Introduction to FS module

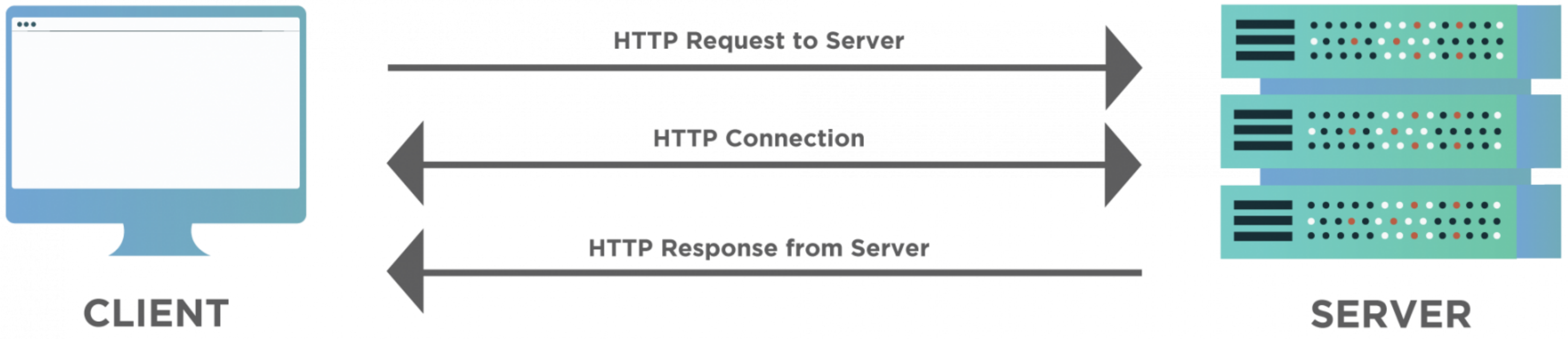
Akash Pundir

System Programming –I

School of Computer Science and Engineering

What is a HTTP server?

An HTTP (Hypertext Transfer Protocol) server is a software application or hardware device that processes and responds to requests made by clients over the Hypertext Transfer Protocol. HTTP is the foundation of data communication on the World Wide Web, and it defines how messages are formatted and transmitted, as well as what actions web servers and browsers should take in response to various commands.


HTTP Request to Server
HTTP Connection
HTTP Response from Server
CLIENT
SERVER

The basic interaction between a client (typically a web browser) and an HTTP server involves the client sending an HTTP request to the server, and the server responding with the requested resource or an appropriate error message. This communication is typically initiated when a user enters a URL (Uniform Resource Locator) in a web browser or clicks on a link.

Let’s make our first node application

- Open Command Prompt, make a new folder, and run this command.

```
npm init
```

http Module

http is one of the built-in modules that comes with Node.js. It provides low-level features for running a server to accept requests and return responses.

Make a new index.js file in same directory

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

const server = http.createServer((req, res) => {
  res.end('Hello, this is your Node.js server!');
});

const port = 3000;

server.listen(port, () => console.log(`Server is running
on http://localhost:${port}`));
```

What is FS module?

The fs (File System) module in Node.js provides a way to interact with the file system. **It allows you to perform various file-related operations such as reading, writing, and manipulating files and directories.**

Common fs Methods

- **fs.readFile(path, options, callback):** Asynchronously reads the entire contents of a file.
- **fs.readFileSync(path, options):** Synchronously reads the entire contents of a file.
- **fs.writeFile(file, data, options, callback):** Asynchronously writes data to a file, replacing the file if it already exists.
- **fs.writeFileSync(file, data, options):** Synchronously writes data to a file.

Http Header

An HTTP header is a set of key-value pairs sent by a web browser or client to a web server, or by a web server to a web browser or client, as part of the HTTP protocol. **HTTP headers convey additional information about the request or the response, providing metadata that helps both the client and the server understand and process the communication effectively**.

Request Headers:

Host: Specifies the domain name of the server (required in HTTP/1.1).

User-Agent: Provides information about the user agent (web browser or other client) making the request.

Accept: Informs the server about the types of media that the client can process.

Cookie: Contains stored HTTP cookies, allowing state to be maintained between requests.

...and many others.

Response Headers:

Content-Type: Indicates the media type (e.g., HTML, JSON) of the resource sent by the server.

Content-Length: Specifies the size of the response body in octets (8-bit bytes).

Server: Identifies the software used by the origin server.

Set-Cookie: Sets an HTTP cookie on the client's browser.

...and many others.

Http Status Codes

HTTP status codes are **three-digit numbers that the server includes in its responses to provide information about the status of a requested resource or operation**. These codes are part of the HTTP (Hypertext Transfer Protocol) standard and are grouped into different classes to indicate the nature of the response

HTTP Status Codes

🟩 **1XX**
INFORMATIONAL

🟧 **2XX**
SUCCESS

🟥 **3XX**
REDIRECTION

🟦 **4XX**
CLIENT ERROR

🟪 **5XX**
SERVER ERROR

1xx (Informational): The request was received, continuing process.

Example: 100 Continue (The server has received the request headers and the client should proceed to send the request body).

2xx (Success): The request was successfully received, understood, and accepted.

Example: 200 OK (The request was successful).

3xx (Redirection): Further action needs to be taken to complete the request.

Example: 302 Found (The requested resource has been temporarily moved to another location).

4xx (Client Error): The request contains bad syntax or cannot be fulfilled by the server.

Example: 404 Not Found (The requested resource could not be found on the server).

5xx (Server Error): The server failed to fulfill a valid request.

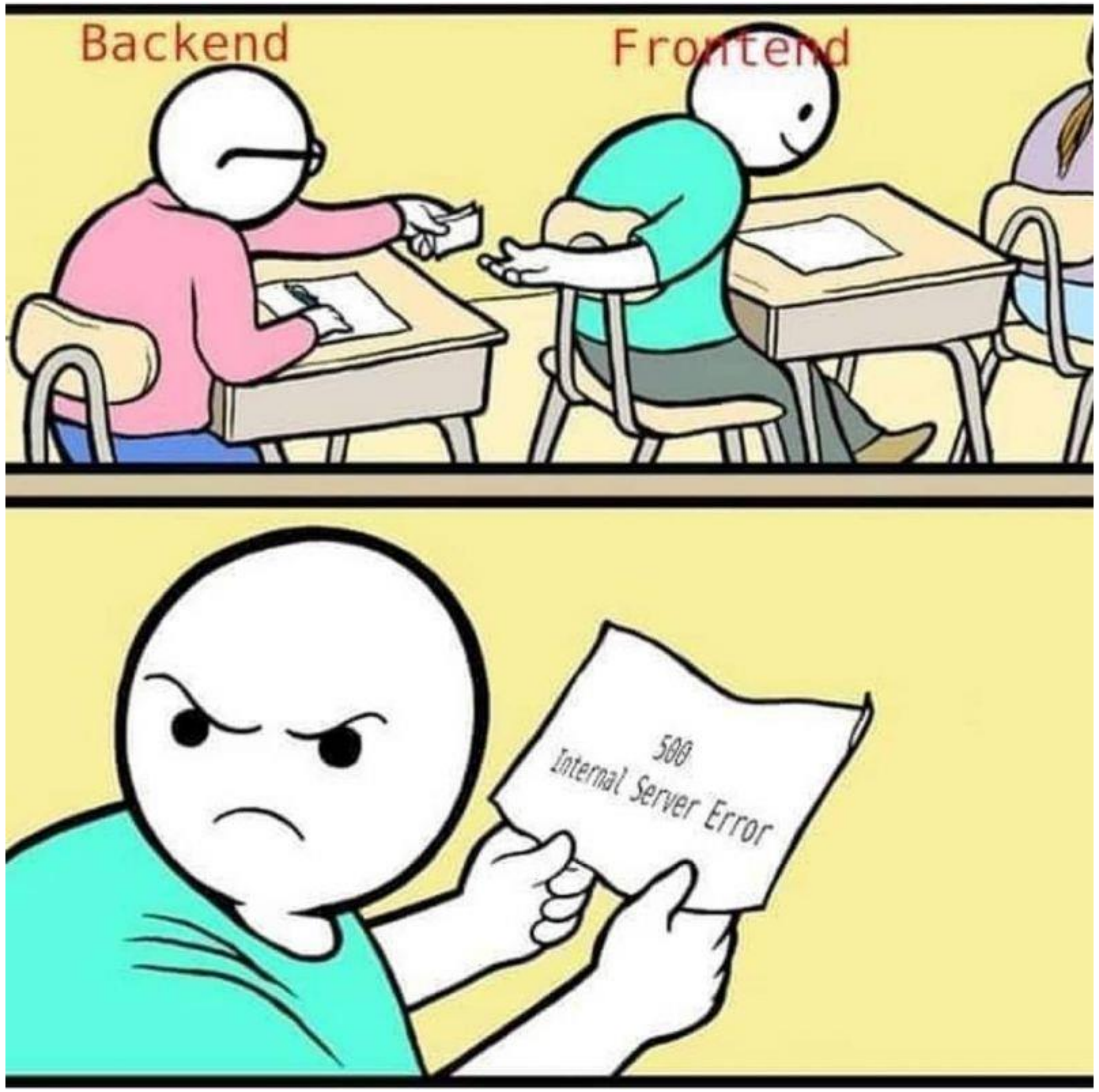
Example: 500 Internal Server Error (A generic error message returned when an unexpected condition was encountered on the server).

Reading a File Asynchronously

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

const server = http.createServer((req, res) => {
  if(req.url=="/"){
    fs.readFile('FileSystem/example.txt','utf8',(err,data)=>{
      if(err){
        console.error(err);
        return;
      }
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end('File Content:'+data);
    })
  }
});

const port = 3000;

server.listen(port, () => console.log(`Server is running on http://localhost:${port}`));
```

Writing to a File Asynchronously

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

const server = http.createServer((req, res) => {
  if (req.url === "/") {
    const message = 'Hello, this is your Node.js server!';
    // Write the message to a file asynchronously
    fs.writeFile('FileSystem/output.txt', message, 'utf8', (err) => {
      if (err) {
        console.error(err);
        res.end('Internal Server Error');
        return;
      }
      res.end('File Content: ' + message);
    });
  }
});

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


Backend
Frontend
500
Internal Server Error

Reading a File Synchronously

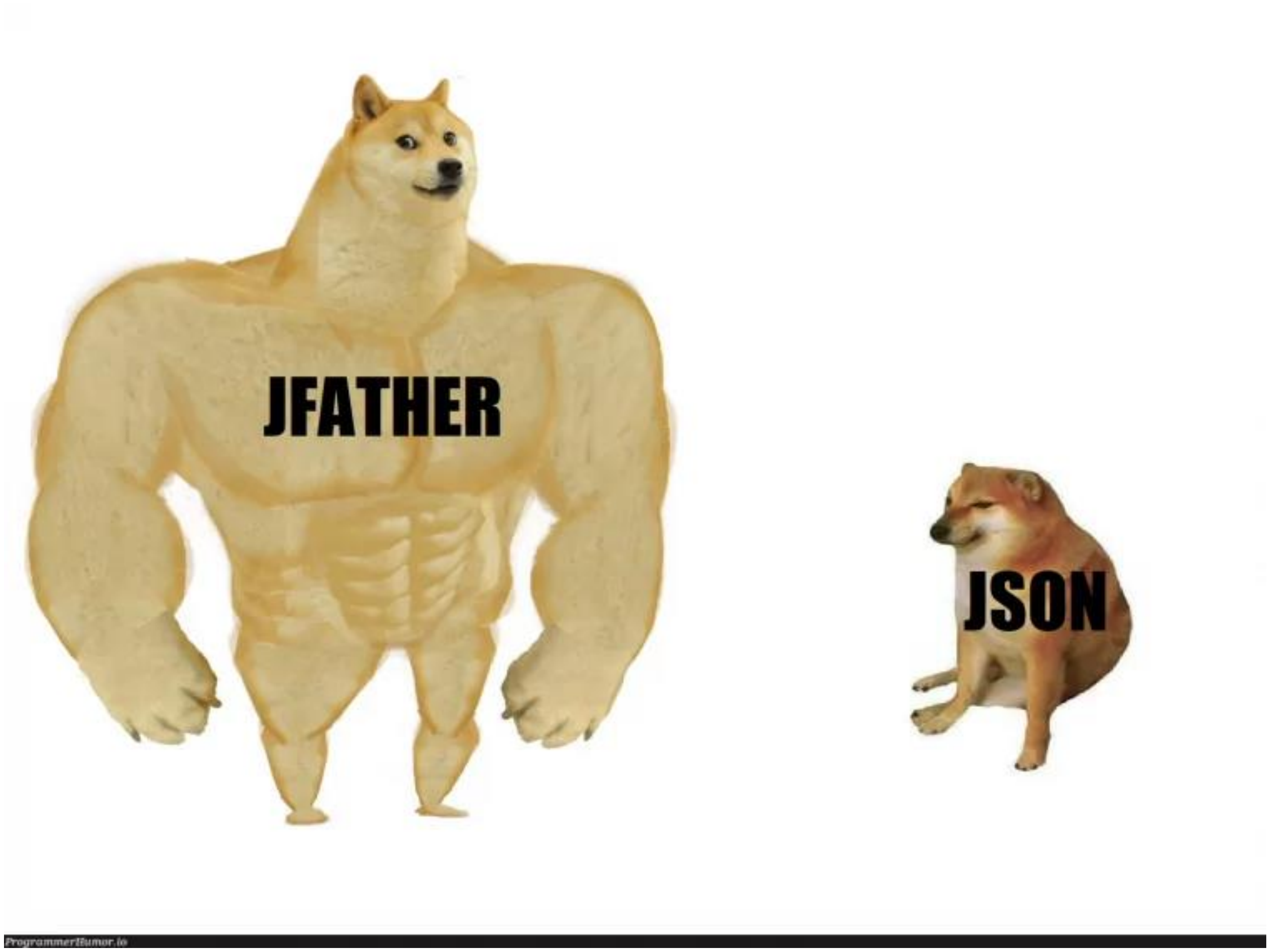
```
const http = require('http');
const fs = require('fs');

const server = http.createServer((req, res) => {
  if (req.url === "/") {
    // Synchronously read the content from a file
    try {
      const data = fs.readFileSync('FileSystem/example.txt', 'utf8’);
      res.end('File Content: ' + data);
    } catch (err) {
      console.error(err);
      res.end('Internal Server Error');
    }
  }
});

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

Writing a File Synchronously

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

const server = http.createServer((req, res) => {
  if (req.url === "/") {
    const message = 'Hello, this file is written using Sync!';
    // Synchronously write the message to a file
    try {
      fs.writeFileSync('FileSystem/output.txt', message, 'utf8’);
      res.end('File Content: ' + message);
    } catch (err) {
      console.error(err);
      res.end('Internal Server Error');
    }
  }
});

const port = 4000;
server.listen(port, () => console.log(`Server is running on http://localhost:${port}`));
```

Appending File in Asynchronous Manner

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

const server = http.createServer((req, res) => {
  if (req.url === "/") {
    const message = 'Hello, this file is appended using Callback!';

    fs.appendFile('FileSystem/output.txt', message + '\n', 'utf8', (err) => {
      if (err) {
        console.error(err);
        res.end('Internal Server Error');
      } else {
        res.end('File Content Appended: ' + message);
      }
    });
  }
});

const port = 4000;
server.listen(port, () => console.log(`Server is running on http://localhost:${port}`));
```


JFATHER
JSON
ProgrammerHumor.io

Working with JSON

- JSON is **a lightweight data-interchange format.**
- It is easy for humans to read and write and easy for machines to parse and generate.

Lightweight

Low Overhead:

JSON has a simple and concise syntax, which results in minimal overhead in terms of file size or data representation. This simplicity makes it easy for both humans to read and machines to parse.

Easy to Understand:

JSON's structure is straightforward and closely resembles the data structures used in many programming languages. This simplicity contributes to the ease of understanding, writing, and parsing JSON data.

Efficient Processing:

Due to its simplicity, JSON can be processed quickly by software applications. This is particularly important in scenarios where efficient data transfer and processing are crucial, such as in web development or when communicating between networked systems.

Data Interchange

Format for Data Exchange:

JSON is primarily used as a format for exchanging data between different systems or platforms. It serves as a common language that applications can use to share information.

Communication Between Systems:

In web development and many other domains, different systems (like a web server and a web browser) need to communicate and exchange data. JSON provides a standardized way for these systems to structure and understand the information being shared.

Platform-Neutral:

JSON is not tied to any specific programming language or platform. It is a language-agnostic format, meaning that systems implemented in different languages can easily understand and process JSON data.

Human-Readable:

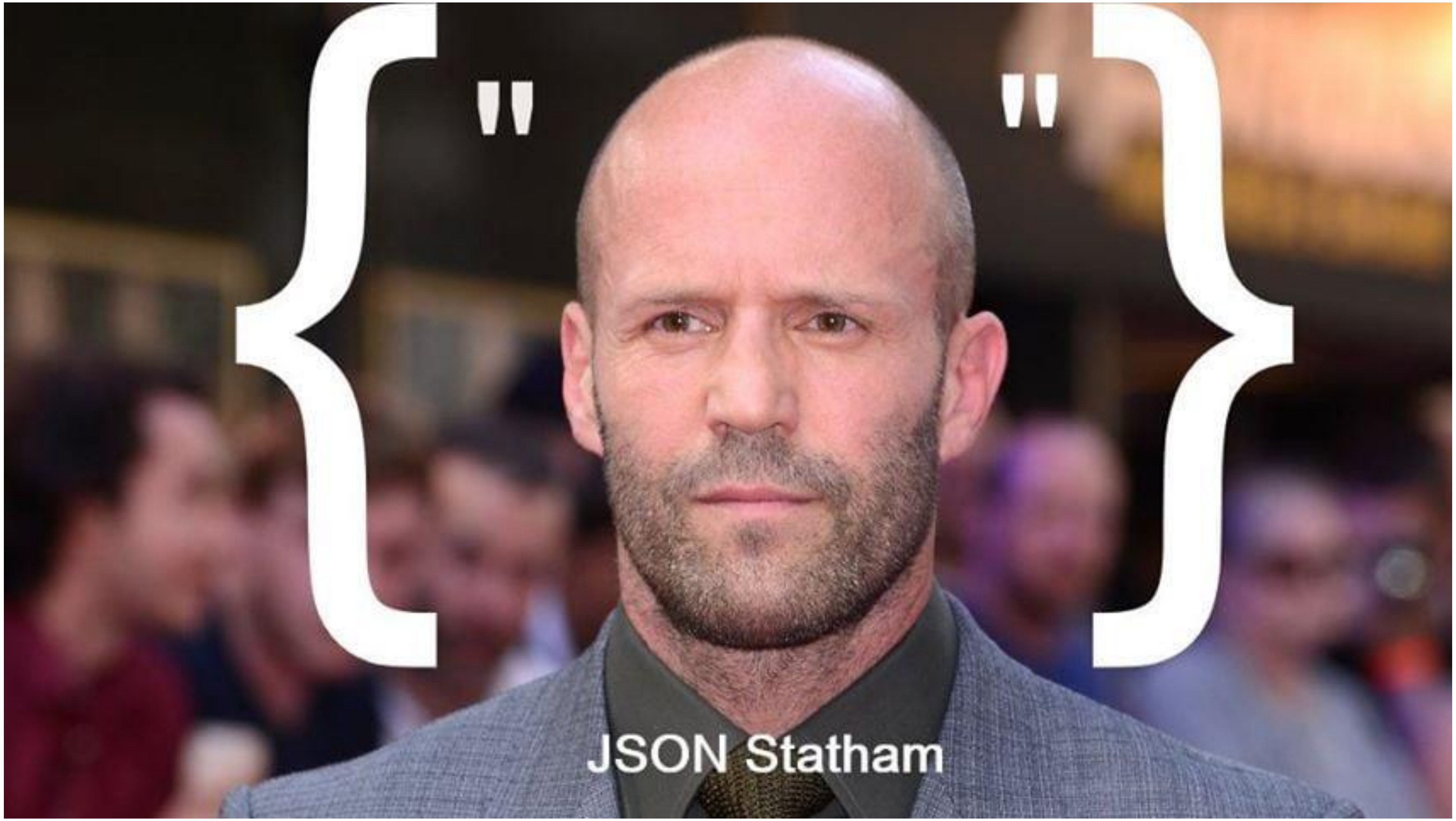
While designed for machine consumption, JSON is also human-readable. This makes it easy for developers to inspect data, troubleshoot issues, and understand the content being exchanged.

Used in APIs:

JSON is widely used in web APIs (Application Programming Interfaces) where servers and clients communicate over the internet. APIs often send and receive data in JSON format due to its simplicity and ease of use.

JSON Format

```
{
  "name": "John",
  "age": 25,
  "city": "New York"
}
```


{" "}
JSON Statham

Difference between JavaScript Object vs JSON Format

Syntax:

In JavaScript, objects are defined using curly braces {} and can have keys without quotes. In JSON, keys must be enclosed in double-quotes.

Usage:

JavaScript objects are used within JavaScript programs for modeling and organizing data. JSON, on the other hand, is a data interchange format often used for communication between different systems, including between a web server and a web client.

Methods:

JavaScript objects can have methods (functions as values), while JSON is limited to representing data and does not include functions.

Encoding Data to JSON format

```javascript
const dataObject = {
  name: "John",
  age: 30,
  city: "New York"
};

const jsonString = JSON.stringify(dataObject);
console.log(jsonString);
```

Decoding from JSON

```
const jsonString = '{"name": "John", "age": 30,
"city": "New York"}';

const dataObject = JSON.parse(jsonString);
console.log(dataObject);
```

Reading a JSON file

Make a **data.json** file in the FileSystem directory

```
{
    "name": "John Doe",
    "age": 30,
    "city": "Example City"
  }
```

In index.js file

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

const server = http.createServer((req, res) => {
  if (req.url === '/') {
    // Read the JSON file
    fs.readFile('FileSystem/data.json', 'utf8', (err, data) => {
      if (err) {
        console.error(err);
        res.end('Internal Server Error');
        return;
      }
      // Set response headers
      res.writeHead(200, { 'Content-Type': 'application/json' });

      // Send JSON data as plain text
      res.end(data);
    });
  }
```

```javascript
else {
    // Handle other routes (if any)
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});

const port = 3000;

server.listen(port, () => {
  console.log(`Server is running on 
http://localhost:${port}`);
});
```

Writing Data in a JSON file

- In this practical we are going to write some data into a JSON file while taking input from the user through the browser.
- Before you start...
- Your Project Directory should look like this
- **project-root/**

```
|-- index.js
|-- public/
|   |-- form.html
|-- FileSystem/
|   |-- output.json
```

Let’s start by doing the simple job first, that is by my making our form.html file

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>JSON Input Form</title>
</head>
<body>
  <h2>Enter JSON Data:</h2>
  <form action="/save-json" method="post">
    <textarea name="jsonData" rows="4" cols="50" required></textarea>
    <br>
    <button type="submit">Save JSON Data</button>
  </form>
</body>
</html>
```

Alright, let’s get our hand dirty…In index.js

- **Let’s start by importing the necessary files.**

```
const http = require('http');
const fs = require('fs');
const path = require('path');
const querystring = require('querystring’);
```

- **The http module is a built-in module in Node.js that provides functionality for creating an HTTP server.**
- **The fs module is a built-in module in Node.js that provides file system-related functionality.**
- **The path module is a built-in module in Node.js that provides utilities for working with file and directory paths.**
- **It's used to construct the correct path to the HTML form file (form.html).**
- **The querystring module is a built-in module in Node.js that provides utilities for working with query strings.**
- **It's used to parse the data received in the POST request body (in this case, the JSON data from the form).**

Now, let’s create server with createServer method

```
const server = http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/') {
    // Serve HTML form
    const formPath = path.join(__dirname, 'public', 'form.html');
    fs.readFile(formPath, 'utf8', (err, data) => {
      if (err) {
        console.error(err);
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end('Internal Server Error');
        return;
      }
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(data);
    });
}
```

Now, let’s write code for what will have happen when user submits the form.

```
else if (req.method === 'POST' && req.url === '/save-json') {
    // Handle POST request to save JSON data
    let body = '';

    req.on('data', (chunk) => {
      body += chunk;
    });

    req.on('end', () => {
      const jsonData = querystring.parse(body).jsonData;
```

Now, let’s convert received data to string and write it in a file.

```
// Convert JSON data to a string
      const jsonString = JSON.stringify(JSON.parse(jsonData));

      // Write JSON data to a file
      fs.writeFile('FileSystem/output.json', jsonString, 'utf8', (err) => {
        if (err) {
          console.error(err);
          res.writeHead(500, { 'Content-Type': 'text/plain' });
          res.end('Internal Server Error');
          return;
        }

        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('JSON data successfully saved to output.json');
      });
    });
```

And let’s add code for what should happen in case a user enters a different route.

```
else {
    // Handle other routes (if any)
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});

const port = 3000;

server.listen(port, () => {
  console.log(`Server is running on 
http://localhost:${port}`);
});
```

Introduction to Buffer Module

Buffer module is **commonly used to work with binary data directly**. It is a part of Node.js and allows you to work with raw binary data in a variety of encodings.

But why to work with binary data directly?

- **Performance**: Operations on buffers are generally faster compared to string manipulation when dealing with binary data, especially when working with large datasets.
- **Flexibility**: Buffers can be resized, sliced, concatenated, and manipulated in various ways to suit different use cases.
- **Interoperability:** Buffers can be easily converted to and from other data types, such as strings, arrays, and TypedArrays, facilitating interoperability with different parts of an application or external systems.

Creating a Buffer:

```
// Creating a Buffer of size 10 bytes
const buffer = Buffer.alloc(4);
console.log(buffer);
```

Writing and Reading Data:

- You can write data into the buffer and read it back:

```
// Writing data to the buffer
buffer.write('Hello', 'utf-8');
// Reading data from the buffer
const data = buffer.toString('utf-8');
console.log(data); // Output: Hello
```

Concatenating Buffers:

```
const buffer1 = Buffer.from('Hello', 'utf-8');
const buffer2 = Buffer.from(' World', 'utf-8');

// Concatenating buffers
const concatenatedBuffer = Buffer.concat([buffer1,
buffer2]);
console.log(concatenatedBuffer.toString('utf-8'));
// Output: Hello World
```