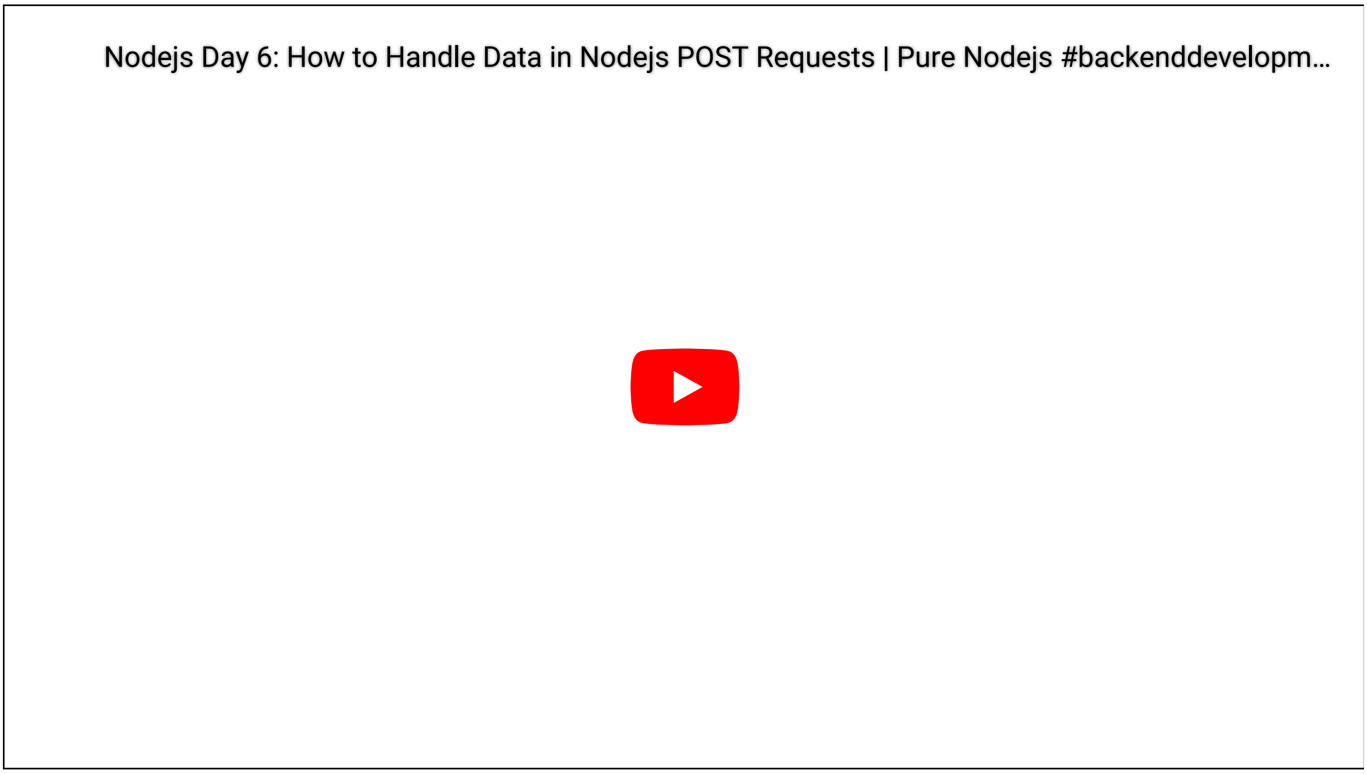
Nodejs Day5: Exploring HTTP Methods — GET, POST, DELETE, PATCH, PUT | #backenddevelopm…



Nodejs Day 6: How to Handle Data in Nodejs POST Requests | Pure Nodejs #backenddevelopm…

1. Introduction to HTTP Methods

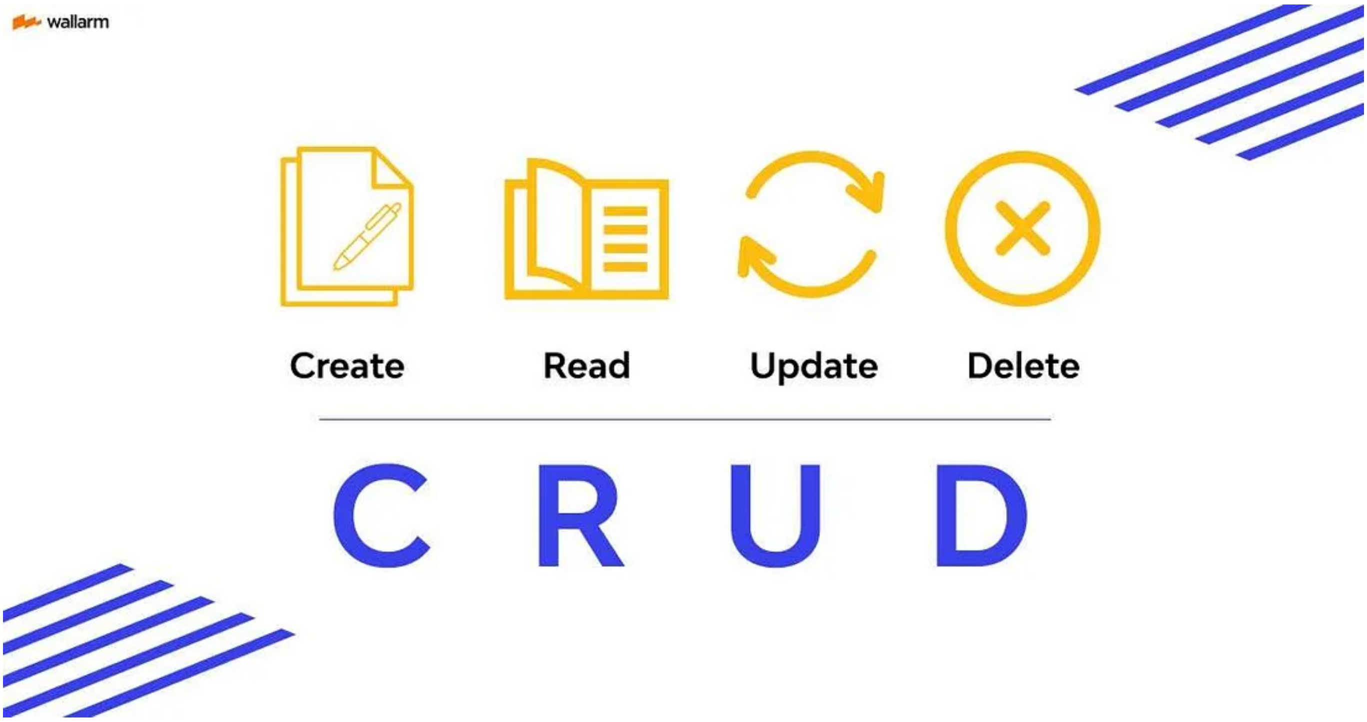
HTTP methods, also known as HTTP verbs, are the foundation of communication between a client (usually a browser) and a server.

They define the type of request being made and what action should be taken.

In web development, you'll commonly encounter six HTTP methods: GET, POST, DELETE, PATCH, PUT, and OPTIONS.

Explaining CRUD Operations

What is CRUD?



CRUD stands for Create, Read, Update, and Delete. It represents the four basic operations you can perform on data in a database or any persistent storage system.

- **Create:** This operation involves adding new data to the system. It's typically done using the HTTP POST method when creating new resources.

PATCH updates specific fields, while PUT replaces the entire resource.

- **Delete:** Removing data from the system. The DELETE HTTP method is used to delete resources.

How HTTP Methods Help Implement CRUD Operations

This Image is from Geekforgeeks

- **Create:** We use the HTTP POST method to create new resources. For example, when a user submits a form on a website, the form data is sent to the server using POST to create a new record in a database.
- **Read:** To retrieve data, we employ the HTTP GET method. When you visit a webpage, your browser sends GET requests to the server to fetch HTML, images, and other resources.

- **Update:** HTTP methods PATCH and PUT are used for updating data. PATCH allows for partial updates, like changing a user's email address. PUT, on the other hand, replaces the entire resource, useful when you need to update all fields.
- **Delete:** The DELETE HTTP method is used for removing resources. When you delete a post on a social media platform, a DELETE request is sent to the server to delete that specific post.

2. Setting Up Node.js

Before we dive into HTTP methods, let's ensure you have Node.js installed. If not, head to nodejs.org and follow the installation instructions. Once Node.js is set up, create a basic server using the following code:

```
const http = require('http');
const server = http.createServer((req, res) => {
  // Handle requests here
});
const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

This code initializes a Node.js server listening on port 3000.

Understanding the GET Method

The HTTP GET method is one of the fundamental ways browsers and servers communicate. It's used when you want to request information from a server, typically to retrieve data like web pages or resources. In simple terms, it's like asking a server to "give me this."

The Code Example

Now, let's dive into the code example to see how a simple GET request handler is created using Node.js. Don't worry if you're new to programming; we'll explain each part carefully.

```
// Import the 'http' module to work with HTTP requests and responses
const http = require('http');

// Create an HTTP server
const server = http.createServer((req, res) => {
  // Check if the request method is 'GET'
  if (req.method === 'GET') {
    // Handle GET request
    // Check if the request URL (the path in the address bar) is '/products'
    if (req.url === '/products') {
      // Create some dummy data for products (like items in an online store)
      const products = [
        { id: 1, name: 'Product A' },
        { id: 2, name: 'Product B' },
        { id: 3, name: 'Product C' },
      ];
      // Respond to the client (the web browser in this case)
      // Set the HTTP status code to 200, which means 'OK'
      // Also, specify that we are sending JSON data
      res.writeHead(200, { 'Content-Type': 'application/json' });
      // Send the products as a JSON string to the client
      res.end(JSON.stringify(products));
    }
  }
});
// Start the server and specify the port it should listen on
const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Explaining the Code

1. We begin by importing the `http` module, which is a part of Node.js. This module allows us to work with HTTP requests and responses.

2. We create an HTTP server using the `http.createServer` method. This server listens for incoming requests and handles them.

3. Inside the server creation function, we check if the request method is ‘**GET**’. This ensures that we’re only interested in handling GET requests.

4. If the request method is ‘GET’, we further check if the request URL is ‘**/products**’. This means we are looking for requests to a specific endpoint (in this case, ‘/products’).

5. If both conditions are met (it’s a GET request and the URL is ‘/products’), we proceed to create some dummy data. In this example, we’re simulating a list of products with their IDs and names.

6. We prepare the response to send back to the client (typically a web browser). We set the **HTTP status code to 200** (*If you don’t know about HTTP status codes, Just scroll down and you will get an Image showing Status codes with respective meanings*), which means ‘OK,’ indicating that the request was successful. We also specify that we’re sending JSON data as the response.

7. Finally, we send the list of products as a JSON string in the response body and end the response.

8. At the end of the code, we start the server and specify the port it should listen on. If you’re running this code locally, it will typically listen on port 3000. You’ll see a log message in the console indicating that the server is running.

> *Note: If you don’t know about HTTP status codes, Just scroll down 👇 and you will get an Image showing Status codes with respective meanings*

Handling POST Requests

The HTTP POST method is used when you want to send data to a server. This method is commonly used when submitting forms on websites or when creating new resources on the server. Think of it as sending information to the server, like when you submit a registration form on a website.

The Code Example

Now, let’s go through the code example that shows how to create a POST request handler using Node.js:

```
  // Import the 'http' module to work with HTTP requests and responses
  const http = require('http');

  // Import the 'parse' function from the 'querystring' module to parse incoming data
  const { parse } = require('querystring');
  // Create an HTTP server
  const server = http.createServer((req, res) => {
```

```
  // Check if the request method is 'POST'
  if (req.method === 'POST') {
    // Handle POST request
    // Check if the request URL (the path in the address bar) is '/feedback'
    if (req.url === '/feedback') {
      // Initialize an empty string to store the incoming data
      let body = '';
      // Listen for data chunks being sent by the client
      req.on('data', (chunk) => {
        // Append the data chunk to the 'body' string
        body += chunk.toString();
      });
      // When all data has been received
      req.on('end', () => {
        // Parse the received data into a usable format (typically an object)
        const feedbackData = parse(body);
        // At this point, you can store or process the received data
        // In this example, we're keeping it simple and sending a success message
        // Prepare the response to send back to the client
        // Set the HTTP status code to 200 (OK)
        // Specify that we are sending plain text as the response
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        // Send a simple success message as the response
        res.end('Feedback submitted successfully');
      });
    }
  }
});
// Start the server and specify the port it should listen on
const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Explaining the Code

1. We start by importing two modules: `http` for handling HTTP requests and responses, and `querystring` to help parse incoming data.

2. Next, we create an HTTP server using `http.createServer()`. This server will listen for incoming requests and handle them accordingly.

3. Inside the server creation function, we first check if the request method is 'POST'. This ensures that we're only interested in handling POST requests.

4. If the request method is indeed 'POST', we further check if the request URL is '/feedback'. This means we're specifically looking for POST requests to the '/feedback' endpoint.

5. If both conditions are met, we proceed to handle the POST request. To do this, we initialize an empty string called `body` to store the incoming data.

6. We listen for data chunks being sent by the client using the `req.on('data', ...)` method. As data arrives, we append it to the `body` string.

7. When all the data has been received (the `req.on('end', ...)` event), we parse the `body` string into a usable format. In this example, we're using the `parse` function from the `querystring` module to convert the data into an object. This allows us to work with the data more easily.

8. At this point, you can perform various actions with the received data, such as storing it in a database or processing it in some way. In our simple example, we're just preparing a response.

9. We prepare the response by setting the HTTP status code to 200 (indicating success) and specifying that we're sending plain text as the response.

10. Finally, we send a plain text success message as the response body and end the response.

11. At the end of the code, we start the server and specify the port it should listen on. If you're running this code locally, it will typically listen on port 3000. You'll see a log message in the console indicating that the server is running.

> *Note: If you don't know about HTTP status codes, Just scroll down 👇 and you will get an Image showing Status codes with respective meanings*

DELETE Requests

HTTP DELETE requests are used to remove resources from a server. They are especially useful when you want to delete data permanently, such as removing a user account or a product from a database. Think of it as telling the server, "Please get rid of this."

The Code Example

Now, let's go through the code example that shows how to implement a DELETE request handler using Node.js:

```
// Import the 'http' module to work with HTTP requests and responses
const http = require('http');

// Create an HTTP server
const server = http.createServer((req, res) => {
  // Check if the request method is 'DELETE'
  if (req.method === 'DELETE') {
    // Handle DELETE request
    // Check if the request URL (the path in the address bar) is '/delete-product'
    if (req.url === '/delete-product') {
      // Create some dummy data for products
      let products = [
        { id: 1, name: 'Product A' },
        { id: 2, name: 'Product B' },
        { id: 3, name: 'Product C' },
      ];
      // Parse product ID from the URL
      const productID = parseInt(req.url.split('/')[2]);
      // Find and delete the product with the matching ID
      const updatedProducts = products.filter((product) => product.id !== productID);
      // Replace the old product list with the updated list
      products = updatedProducts;
      // Prepare the response to send back to the client
      // Set the HTTP status code to 200 (OK)
      // Specify that we are sending plain text as the response
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      // Send a success message as the response body
      res.end('Product deleted successfully');
    }
  }
});
// Start the server and specify the port it should listen on
const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Explaining the Code

1. We start by importing the `http` module, which allows us to handle HTTP requests and responses.
2. Next, we create an HTTP server using `http.createServer()` . This server will listen for incoming requests and handle them accordingly.
3. Inside the server creation function, we check if the request method is 'DELETE'. This ensures that we're only interested in handling DELETE requests.
4. If the request method is indeed 'DELETE', we further check if the request URL is '/delete-product'. This means we're specifically looking for DELETE requests to the '/delete-product' endpoint.
5. If both conditions are met, we proceed to handle the DELETE request. To do this, we create some dummy data for products. In a real-world scenario, this data might come from a database.
6. We parse the product ID from the URL. In this example, we assume that the product ID is the last part of the URL path.
7. We use the `filter()` method to find and delete the product with the matching ID from the `products` array. This results in a new array called `updatedProducts` that doesn't include the deleted product.
8. We replace the old `products` array with the `updatedProducts` array, effectively removing the product from our data.
9. We prepare the response to send back to the client by setting the HTTP status code to 200, indicating success, and specifying that we're sending plain text as the response.
10. Finally, we send a plain text success message as the response body and end the response.
11. At the end of the code, we start the server and specify the port it should listen on. If you're running this code locally, it will typically listen on port 3000. You'll see a log message in the console indicating that the server is running.

UPDATE (PUT) Requests

HTTP PUT requests are used to update or replace an entire resource on the server. They are especially useful when you want to replace an existing resource with new data. Think of it as telling the server, "Please update this resource with this new information."

The Code Example

Now, let's go through the code example that shows how to implement an UPDATE (PUT) request handler using Node.js:

```
// Import the 'http' module to work with HTTP requests and responses
const http = require('http');

// Create an HTTP server
const server = http.createServer((req, res) => {
  // Check if the request method is 'PUT'
  if (req.method === 'PUT') {
    // Handle PUT request
    // Check if the request URL (the path in the address bar) is '/replace-product'
    if (req.url === '/replace-product') {
      // Create some dummy data for products
      let products = [
        { id: 1, name: 'Product A' },
        { id: 2, name: 'Product B' },
        { id: 3, name: 'Product C' },
      ];
      // Parse product ID from the URL
      const productID = parseInt(req.url.split('/')[2]);
      // Find the product to replace
      const productToReplace = products.find((product) => product.id === productID);
      if (productToReplace) {
        // Dummy replacement data
        const replacementData = { id: productID, name: 'New Product Name' };
        // Replace the product with the new data
        const index = products.indexOf(productToReplace);
        products[index] = replacementData;
        // Prepare the response to send back to the client
        // Set the HTTP status code to 200 (OK)
        // Specify that we are sending JSON data as the response
        res.writeHead(200, { 'Content-Type': 'application/json' });
        // Send the updated product as the response body in JSON format
        res.end(JSON.stringify(replacementData));
      } else {
        // If the product to replace is not found, send a 404 error response
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('Product not found');
      }
    }
  }
});
// Start the server and specify the port it should listen on
const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Explaining the Code

1. We start by importing the `http` module, which allows us to handle HTTP requests and responses.

2. Next, we create an HTTP server using `http.createServer()`. This server will listen for incoming requests and handle them accordingly.

3. Inside the server creation function, we check if the request method is 'PUT'. This ensures that we're only interested in handling PUT requests.

4. If the request method is indeed 'PUT', we further check if the request URL is '/replace-product'. This means we're specifically looking for PUT requests to the '/replace-product' endpoint.

5. If both conditions are met, we proceed to handle the PUT request. To do this, we create some dummy data for products. In a real-world scenario, this data might come from a database.

6. We parse the product ID from the URL. In this example, we assume that the product ID is the last part of the URL path.

7. We use the `find()` method to locate the product with the matching ID in the `products` array.

8. If the product to replace is found, we create some dummy replacement data. This data represents the updated product information.

9. We replace the old product with the new data by finding its index in the `products` array and updating that element.

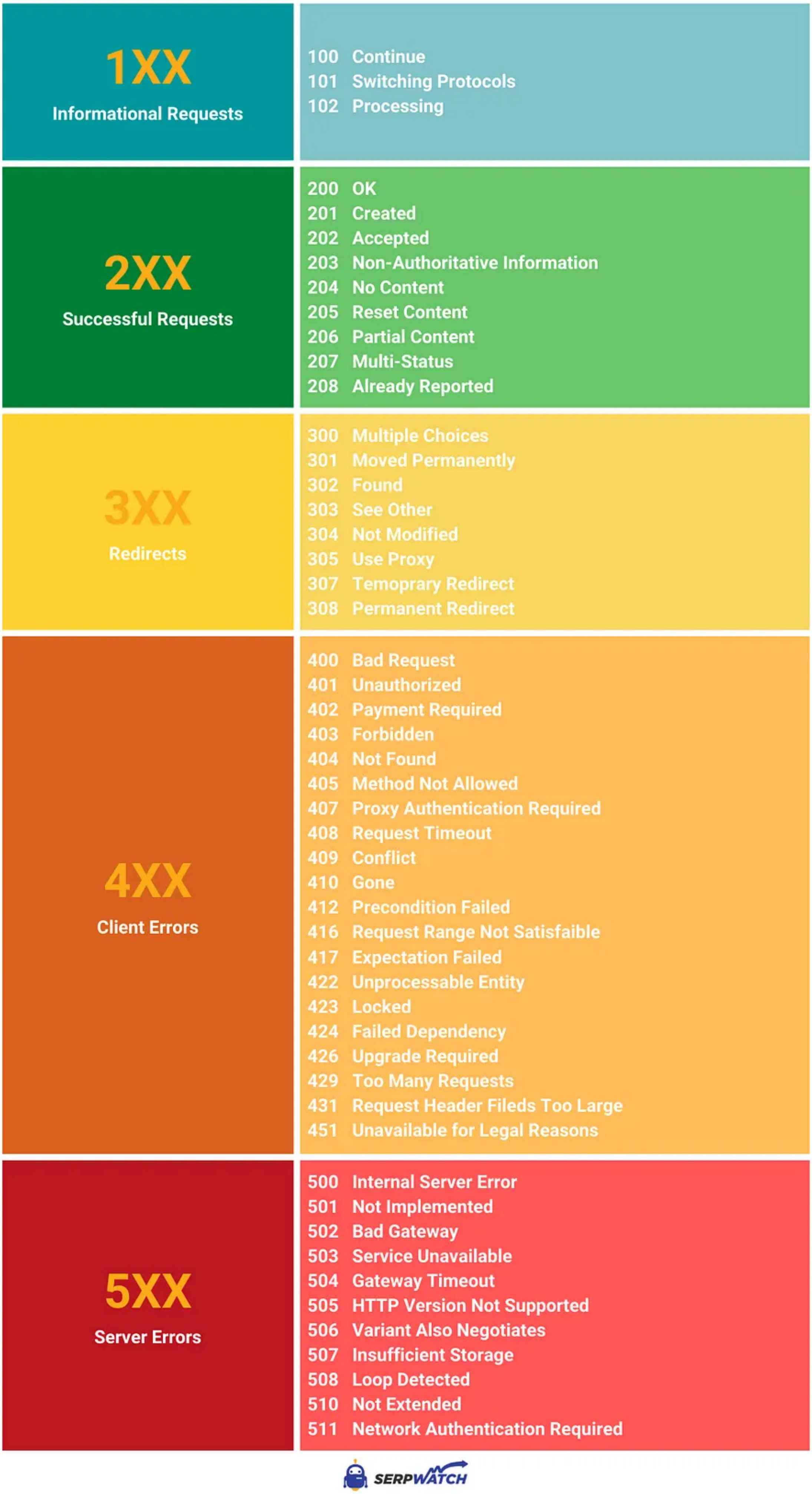
10. We prepare the response to send back to the client by setting the HTTP status code to 200 (indicating success) and specifying that we're sending JSON data as the response.

11. We send the updated product as the response body in JSON format.

12. If the product to replace is not found (meaning the product ID provided in the URL doesn't match any existing product), we send a 404 error response with a simple text message indicating that the product was not found.

13. At the end of the code, we start the server and specify the port it should listen on. If you're running this code locally, it will typically listen on port 3000. You'll see a log message in the console indicating that the server is running.

HTTP Status Codes



1XX Informational Requests	100 Continue 101 Switching Protocols 102 Processing
2XX Successful Requests	200 OK 201 Created 202 Accepted 203 Non-Authoritative Information 204 No Content 205 Reset Content 206 Partial Content 207 Multi-Status 208 Already Reported
3XX Redirects	300 Multiple Choices 301 Moved Permanently 302 Found 303 See Other 304 Not Modified 305 Use Proxy 307 Temoprary Redirect 308 Permanent Redirect
4XX Client Errors	400 Bad Request 401 Unauthorized 402 Payment Required 403 Forbidden 404 Not Found 405 Method Not Allowed 407 Proxy Authentication Required 408 Request Timeout 409 Conflict 410 Gone 412 Precondition Failed 416 Request Range Not Satisfaible 417 Expectation Failed 422 Unprocessable Entity 423 Locked 424 Failed Dependency 426 Upgrade Required 429 Too Many Requests 431 Request Header Fileds Too Large 451 Unavailable for Legal Reasons
5XX Server Errors	500 Internal Server Error 501 Not Implemented 502 Bad Gateway 503 Service Unavailable 504 Gateway Timeout 505 HTTP Version Not Supported 506 Variant Also Negotiates 507 Insufficient Storage 508 Loop Detected 510 Not Extended 511 Network Authentication Required

SERPWATCH

This image is from serpwatch.io

Error Handling

In any application, error handling is crucial. Node.js provides mechanisms for handling errors gracefully. You can use **try-catch** blocks or handle errors using middleware, depending on your application's complexity.

Conclusion

In this article, we've explored the essential HTTP methods — GET, POST, DELETE, PUT — in the context of Node.js. You've seen real-world examples and learned how to implement these methods in your Node.js applications. Understanding HTTP methods is fundamental to building robust and interactive web applications.

Now that you're equipped with this knowledge, go ahead and build amazing Node.js applications that handle different types of HTTP requests.

Backend Development Api Development Node Js Tutorial Tutorial Expressjs

Follow

Written by Sandeep Singh (Full Stack Dev.)