#)

Reading Time: 3 min

Authors: [F](#), [P](#), [MB](#), [R](#), [V](#), [T](#), [O](#), [O](#), [M](#) +10

Contribute: [Edit this page](#)

Table of Contents

1. [An Example Node.js Application](#)
-

Navigate to:

[Home](#) > [Getting Started](#) > [Introduction to Node.js](#)