**Core Modules**

Node.js comes with a set of built-in modules that you can use without any additional installation. Some commonly used core modules include:

* **fs**: File system operations (reading/writing files).
* **http**: Creating HTTP servers and clients.
* **path**: Working with file and directory paths.
* **os**: Operating system-related utility methods and properties.
* **events**: Event-driven programming.

**Local Modules**

* You can create your own modules by defining JavaScript files and exporting functionalities. For

**Third-Party Modules**

These are modules that you can install from npm (Node Package Manager).



1. name:

- This is the name of your package or project. It should be lowercase and can contain hyphens or underscores. It must be unique if you plan to publish it to the npm registry.

2. version:

- This specifies the version of your package using semantic versioning (semver). It follows the format major.minor.patch. For example, 1.0.0 indicates the first stable version.

3. description:

- A short description of your project. This is often used when publishing to npm, helping others understand what your package does.

4. type:

- Setting this to "module" allows you to use ES module syntax (i.e., import and export) in your .js files without needing to use the .mjs extension.

5. main:

- This specifies the entry point of your application. In your case, index.js is the file that will be loaded when your package is required by another module.

6. scripts:

- This section defines command-line scripts that you can run using npm.

- test: This script is a placeholder that echoes an error message if you run npm test. It's common to see this in new projects.

- start: This script runs nodemon index.js, which starts your application and automatically restarts it when file changes are detected.

7. keywords:

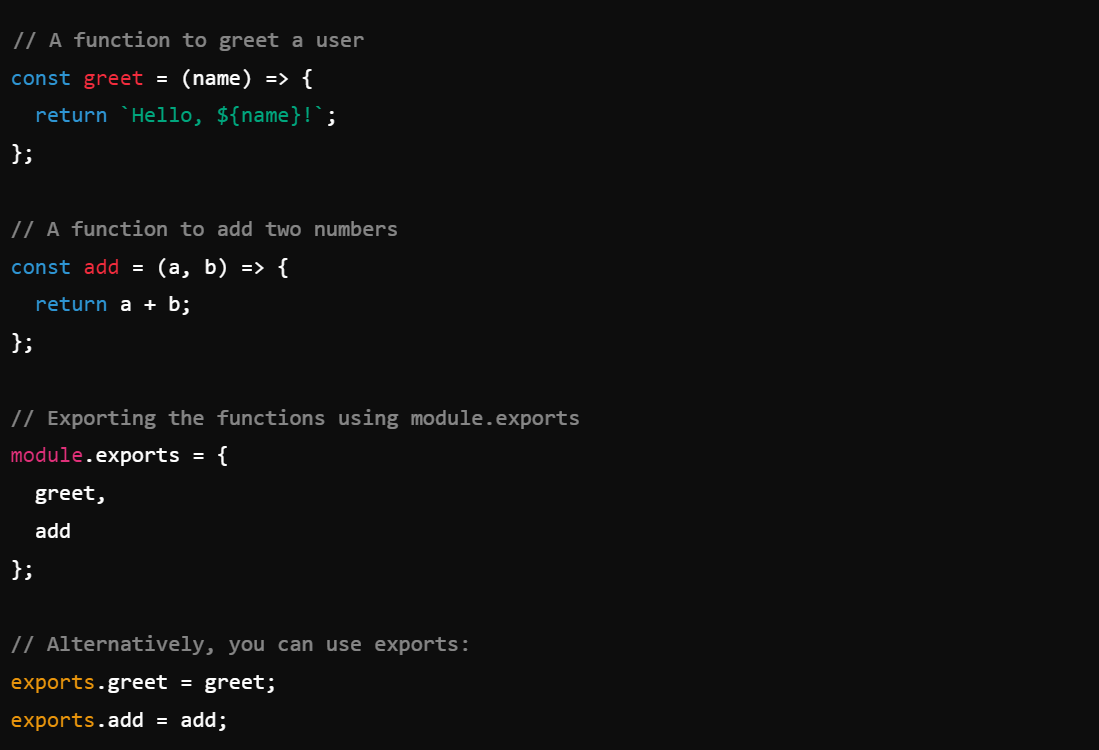
- An array of keywords that relate to your project. These can help others find your package in the npm registry. It’s often left empty in personal projects.

8. author:

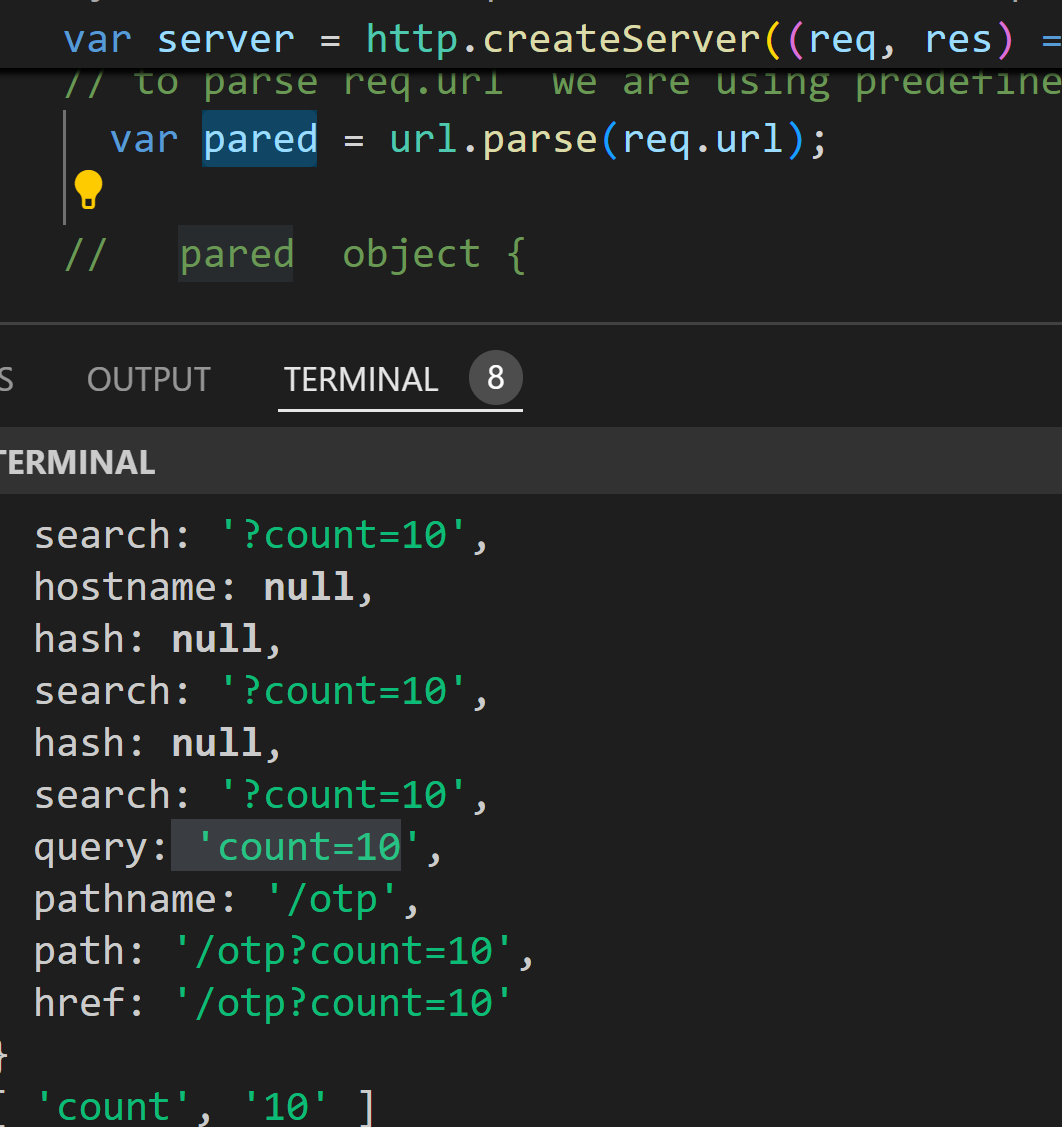
- This field can specify the author of the package. It can be a name or an email address, but it's often left blank in personal projects.

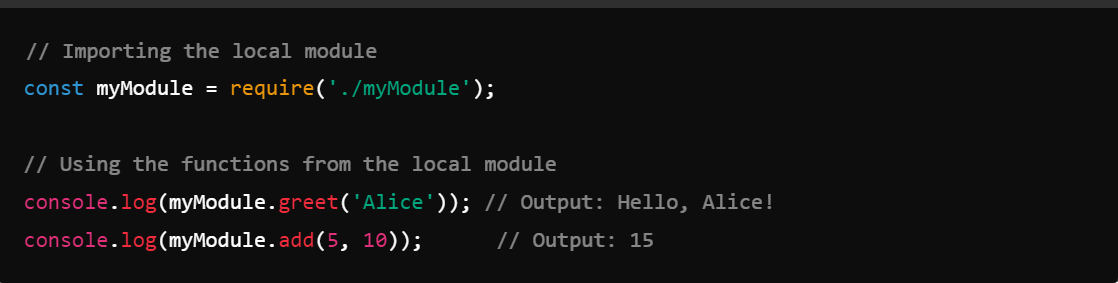
9. license:

- This field specifies the license under which your package is distributed. ISC is a permissive license, similar to the MIT license.









**Overview of the HTTP Module**

1. **Core Module**: The HTTP module is built into Node.js, so you don’t need to install any additional packages.
2. **Creating Servers and Clients**: It provides functionality to create web servers that can handle incoming requests and send responses.

**Basic Concepts**

1. **Server**: A Node.js application that listens for HTTP requests and sends responses.
2. **Client**: The entity (like a web browser) that makes HTTP requests to the server



Important Methods and Properties for `req` (Request)

1. req.method:

- Contains the HTTP method used for the request (e.g., GET, POST, PUT, DELETE).

- Example: `console.log(req.method);`

2. req.url:

- Contains the URL of the request, which can be used to route requests or serve different content.

- Example: `console.log(req.url);`

3. req.headers:

- An object containing the headers sent by the client.

- Example: `console.log(req.headers);`

4. req.on(event, callback):

- Allows you to listen for events on the request stream, such as 'data' and 'end'

5.req.pipe(destination):

- Pipes the readable stream from the request to a writable stream (useful for uploading files).

1. res.statusCode:

- Sets the HTTP status code for the response.

- Example: `res.statusCode = 404;`

2. res.setHeader(name, value):

- Sets a specific header in the response.

- Example: `res.setHeader('Content-Type', 'application/json');`

3. res.getHeader(name):

- Retrieves the value of a specified response header.

- Example: `const contentType = res.getHeader('Content-Type');`

4. res.removeHeader(name):

- Removes a specified header from the response.

- Example: `res.removeHeader('X-Powered-By');`

5. res.write(chunk):

- Sends a chunk of data to the client. You can use this method to send parts of the response before calling `res.end()`.

- Example:

```javascript

res.write('Hello, ');

res.write('World!');

```

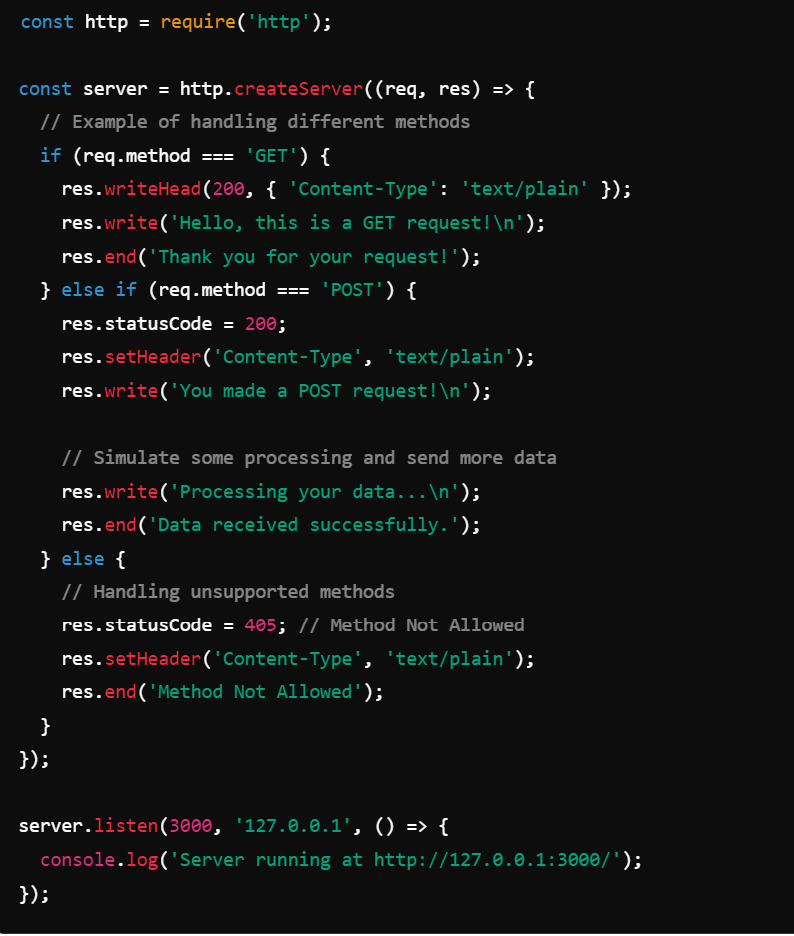
6. res.end(data):

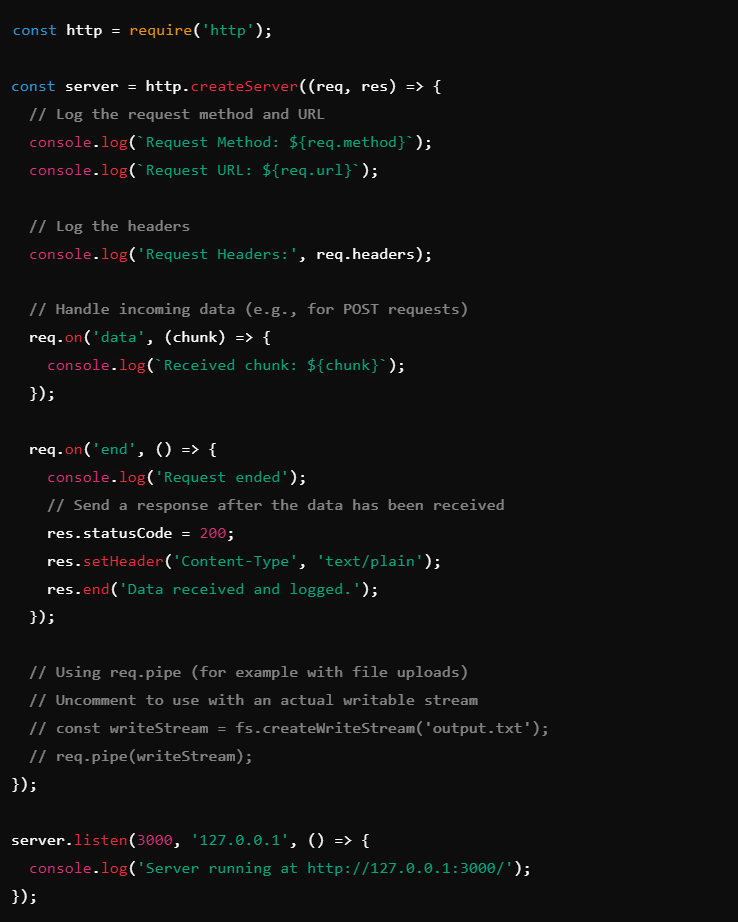
- Signals that the response is complete. You can also send a final piece of data.

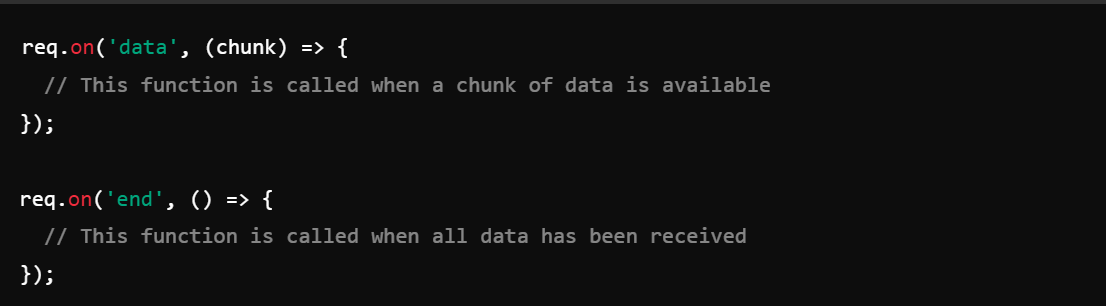
- Example: `res.end('Goodbye!');`

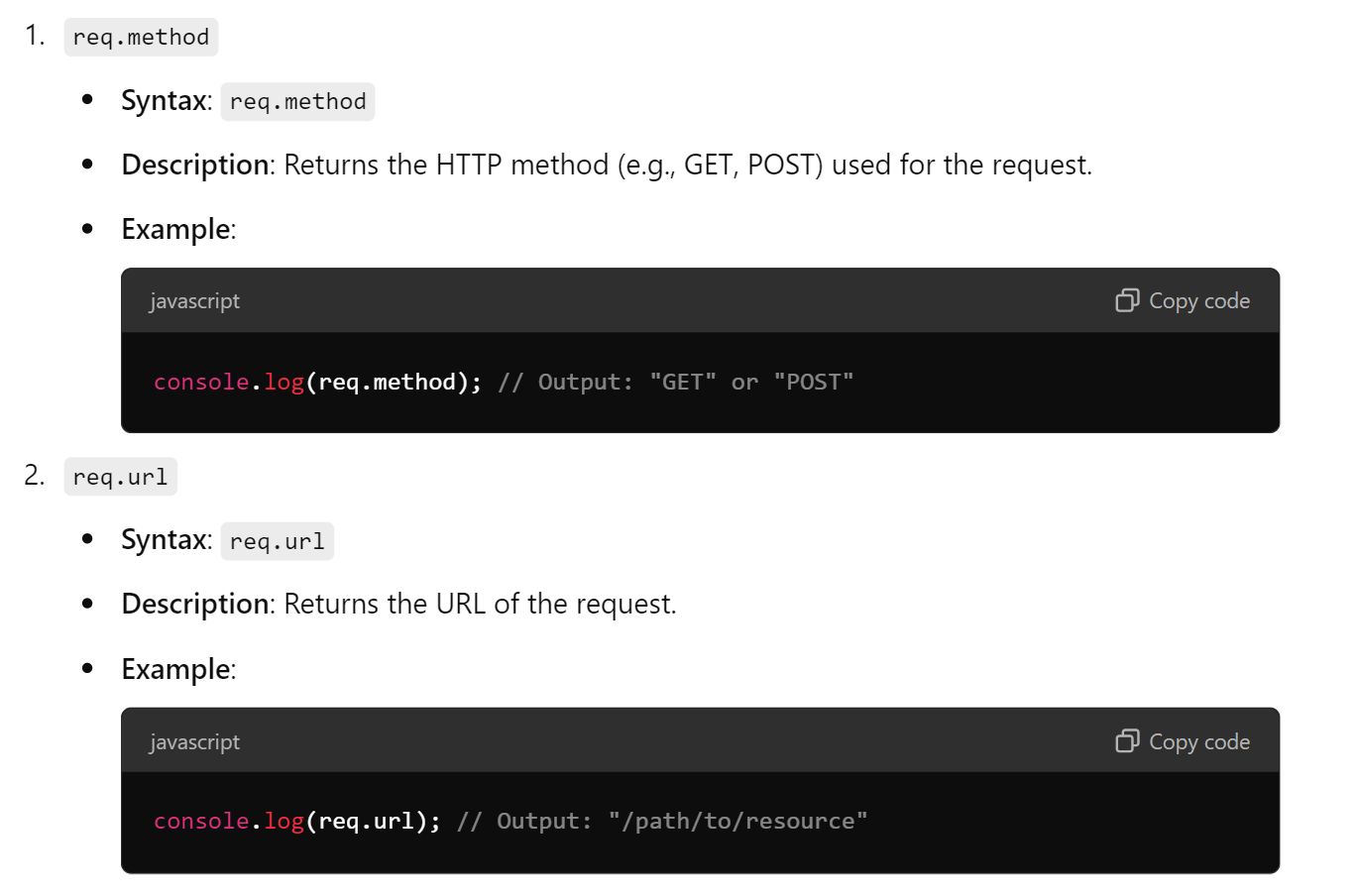
7. res.writeHead(statusCode, [headers]):

- Sets the status code and headers for the response in one go.



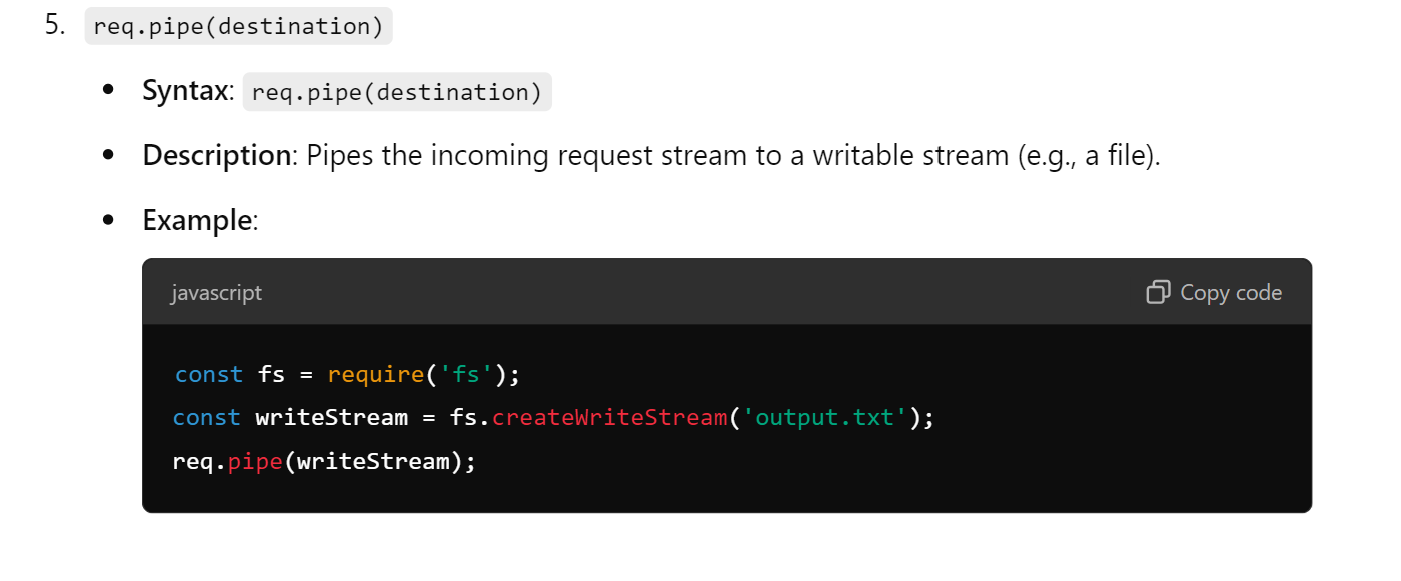




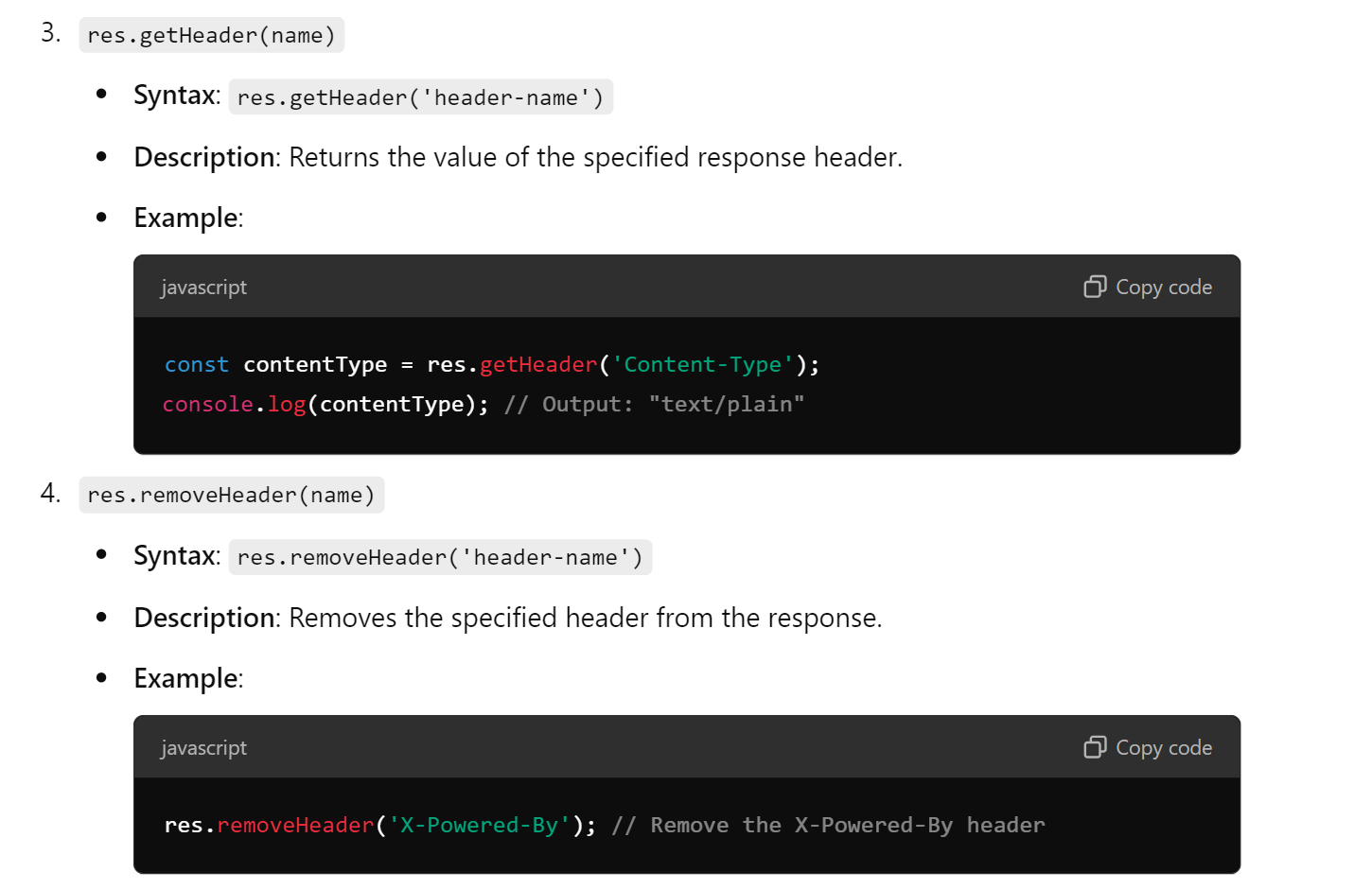


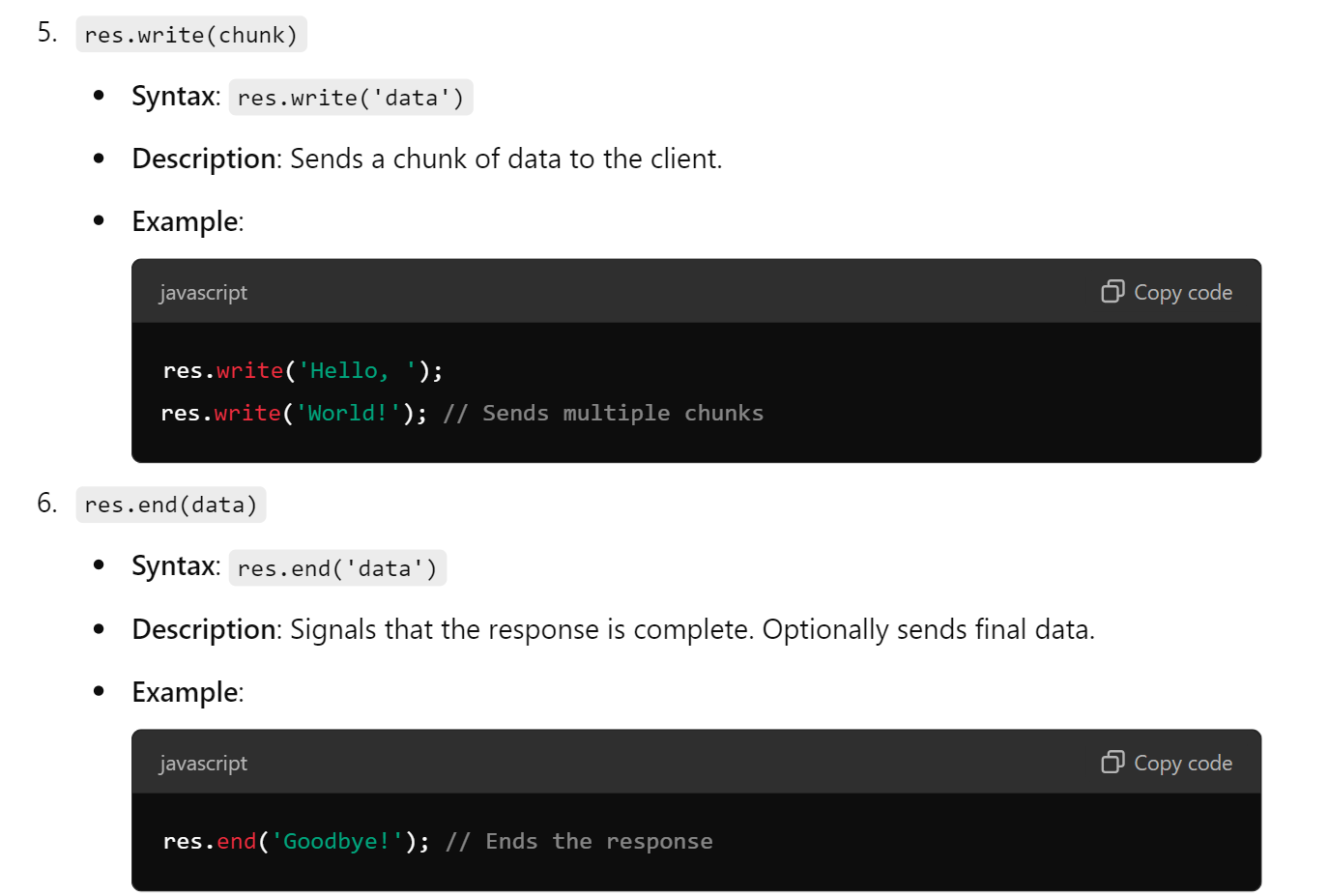








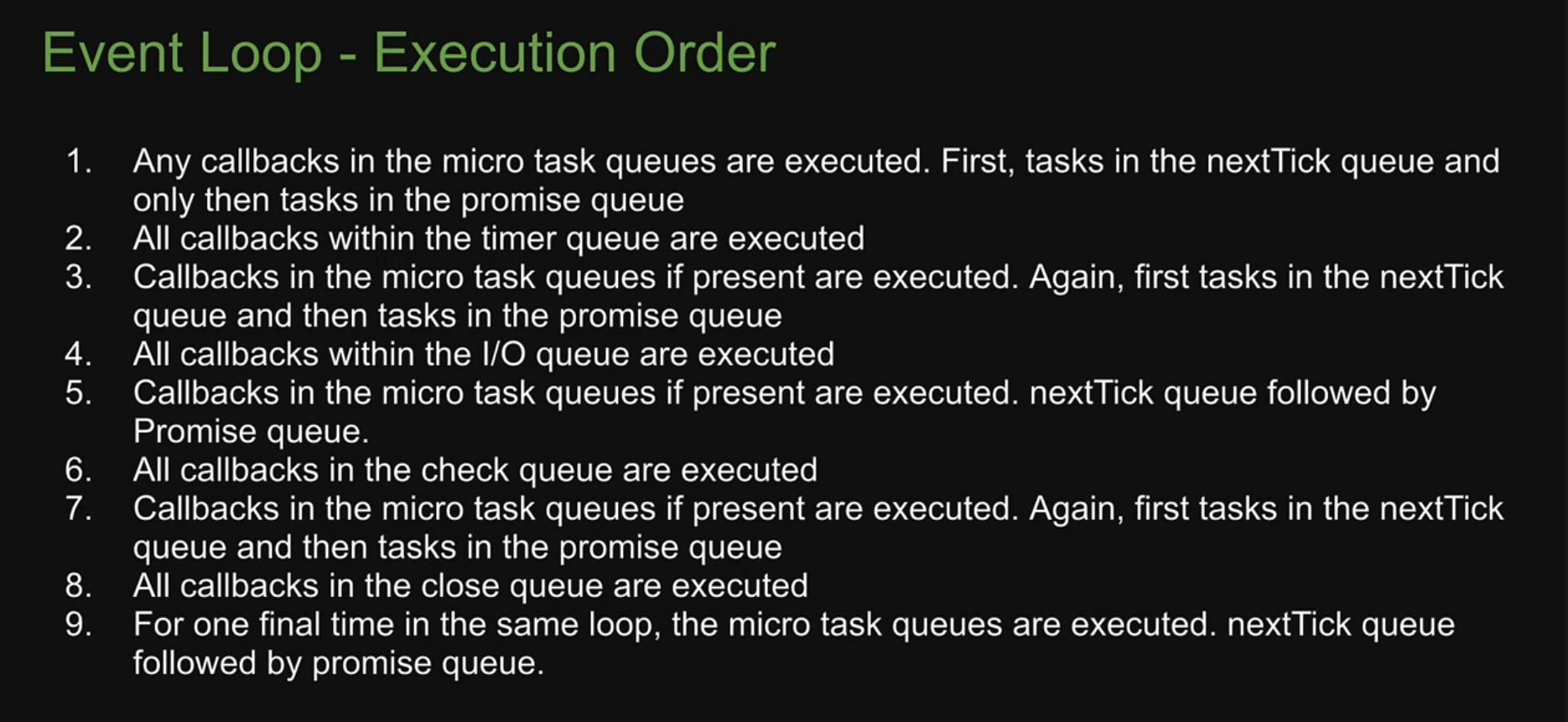


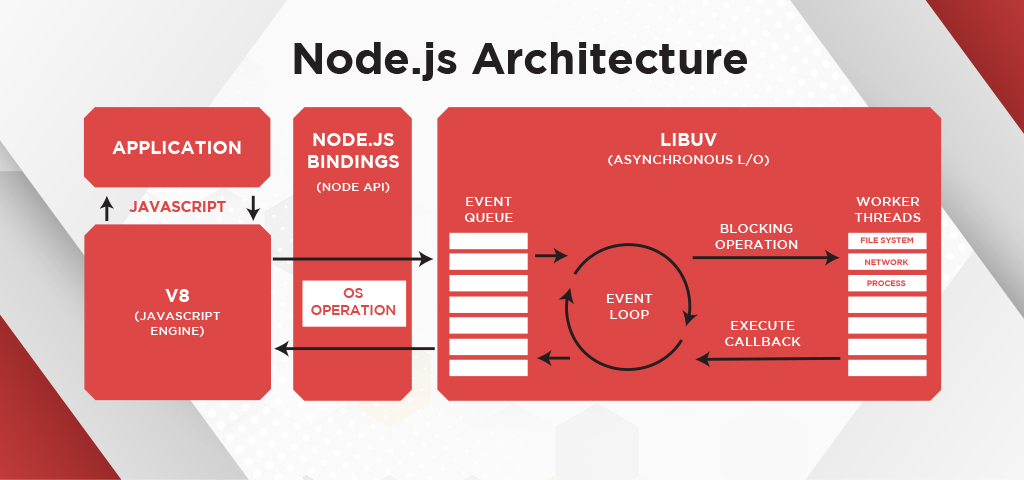


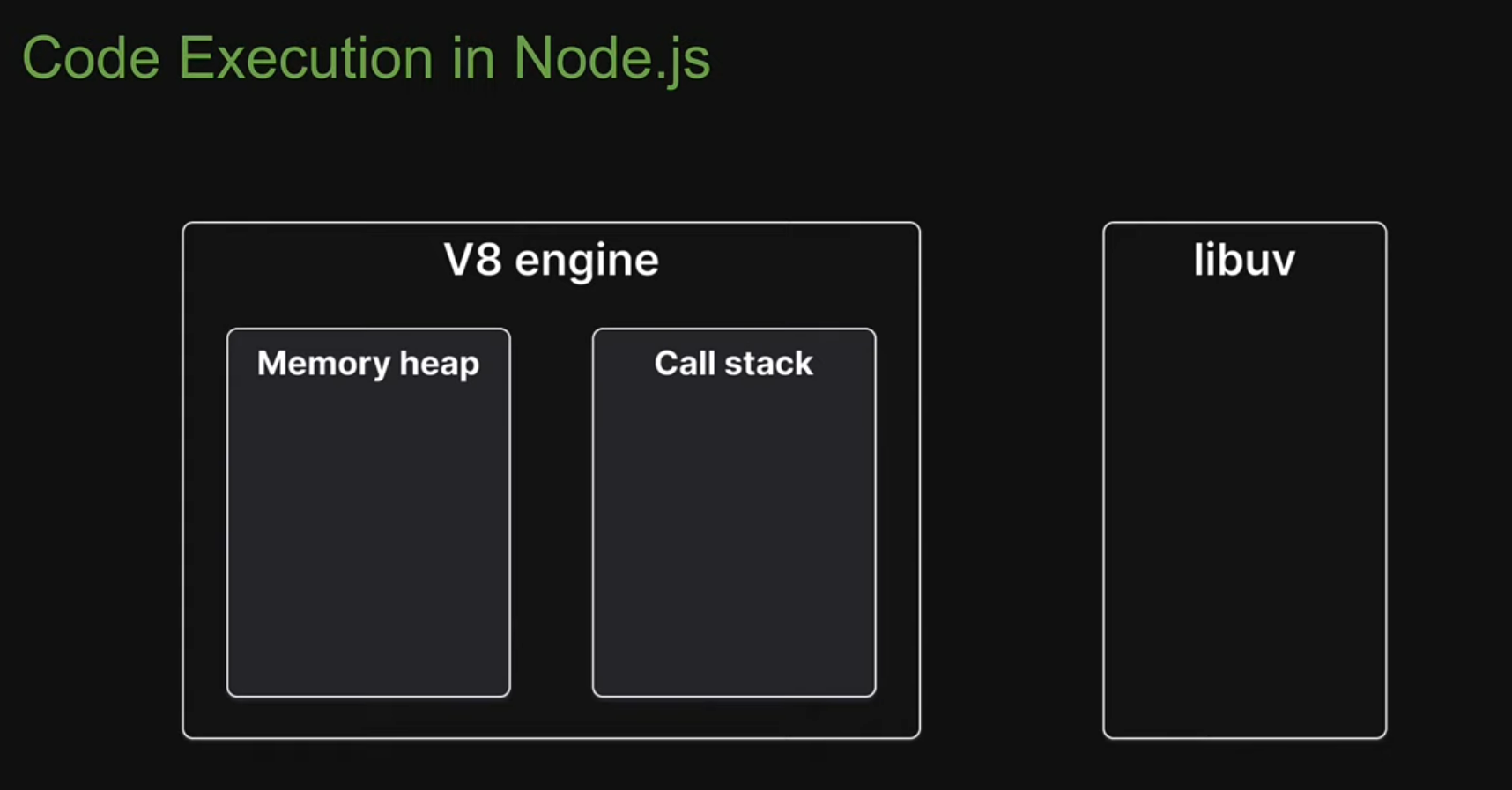


"scripts": { "start": "concurrently \"json-server --watch db.json --port 3000\" \"node yourNodeFile.js\"" }









**1. Introduction to Postman**

* **What is Postman?**: Explain that Postman is a collaboration platform for API development that helps developers build, test, and document APIs.
* **Purpose**: Emphasize its role in simplifying the process of working with APIs by providing a user-friendly interface.

**2. Core Features**

* **API Requests**: Show how to create different types of requests (GET, POST, PUT, DELETE) and explain their purposes.
* **Collections**: Discuss how users can organize requests into collections for better management and sharing.
* **Environment Variables**: Explain how to set up different environments (like development, testing, production) to easily switch contexts.
* **Testing and Automation**: Introduce how Postman allows users to write tests for API responses, ensuring they meet expected criteria.

**Query Parameters**

Query parameters are key-value pairs added to the end of a URL, typically following a question mark (?). They are commonly used to pass data to a web server or to specify certain options or filters when making a request.

**Why are Query Parameters Used?**

1. **Data Filtering**: They allow clients to filter results or request specific information. For example, in an API request for a list of products, you might use query parameters to filter by category or price range.

Example: /api/products?category=electronics&priceRange=100-500

1. **Pagination**: Query parameters can control pagination in API responses. They help in fetching a specific subset of data.

Example: /api/products?page=2&limit=10 (to get the second page of results with 10 items per page).

1. **Search Queries**: They can represent search terms in a query, enabling users to search for specific data.

Example: /search?query=laptop

1. **Sorting**: Query parameters can specify how results should be sorted, such as by date or price.

Example: /api/products?sort=price&order=asc

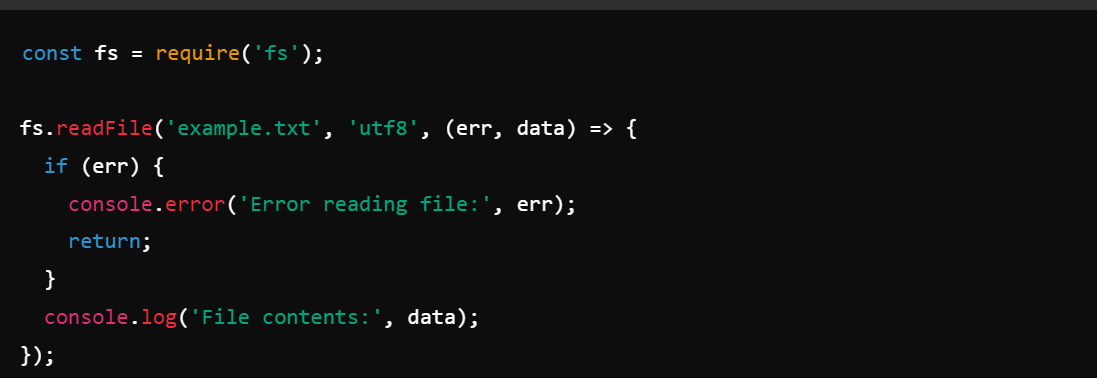
1. **State Management**: They can maintain state information for web applications, such as user preferences or session identifiers.

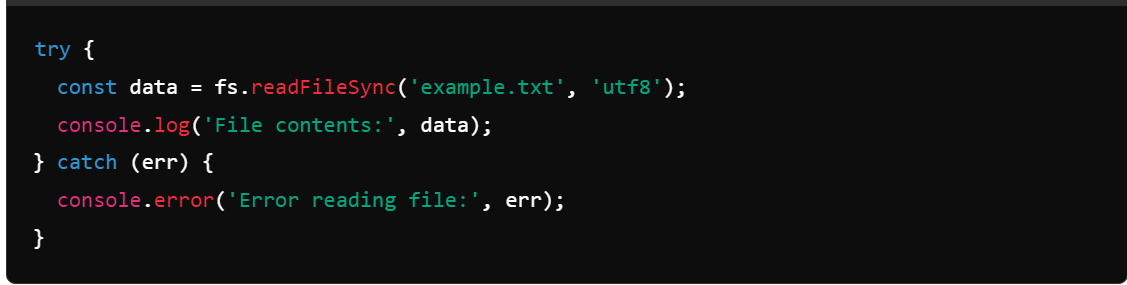
**Introduction to the fs Module**

The fs (File System) module in Node.js provides an API to interact with the file system. It allows us to perform operations such as reading, writing, updating, and deleting files and directories. Understanding how to use this module is essential for tasks like data storage, configuration management, and much more.

**Key Concepts**

1. **Asynchronous vs. Synchronous**:
   * **Asynchronous** methods allow other code to run while waiting for the file operation to complete. This is generally preferred in Node.js for better performance and responsiveness.
   * **Synchronous** methods block the execution of code until the operation completes. They can lead to performance bottlenecks, especially in server applications.
2. **Callbacks**: Most fs methods use callbacks to handle the results of file operations. This means we pass a function as an argument that gets called once the operation is done.
3. Reading Files

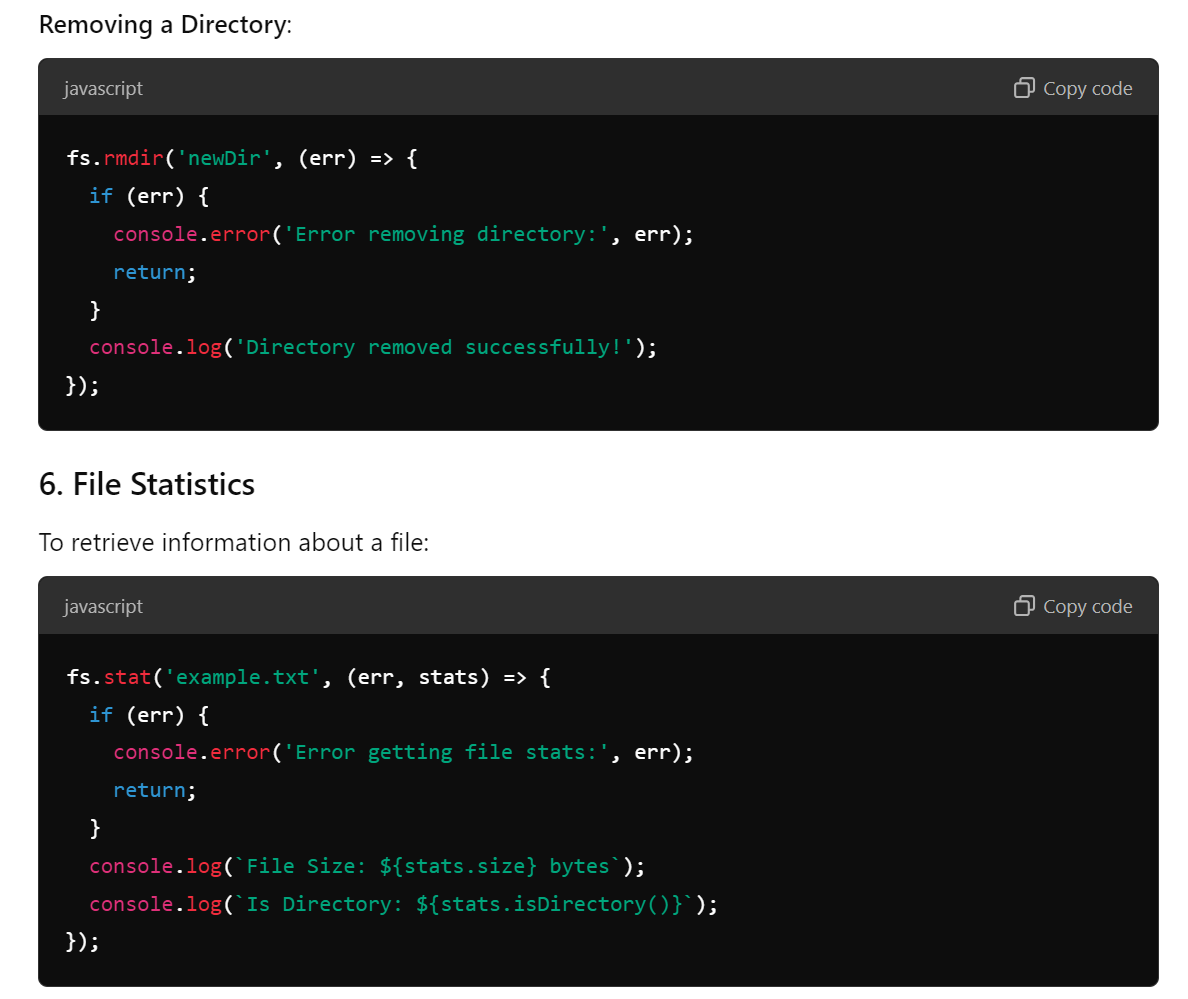


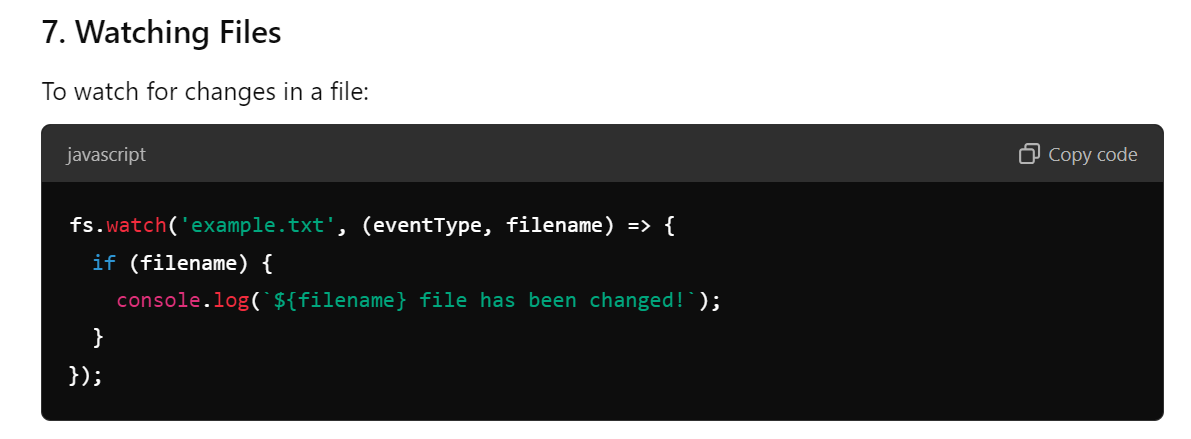








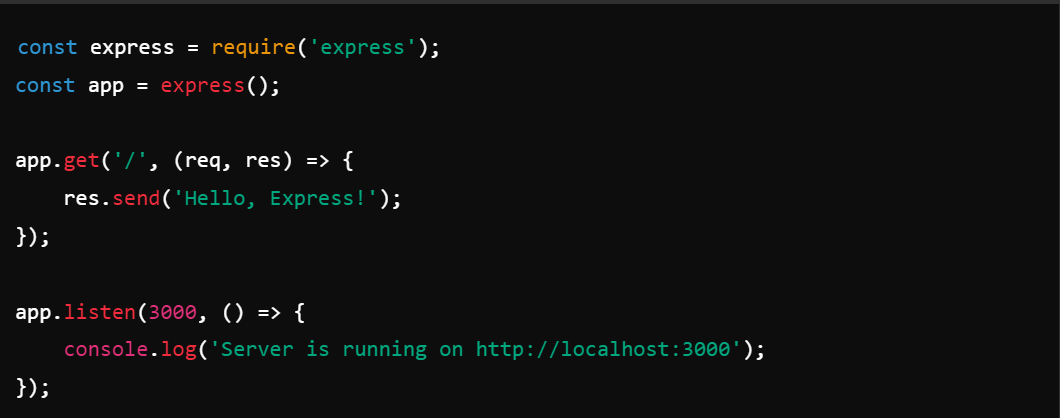




Express

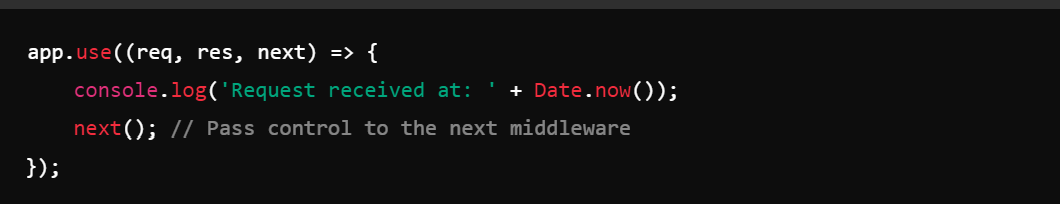
**1. Introduction to Express.js**

* **What is Express.js?**
  + A minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.
* **Advantages of Express:**
  + Fast and lightweight.
  + Middleware support for handling requests.
  + Easy to set up and scale.
  + Robust routing system.



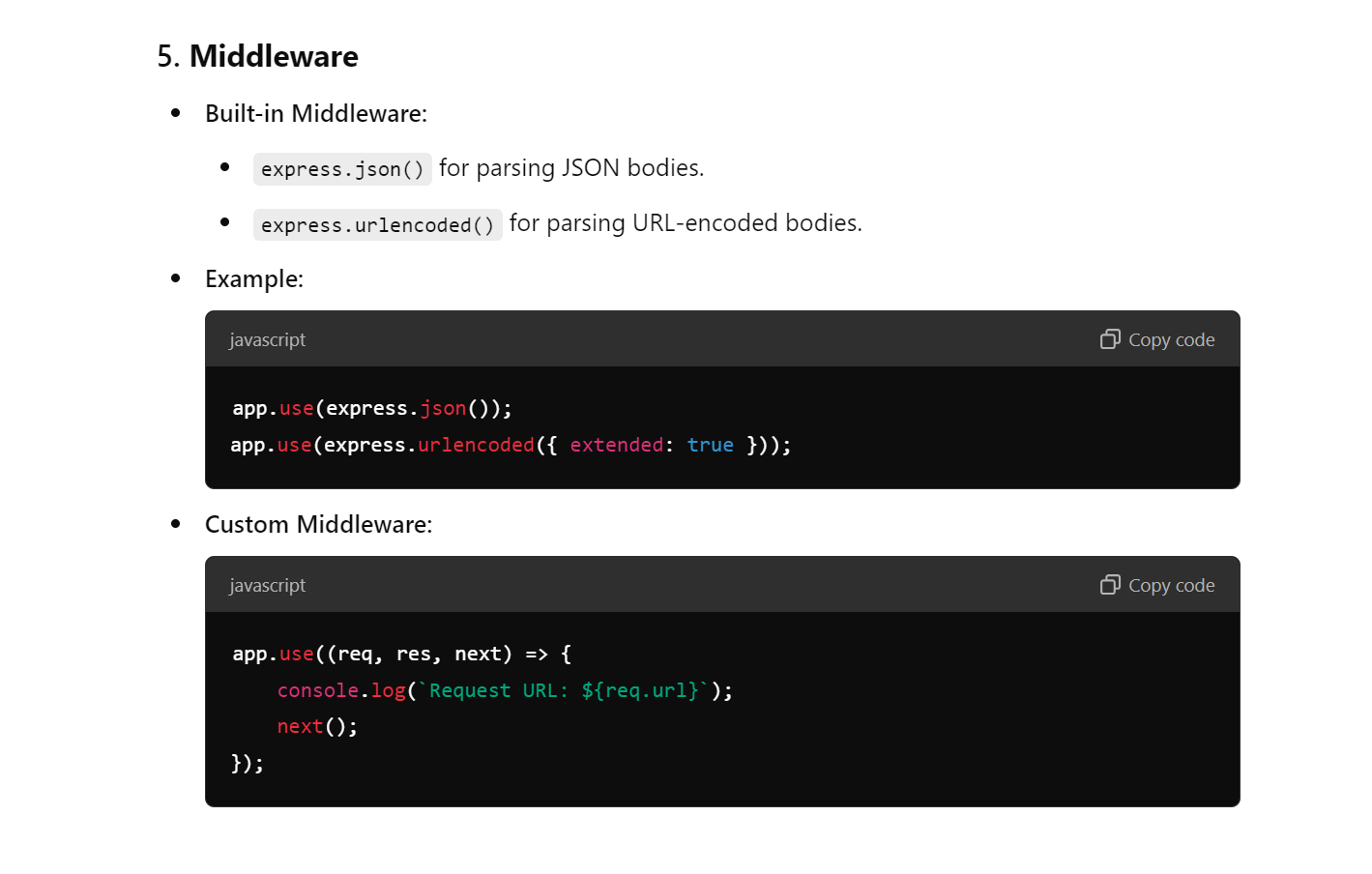
**Understanding Middleware:**

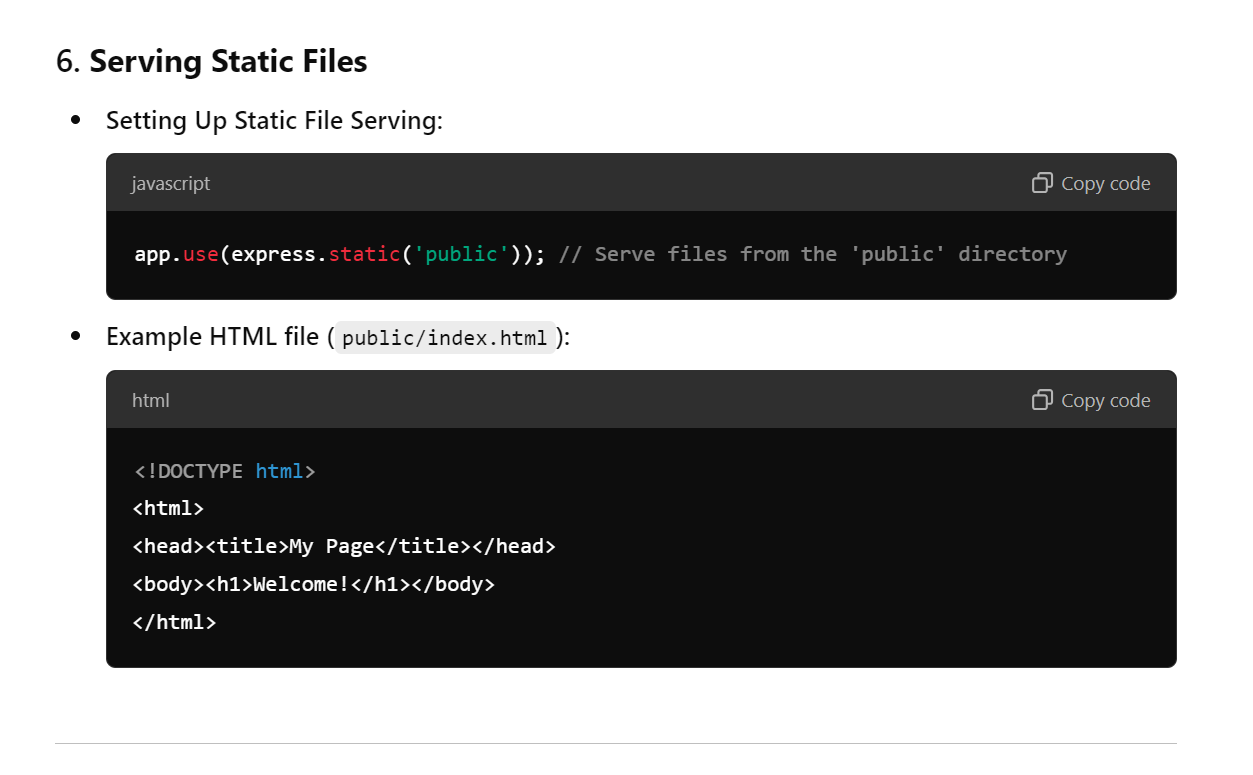
* Functions that execute during the request-response cycle.
* Can modify the request, response, or end the request-response cycle.



**Routing**

* **Defining Routes:**
  + Use different HTTP methods (GET, POST, etc.) to define routes.







Course Outline for Express.js

1. Introduction to Express.js

- What is Express.js?

- Advantages of using Express in Node.js applications

- Differences between Express and other frameworks (e.g., Koa, Hapi)

2. Setting Up the Environment

- Installing Node.js and npm

- Creating a new project and installing Express

- Directory structure of a typical Express application

3. Basic Concepts

- Understanding middleware

- Setting up a simple server

- Handling requests and responses

- Route definitions and HTTP methods (GET, POST, PUT, DELETE)

4. Routing

- Defining routes

- Route parameters and query strings

- Nested routes

- Using Express Router for modular routing

5. Middleware

- What is middleware?

- Built-in middleware (e.g., express.json(), express.urlencoded())

- Custom middleware functions

- Error-handling middleware

6. Serving Static Files

- Setting up static file serving

- Serving HTML, CSS, JavaScript, and images

- Using a template engine (e.g., Pug, EJS)

7. Working with JSON and Form Data

- Sending and receiving JSON data

- Handling form submissions

- Using middleware for file uploads (e.g., multer)

8. Error Handling

- Best practices for error handling in Express

- Custom error handling middleware

- Using try-catch for async routes

9. Connecting to a Database

- Introduction to databases (MongoDB, mySQL, etc.)

- Using Mongoose with MongoDB or Sequelize with SQL databases

- CRUD operations with the database

10. Authentication and Authorization

- Introduction to authentication methods (JWT, OAuth)

- Setting up user authentication with Passport.js or similar

- Securing routes based on user roles

11. Testing and Debugging

- Introduction to testing frameworks (Mocha, Chai, Jest)

- Writing tests for routes and middleware

- Debugging techniques in Express

12. Deployment

- Preparing an Express app for production

- Hosting options (Heroku, Vercel, DigitalOcean)

- Environment variables and configuration

13. Final Project

- Building a simple application using Express (e.g., a blog, a to-do app)

- Presenting the projects to the class

- Code reviews and feedback

Request Methods (`req`)

1. GET Retrieve data from the server.

- Example

javascript

app.get("/api/users", (req, res) => {

// Logic to get users

res.json(users); // Respond with JSON

});

2. POST Send data to the server to create a new resource.

- Example

javascript

app.post("/api/users", (req, res) => {

const newUser = req.body; // Get the new user data from the request body

users.push(newUser); // Add user to the array

res.status(201).json(newUser); // Respond with the created user

});

3. PUT Update an existing resource.

- Example

javascript

app.put("/api/users/:id", (req, res) => {

const userId = req.params.id; // Get user ID from URL

const updatedUser = req.body; // Get updated data

// Logic to update user in the array

res.json(updatedUser); // Respond with updated user

});

4. DELETE Remove a resource from the server.

- Example

javascript

app.delete("/api/users/:id", (req, res) => {

const userId = req.params.id; // Get user ID from URL

// Logic to delete user from the array

res.status(204).send(); // Respond with no content

});

5. PATCH Partially update an existing resource.

- Example

javascript

app.patch("/api/users/:id", (req, res) => {

const userId = req.params.id; // Get user ID from URL

const updates = req.body; // Get updates

// Logic to apply updates to the user

res.json(updatedUser); // Respond with updated user

});

Response Methods (`res`)

1. res.send() Send a response of various types (string, buffer, object, etc.).

- Example

javascript

app.get("/text", (req, res) => {

res.send("Hello, World!"); // Send a plain text response

});

2. res.json() Send a JSON response.

- Example

javascript

app.get("/api/data", (req, res) => {

const data = { message: "This is JSON data." };

res.json(data); // Send a JSON response

});

3. res.status() Set the HTTP status code for the response.

- Example

javascript

app.get("/notfound", (req, res) => {

res.status(404).send("Not Found"); // Send a 404 response

});

4. res.redirect() Redirect the client to a different URL.

- Example

javascript

app.get("/old-route", (req, res) => {

res.redirect("/new-route"); // Redirect to a new route

});

5. res.render() Render a view template (when using a templating engine).

- Example

javascript

app.get("/profile", (req, res) => {

res.render("profile", { user: req.user }); // Render a profile view

});

Request Methods (`req`)

1. req.params: Access route parameters.

- Usage: `const userId = req.params.id;`

2. req.query: Access query string parameters.

- Usage: `const page = req.query.page;`

3. req.body: Access data sent in the request body (requires body-parser or express.json()).

- Usage: `const userData = req.body;`

4. req.headers: Access request headers.

- Usage: `const authToken = req.headers['authorization'];`

5. req.method: Get the HTTP method of the request.

- Usage: `const method = req.method;`

6. req.url: Get the requested URL.

- Usage: `const url = req.url;`

---

Response Methods (`res`)

1. res.send(): Send a response of various types (string, buffer, object, etc.).

- Usage: `res.send('Hello, World!');`

2. res.json(): Send a JSON response.

- Usage: `res.json({ key: 'value' });`

3. res.status(): Set the HTTP status code for the response.

- Usage: `res.status(404).send('Not Found');`

4. res.redirect(): Redirect the client to a different URL.

- Usage: `res.redirect('/new-route');`

5. res.render(): Render a view template (when using a templating engine).

- Usage: `res.render('templateName', { data: 'value' });`

6. res.sendFile(): Send a file as a response.

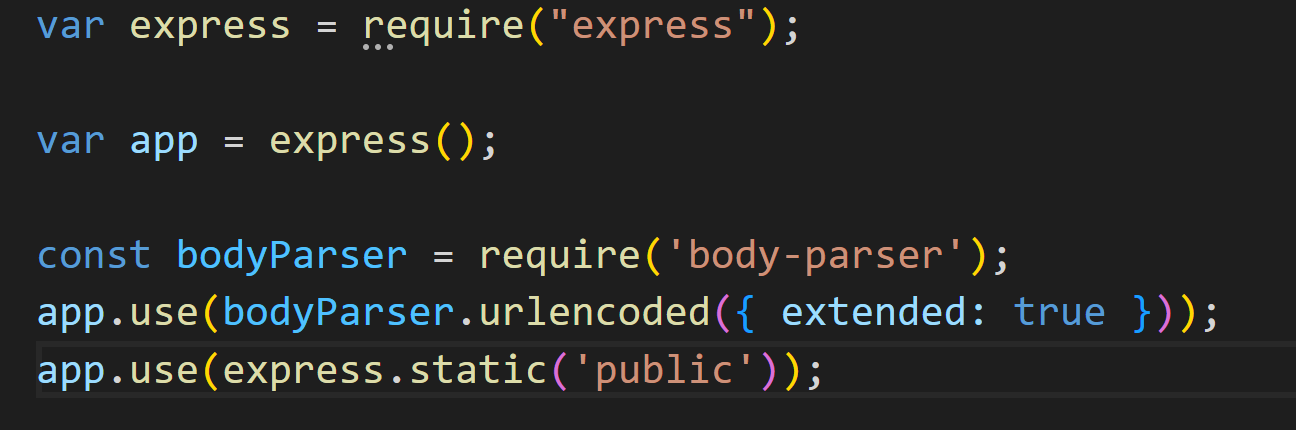
- Usage: `res.sendFile('/path/to/file');`

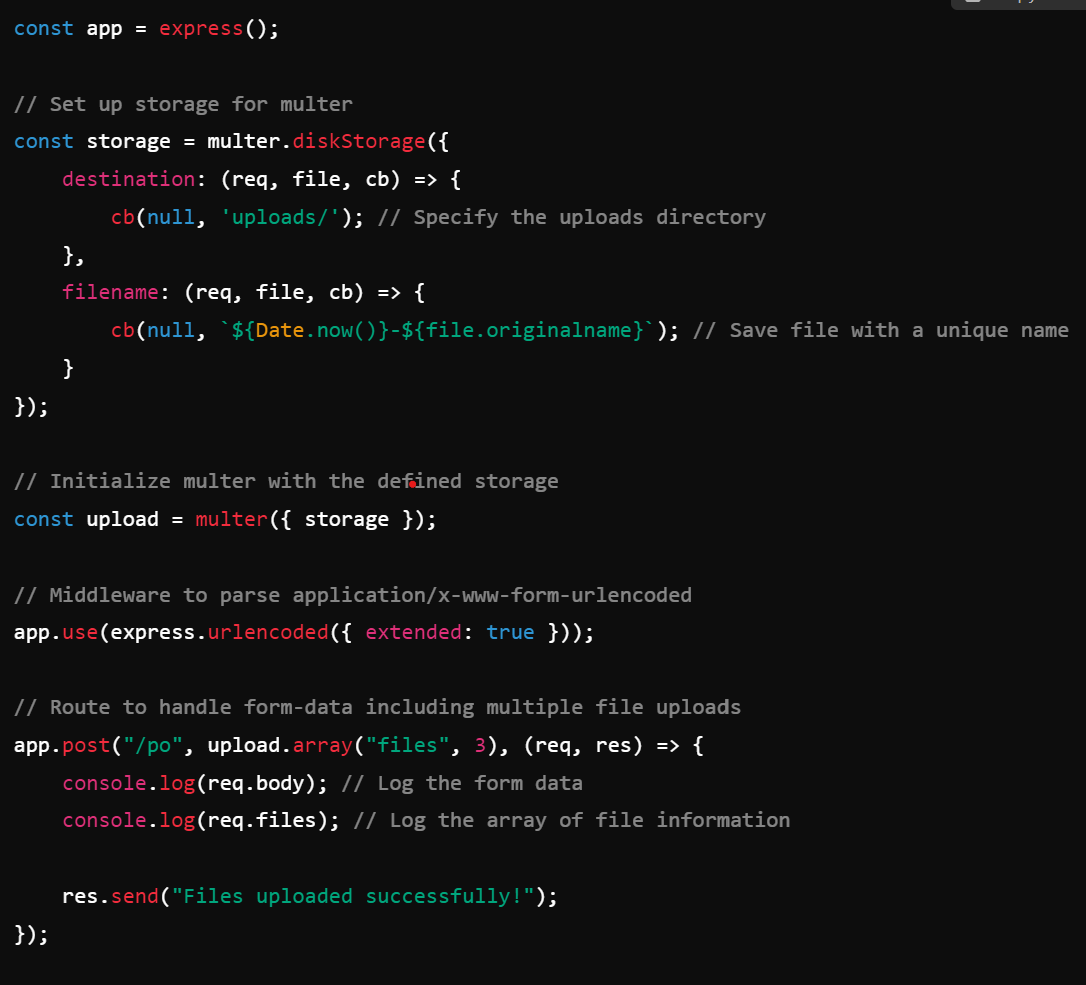
7. res.cookie(): Set a cookie on the response.

- Usage: `res.cookie('name', 'value');`

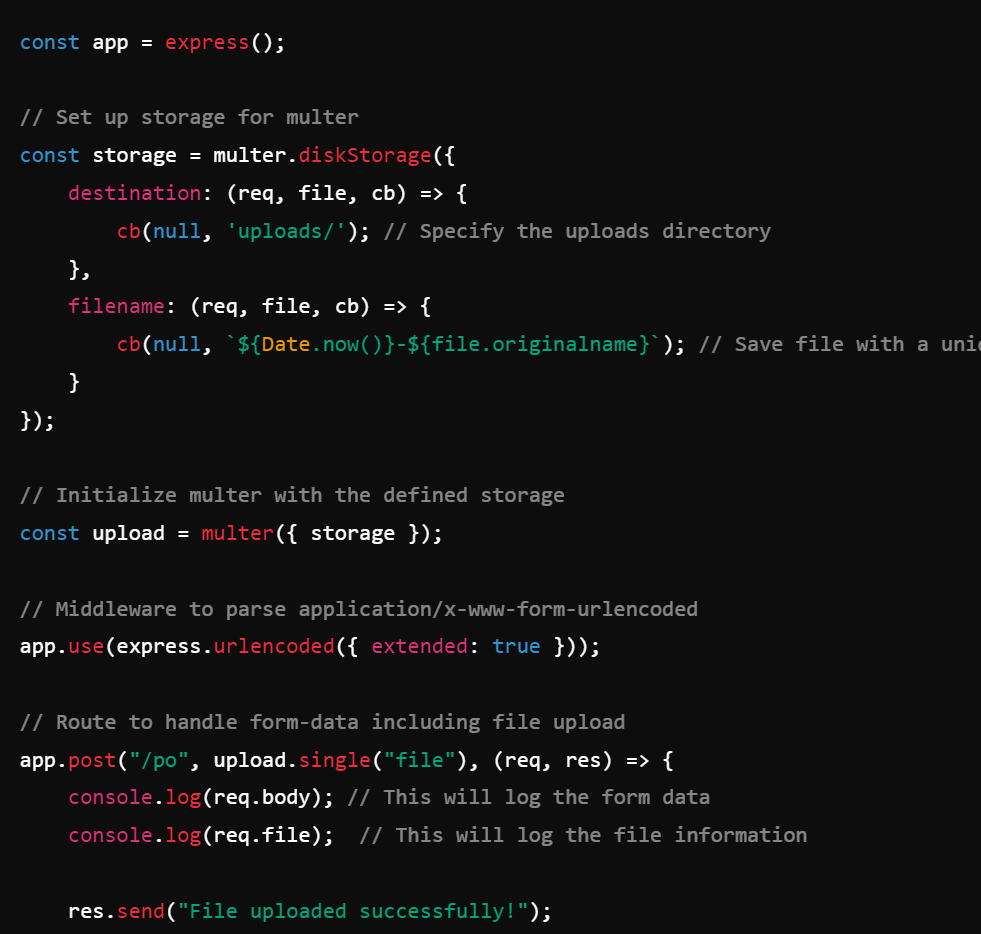
8. res.clearCookie(): Clear a cookie.

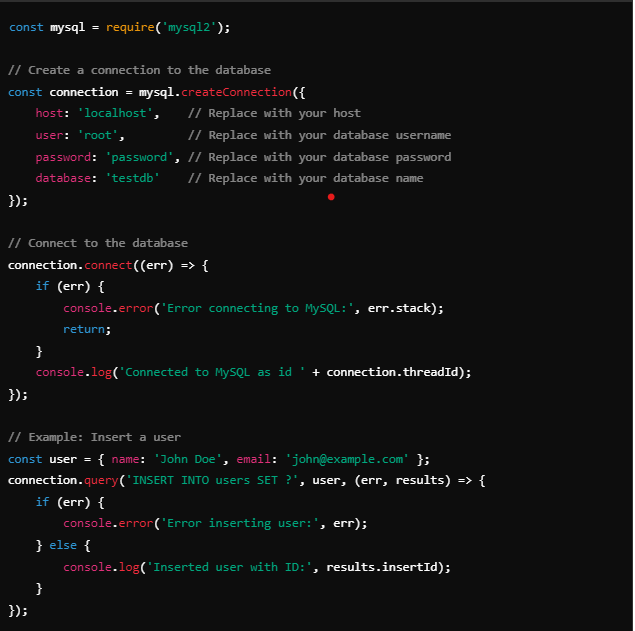
- Usage: `res.clearCookie('name');`

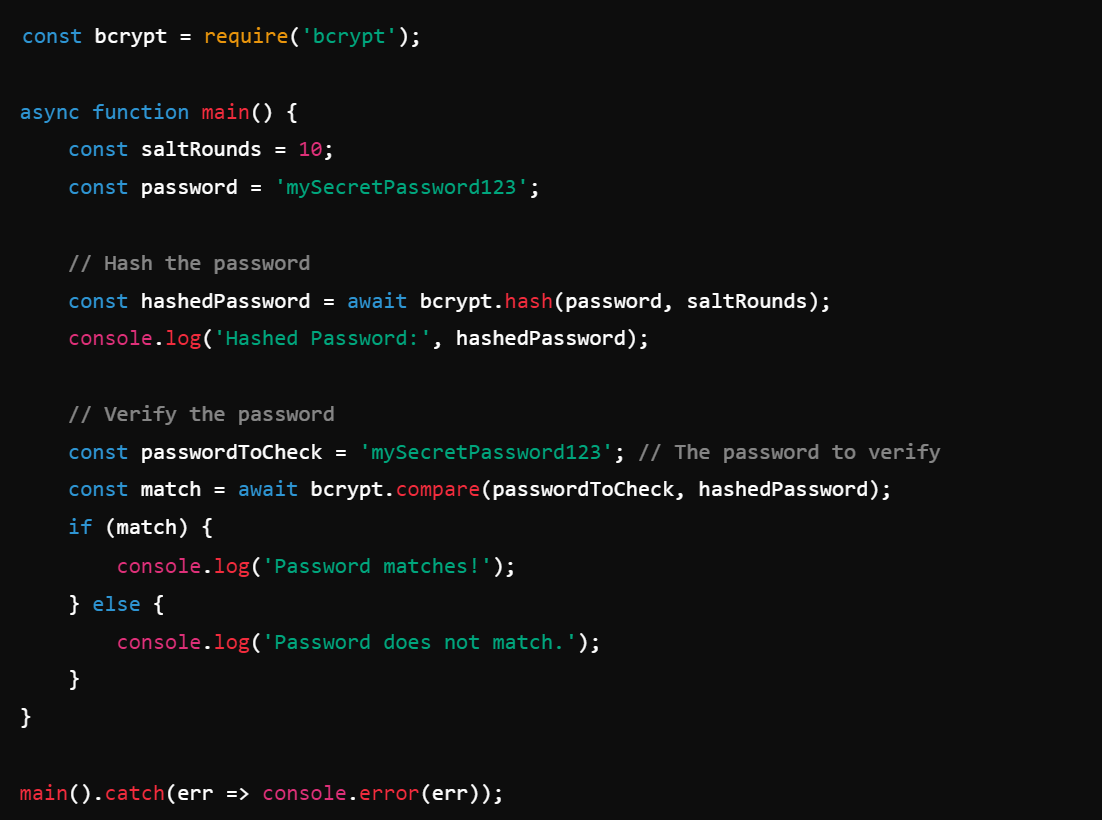












Middleware in Node.js, particularly in the context of Express, refers to functions that have access to the request and response objects in an application. Middleware can perform various tasks, such as executing code, modifying the request and response objects, ending the request-response cycle, or calling the next middleware in the stack.

const loggerMiddleware = (req, res, next) => { console.log(`${req.method} request for '${req.url}'`); next(); // Pass control to the next middleware };

In Express, the express.urlencoded({ extended: true }) middleware is used to parse incoming requests with URL-encoded payloads. The extended option determines how the parsing is done:

* **extended: true**: This option uses the qs library to parse the URL-encoded data, allowing for rich objects and arrays. It supports nested objects, so you can send complex data structures.
* **extended: false**: This option uses the querystring library to parse the data. It only supports simpler key-value pairs and does not handle nested objects or arrays.

**1. application/x-www-form-urlencoded**

* **Default Encoding**: This is the default encoding type for HTML forms.
* **How It Works**: Data is encoded as key-value pairs. Spaces are replaced with + signs, and special characters are URL-encoded (e.g., & becomes %26).
* **Use Case**: Best for simple forms that only contain text fields and do not require file uploads. It works well for most standard form submissions.

**2. multipart/form-data**

* **Used for File Uploads**: This encoding type is required when a form includes file uploads.
* **How It Works**: Data is divided into parts, each with its own content type, and it can include binary data. Each field is separated by a boundary string.
* **Use Case**: Ideal for forms that include file input elements (like <input type="file">) alongside regular text fields.

**Authentication**

Authentication is the process of verifying the identity of a user, device, or system. It typically involves checking credentials such as usernames and passwords, biometric data, or security tokens. The goal is to ensure that the entity trying to access a system is who they claim to be.

**Authorization**

Authorization, on the other hand, determines what an authenticated user is allowed to do. Once a user’s identity has been verified, the system checks their permissions to decide what resources they can access and what actions they can perform (like read, write, or delete data).

Tokens

What Are Tokens?

Think of a Token as a Ticket

- Imagine you’re going to a concert. When you buy your ticket, it gives you access to the event. The ticket has your name and shows that you’ve paid to get in.

- Similarly, a token is like a digital ticket that proves you’re allowed to access certain parts of a website or application.

Why Do We Use Tokens?

1. Access Control

- Just like you can’t get into the concert without a ticket, you can’t access certain features of an app without a token. It tells the app, "This user has permission."

2. No Need to Remember Passwords

- Once you have your concert ticket, you don’t need to keep showing your ID to get in again. With tokens, after you log in once, you don’t have to keep entering your password for every action you want to perform.

3. Safety

- If someone loses their concert ticket, they can’t get in without it. Similarly, tokens expire after a certain time, which keeps things secure. If someone tries to use a stolen token after it expires, it won’t work.

4. Easy to Use Across Different Places

- If you have a ticket for a music festival, you can use it for different stages. Tokens work similarly; you can use the same token to access different parts of an app or even different apps without logging in again.

Example in Real Life:

- Logging into a Website

- When you log in to a website, it checks your username and password. If they’re correct, the website gives you a token (like a digital ticket).

- You can use that token to move around the site without logging in again until the token expires.

tokens are like digital tickets that help you access certain areas of an app or website. They make it easier for you to log in, keep things secure, and allow you to use multiple features without hassle. Just remember, like a ticket, if it’s lost or expires, you won’t be able to get in