



The background features a black field with four large, irregular geometric shapes in teal, light green, and dark teal. The text 'NODE JS' is centered, with 'NODE' in teal and 'JS' in yellow. The letters are bold and have a distressed, hand-painted appearance with visible brush strokes.

NODE JS

INTRODUCTION

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a browser.

It is built on Chrome's V8 JavaScript engine, and uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js is commonly used for developing server-side applications, but can also be used for developing desktop and mobile applications.

KEY FEATURES

JavaScript: Node.js utilizes JavaScript as its primary programming language, allowing developers to use the same language for both client-side and server-side development, resulting in improved code reusability.

KEY FEATURES

Asynchronous and Non-blocking: Node.js operates on an event-driven architecture, which means it can handle multiple concurrent requests efficiently without blocking other operations. This asynchronous nature allows for scalable and high-performance applications.

KEY FEATURES

NPM (Node Package Manager): Node.js comes bundled with npm, a powerful package manager that provides access to a vast ecosystem of open-source libraries and frameworks. This extensive repository of packages allows developers to easily integrate third-party modules into their applications.

KEY FEATURES

Lightweight and Fast: Node.js is known for its lightweight nature, making it quick to install and execute. Its efficient event loop and non-blocking I/O operations contribute to its fast and responsive performance.

KEY FEATURES

Large Community and Support: Node.js has a thriving community of developers and contributors, providing extensive documentation, tutorials, and support. This vibrant community ensures continuous improvement and innovation within the Node.js ecosystem.

ASYNCHRONOUS CODE

In asynchronous code, operations are initiated and then the program continues to execute without waiting for the operation to complete. A callback function is usually provided to handle the result when the operation finishes.

CALLBACKS

Callbacks are functions passed as arguments to asynchronous functions. They are executed once the asynchronous operation is completed.

EVENT DRIVEN PROGRAMMING

Event-driven programming is a programming paradigm that focuses on the flow of events or actions that occur within a program. In this paradigm, the program responds to events as they occur, rather than executing code in a linear manner. This approach is particularly useful for building interactive applications, such as web applications.

EVENT DRIVEN PROGRAMMING

EVENT: An event is a signal that something has happened.
It can be triggered by a specific action or a change in state.



```
console.log("Starting Node.js");
```

```
db.query("SELECT * FROM public.cars", function (err, res) {  
  console.log("Query executed");  
});
```

```
console.log("Before query result");
```

An invoked function is added to the call stack. Once it returns a value, it is popped off.



```
console.log("Starting Node.js");

db.query("SELECT * FROM public.cars", function (err, res) {
  console.log("Query executed");
});

console.log("Before query result");
```

OUTPUT

V8 ENGINE

CALL STACK

LIBUV

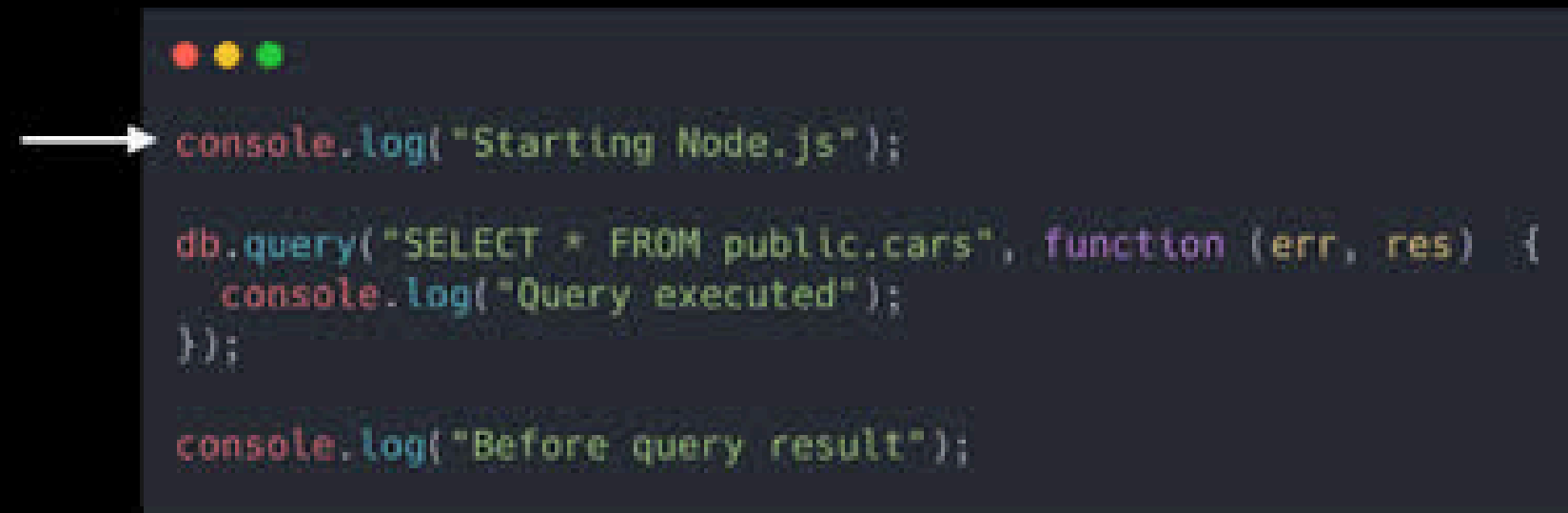
LIBUV API

EVENT
LOOP



EVENT QUEUE

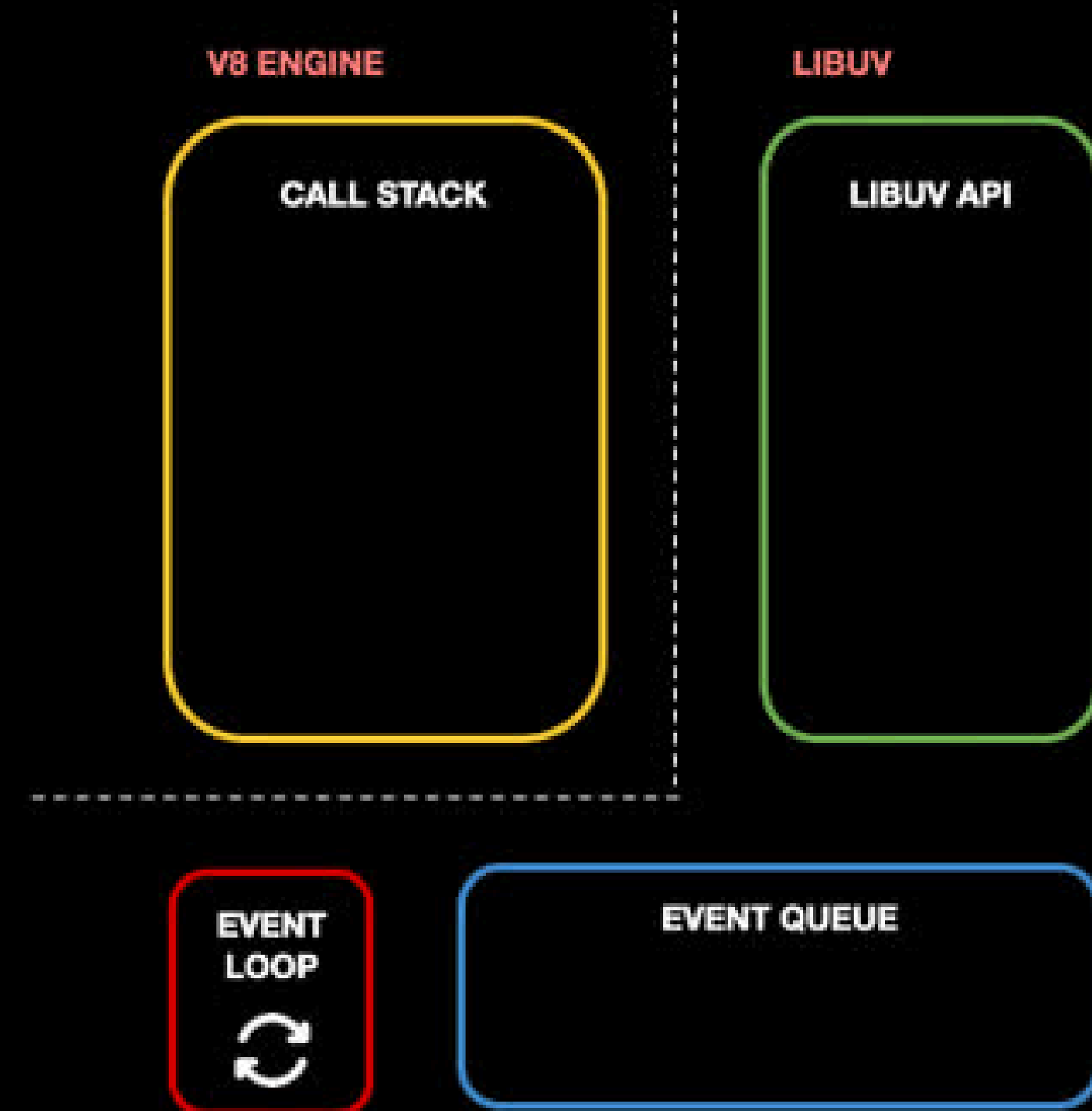
Database queries or other I/O ops do not block Node.js single thread because Libuv API handles them.



```
console.log("Starting Node.js");  
  
db.query("SELECT * FROM public.cars", function (err, res) {  
  console.log("Query executed");  
});  
  
console.log("Before query result");
```

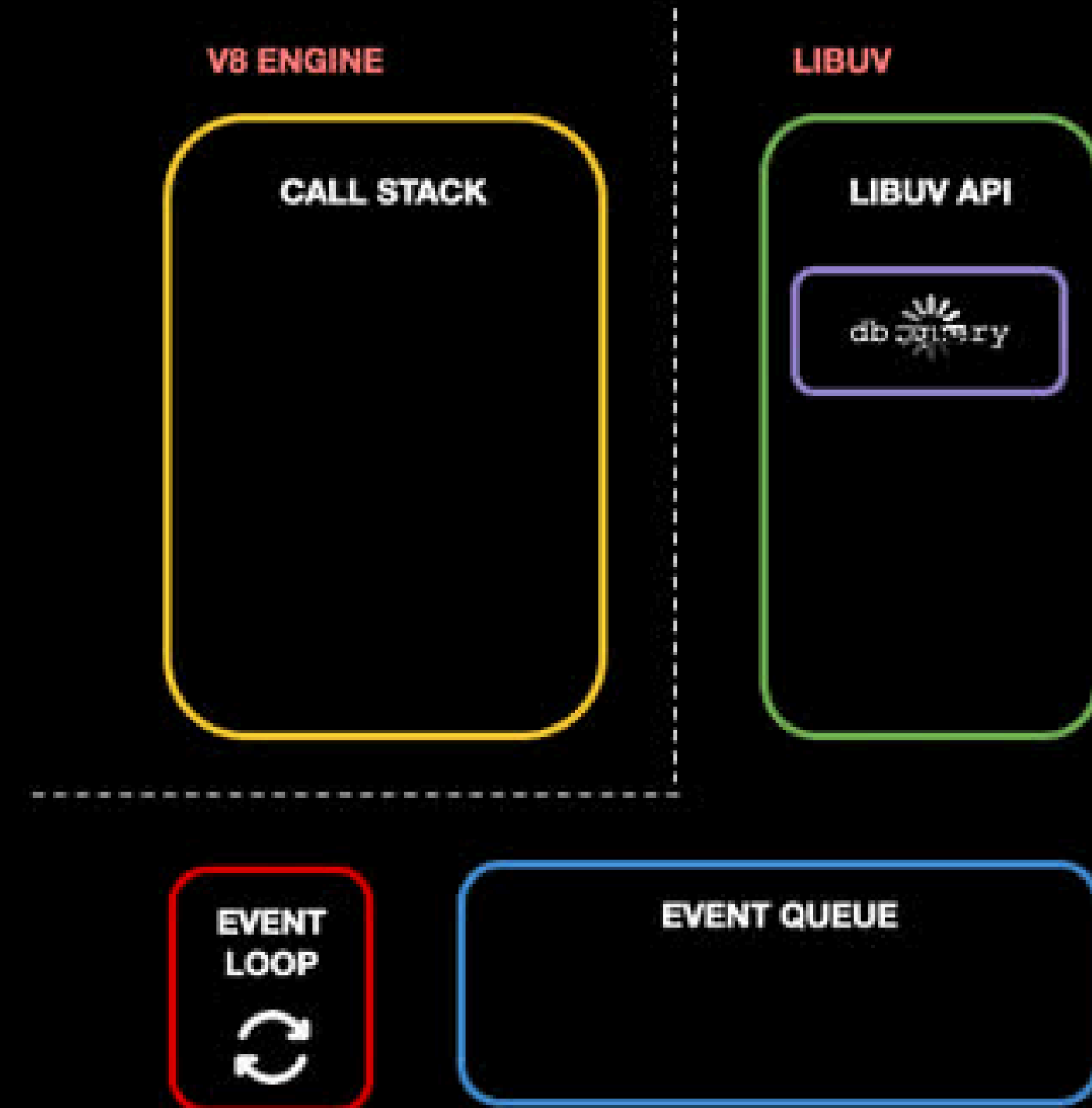
OUTPUT

```
Starting Node.js
```



While Libuv asynchronously handles I/O operations, Node.js single thread keeps running code.

```
console.log("Starting Node.js");  
→ db.query("SELECT * FROM public.cars", function (err, res) {  
  console.log("Query executed");  
});  
console.log("Before query result");
```



OUTPUT

```
Starting Node.js
```

Callbacks of completed queries are moved to the event queue. If the call stack is empty, the event loop checks for callbacks and transfers the first.

```
console.log("Starting Node.js");

db.query("SELECT * FROM public.cars", function (err, res) {
  console.log("Query executed");
});

→ console.log("Before query result");
```

V8 ENGINE

CALL STACK

console.log

LIBUV

LIBUV API

db.query

OUTPUT

Starting Node.js

Before query result

EVENT
LOOP



EVENT QUEUE

PROMISES

Promises are an alternative to callbacks, providing a cleaner way to handle asynchronous code. A Promise represents a value that may be available now, or in the future, or never.

ASYNC/AWAIT

Async/await is a syntax built on top of Promises, providing a more readable and concise way to write asynchronous code.

COMMON ASYNCHRONOUS OPERATIONS

- File I/O: Reading/writing files asynchronously.
- Network Operations: Making HTTP requests, handling database queries.

ERROR HANDLING

Asynchronous operations may result in errors. It's essential to handle errors properly to prevent them from crashing the application.

setTimeout()

The `setTimeout()` function is a built-in JavaScript function that allows you to schedule the execution of a function or a piece of code after a specified delay (in milliseconds). It takes two arguments: a callback function or code to be executed and the delay in milliseconds.

```
function sayHello() {  
    console.log('Hello!');  
}  
  
setTimeout(sayHello, 2000);
```

clearTimeout()

The clearTimeout() function is used to cancel a timeout set by setTimeout() before it has a chance to execute. It takes one argument: the identifier returned by the setTimeout() function.

```
function sayHello() {  
    console.log('Hello!');  
}
```

```
const timeoutId = setTimeout(sayHello, 2000);  
clearTimeout(timeoutId); // Cancels the timeout
```


setInterval()

The `setInterval()` function is similar to `setTimeout()`, but instead of executing the code once after a delay, it repeatedly executes the provided function or code at a specified interval.

```
let count = 0;
```

```
function incrementCount() {  
    count++;  
    console.log('Count:', count);  
}
```

```
setInterval(incrementCount, 1000);
```

BUILT-IN MODULES

Node.js provides a rich set of built-in modules that extend the capabilities of JavaScript for server-side development. These modules cover a wide range of functionalities, from handling file I/O to working with HTTP, events, and more.

BUILT-IN MODULES

FS (File System).

In JavaScript, the File System module is used to interact with the file system on the server or in a Node.js environment. The File System module provides methods and APIs to perform various file-related operations, such as reading and writing files, creating directories, deleting files, and more. It allows you to manipulate files and directories on the local file system.

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

```
import fs from 'fs';
```

BUILT-IN MODULES

Reading Files: The `fs.readFile()` method is used to read the contents of a file asynchronously. It takes the file path and an optional encoding as parameters and returns the contents of the file in the provided encoding.

```
fs.readFile('file.txt', 'utf-8', (err, data) => {  
  if (err) throw err;  
  console.log(data);  
});
```

BUILT-IN MODULES

Writing file: `fs.writeFile()` method is used to write data to a file. It takes the file path, data to write, and an optional encoding as parameters. It creates a new file if it doesn't exist or replaces the existing file with the new data.

```
const content = 'This is some data to write to the file.';
fs.writeFile('file.txt', content, 'utf-8', (err) => {
  if (err) throw err;
  console.log('File written successfully.');
```

```
});
```

BUILT-IN MODULES

Creating Directories: The `fs.mkdir()` method is used to create a new directory asynchronously. It takes the directory path and an optional options object as parameters.

```
fs.mkdir('newDir', (err) => {  
  if (err) throw err;  
  console.log('Directory created successfully.');
```

```
});
```


BUILT-IN MODULES

deleting Files or Directories: The `fs.unlink()` method is used to delete a file asynchronously, while the `fs.rmdir()` method is used to delete an empty directory asynchronously.

```
fs.unlink('file.txt', (err) => {  
  if (err) throw err;  
  console.log('File deleted successfully.');
```

```
});  
  
fs.rmdir('emptyDir', (err) => {  
  if (err) throw err;  
  console.log('Directory deleted successfully.');
```

BUILT-IN MODULES

OS MODULE

In Node.js, the `os` module is a built-in module that provides various methods for interacting with the operating system. You can use the `os` module to retrieve information about the operating system, and network interfaces, and perform other OS-related tasks.

```
console.log(os.platform());  
console.log(os.arch());  
console.log(os.type());  
console.log(os.release());  
console.log(os.hostname());
```

BUILT-IN MODULES

HTTP MODULE

In Node.js, the http module is a built-in module that provides functionality for creating HTTP servers and making HTTP requests. It allows you to handle incoming HTTP requests and send HTTP requests to external servers.

SERVER

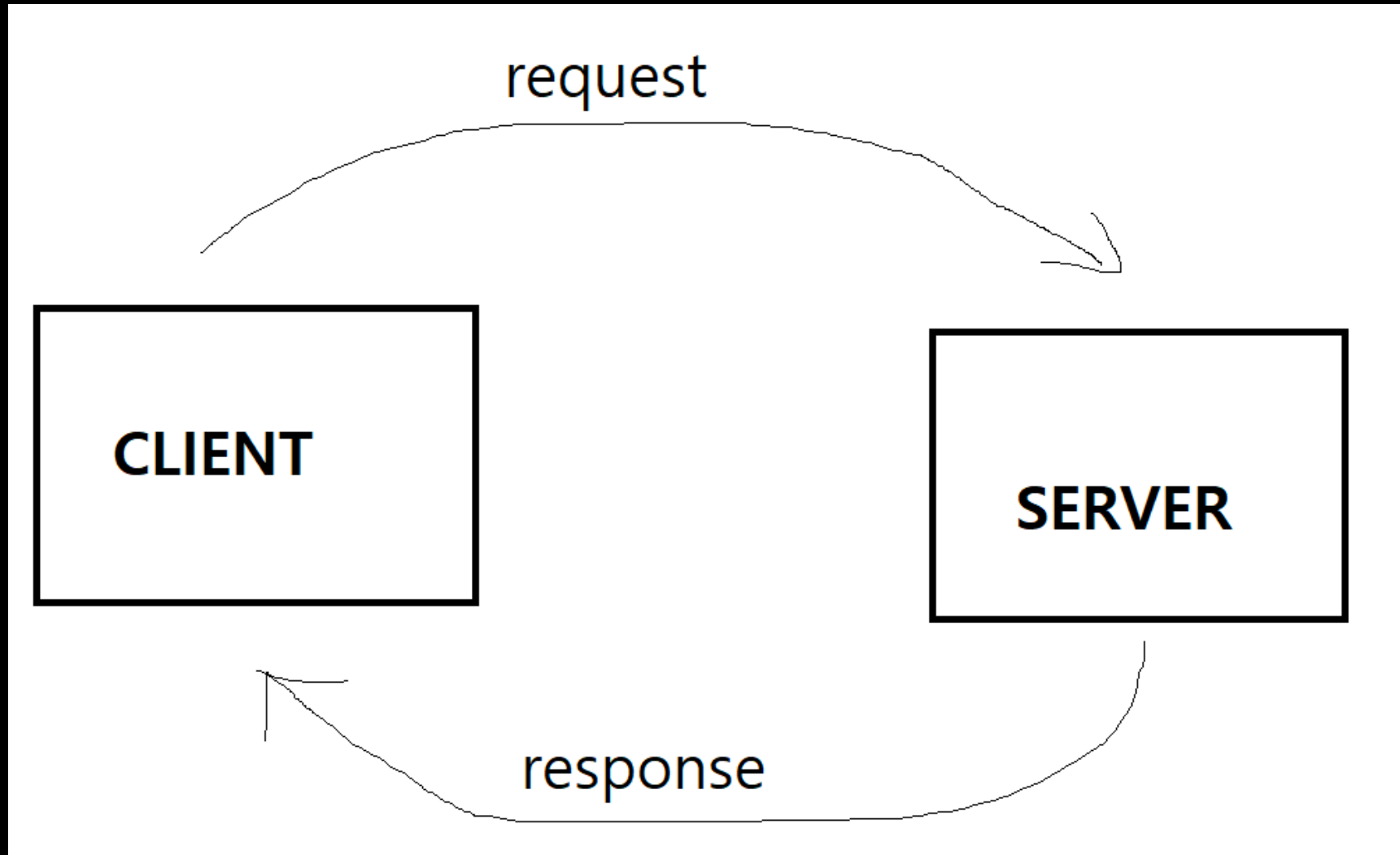
A server is a computer program or a device that provides services or resources to other programs or devices, known as clients, over a network. Servers are a fundamental part of the client-server architecture, a model widely used in computing to distribute workloads and facilitate communication between different devices.

SERVER

Service Provider

Servers are designed to provide specific services, resources, or functionality to clients. These services can include file storage, web hosting, email processing, database management, and more.

SERVER



TYPES OF SERVERS

- **Web Servers**: Serve web pages and handle HTTP requests.
- **Database Servers**: Store and manage databases, handling database-related requests.
- **Mail Servers**: Handle email communication and storage.
- **Application Servers**: Host and execute applications, providing services to clients.
- **DNS Servers**: Resolve domain names to IP addresses.

CREATE A SERVER

In Node.js, the http module provides functionality for creating HTTP servers

createServer method from the http module is used to create an HTTP server. Then callback function that will be executed whenever there is an incoming HTTP request. The req and res parameters represent the request and response objects, respectively.

```
const server = http.createServer((req, res) => {  
  // Request handling logic goes here  
});
```

req (request)

A request refers to the data that a client sends to a server when making an HTTP call. The request typically consists of two main parts: the request method and headers in the request header, and optional data in the request body.

REQUEST METHOD

The request method is an HTTP verb that specifies the action the client wants to perform on a resource.

- **GET:** Retrieve data from the server.
- **POST:** Submit data to the server to create new resource.
- **PUT:** Update a resource on the server.
- **DELETE:** Delete a resource on the server.
- **PATCH:** Apply partial modifications to a resource.

REQUEST HEADERS

Request headers are key-value pairs sent by the client to provide additional information about the request. They include details such as the client's user agent, accepted content types, and authentication tokens.

REQUEST HEADERS

- **User-Agent:** Indicates the client application or browser making the request.
- **Content-Type:** Specifies the type of data in the request body (e.g., application/json).
- **Authorization:** Contains credentials for authenticating the client.

REQUEST BODY

The request body is the optional data sent by the client in the HTTP request. It is commonly used with methods like POST and PUT to submit data to the server.

res (response)

A response refers to the data that a server sends back to a client in response to an HTTP request. The response typically consists of two main parts: the response header and the response body.

RESPONSE HEADERS

The response header is a set of key-value pairs sent by the server to provide metadata about the response. It contains information such as the HTTP status code, content type, server information, and various other details.

- **Status Code**: Indicates the status of the HTTP request. Common status codes include 200 OK for a successful request, 404 Not Found for a resource not found, and 500 Internal Server Error for server errors.
- **Content Type**: Specifies the type of content included in the response body. Common content types include text/html for HTML documents, application/json for JSON data, and image/jpeg for JPEG images.
- **Server Information**: Some servers include information about the server software and version in the response header. This is often used for debugging or informational purposes.

RESPONSE BODY

The response body is the actual content of the response sent by the server. It can include HTML, JSON, plain text, binary data, or any other content type depending on the nature of the HTTP request.

```
res.write('Hello, ');  
res.write('World!');  
res.end();
```

```
const server = http.createServer((req, res) => {  
  // Setting response headers  
  res.writeHead(200, {  
    'Content-Type': 'text/plain',  
    'Server': 'Node.js Server'  
  });  
  
  // Sending response body  
  res.write('Hello, ');  
  res.write('World!');  
  res.end();  
});
```

COMMON CONTENT TYPES

```
res.writeHead(200, { 'Content-Type': 'text/plain' });  
res.end('Hello, World!\n');
```

```
res.writeHead(200, { 'Content-Type': 'text/html' })  
res.end('<h1>Hello, World!</h1>');
```

```
const data = { message: 'Hello, World!' };  
res.writeHead(200, { 'Content-Type': 'application/json' });  
res.end(JSON.stringify(data));
```

STATUS CODES

2xx Success:

- a. 200 OK: Standard response for successful HTTP requests.
- b. 201 Created: Request has been fulfilled, and a new resource has been created.

3xx Redirection:

- a. 301 Moved Permanently: The requested resource has been permanently moved to a new location.
- b. 302 Found or 307 Temporary Redirect: The requested resource resides temporarily under a different URI.

STATUS CODES

4xx Client Errors:

- 400 Bad Request: The server cannot or will not process the request due to a client error.
- 401 Unauthorized: Similar to 403 Forbidden but specifically for authentication.
- 403 Forbidden: The client does not have permission to access the requested resource.
- 404 Not Found: The requested resource could not be found on the server.

STATUS CODES

5xx Server Errors:

- 500 Internal Server Error: A generic error message returned when an unexpected condition was encountered on the server.
- 501 Not Implemented: The server either does not recognize the request method or lacks the ability to fulfill the request.
- 503 Service Unavailable: The server is not ready to handle the request. Common causes are a server that is down for maintenance or overloaded.

METHODS AND ROUTING

```
const server = http.createServer((req, res) => {  
  // Accessing request properties  
  const method = req.method;  
  const url = req.url;
```

HTML

```
res.writeHead(200, { 'Content-Type': 'text/html' });
```

```
// Send the HTML response
```

```
res.end(`
  <!DOCTYPE html>
  <html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width
    <title>HTML Response</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
`);
```

JSON

JSON, which stands for JavaScript Object Notation, is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. JSON is often used to transmit data between a server and a web application

JSON represents data as key-value pairs, similar to JavaScript object literals. Data is organized into objects and arrays, allowing for nested structures.

FRAMEWORK

A framework is a pre-built set of tools, libraries, and conventions that provide a structured way to build and organize software applications. Frameworks aim to simplify the development process by offering reusable components, a defined structure, and often follow certain design patterns. They can be applied to various types of software development, including web development.

EXPRESS

Express is a minimal and flexible web application framework for Node.js. It is one of the most popular frameworks in the Node.js ecosystem.

CREATING A SERVER IN EXPRESS

```
import express from 'express';

// Create an instance of the Express application
const app = express();

// Set a port for the server to listen on
const PORT = 3000;

// Define a route
app.get('/', (req, res) => {
  res.send('Hello, World!');
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server is listening on port ${PORT}`);
});
```



PARAMS

In Express, route parameters are used to capture values specified in the URL and make them accessible within the route handler. These parameters are defined in the route pattern and are prefixed with a colon (:). Express extracts the values from the URL and provides them as properties in the `req.params` object.


```
// Route with a parameter
app.get('/users/:userId', (req, res) => {
  const userId = req.params.userId;
  res.send(`User ID: ${userId}`);
});
```

```
app.get('/users/:userId/posts/:postId', (req, res) => {  
  const userId = req.params.userId;  
  const postId = req.params.postId;  
  res.send(`User ID: ${userId}, Post ID: ${postId}`);  
});
```

MAKE A CALL FROM FRONTEND TO SERVER

QUERY STRINGS

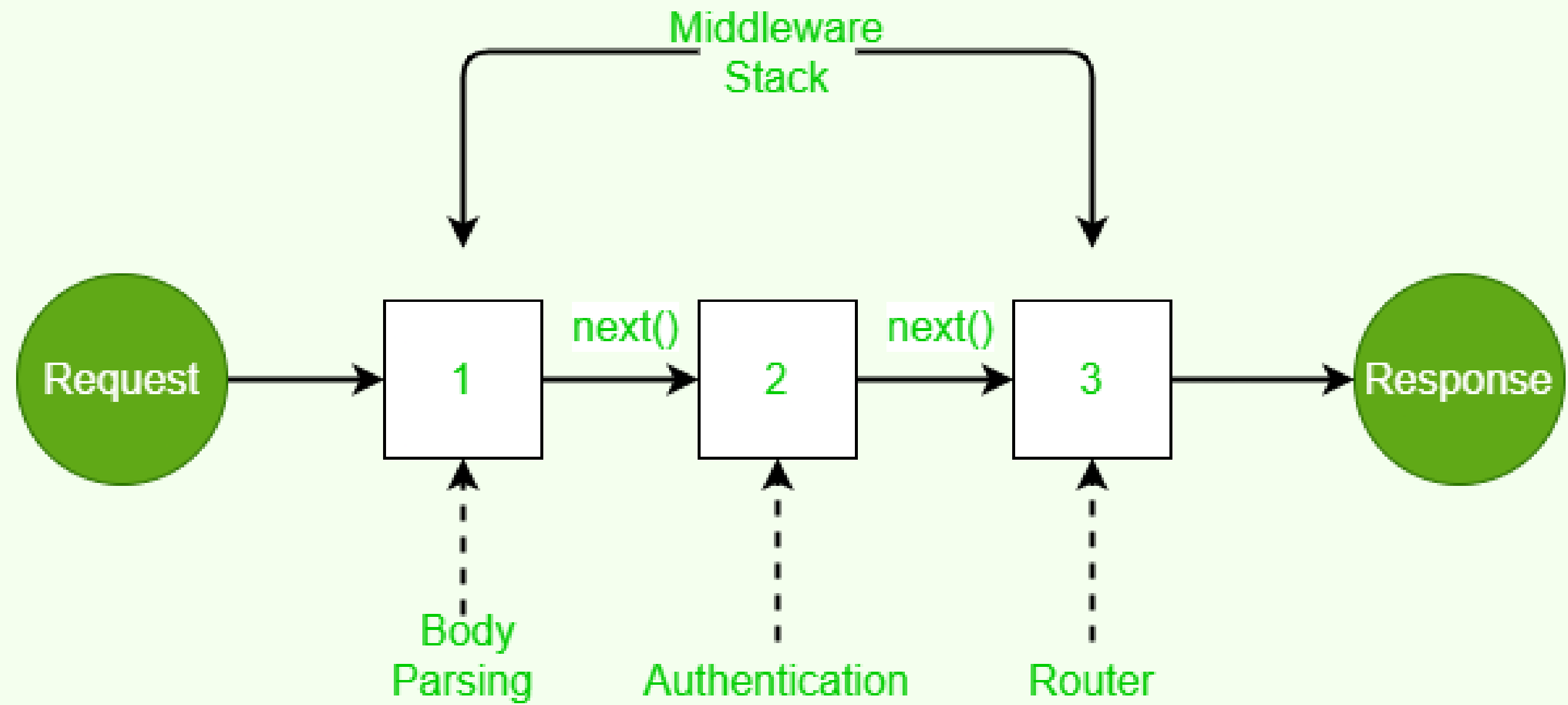
In Express, query strings are a way to send data to the server as part of the URL. Query strings are usually added to the end of a URL and are preceded by a question mark (?). They consist of key-value pairs separated by ampersands (&). Express makes it easy to extract and parse query strings from incoming requests using the `req.query` object.

/api?name=jhon&age=52

```
app.get('/users', (req, res) => {  
  const name = req.query.name;  
  const age = req.query.age;  
  
  res.send(`User Name: ${name}, Age: ${age}`);  
});
```

MIDDLEWARE

Middleware functions in Node.js, and particularly in the context of web frameworks like Express, are functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle. These functions can modify the request and response objects, end the request-response cycle, or call the next middleware function in the stack.



MIDDLEWARE

- **Executing code on every request**: Middleware functions can execute code that needs to run on every incoming request, regardless of the route.
- **Modifying the request or response**: Middleware functions can modify the req and res objects, adding or altering properties as needed.
- **Handling errors**: Middleware functions can catch errors and handle them in a centralized way.
- **Authentication and Authorization**: Middleware functions are often used for implementing authentication and authorization checks before allowing access to certain routes.


```
// Middleware function
const loggerMiddleware = (req, res, next) => {
  console.log(`Request received at ${new Date()}`);
  next(); // Call the next middleware function in the stack
};

// Use the middleware for all routes
app.use(loggerMiddleware);
```

MORGAN

- HTTP request logger middleware.
- Logs incoming requests to the console, providing information about the request method, URL, response status, and response time.

```
import morgan from 'morgan';
```

```
app.use(morgan('dev')); // Log requests in the development format
```

ORDER OF EXECUTION

In an Express application, the order in which you define your routes and middleware is important because Express matches the routes and executes the corresponding handlers based on the order in which they are defined. Express uses a first-match-wins approach when it comes to route matching.

ORDER OF EXECUTION

When a request is received, Express iterates through the defined routes and middleware in the order they were added to the application. It compares the incoming request's URL path against each defined route's path pattern to find a match. The first route that matches the request's path will be executed, and the subsequent routes will be ignored.

SERVING STATICFILES

```
app.use(express.static('public'));
```

FILE UPLOAD

To handle file uploads in an Express.js application, you can use the `multer` middleware, which is a popular package specifically designed for handling multipart/form-data requests (used for file uploads).

1. Install multer

```
npm install multer
```


2. Create an instance of multer and specify the destination folder

```
const express = require('express');  
const multer = require('multer');  
const app = express();  
const upload = multer({ dest: 'uploads/' });
```

3. Set up a route handler that will handle the file upload. Use the upload middleware to specify the field name in the form and handle the uploaded file.

```
app.post('/upload', upload.single('file'), (req, res) => {  
  // Access the uploaded file using req.file  
  const uploadedFile = req.file;  
  // Process the file as needed (e.g., save it to the database, perform  
  // Return a response to the client  
  res.send('File uploaded successfully');  
});
```

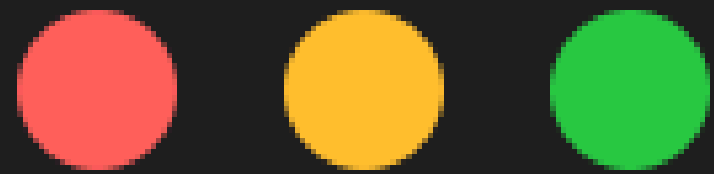
4. To upload complex file

```
exports.multerStorage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, "uploads");
  },
  filename: (req, file, cb) => {
    console.log(file.mimetype);
    const ext = file.mimetype.split("/")[1];
    cb(null, `${file.fieldname}.${ext}`);
  },
});
```

<%= EJS %>

EJS is a simple templating language that lets you generate HTML markup with plain JavaScript.

```
const ejs = require('ejs');  
app.set('view engine', 'ejs');
```



```
1  <h1><%= data %></h1>
```



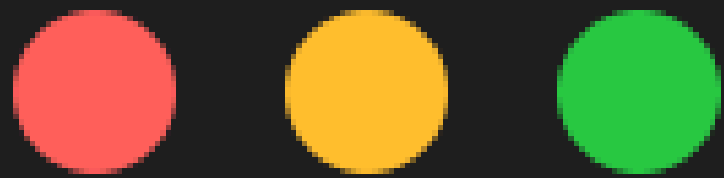
```
1 app.use(express.static("public"));
```




```
1  <% if (data) {%>
2
3    <h1>Hello</h1>
4
5  <%} else {%>
6
7    <h1>Hi</h1>
8
9  <%}%>
```



```
1  <ul>
2      <% articles.map((article)=> { %>
3          <li>
4              <h2><%= article.title %></h2>
5              <p><%= article.body %></p>
6          </li>
7          <hr />
8          <% }) %>
9  </ul>
```



```
1  <%- include( './navbar' )%>
```



```
1 <%- include('./navbar', {title : 'Page Title'})%>
```



```
1 <%= typeof title != 'undefined' ? title : 'Page Title' %>
```

