```
console.log(systemInfo)
```

```
setTimeout(() => {
```

NodeJS

CHEAT SHEET

SCALER Topics

```
// Example paths
```

```
// Enable CORS for all routes
```

01 Introduction to NodeJS

What is NodeJS?

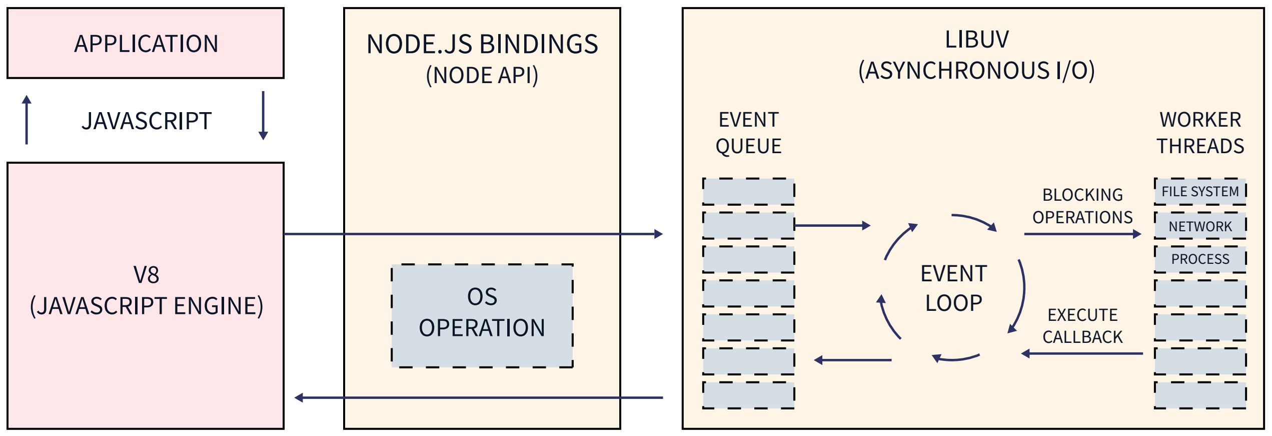
It is a runtime environment that allows JavaScript for server-side programming, enabling the execution of JavaScript code outside of a web browser.

Use case

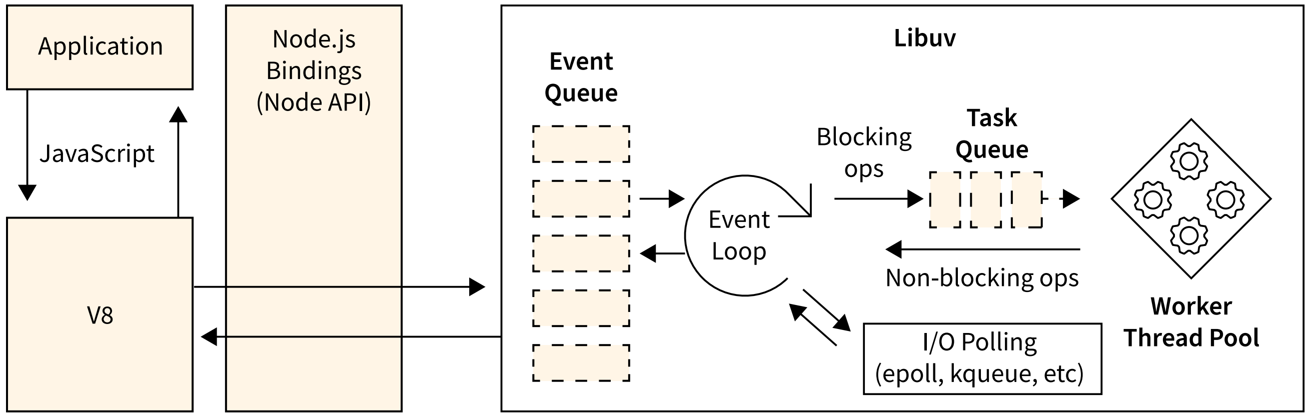
- Building high-performance server -side applications
- Web servers and APIs
- Real-time applications
- Application that needs event-driven architecture

Architecture

THE NODE.JS SYSTEM



Working



1.Start - Node.js application begins its execution.

2.Event Loop - Node.js operates on an event-driven, non-blocking I/O model. It is constantly checking for events such as incoming requests.

3.Event Triggered - When an event (e.g., an HTTP request, or a file system operation) occurs, it triggers a callback function.

4.Callback Execution - The callback function associated with the event is executed. Node.js uses asynchronous functions, allowing other tasks to continue while waiting for non-blocking operations to complete.

5.Concurrency - Multiple events can be processed concurrently, as the event loop manages the execution of callback functions.

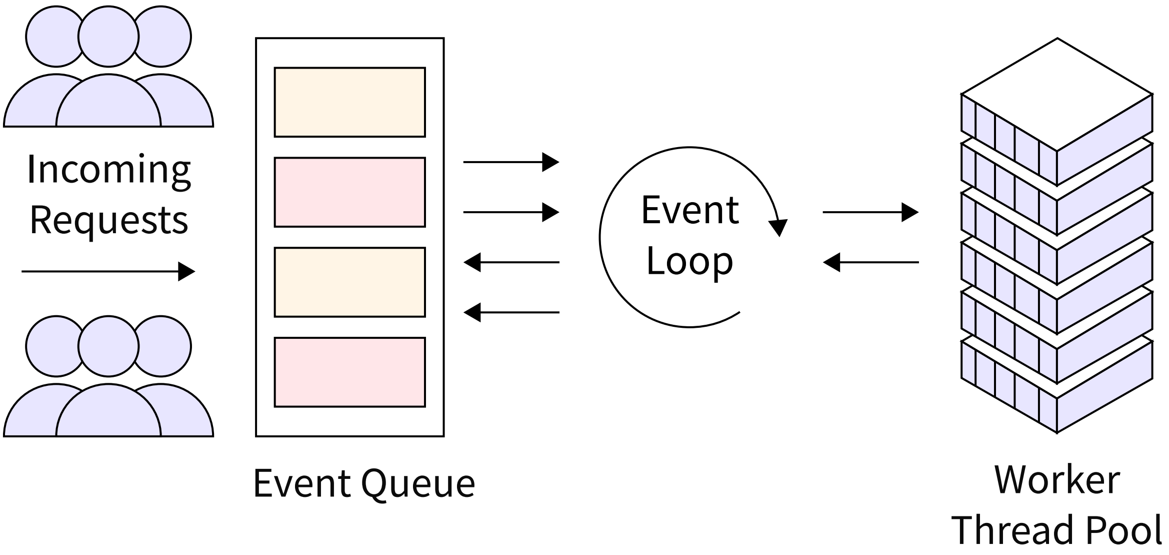
6.I/O Operations - It is used for reading from or writing to a file, making network requests, or interacting with databases.

7.Non-Blocking - The non-blocking nature of Node.js ensures that the application can continue processing other tasks while waiting for I/O operations to complete, improving overall efficiency.

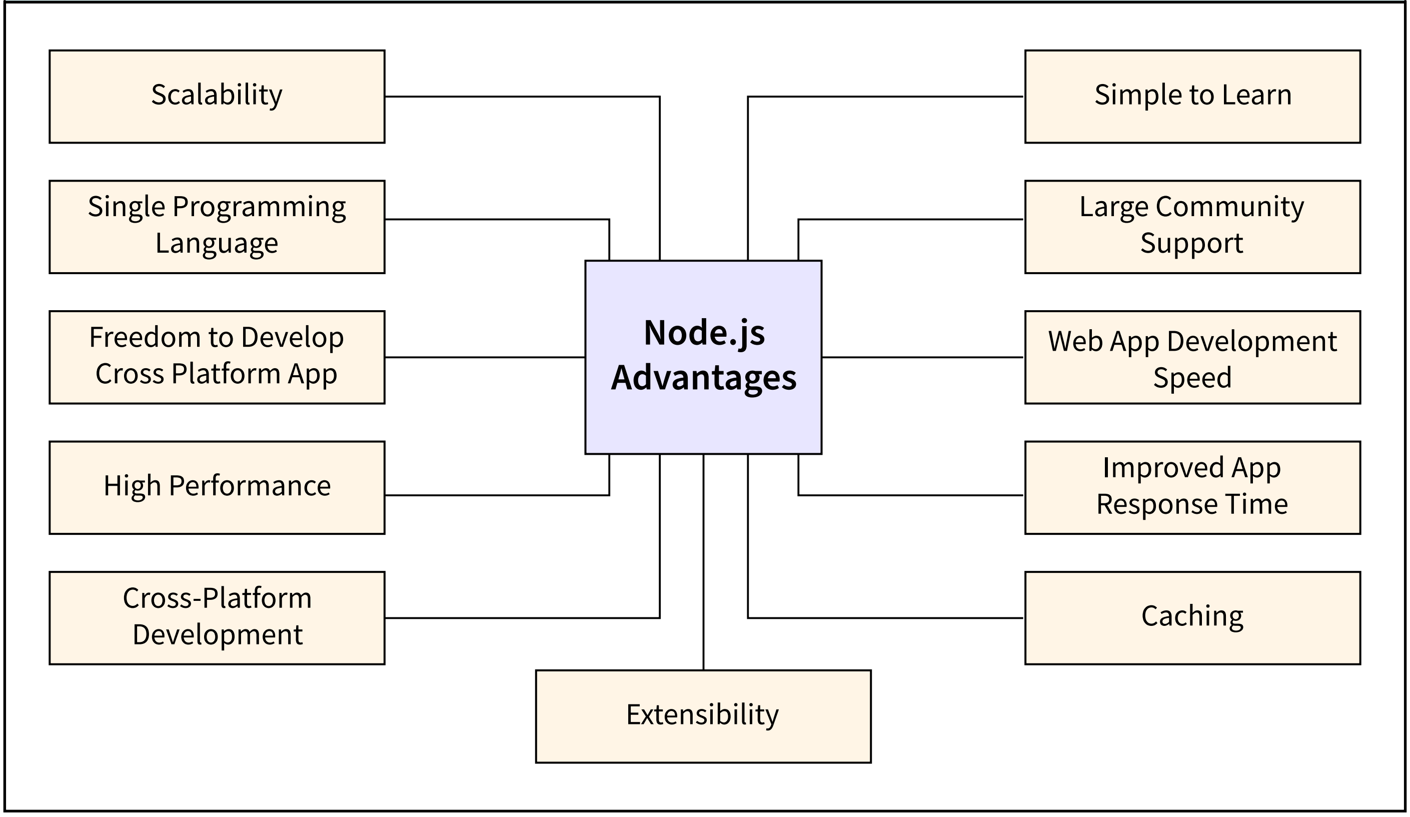
8.Callback Queue - Completed callback functions are placed in a callback queue..

9.Event Loop (Again) - The event loop checks for events and executes the associated callbacks, pulling them from the callback queue

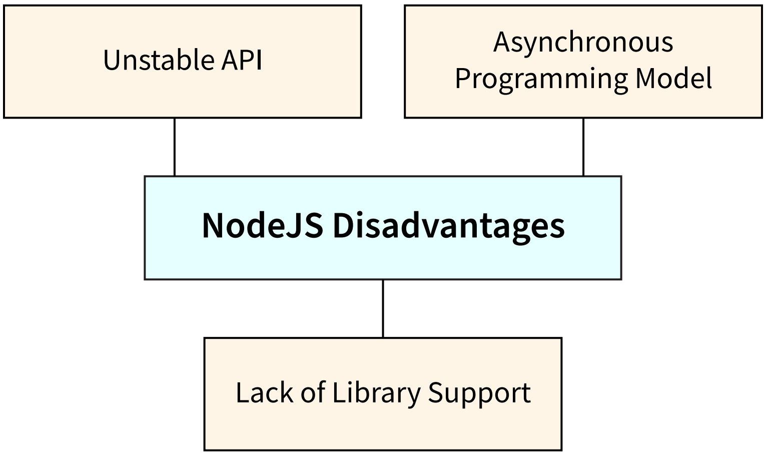
10.Repeat - The process repeats, allowing Node.js to efficiently handle multiple events simultaneously.



Advantages and Disadvantages


Node.js Advantages

- Scalability
- Single Programming Language
- Freedom to Develop Cross Platform App
- High Performance
- Cross-Platform Development
- Extensibility
- Simple to Learn
- Large Community Support
- Web App Development Speed
- Improved App Response Time
- Caching



02 Installing and running NodeJS

<table>
<tr><th>Mac OS</th><th>Windows</th></tr>
<tr><td>We can install Node.js on MacOS easily by using the installer from the official Node.js website.</td><td>We can install Node.js on Windows easily by using the installer from the official Node.js website.</td></tr>
<tr><th colspan="2">Linux (Ubuntu</th></tr>
<tr><td colspan="2">There are various package managers available for each distribution. We can install Node.js by using the Ubuntu official repository.</td></tr>
</table>

Command	Comments
node	Run node in your terminal
node --version	To check node version if installed
node filename.js	Executing the node code in file named filename.js

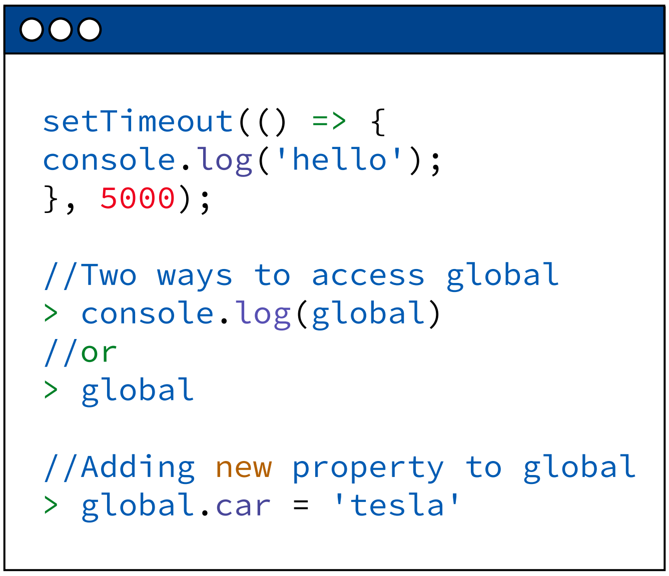
03 Global Object in NodeJS

Global object

Can be accessed anywhere. Used where a feature is expected to be available everywhere.

For example, to have some code execute after 5 seconds we can use either **global.setTimeout** or just **setTimeout**

Note: The global keyword is optional.



```
setTimeout(() => {
console.log('hello');
}, 5000);

//Two ways to access global
> console.log(global)
//or
> global

//Adding new property to global
> global.car = 'tesla'
```

You can also write **global.console.log()** to access global objects.

Note: It's important to note that while the global object provides accessibility across modules, relying too heavily on global variables and functions can lead to code that is harder to maintain and understand. Therefore, it's advisable to use globals judiciously and consider alternative approaches for better code organization and readability.

04 Process object in NodeJS

A process is the instance of a computer program that is being executed. Node has a global process object **NODE_ENV** which can be used to determine the environment in which the application is running

```
if (process.env.NODE_ENV === 'development'){
  console.log('Do not deploy!! Do not deploy!!');
}
```

process.argv in NodeJS

It is a property that holds an array of command-line values provided when the current process was initiated.

- First element→ absolute path
- Next element→ file path that is running
- Finally→ command-line arguments were provided when the process was initiated

```
// Command line values: node web.js
testing several features
```

```
console.log(process.argv[2]);
// 'testing' will be printed
```

process.memoryUsage in NodeJS

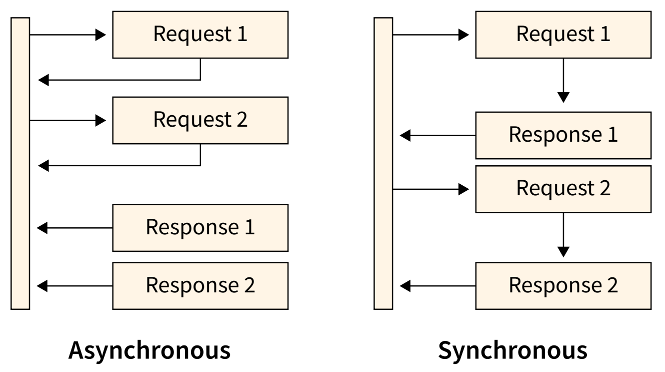
It is a method that can be used to return information on the CPU demands of the current process

```
//using process.memoryUsage() will
return an object in a format
like this:

{
  rss: 26247168,
  heapTotal: 5767168,
  heapUsed: 3573032,
  external: 8772
 }
```

05 Node.js and Asynchronous Programming

Node.js leverages asynchronous programming, employing a **non-blocking event-driven** architecture with **callback functions**, to efficiently **handle concurrent tasks** like I/O operations, enhancing scalability and responsiveness.

Async VS Sync



06 Module system in NodeJS

1.Core module

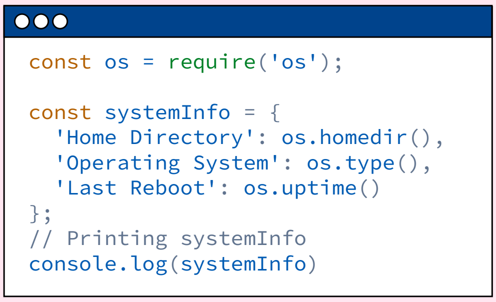
Modules included within the environment to efficiently perform common tasks.

console module

Exports a global console object **console.log()** and other familiar methods for debugging..

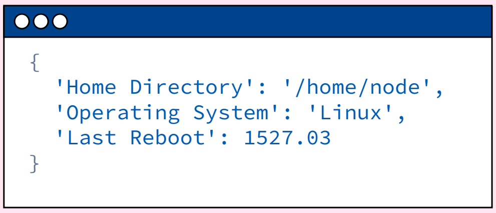
os module

os module can be used to get **information about the computer and operating system** on which a program is running.



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

const systemInfo = {
  'Home Directory': os.homedir(),
  'Operating System': os.type(),
  'Last Reboot': os.uptime()
};
// Printing systemInfo
console.log(systemInfo)
```

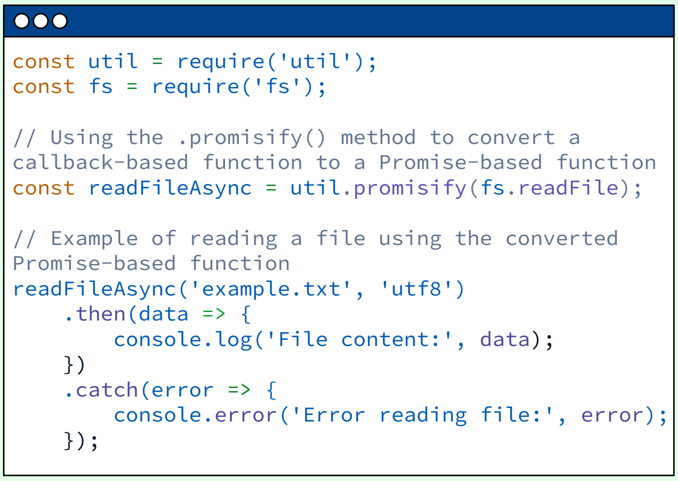
Output:



```
{
  'Home Directory': '/home/node',
  'Operating System': 'Linux',
  'Last Reboot': 1527.03
}
```

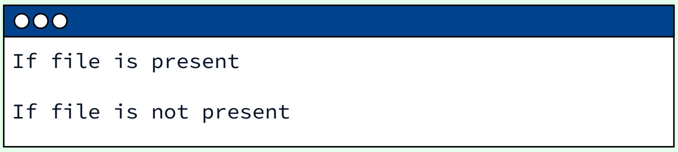
util module

Contains utility functions. Common uses include runtime type checking with types and turning callback functions into promises with the .promisify() method.



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

// Using the .promisify() method to convert a
callback-based function to a Promise-based function
const readFileAsync = util.promisify(fs.readFile);

// Example of reading a file using the converted
Promise-based function
readFileAsync('example.txt', 'utf8')
    .then(data => {
        console.log('File content:', data);
    })
    .catch(error => {
        console.error('Error reading file:', error);
    });
```

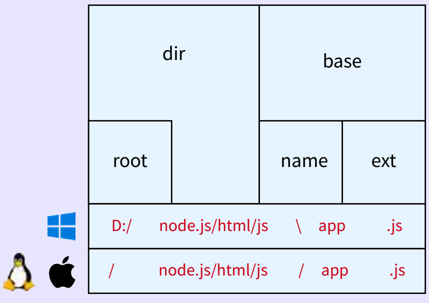
Output:



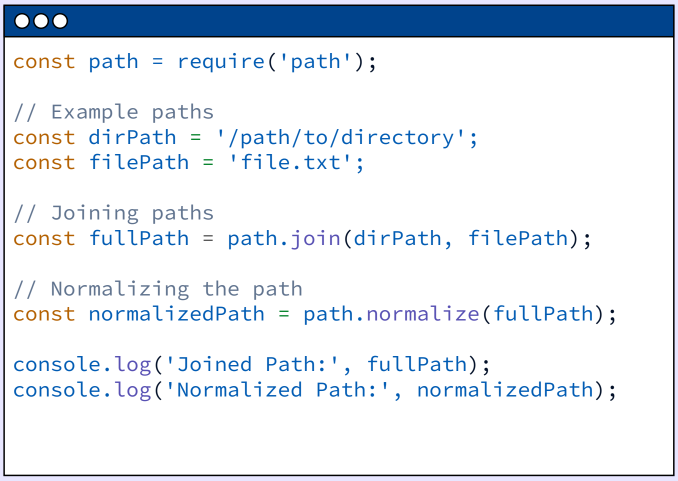
```
If file is present

If file is not present
```

path

The `path` module in Node.js provides utilities for working with file and directory paths.

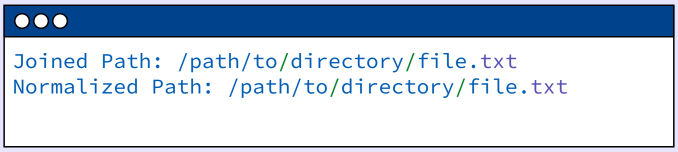


Node.js path Module



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

// Example paths
const dirPath = '/path/to/directory';
const filePath = 'file.txt';

// Joining paths
const fullPath = path.join(dirPath, filePath);

// Normalizing the path
const normalizedPath = path.normalize(fullPath);

console.log('Joined Path:', fullPath);
console.log('Normalized Path:', normalizedPath);
```

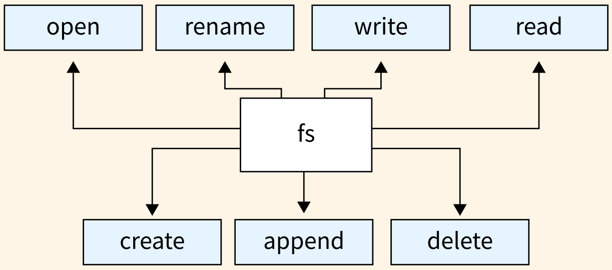
Output:

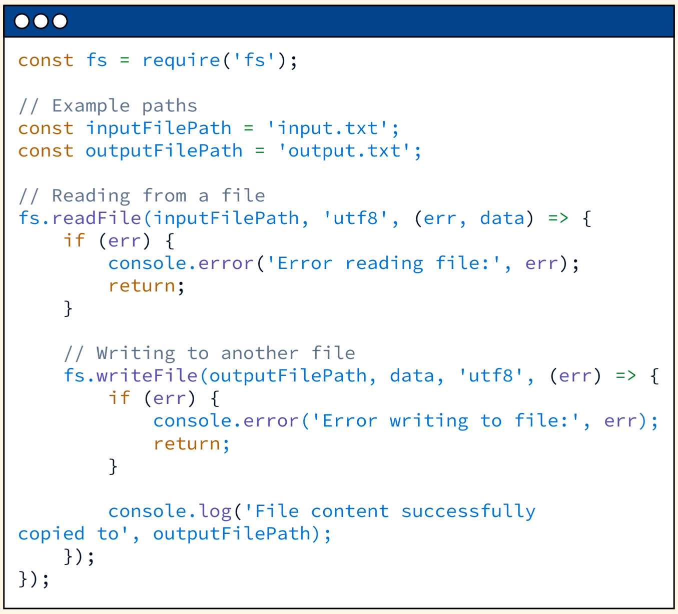


```
Joined Path: /path/to/directory/file.txt
Normalized Path: /path/to/directory/file.txt
```

fs

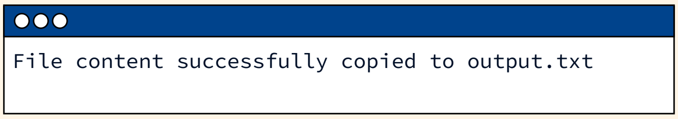
Read and write files on your file system.





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

// Example paths
const inputFilePath = 'input.txt';
const outputFilePath = 'output.txt';

// Reading from a file
fs.readFile(inputFilePath, 'utf8', (err, data) => {
    if (err) {
        console.error('Error reading file:', err);
        return;
    }

    // Writing to another file
    fs.writeFile(outputFilePath, data, 'utf8', (err) => {
        if (err) {
            console.error('Error writing to file:', err);
            return;
        }

        console.log('File content successfully
copied to', outputFilePath);
    });
});
```

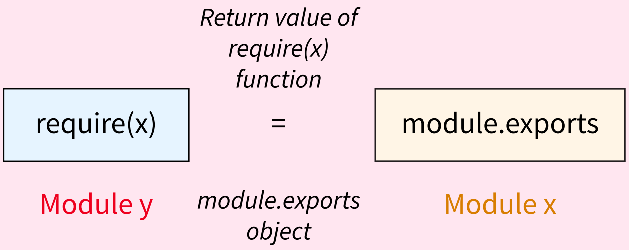
Output:



```
File content successfully copied to output.txt
```

2. require function

We can re-use existing code by using the Node built-in require() function. This function imports code from another module.



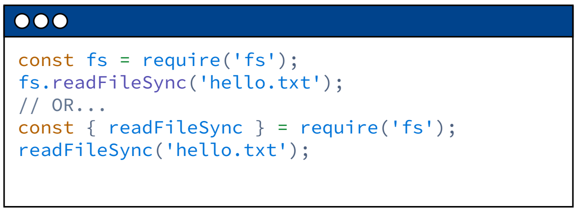
Return value of require(x) function

require(x) = module.exports

Module y

module.exports object

Module x



```
const fs = require('fs');
fs.readFileSync('hello.txt');
// OR...
const { readFileSync } = require('fs');
readFileSync('hello.txt');
```

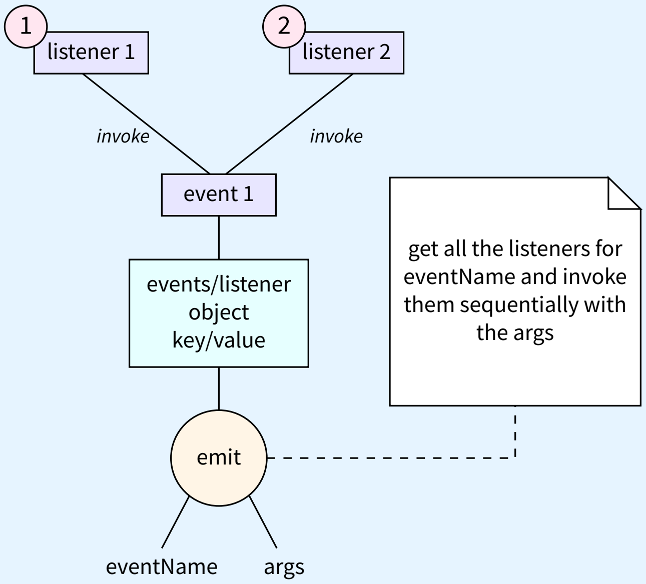
3. Built-in modules

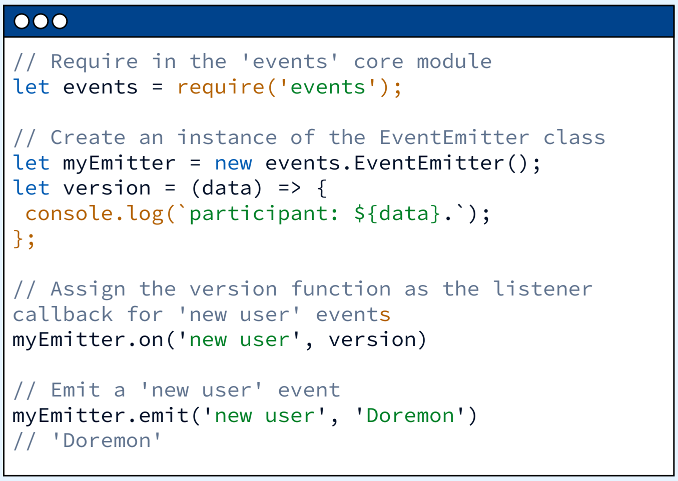
Key built-in modules include:

- **fs** → read and write files on your file system
- **path** → combine paths regardless of which OS you're using
- **process** → information about the currently running process, e.g. process.argv for arguments passed in or process.env for environment variables
- **http** → make requests and create HTTP servers
- **https** → work with secure HTTP servers using SSL/TLS
- **events** → work with the EventEmitter
- **crypto** → cryptography tools like encryption and hashing

4. events module

Used for EventEmitter, also has an .emit() method which announces a named event that has occured.


1
listener 1
2
listener 2
invoke
invoke
event 1
events/listener object key/value
get all the listeners for eventName and invoke them sequentially with the args
emit
eventName
args



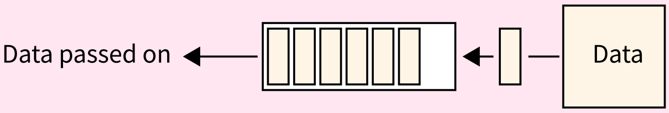
```
// Require in the 'events' core module
let events = require('events');

// Create an instance of the EventEmitter class
let myEmitter = new events.EventEmitter();
let version = (data) => {
 console.log(`participant: ${data}.`);
};

// Assign the version function as the listener
callback for 'new user' events
myEmitter.on('new user', version)

// Emit a 'new user' event
myEmitter.emit('new user', 'Doremon')
// 'Doremon'
```

5. error module

It doesn’t have a specific error module. The callback function has an error passed as a first parameter. If there is no error then that parameter is undefined.

6. buffer object

A Buffer is an object that represents a static amount of memory that can’t be resized. The Buffer class is within the global buffer module, meaning it can be used without the require() statement.

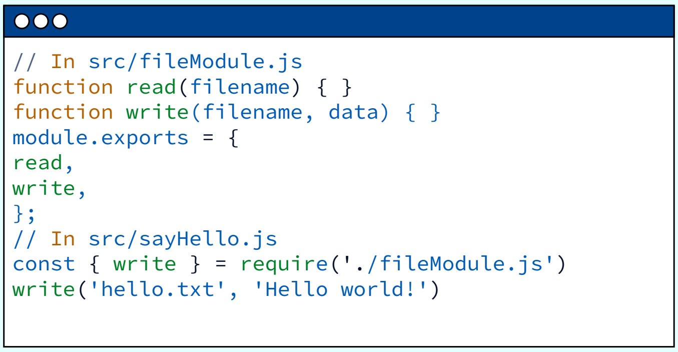


7. timers module

The global timers module contains scheduling functions such as setTimeout(), setInterval(), and setImmediate(). These functions are put into a queue processed at every iteration of the Node.js event loop.

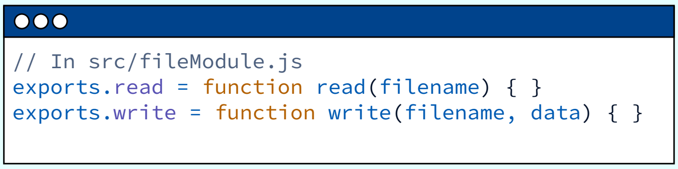
8. How to create modules

We can create our own modules by exporting a function from a file and importing it in another module.



```
// In src/fileModule.js
function read(filename) { }
function write(filename, data) { }
module.exports = {
read,
write,
};
// In src/sayHello.js
const { write } = require('./fileModule.js')
write('hello.txt', 'Hello world!')
```

Some Node modules may instead use the shorthand syntax to export functions.

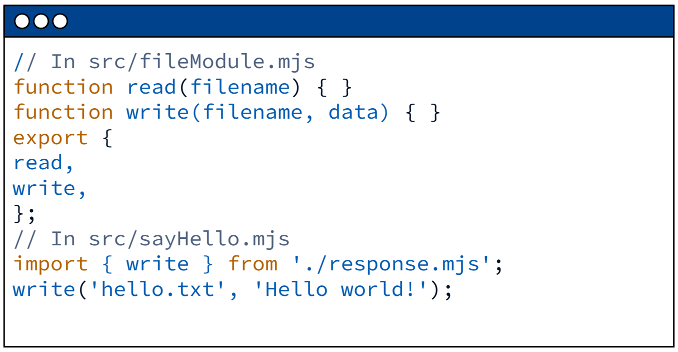


```
// In src/fileModule.js
exports.read = function read(filename) { }
exports.write = function write(filename, data) { }
```

9. ECMAScript modules

The imports above use a syntax known as CommonJS (CJS) modules. Node treats JavaScript code as CommonJS modules by default. More recently, you may have seen the ECMAScript module (ESM) syntax. This is the syntax that is used by TypeScript

Extension: .mjs



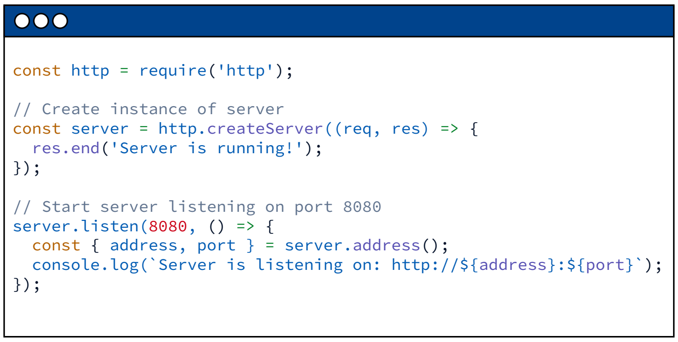
```
// In src/fileModule.mjs
function read(filename) { }
function write(filename, data) { }
export {
read,
write,
};
// In src/sayHello.mjs
import { write } from './response.mjs';
write('hello.txt', 'Hello world!');
```

07 Setting up the server with HTTP

HTTP messages can be protected using Transort Layer Security (TLS), a protocol designed to facilitate secure data transmission via encryption.

1. .createServer() Method

- Create an HTTP server
- Takes a single argument in the form of a callback function
- Callback function has two primary arguments; the request (commonly written as req) and the response (commonly written as res)



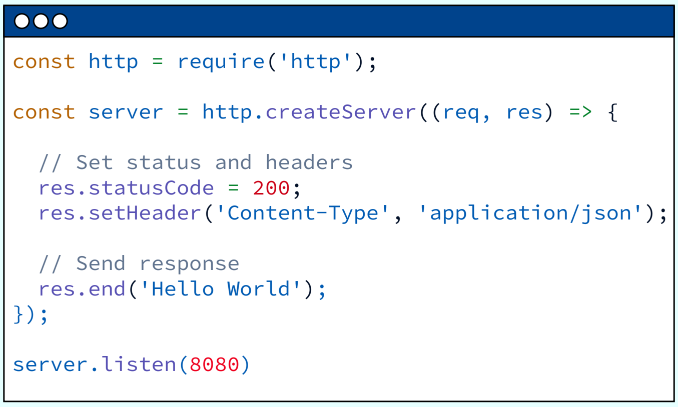
```
const http = require('http');

// Create instance of server
const server = http.createServer((req, res) => {
  res.end('Server is running!');
});

// Start server listening on port 8080
server.listen(8080, () => {
  const { address, port } = server.address();
  console.log(`Server is listening on: http://${address}:${port}`);
});
```

2. Request Object

- Contains information about the incoming HTTP request
- Handle server requests such as routing and data processing

3. Response Object

- Contains information about the outgoing HTTP response
- Contains various properties and methods that can be used to configure and send the response such as **.statusCode, .setHeader(), and.end()**

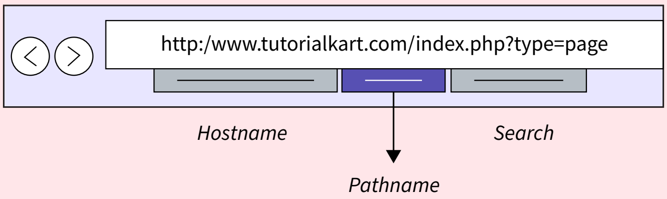


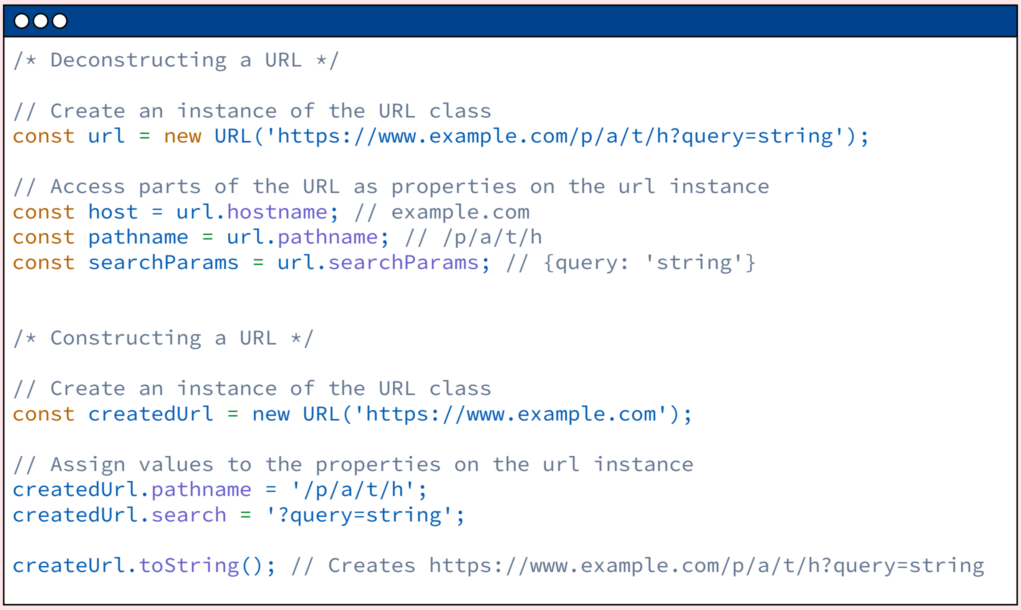
```
const http = require('http');

const server = http.createServer((req, res) => {

  // Set status and headers
  res.statusCode = 200;
  res.setHeader('Content-Type', 'application/json');

  // Send response
  res.end('Hello World');
});

server.listen(8080)
```

4. url Module

Used to break down URLs into their constituent parts.

Parse URL





```
/* Deconstructing a URL */

// Create an instance of the URL class
const url = new URL('https://www.example.com/p/a/t/h?query=string');

// Access parts of the URL as properties on the url instance
const host = url.hostname; // example.com
const pathname = url.pathname; // /p/a/t/h
const searchParams = url.searchParams; // {query: 'string'}


/* Constructing a URL */

// Create an instance of the URL class
const createdUrl = new URL('https://www.example.com');

// Assign values to the properties on the url instance
createdUrl.pathname = '/p/a/t/h';
createdUrl.search = '?query=string';

createUrl.toString(); // Creates https://www.example.com/p/a/t/h?query=string
```

5. Query String Parameters

- Used in conjunction with GET requests to submit information used in processing a request
- Provide filter criteria for some requested data

6. Request Body

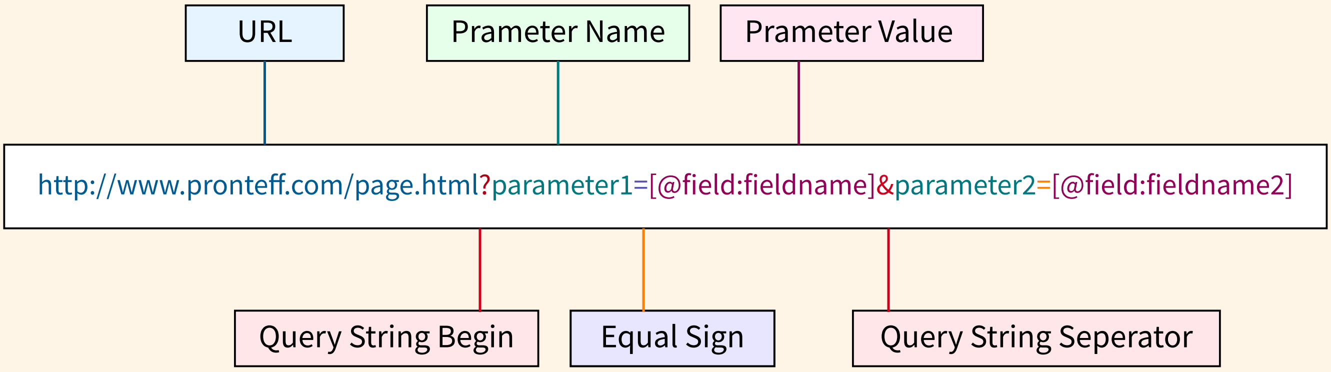
- Data is commonly provided in the body of a request
- The body is most commonly used in POST and PUT requests as they usually have information that needs to be processed by the server

7. HTTP Headers

- Provide metadata that is used by servers to process requests

8. querystring Module

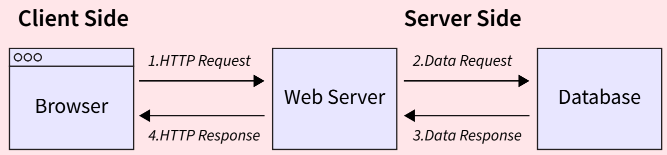
- Used to decode/encode and parse query string parameters into easily usable data structures
- Operates on query string parameters, requiring isolation of the query string before use
- Methods of the module are **.parse(), .stringify() , .escape(), and .unescape()**


URL
Prameter Name
Prameter Value
http://www.pronteff.com/page.html?parameter1=[@field:fieldname]&parameter2=[@field:fieldname2]
Query String Begin
Equal Sign
Query String Seperator

What are Query String Parameters & Methods in NodeJS

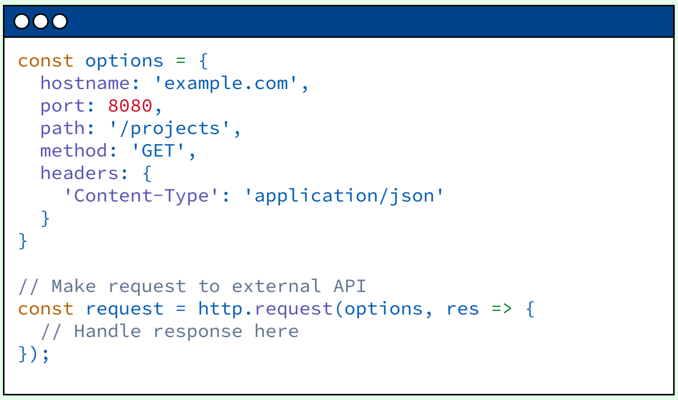
9. HTTP and Databases

- HTTP requests can be used to interact with databases to retrieve remote data



10. HTTP and External APIs

- Used to interact with other external APIs from within a server
- A common way to request another server is through the **.request()** method on the http module

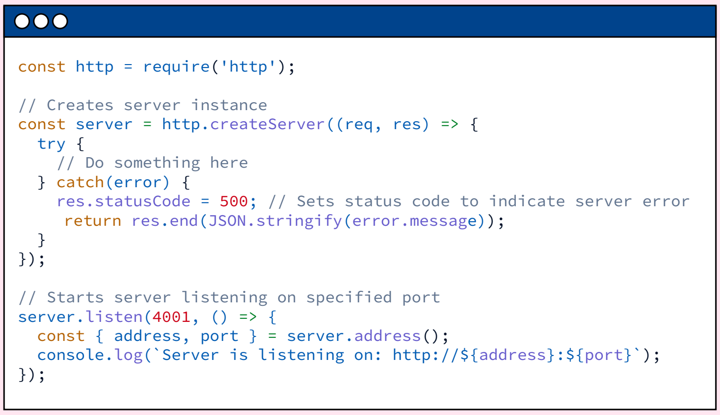


```
const options = {
  hostname: 'example.com',
  port: 8080,
  path: '/projects',
  method: 'GET',
  headers: {
    'Content-Type': 'application/json'
  }
}

// Make request to external API
const request = http.request(options, res => {
  // Handle response here
});
```

11. HTTP Response Status Codes

- Indicate whether a specific HTTP request has been completed
- Conveys information about what happened during the processing of the request

1XX	Informational codes	The server acknowledges and is processing the request.
2XX	Success codes	The server successfully recieved, understood, and processed the requests.
3XX	Redirection codes	The server recieived the request, but there's a redirect to somewhere else (or, in rare cases, some additional action oter than a redirect must be completed.
4XX	Client error codes	The server couldn't find (or reach) the page or website. This is an error on the site's side.
5XX	Server error codes	The client made a valid request, but the server failded to complete the request.



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

// Creates server instance
const server = http.createServer((req, res) => {
  try {
    // Do something here
  } catch(error) {
    res.statusCode = 500; // Sets status code to indicate server error
     return res.end(JSON.stringify(error.message));
  }
});

// Starts server listening on specified port
server.listen(4001, () => {
  const { address, port } = server.address();
  console.log(`Server is listening on: http://${address}:${port}`);
});
```

08 Packages in NodeJS

A package is a collection of Node modules along with a package.json file describing the package

1.npm command

Command	Comments
npm start	Execute the current Node package defined by package.json; defaults to node server.js
npm init	Initialize a fresh package.json file
npm init -y	Initialize a fresh package.json file, accepting all default options (equivalent to npm init --yes).
npm install	Install a package from the NPM registry at www.npmjs.com (equivalent to npm i <package>).
npm install -D	Install a package as a development dependency (equivalent to npm install --save-dev <package>).
npm install -g	Install a package globally.
npm update <package>	Update an already installed package (equivalent to npm up <package>).
npm uninstall <package>	Uninstall a package from your node_modules/ folder (equivalent to npm un <package>).
npm outdated	Check for outdated package dependencies.
npm audit	Check for security vulnerabilities in package dependencies.
npm audit fix	Try to fix any security vulnerabilities by automatically updating vulnerable packages.

2. package.json file

It contains:

- Name, version, description, and license of the current package.
- Scripts to automate tasks like starting, testing, and installing the current package.
- Lists of dependencies that are required to be installed by the current package

3. node_modules

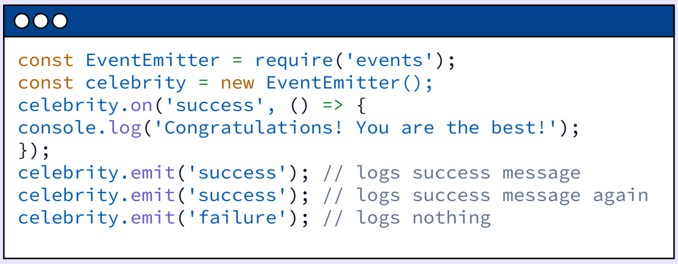
- Next to your package.json file
- When you run npm install the packages listed as dependencies in your package.json are downloaded from the NPM registry and put in the node_modules folder. It contains not just your direct dependencies, but also the dependencies of those dependencies. The entire dependency tree lives in node_modules.

4. package-lock.json

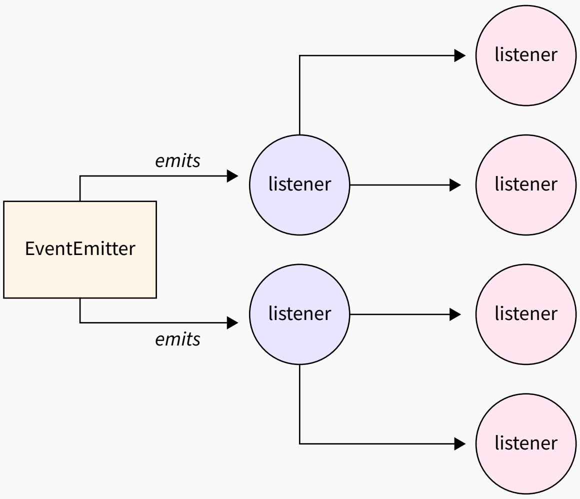
- The package-lock.json file is automatically created by NPM to track the exact versions of packages that are installed in your node_modules folder.

09 Event emitter in NodeJS

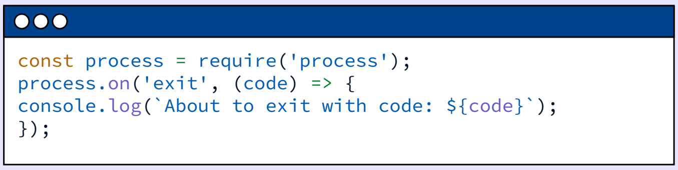
Node.js provides a built-in module to work with events



```
const EventEmitter = require('events');
const celebrity = new EventEmitter();
celebrity.on('success', () => {
console.log('Congratulations! You are the best!');
});
celebrity.emit('success'); // logs success message
celebrity.emit('success'); // logs success message again
celebrity.emit('failure'); // logs nothing
```

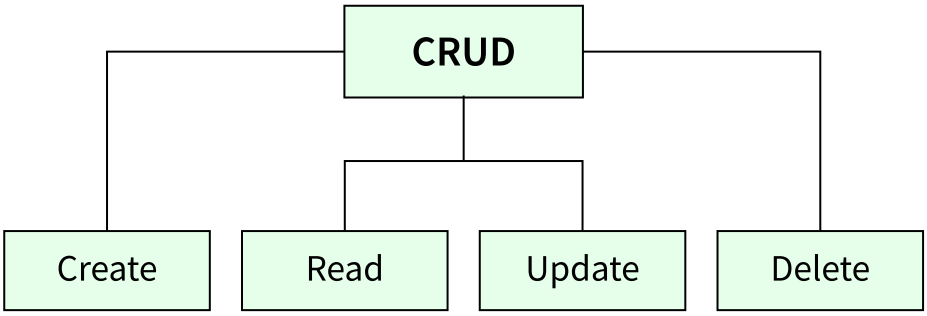


Some examples include the currently running Node process, a running HTTP server, and web sockets. They all emit events that can then be listened for using a listener function like on().



```
const process = require('process');
process.on('exit', (code) => {
console.log(`About to exit with code: ${code}`);
});
```

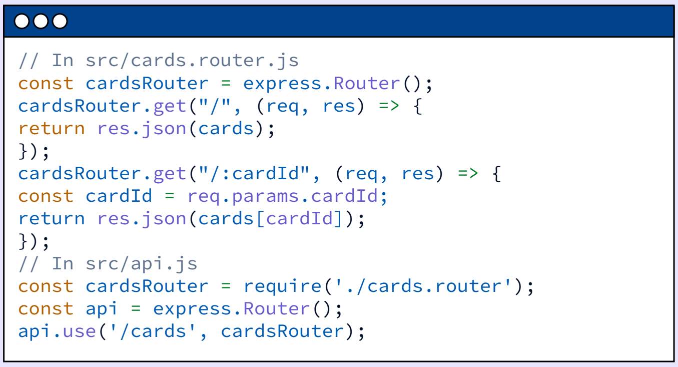
10 Concept of the backend in NodeJS



CRUD Operation	HTTP Method	Example
Create	POST	POST /cards - **Save a new card to the cards collection**
Read	GET	GET /cards - **Get the whole cards collection**
Read	GET	GET /cards/:cardId - **Get an individual card**
Update	PUT (or PATCH)	PUT /cards/:cardId - **Update an individual card**
Delete	DELETE	DELETE /cards/:cardId - **Delete an individual card**
Delete	DELETE	DELETE /cards - **Delete the entire collection of cards**

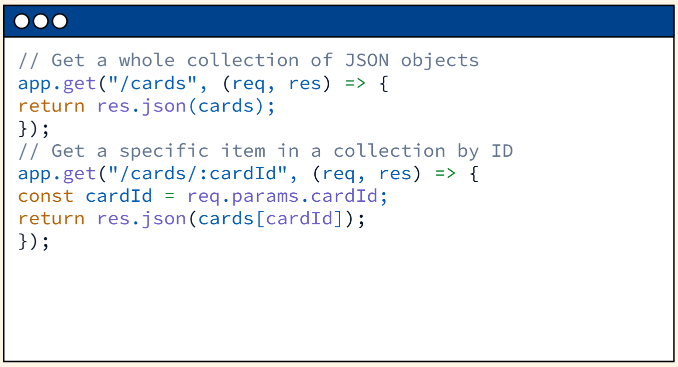
11 Express.js

Routers



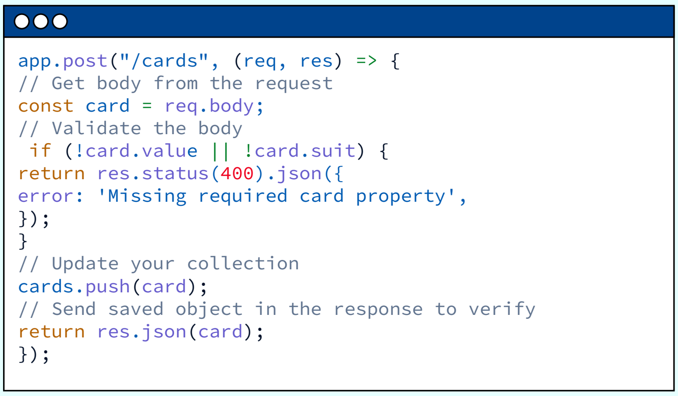
```
// In src/cards.router.js
const cardsRouter = express.Router();
cardsRouter.get("/", (req, res) => {
return res.json(cards);
});
cardsRouter.get("/:cardId", (req, res) => {
const cardId = req.params.cardId;
return res.json(cards[cardId]);
});
// In src/api.js
const cardsRouter = require('./cards.router');
const api = express.Router();
api.use('/cards', cardsRouter);
```

GET route



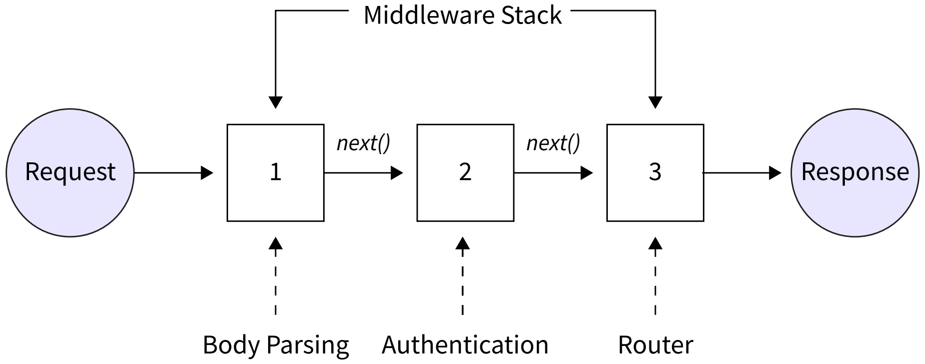
```
// Get a whole collection of JSON objects
app.get("/cards", (req, res) => {
return res.json(cards);
});
// Get a specific item in a collection by ID
app.get("/cards/:cardId", (req, res) => {
const cardId = req.params.cardId;
return res.json(cards[cardId]);
});
```

POST route



```
app.post("/cards", (req, res) => {
// Get body from the request
const card = req.body;
// Validate the body
 if (!card.value || !card.suit) {
return res.status(400).json({
error: 'Missing required card property',
});
}
// Update your collection
cards.push(card);
// Send saved object in the response to verify
return res.json(card);
});
```

12 Middlewares in Express



1. What are Middlewares

- Functions that have access to the request, response objects, and the next function in the application's request-response cycle, allowing for pre-processing, modification, or termination of the request flow based on specific conditions.
- Enhance the functionality of the server by adding layers of processing between the client and the final route handlers

2. Custom Middlewares

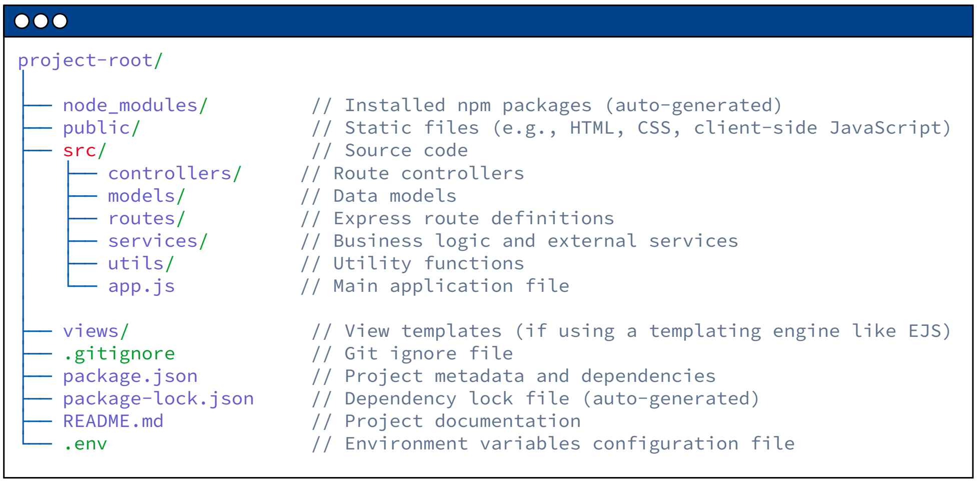
Functions that can be created to perform specific tasks or validations in the request-response cycle, adding customized processing to routes.

3. Third-Party Middlewares

Third-party Middlewares in Node.js are pre-built middleware functions provided by external libraries, often used to add common functionality (e.g., authentication, logging) to an application with minimal setup

13 Folder structure in NodeJS

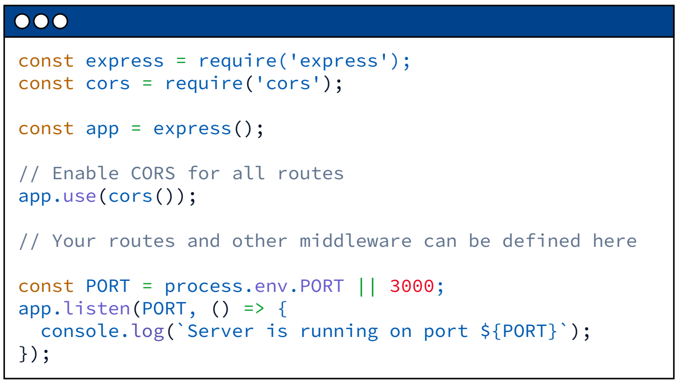
Here's a commonly used and recommended structure for a basic Node.js application.



```
project-root/
│
├── node_modules/         // Installed npm packages (auto-generated)
├── public/               // Static files (e.g., HTML, CSS, client-side JavaScript)
├── src/                  // Source code
│   ├── controllers/     // Route controllers
│   ├── models/          // Data models
│   ├── routes/          // Express route definitions
│   ├── services/        // Business logic and external services
│   ├── utils/           // Utility functions
│   └── app.js           // Main application file
│
├── views/                // View templates (if using a templating engine like EJS)
├── .gitignore            // Git ignore file
├── package.json          // Project metadata and dependencies
├── package-lock.json     // Dependency lock file (auto-generated)
├── README.md             // Project documentation
└── .env                  // Environment variables configuration file
```

14 Cross-origin resource sharing

Security feature implemented by web browsers to control which web pages can make requests to a given resource from a different domain.



```
const express = require('express');
const cors = require('cors');

const app = express();

// Enable CORS for all routes
app.use(cors());

// Your routes and other middleware can be defined here

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

15 PM2 commands

- PM2 is a tool we use to create and manage Node.js clusters. It allows us to create clusters of processes, manage those processes in production, and keep our applications running forever.
- We can install the PM2 tool globally using npm install -g pm2

HTTP Method	Example
pm2 list	List the status of all processes managed by PM2
pm2 start server.js -i 4	Start server.js in cluster mode with 4 processes
pm2 start server.js -i 0	Start server.js in cluster mode with the maximum number of processes.
pm2 logs	Show logs from all processes.
pm2 logs --lines 200	Show older logs up to 200 lines long.
pm2 monit	Display a real-time dashboard in your terminal with statistics for all processes.
pm2 stop 0	Stop running process with ID 0.
pm2 restart 0	Restart process with ID 0.
pm2 delete 0	Remove process with ID 0 from PM2's list of managed processes.
pm2 delete all	Remove all processes from PM2's list.
pm2 reload all	Zero downtime reload of all processes managed by PM2. For updating and reloading server code already running in production.

SCALER TOPICS

Unlock your potential in software development with **FREE COURSES** from **SCALER TOPICS!**

Register now and take the first step towards your future Success!

C++ for Beginners

5.9k enrolled ₹ Free

Java for Beginners

6.8k enrolled ₹ Free

That's not it. Explore 20+ Courses by clicking below

Explore Other Courses

Practice **CHALLENGES**
and become 1% better everyday

CIFAR-10 Image Classification Using PyTorch

Article

No. Of Questions : 3

Go to Challenge >

How to Build a Snake Game in JavaScript?

Article

No. Of Questions : 3

Go to Challenge >

Explore Other Challenges