

## Header in NodeJS

- In Node.js, HTTP headers are key-value pairs sent between the client and the server as part of an HTTP request or response.
- Headers provide metadata about the data being transmitted and help both the client and server understand how to handle the request or response.
- Headers play a crucial role in HTTP communication as they enable clients and servers to exchange metadata about the request and response.
- They facilitate various functionalities such as content negotiation, authentication, caching, redirection, and more.
- Understanding and properly handling headers is essential for building robust and secure web applications in Node.js.

### Request Headers

**Accessing Request Headers:** In Node.js, request headers are accessible through the headers property of the request object.

Example:

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

const server = http.createServer((req, res) => {
  // Accessing request headers
  const userAgent = req.headers['user-agent'];
  console.log('User-Agent:', userAgent);
  res.end();
});

server.listen(3000);
```

### Common Request Headers

- **Accept:** Specifies the media types that are acceptable for the response.
- **User-Agent:** Identifies the client making the request, typically the web browser or application.
- **Authorization:** Provides credentials for authentication purposes.
- **Content-Type:** Specifies the MIME type of the request body for POST and PUT requests.

## Response Headers

**Setting Response Headers:** In Node.js, response headers can be set using the `setHeader()` or `writeHead()` method of the response object.

**Example:**

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

const server = http.createServer((req, res) => {
  // Setting response headers
  res.setHeader('Content-Type', 'text/plain');
  res.writeHead(200, { 'Custom-Header': 'Custom Value' });
  res.end('Hello, World!');
});

server.listen(3000);
```

## Common Response Headers

- **Content-Type:** Specifies the MIME type of the response body.
- **Cache-Control:** Directives for caching behavior, instructing clients or intermediaries on whether and how to cache the response.
- **Set-Cookie:** Used to set cookies in the client's browser.
- **Location:** Specifies a URL to redirect the client to, commonly used for HTTP redirection.

## Manipulating Headers

- **Reading Headers:** Use the `req.headers` object to access request headers.

**Example:**

```
const contentType = req.headers['content-type'];
```

- **Setting Headers:** Use the `res.setHeader()` or `res.writeHead()` method to set response headers.

**Example:**

```
res.setHeader('Content-Type', 'application/json');
res.writeHead(200, { 'Custom-Header': 'Custom Value' });
```

- **Removing Headers:** Headers can be removed using the `removeHeader()` method.

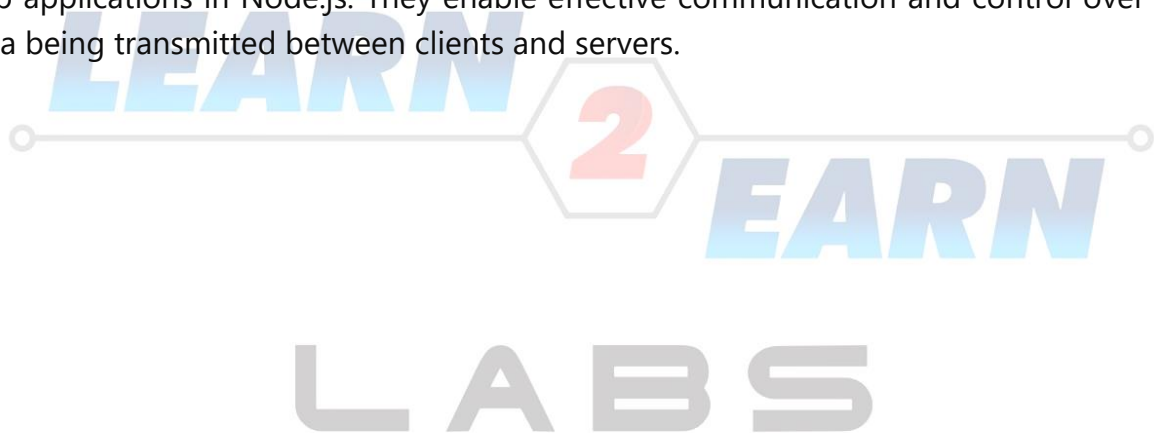
Example:

```
res.removeHeader('Content-Type');
```

## Importance of Headers

- **Content negotiation:** determining the format of the response (e.g., JSON, HTML).
- **Authentication:** passing credentials for authorization purposes.
- **Caching:** specifying caching directives to control cache behavior.
- **Redirection:** indicating a URL to redirect the client to.
- **Security:** including security-related headers to prevent common attacks.

Understanding headers and their usage is fundamental for building robust and secure web applications in Node.js. They enable effective communication and control over the data being transmitted between clients and servers.



### Example: work with headers in an HTTP server


#### App.js

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

// Create an HTTP server
const server = http.createServer((req, res) => {
  // Set response headers
  res.setHeader('Content-Type', 'text/plain');
  res.setHeader('Custom-Header', 'Full Stack Web Development');

  // Send the response
  res.end('Hello Students, Hope you are doing well!');
});

// Listen on port 3000
const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

The logo for Learn 2 Earn Labs is centered in the background. It features the words 'LEARN' and 'EARN' in a large, blue, sans-serif font. A large red number '2' is positioned between them, enclosed within a hexagonal frame. Below this, the word 'LABS' is written in a smaller, grey, sans-serif font. The entire logo is overlaid on a faint, light grey circuit-like pattern.

Example: handle custom headers in both the request and response of an HTTP server in Node.js

App.js

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

// Create an HTTP server
const server = http.createServer((req, res) => {
  // Access request headers
  const userAgent = req.headers['user-agent'];
  console.log('User-Agent:', userAgent);

  // Set response headers
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Powered-By', 'Learn2Earn Labs');

  // Send the response
  res.end('<h1>Hello Students, Hope you are doing well</h1>');
});

// Listen on port 3000
const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

## Status codes

- In Node.js, when you are building a web application using frameworks like Express, you often need to send HTTP responses to clients (browsers or other services) that make requests to your server. These responses typically include a status code, headers, and an optional body.
- Status codes are a standardized way to indicate the result of the server's attempt to fulfill the client's request. They are three-digit integers where the first digit defines the class of response (e.g., 2xx for successful responses, 4xx for client errors, 5xx for server errors).

### Commonly used status codes

- **200 OK:** This status code indicates that the request was successful. The server typically returns this code along with the requested content.
- **201 Created:** This status code indicates that the request has been fulfilled and a new resource has been created. It is often used in conjunction with POST requests to create new resources on the server.
- **400 Bad Request:** This status code indicates that the server cannot process the request due to a client error, such as malformed syntax or invalid parameters.
- **401 Unauthorized:** This status code indicates that the client must authenticate itself to get the requested response. It is commonly used when access to a resource is restricted to authenticated users.
- **404 Not Found:** This status code indicates that the server cannot find the requested resource. It is commonly used when a URL is mistyped or when the requested resource no longer exists.
- **500 Internal Server Error:** This status code indicates a generic error message that indicates that something has gone wrong on the server. It is often used to indicate that the server encountered an unexpected condition that prevented it from fulfilling the request.

In Node.js, you can set the status code of an HTTP response using the `res.status()` method provided by Express (or other similar frameworks).

For example, to send a 404 status code along with a message 'Not Found', you need to use the following code:

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

This sets the status code of the response to 404 and sends the string 'Not Found' as the response body.

### Example: setting different HTTP status codes and sending responses

#### App.js

```
const express = require('express');
const app = express();
// Route to handle GET requests
app.get('/', (req, res) => {
  // Send a 200 OK status code with a message
  res.status(200).send('Hello, Students!');
});
// Route to handle POST requests
app.post('/create', (req, res) => {
  // Send a 201 Created status code with a message
  res.status(201).send('Resource created successfully');
});
// Route to handle invalid requests
app.get('/error', (req, res) => {
  // Send a 400 Bad Request status code with a message
  res.status(400).send('Bad Request');
});
// Route to handle unauthorized access
app.get('/unauthorized', (req, res) => {
  // Send a 401 Unauthorized status code with a message
  res.status(401).send('Unauthorized');
});
// Route to handle not found resources
app.get('/notfound', (req, res) => {
  // Send a 404 Not Found status code with a message
  res.status(404).send('Not Found');
});
// Route to simulate internal server error
app.get('/error500', (req, res) => {
  // Send a 500 Internal Server Error status code with a message
  res.status(500).send('Internal Server Error');
});
// Start the server
const PORT = 3000;
app.listen(PORT, () => { console.log(`Server running on port ${PORT}`); });
```

## Checking the status codes

To check the status codes of HTTP responses in your Node.js application, you can use tools like Postman, curl, or simply your web browser's developer tools.

**Using Postman:** If you're using Postman, you can make requests to your server and view the response details, including the status code. Postman provides a detailed view of the response, including headers, body, and status code.

**Using Browser Developer Tools:** If you're testing your application in a web browser, you can use the browser's developer tools to view the status code of the responses. In Chrome, for example, you can open the developer tools (F12 or right-click -> Inspect) and go to the Network tab. Then, reload the page or make a request to your server, and you'll see a list of requests along with their status codes.

**Using curl:** You can use the curl command in your terminal to make requests to your server and view the response status code.

For example:

```
curl -I http://localhost:3000
```

This will send a HEAD request to your server (using the -I flag) and display the response headers, including the status code.

By using these tools, you can easily check the status codes of HTTP responses in your Node.js application and debug any issues related to status codes.



### Example: setting HTTP status codes and sending responses in json format

#### Server.js

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

// Route to handle GET requests
app.get('/', (req, res) => {
  // Send a 200 OK status code with a JSON response
  res.status(200).json({ message: 'Success', data: { name: 'Neha', age: 27,
gender:'female' } });
});

// Route to handle a resource that is not found
app.get('/notfound', (req, res) => {
  // Send a 404 Not Found status code with a message
  res.status(404).json({ error: 'Resource not found' });
});

// Route to handle a server error
app.get('/error', (req, res) => {
  // Simulate an error
  try {
    throw new Error('Something went wrong');
  } catch (err) {
    // Send a 500 Internal Server Error status code with an error message
    res.status(500).json({ error: err.message });
  }
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

## Application Programming Interface

- API (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other.
- It defines the methods and data formats that applications can use to request and exchange information.
- APIs are commonly used in web development to enable communication between a web server and client-side applications.

### Concept of API

APIs allow developers to access the functionality of a system or service without having to understand its internal workings. They provide a level of abstraction that simplifies the development process and promotes code reusability. APIs can be used to perform a wide range of tasks, such as retrieving data from a database, sending emails, or interacting with external services.

### Types of APIs

- **Web APIs (HTTP APIs)**
  - These APIs are accessed over the internet using the HTTP protocol.
  - They are commonly used to enable communication between web servers and client-side applications.
  - Web APIs are often used to retrieve or manipulate data, such as fetching weather information, sending messages, or accessing a database.

**Example:** The OpenWeatherMap API allows developers to retrieve weather data for a specific location by sending an HTTP request to their API endpoint.

- **Library APIs**
  - These APIs are provided by software libraries and allow developers to access the functionality of the library in their own code.
  - Library APIs are used to perform specific tasks, such as manipulating images, parsing JSON data, or encrypting data.

**Example:** The axios library in Node.js provides an API for making HTTP requests. Developers can use the `axios.get()` method to send a GET request to a server and retrieve data.

- **Operating System APIs**

- These APIs are provided by operating systems and allow developers to access system resources, such as files, processes, and network connections.
- Operating system APIs are used to perform low-level tasks that require interaction with the underlying operating system.

**Example:** The Windows API provides a set of functions that allow developers to create and manage windows, handle user input, and perform other tasks related to the Windows operating system.

### **Common types of Web API's**

There are several types of APIs, each with its own set of characteristics and use cases. The choice of API type depends on the specific requirements of your application, including the desired level of complexity, scalability, and performance.

Some of the most common types include:

- **RESTful API (Representational State Transfer)**

- REST is an architectural style for designing networked applications. RESTful APIs are designed to be stateless and utilize HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources.
- They typically use JSON or XML as the data format. RESTful APIs are widely used for web and mobile applications due to their simplicity and scalability.

- **SOAP API (Simple Object Access Protocol)**

- SOAP is a protocol for exchanging structured information in the implementation of web services.
- SOAP APIs use XML as the data format and typically require a more complex setup compared to RESTful APIs.
- They are often used in enterprise environments where a high level of security and reliability is required.

- **GraphQL API**

- GraphQL is a query language for APIs that allows clients to request only the data they need.
- Unlike RESTful APIs, which expose a fixed set of endpoints, GraphQL APIs have a single endpoint and allow clients to specify the structure of the response.
- This makes GraphQL APIs more flexible and efficient for fetching data in complex applications.

- **WebSocket API**

- WebSocket is a communication protocol that provides full-duplex communication channels over a single TCP connection.
- WebSocket APIs allow for real-time, bidirectional communication between clients and servers, making them ideal for applications that require low-latency updates, such as chat applications and online gaming platforms.

- **RPC API (Remote Procedure Call)**

- RPC is a protocol that allows a program to execute code on a remote server.
- RPC APIs enable applications to call functions or procedures on a remote server as if they were local, making them useful for distributed systems and microservices architectures.

- **gRPC (Google Remote Procedure Call)**

- gRPC is a high-performance RPC framework developed by Google.
- It uses HTTP/2 for transport and Protocol Buffers as the interface description language.
- gRPC APIs are often used in microservices architectures and distributed systems where performance and efficiency are critical.

## API Use Cases

- **Social Media APIs:** Social media platforms like Facebook, Twitter, and Instagram provide APIs that allow developers to access their platform's features, such as posting updates, retrieving user information, and interacting with friends.
- **Payment Gateway APIs:** Payment gateway services like PayPal, Stripe, and Braintree provide APIs that allow developers to integrate payment processing into their applications. Developers can use these APIs to accept payments, issue refunds, and manage transactions.
- **Maps APIs:** Mapping services like Google Maps and Mapbox provide APIs that allow developers to integrate mapping functionality into their applications. Developers can use these APIs to display maps, get directions, and perform geocoding tasks.
- **Weather APIs:** Weather services like OpenWeatherMap and WeatherAPI provide APIs that allow developers to retrieve weather information for a specific location. Developers can use these APIs to display current weather conditions, forecasts, and historical data.

APIs play a crucial role in modern software development by enabling interoperability between different systems and services. They provide a standardized way for applications to communicate and exchange data, making it easier for developers to build complex and feature-rich applications.

## API in Node.js

- In Node.js, an API (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other.
- APIs are commonly used to define how clients can request and receive data from a server.
- APIs in Node.js allow you to create web services that can be accessed by clients to retrieve or manipulate data.
- They are a fundamental part of building modern web applications and services.

## How APIs work in Node.js

- **Creating an API:** In Node.js, you can create an API using frameworks like Express. Express provides a set of methods to define routes and handle HTTP requests.
- **Defining Routes:** Routes in an API correspond to different endpoints that clients can access. For example, you can define a route to handle GET requests to the /users endpoint, which would return a list of users.
- **Handling Requests:** When a client makes a request to your API, Express (or another framework) will invoke the appropriate route handler based on the request method (GET, POST, PUT, DELETE, etc.) and endpoint.
- **Sending Responses:** In the route handler, you can send a response back to the client. This response can include data in various formats, such as JSON, HTML, or plain text.
- **Middleware:** Middleware functions can be used to process requests before they reach the route handler. This can include tasks like authentication, logging, or data parsing.
- **Error Handling:** APIs should include error handling mechanisms to handle situations where requests cannot be fulfilled. This can include sending appropriate HTTP status codes (e.g., 404 for not found, 500 for internal server error) and error messages.
- **Testing:** It's important to test your API to ensure that it behaves as expected. You can use tools like Postman or automated testing frameworks to test different endpoints and scenarios.

### Example : Node.js API using Express

First you need to install express using following command  
npm install express

#### App.js

```
const express = require('express');
const app = express();
const PORT = 3000;

// Sample data
let students = [
  { id: 1, name: 'shivam', status:'fulltime' },
  { id: 2, name: 'manisha', status:'parttime' },
  { id: 3, name: 'neeraj', status:'fulltime' }
];

// GET /students - Get all student records
app.get('/students', (req, res) => {
  res.json(students);
});

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