




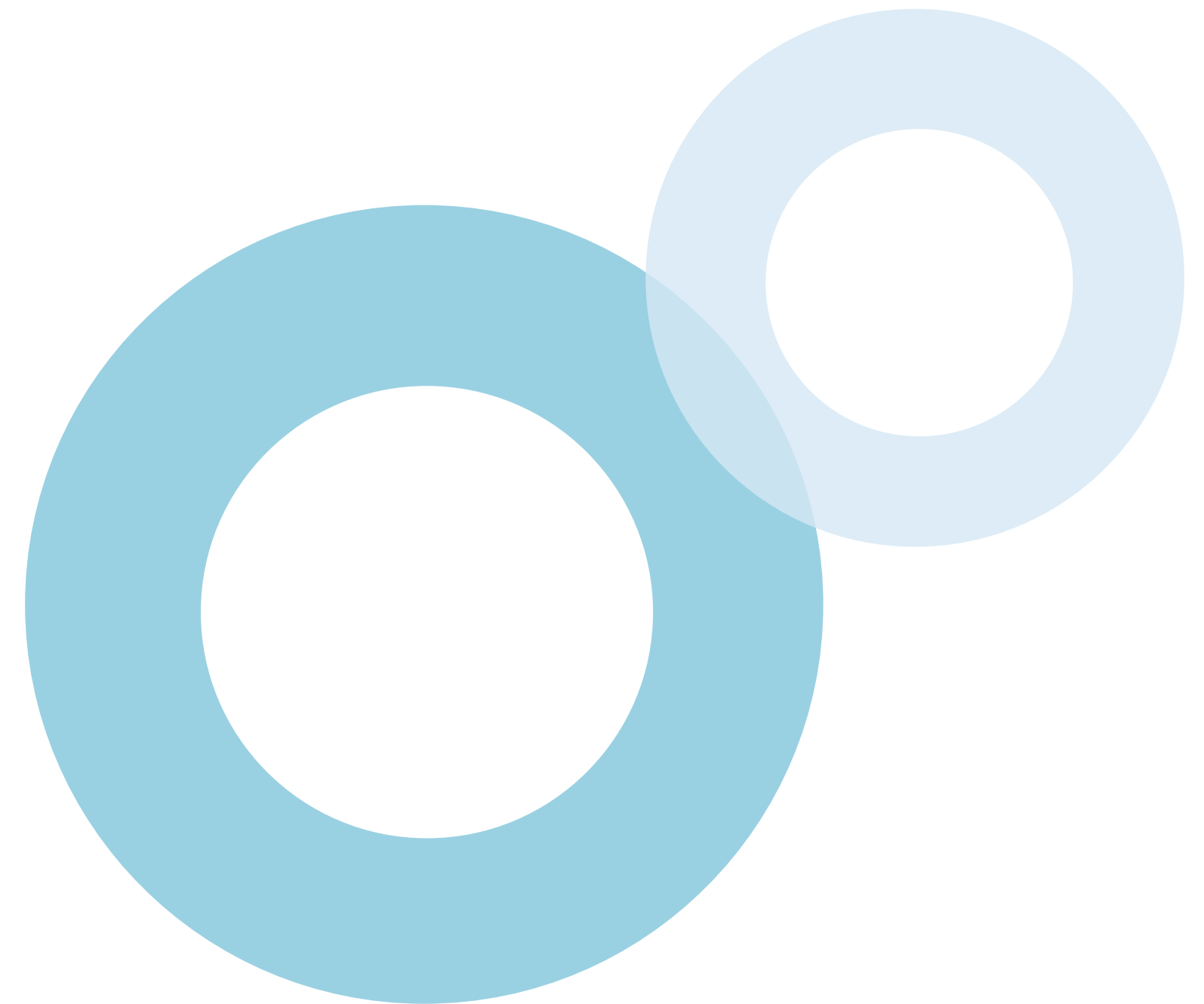

After studying this topic, you should be able to:

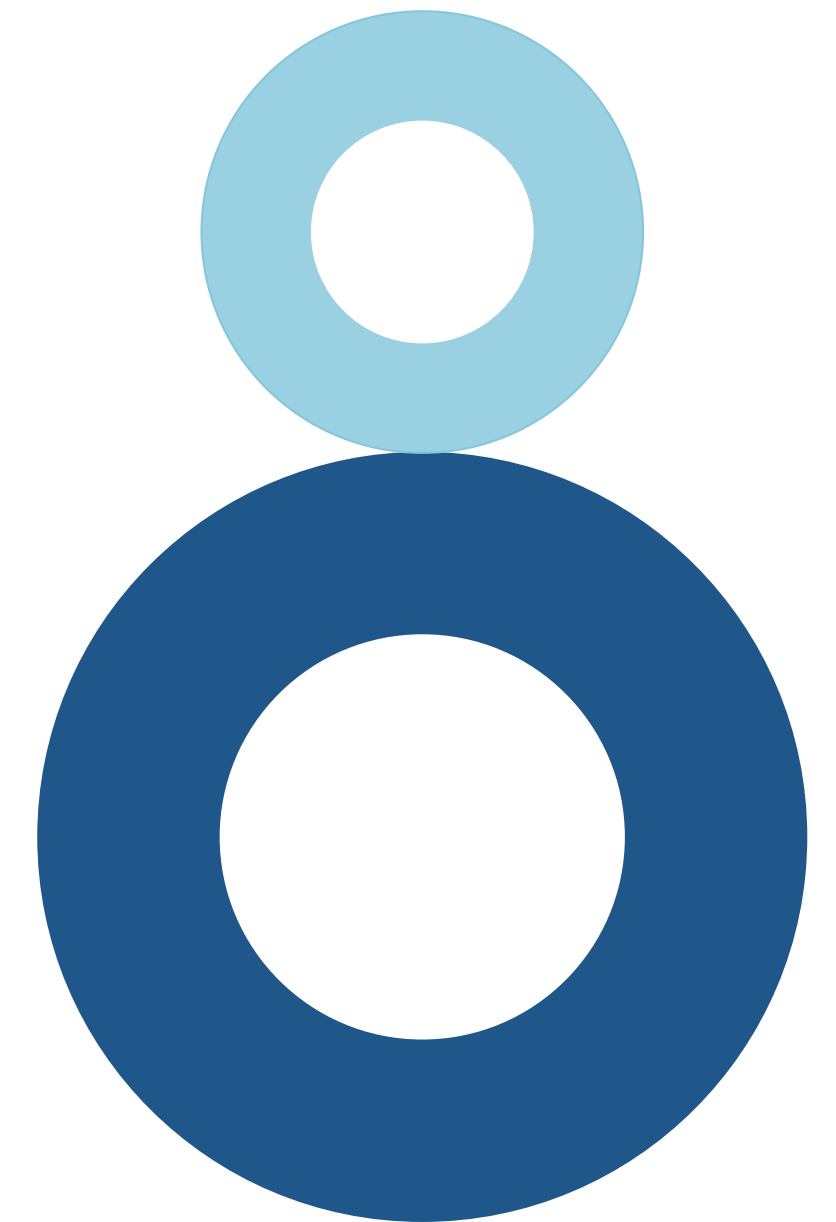
-  Identify the main features and use cases of Node.js.
-  Describe how to build an HTTP server and handle incoming HTTP requests in Node.js.
-  Describe how to make outbound HTTP requests in Node.js.
-  Describe how to use other Node.js modules, such as 'fs' and 'events'.
-  Explain how to install an npm package to meet a particular Node.js requirement.



Introduction

Node.js is a back-end JavaScript runtime environment that can be used to execute JavaScript code outside a browser. A single **non-blocking** process is utilized to run a Node.js app. Node.js can be used to generate **dynamic web page content**, build **real-time** or **single-page apps**, and also develop **APIs** and **microservices**.

After installing Node.js, a web server can be built by using the **'http'** module. It is also possible to use it to perform outbound HTTP requests, such as GET and POST. The **'fs'** module allows working with the file system. The **'events'** module can be used to emit and handle events in Node.js. **Node package manager (npm)** can be used to install open-source packages that are available in the npm registry. They can be used for specific requirements that cannot be met by using the core Node.js modules.



Overview

Node.js Features

Node.js is a server-side JavaScript runtime environment that utilizes a **non-blocking** process to run apps. It uses the **ECMAScript** standards.

Use Cases

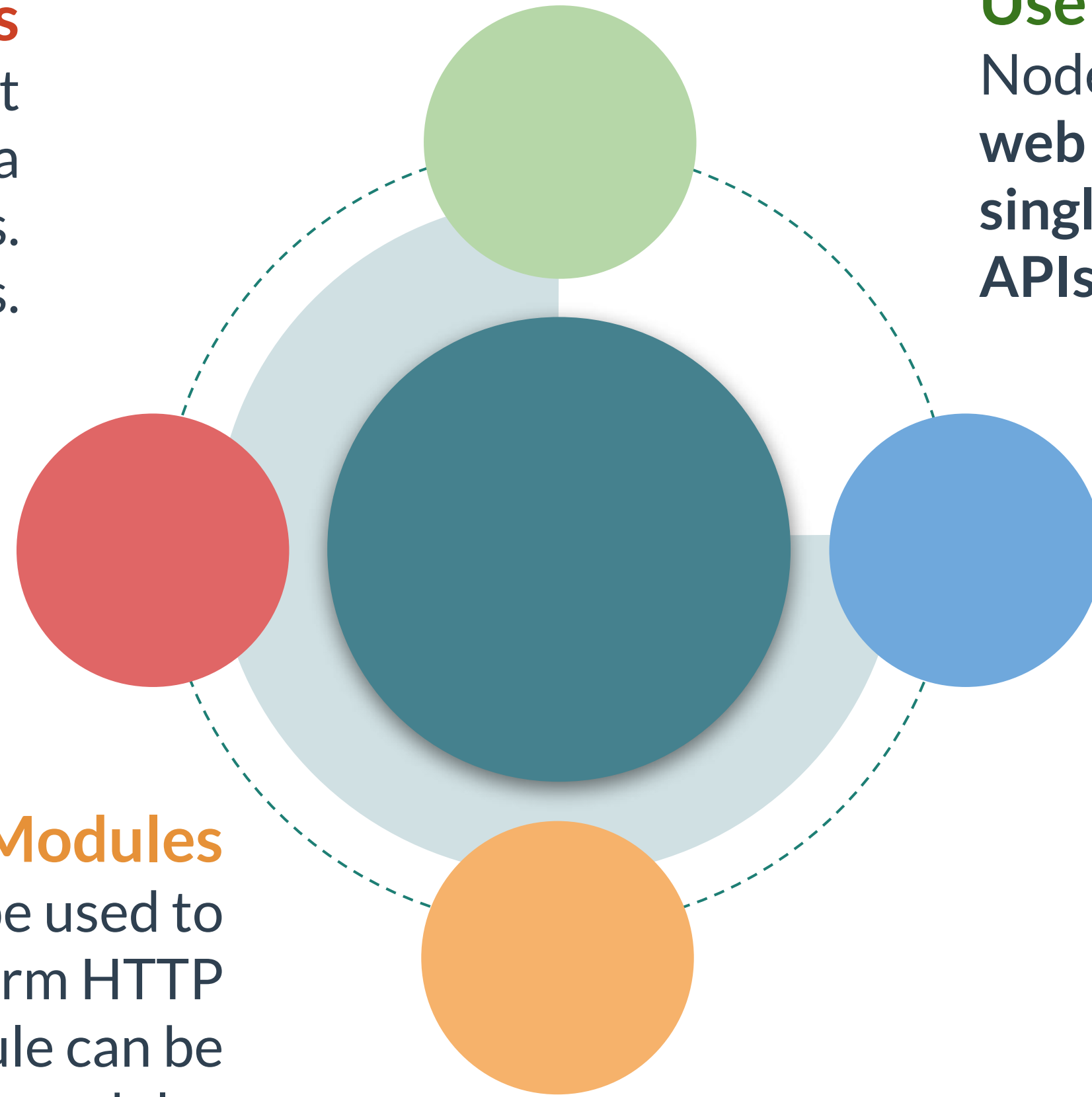
Node.js can be used generate **dynamic web page content**, build **real-time** or **single-page apps**, and also develop **APIs** and **microservices**.

Core Modules

The **'http'** module can be used to create a server and perform HTTP requests. The **'fs'** module can be used work with files. Other modules such as **'events'** are also available.

NPM

Node package manager (npm) is the package manager for Node.js. It can be used to install open-source packages available in the npm registry.



Node.js

Node.js is a **back-end JavaScript runtime environment** that allows executing JavaScript code outside a web browser. It can be used to write server-side code. It uses **event-driven architecture** that allows **asynchronous I/O**.

```
// This example shows server-side code written using Node.js.
```

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

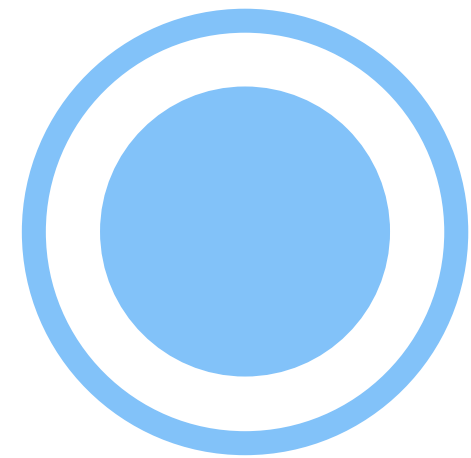
```
const port = 3000;
```

```
const server = http.createServer((req, res) => {  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Welcome to the Node.js app!');  
});
```

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

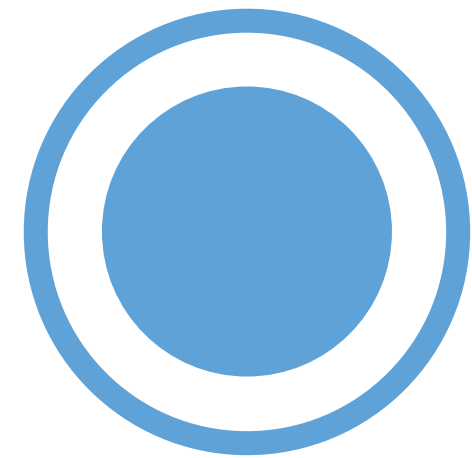
Node.js Features

Node.js offers the following main features:



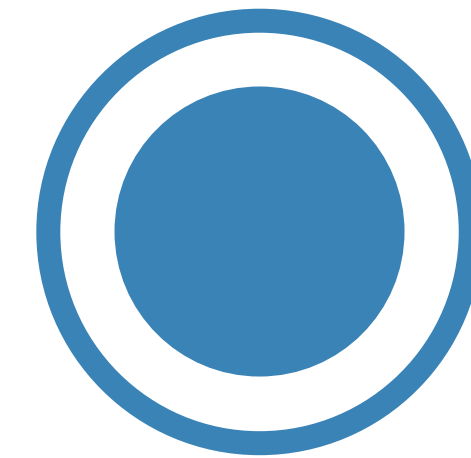
V8 ENGINE

The **V8 JavaScript engine** is used to run Node.js apps outside the browser. It is also the core of the Chrome web browser.



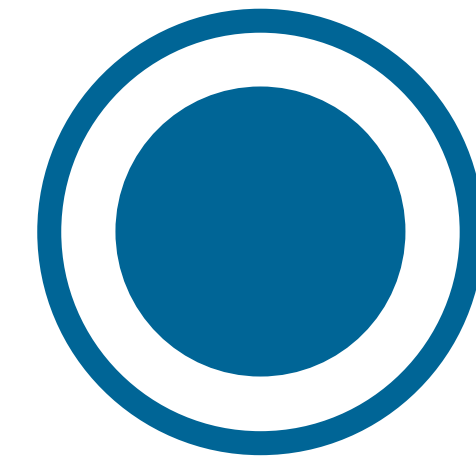
NON-BLOCKING

A **single non-blocking process** is utilized to run a Node.js app. A new thread is not created for every request.



ECMAScript

Node.js supports the new **ECMAScript** standards. Developers can use them for both client-side and server-side code.

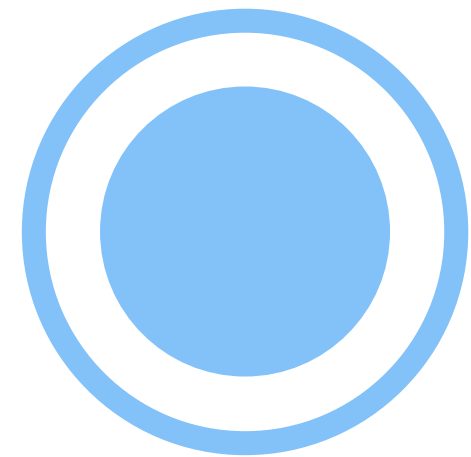


NPM

The **npm** registry hosts open-source packages, which can be used to build Node.js apps.

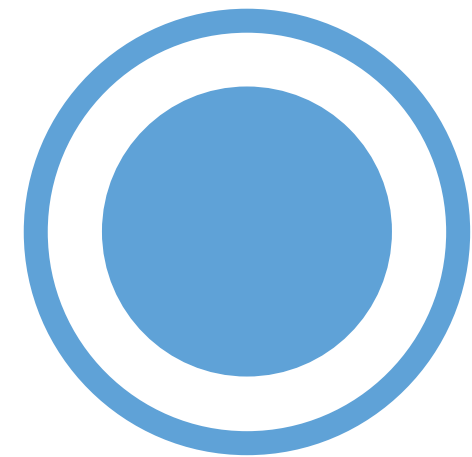
Use Cases of Node.js

Some common use cases of Node.js are as follows:



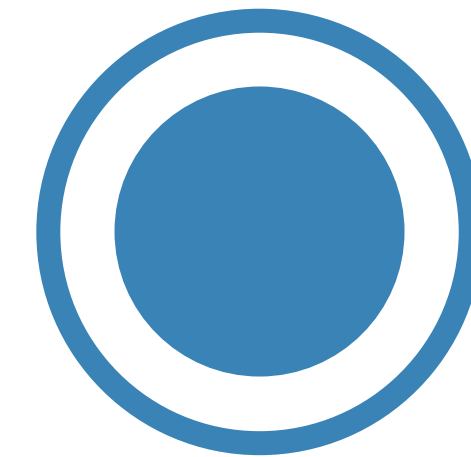
DYNAMIC CONTENT

Node.js can be used to produce **dynamic web page content** by running server-side scripts.



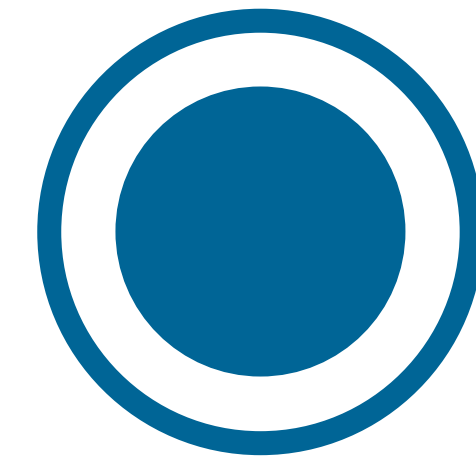
REAL-TIME APPS

Real-time web applications, such as **chat apps**, can be built using Node.js due to its ability to handle **multiple I/O requests concurrently**.



SINGLE-PAGE APPS

Node.js can be used for **complex, single-page apps (SPAs)** that require efficient handling of **asynchronous calls** and **heavy I/O operations**.



API

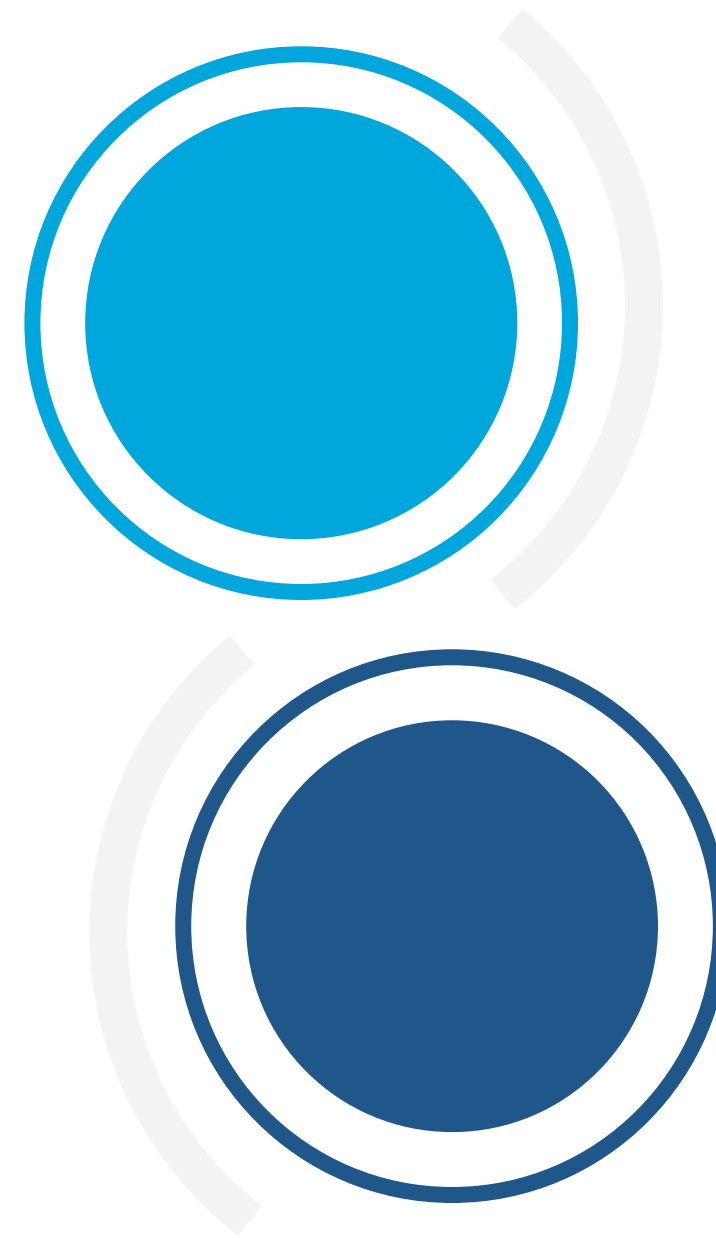
Node.js is often used for developing **RESTful APIs** and independent **microservices** for web applications.

Installing Node.js

Node.js can be installed by downloading the **official installer** for Windows or macOS from **nodejs.org**.

Versions

Official installers are available for the **LTS (Long Term Support)** version and the **Current** version. The LTS version is recommended for most users.



Package Manager

Node.js can also be installed using a package manager. **Chocolatey** can be used for Windows. **Homebrew** can be used for macOS.

Building a Web Server

Once Node.js has been installed, a **JavaScript file** can be created to build a **web server**. The **http** module can be used to create a server. The example below shows the definition of a web server in a file named **app.js**.

```
// This example shows how to build a web server using Node.js.
//Include the 'http' module by using the built-in 'require' function
const http = require('http');
const port = 3000;

//Use http.createServer() to create a new instance of http.Server
const server = http.createServer((req, res) => {
  //Define the callback function that handles incoming requests
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Welcome!');
});

// Use server.listen() to start the server listening for connections.
server.listen(port, () => {
  console.log(`Server running at port ${port}`);
});
```


Handling Incoming HTTP Requests

Routing can be defined in a Node.js web server to handle incoming **HTTP(S) requests**, such as **GET, POST, PATCH**, etc. It can be implemented using a framework or without it.

```
// This example shows how to handle incoming HTTP requests in a Node.js app. (Part 1 of 2)

const http = require('http');
const port = 3000;
const server = http.createServer( (req, res) => {
  // Define a callback function to handle incoming GET requests to '/users' endpoint.
  if(req.url === '/users' && req.method === 'GET') {
    // Get the data, typically from a database.
    const data = getUsers();
    // Set the status code of the response.
    res.statusCode = 200;
    // Set the response header.
    res.setHeader('Content-Type', 'text/plain');
    // Call the required end() method, passing the data.
    res.end(data);
  }
});
```

Continued...

Handling Incoming HTTP Requests

Routing can be defined in a Node.js web server to handle incoming **HTTP(S) requests**, such as **GET, POST, PATCH**, etc. It can be implemented using a framework or without it.

```
// This example shows how to handle incoming HTTP requests in a Node.js app. (Part 2 of 2)

// A function that returns sample data.
function getUsers() {
  return 'Ashley, John, and Olivia';
}

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

Output

← → ↻ ⓘ localhost:3000/users

Ashley, John, and Olivia

Making Outbound HTTP Requests

Node.js can also be used to perform **outbound HTTP requests**. For example, a **GET request** can be perform to **retrieve data** from an external web server.

```
// This example shows how to perform an outbound GET request in a Node.js app. (Part 1 of 2)

const http = require('http');

const options = {
  hostname: 'jsonplaceholder.typicode.com',
  path: '/todos/1',
  method: 'GET',
  headers: {
    'Content-Type': 'application/json'
  }
};
```

Continued...

Making Outbound HTTP Requests

Node.js can also be used to perform **outbound HTTP requests**. For example, a **GET request** can be perform to **retrieve data** from an external web server.

```
// This example shows how to perform an outbound GET request in a Node.js app. (Part 2 of 2)
```

```
const req = http.request(options, res => {  
  console.log(`statusCode: ${res.statusCode}`);
```

```
  
  res.on('data', d => {  
    process.stdout.write(d);  
  })  
})
```

```
req.on('error', error => {  
  console.error(error);  
})
```

```
req.end();
```

Output

```
statusCode: 200  
{  
  "userId": 1,  
  "id": 1,  
  "title": "delectus aut autem",  
  "completed": false  
}
```

Working with Files

The **fs** module allows accessing and interacting with the **file system** on the computer. For example, one can **read**, **copy** and **write** to files. Such operations can be performed **synchronously** (which blocks code execution) or **asynchronously**.

```
// This example shows how to read and write to files in Node.js asynchronously. (Part 1 of 2)

const fs = require('fs');

const content = '{"name": "John", "salary": 75000, "position": "HR Specialist"}';

// The readFile() method is a non-blocking method that reads a file asynchronously.
fs.readFile('./user.json' , 'utf-8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  // Output the data returned from the file.
  console.log(data);
});
```

Continued...

Working with Files

The **fs** module allows accessing and interacting with the **file system** on the computer. For example, one can **read**, **copy** and **write** to files. Such operations can be performed **synchronously** (which blocks code execution) or **asynchronously**.

```
// This example shows how to read and write to files in Node.js asynchronously. (Part 2 of 2)

// Replace the content of the file with the value of 'content' using the writeFile() asynchronous method.
fs.writeFile('./user.json', content, err => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('Content Replaced!');
});
```

Output

```
{"name": "John", "salary": 60000, "position": "HR Specialist"}
Content Replaced!
```

```
{ } user.json ×
```

```
{ } user.json > ...
```

```
1  {"name": "John", "salary": 75000, "position": "HR Specialist"}
```

Working with Files

The **fs** module allows accessing and interacting with the **file system** on the computer. For example, one can **read**, **copy** and **write** to files. Such operations can be performed **synchronously** (which blocks code execution) or **asynchronously**.

```
// This example shows how to read and write to files in Node.js synchronously. (Part 1 of 2)

const fs = require('fs');

const content = '{"name": "Ashley", "salary": 90000, "position": "HR Specialist"}';

// The readFileSync() method is a blocking method that reads a file synchronously.
let data;
try {
  data = fs.readFileSync('./user.json', 'utf-8');
} catch(error) {
  console.error(error);
}

// Output the data returned from the file.
console.log(data);
```

Continued...

Working with Files

The **fs** module allows accessing and interacting with the **file system** on the computer. For example, one can **read**, **copy** and **write** to files. Such operations can be performed **synchronously** (which blocks code execution) or **asynchronously**.

```
// This example shows how to read and write to files in Node.js synchronously. (Part 2 of 2)

// The writeFileSync() method is a blocking method that writes to a file synchronously.
try {
  fs.writeFileSync('./user.json', content);
  console.log('Content Replaced!');
} catch(error) {
  console.error(error);
}
```

Output

```
{"name": "Ashley", "salary": 80000, "position": "HR Specialist"}
Content Replaced!
```

```
{ } user.json ×
```

```
{ } user.json > ...
```

```
1 {"name": "Ashley", "salary": 90000, "position": "HR Specialist"}
```

Working with Events

The **events** module can be used to work with events in Node.js. An **EventEmitter** object can be created to **emit** events, **add an event listener**, **remove an event listener**, etc.

```
// This example shows how to emit and handle events in Node.js.
```

```
const EventEmitter = require('events');
```

```
const object = new EventEmitter();
```

```
// Add an event listener.
```

```
object.addListener('spark', () => {  
    console.log(`Let there be light!`);  
});
```

```
// Emit the event.
```

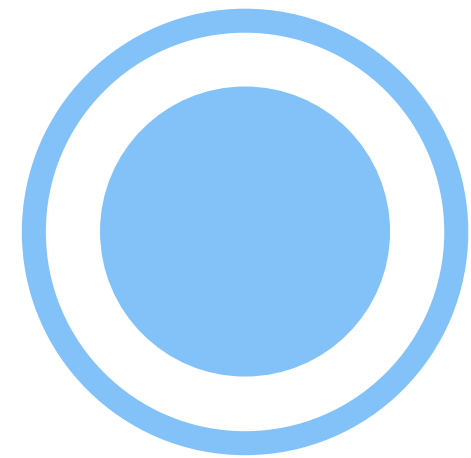
```
console.log('Emitting the event...');  
object.emit('spark');
```

Output

```
Emitting the event...  
Let there be light!
```

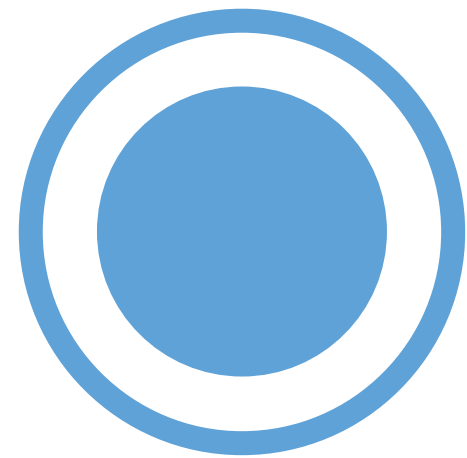
Global Object

Node.js has the **global** object instead of the **window** object. It has many useful methods and properties which can be utilized without using `require()`.



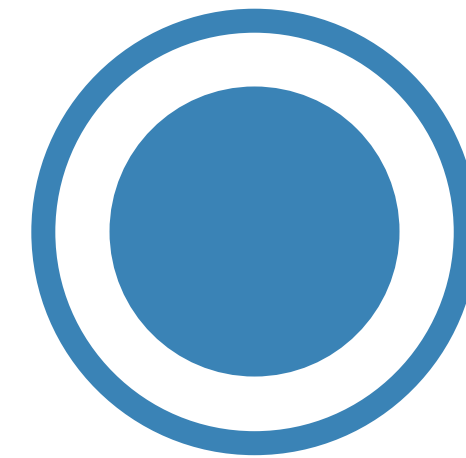
console

The **console** object provides methods such as **console.log()** which can be used to write to standard output of any Node.js stream.



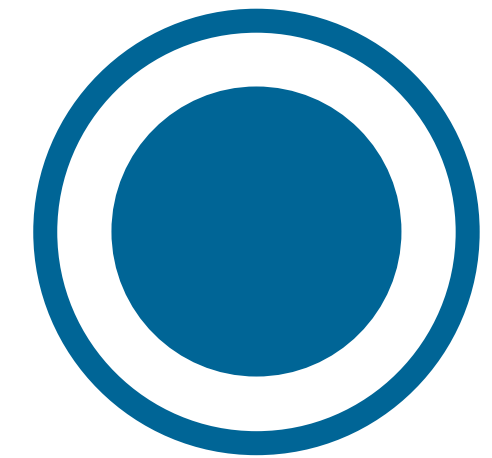
process

The **process** object allows getting information about and controlling the current Node.js process. For example, **process.env** can be used to access environment variables.



Timers

Methods such as **setImmediate()**, **setInterval()** and **setTimeout()** can be used to call a function at a specific time.



WebAssembly

The **WebAssembly** object provides WebAssembly related functionality to work with low-level code. It is used to run code written in multiple languages on the web.

Global Variables

The **global** object stores **global variables**. A **global variable** can be set **explicitly** or **implicitly**. Using the **var**, **let** or **const** keyword sets a variable's scope to the **module**.

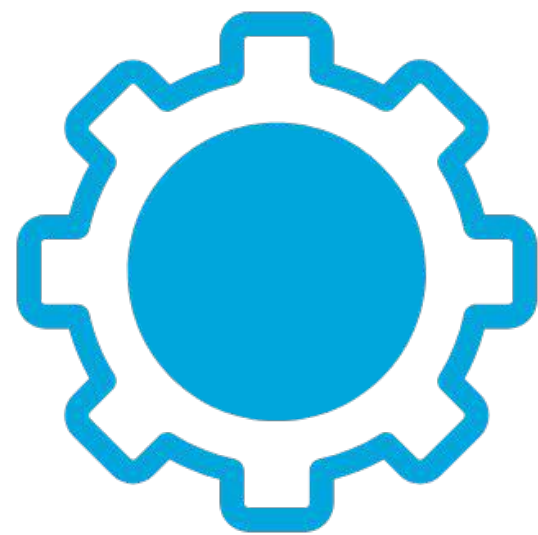
```
firstName = 'Joe'; // A global variable can be set implicitly.
global.lastName = 'Smith'; // A global variable can be set explicitly using 'global'.
// When the var, let or const keyword is used, the variable is scoped to the module.
var position = 'IT Director';
const company = 'Cosmic Solutions';
let salary = 200000;
console.log(`Global Name: ${global.firstName} ${global.lastName}`);
// Trying to access a non-global variable using 'global' returns undefined.
console.log(`Global Position: ${global.position}`);
console.log(`Global Company: ${global.company}`);
console.log(`Global Salary: ${global.salary}`);
console.log('-----');
// A global variable can be accessed without using 'global'.
console.log(`Global Name: ${firstName} ${lastName}`);
console.log(`Module Position: ${position}`);
console.log(`Module Company: ${company}`);
console.log(`Module Salary: ${salary}`);
```

Output

```
Global Name: Joe Smith
Global Position: undefined
Global Company: undefined
Global Salary: undefined
-----
Global Name: Joe Smith
Module Position: IT Director
Module Company: Cosmic Solutions
Module Salary: 200000
```

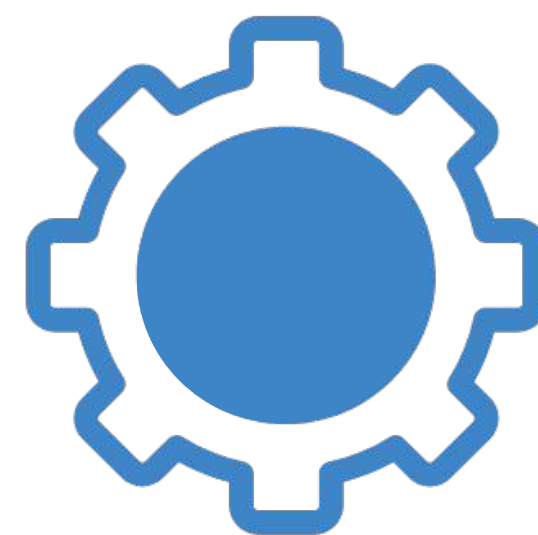

Node Package Manager (npm)

Node Package Manager (npm) is the package manager for Node.js. It consists of an online **registry of open-source packages** and a **command line client (also called npm)**.



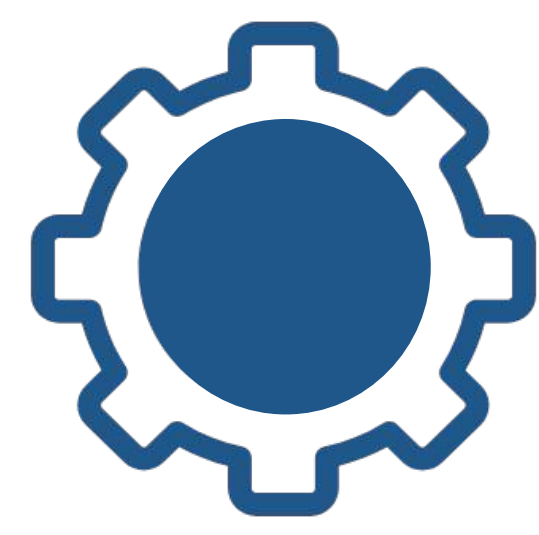
INSTALLING PACKAGES

The command line client can be used to **install an npm package** that is available in the registry. The package dependencies of a project are listed in the **package.json** file.



COMMANDS

The **npm install <package-name>** command can be used to install a specific package. The **npm install** command can be used to install all the dependencies in package.json.



RUNNING SCRIPTS

The command line client can be used to run tasks by using the **npm run <task-name>** command. These are specified as “**scripts**” in the package.json file.

Using Packages

An **npm package** can be installed as a **local install** or a **global install**. To use it in the code, it needs to be **imported** using **require**.

```
/* An npm package, such as the lodash library, can be installed in a project by executing the following command in the terminal.  
It installs the package in the 'node_modules' folder of the project directory. */
```

```
npm install lodash
```

```
// After installation, the library can be used in Node.js by using require() as shown below.
```

```
const _ = require('lodash');
```

```
const user1 = {name: 'Ronald', position: 'IT Analyst'};
```

```
const user2 = {name: 'Ronald', position: 'IT Analyst'};
```

```
const equal = _.isEqual(user1, user2);
```

```
console.log(`The two objects are ${equal ? 'equal': 'not equal'}.`);
```

Output

```
The two objects are equal.
```

Scenario and Solution

Scenario

A developer of Cosmic Solutions is creating a Node.js app that needs to make an outbound GET request to retrieve data about a particular employee. After the data is received, it should be added to the end of a file named employees.txt. The following code has been written so far:

```
const https = require('https');  
// Add code to include a core module for working with the file system.  
const options = {  
  hostname: 'cosmicsolutionsemployees.free.beeceptor.com',  
  path: '/' + employeeId, // employeeId is 713  
  method: 'GET'  
};  
  
const request = https.request(options, response => {  
  // Add code to append the retrieved data to a file named 'employees.txt' in the project directory.  
});  
  
request.on('error', error => {  
  console.error(error);  
});  
  
request.end()
```

Expected Data

Name: Ashley Williams, Role: US Sales Rep, Office Location: Phoenix, AZ

Solution

The following code can be added to include the 'fs' module that would allow working with the file system:

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

In order to access the employee data received from the remote endpoint, `response.on()` can be used to add a listener for the 'data' event. It can use a callback function that receives the data as its parameter.

Inside the callback function, `fs.appendFile()` method can be used to append a newline character ('\n') and the data to the end of the file named 'employees.txt'. This method appends the given content to the end of a given file. It also has a counterpart called `fs.appendFileSync()`, which performs the operation synchronously.

```
response.on('data', data => {
  fs.appendFile('./employees.txt', '\n' + data, err => {
    if (err) {
      console.error(err);
      return;
    }
    console.log('Employee data added to the file.');
```

```
});
```

Output

Employee data added to the file.

employees.txt X

employees.txt

1 Name: John Smith, Role: Sales Manager, Office Location: Richmond, VA

2 Name: Ashley Williams, Role: US Sales Rep, Office Location: Phoenix, AZ

Learn More



nodejs.org



[Node.js Documentation](https://nodejs.org/en/docs/)