JAMAL MOHAMED COLLEGE (Autonomous)

Department of Computer Applications

VALUE ADDED COURSE

III BCA – NODE JS

Unit-1 Introduction – NODE JS

Topic-1 Introduction to Node.js

Node.js is a powerful, event-driven, non-blocking I/O model JavaScript runtime that allows developers to build scalable network applications. It's built on Chrome's V8 JavaScript engine, providing an efficient and lightweight platform for server-side programming.

Key Features of Node.js:

- **Event-driven architecture**: Allows handling multiple connections concurrently.
- **Non-blocking I/O**: Improves performance by allowing other operations to continue while waiting for I/O operations to complete.
- **Single-threaded**: Handles multiple operations with a single thread using event looping.

Topic-2 First Application

Let's write our first Node.js application which will simply print "Hello, World!" to the console.

Step-by-step Guide:

1. Create a new file called app.js.
2. Add the following code:

```
// app.js
// Printing "Hello, World!" to the console
console.log('Hello, World!');
```

Explanation:

1. **// app.js**: A comment indicating the filename for clarity.
2. **console.log('Hello, World!');**: This line prints the string 'Hello, World!' to the console.

Running the Application:

To run your application, open a terminal, navigate to the directory containing app.js, and execute:

```
node app.js
```

Topic-3 REPL Terminal

REPL stands for Read-Eval-Print Loop. It is an interactive environment provided by Node.js to execute JavaScript code on the fly.

Starting the REPL:

To start the REPL, simply type node in your terminal and press Enter.

```
$ node
```

Example Session:

```
> const message = 'Hello, REPL';
> console.log(message);
Hello, REPL
> 2 + 3
5
```

Explanation:

1. **> const message = 'Hello, REPL';**: Declares a constant variable message with the value 'Hello, REPL'.
2. **> console.log(message);**: Prints the value of message to the console.
3. **> 2 + 3**: Evaluates the arithmetic expression 2 + 3 and prints the result, 5.

Topic-4 Online REPL Terminal

Several online platforms offer REPL environments for running Node.js code without any installation:

- **repl.it**: https://replit.com/
- **JSFiddle**: https://jsfiddle.net/
- **CodeSandbox**: https://codesandbox.io/

Example on repl.it:

1. Navigate to https://replit.com/.
2. Select the "Node.js" template.
3. Write your code in the editor and click "Run" to see the output in the integrated terminal.

Topic-5 REPL Commands

1) .help

Displays a list of all available REPL commands.

```
> .help
```

2) .break

Exits from a multiline expression.

> .break

3) .clear

Resets the REPL context, clearing all variables and functions.

> .clear

4) .save

Saves the current REPL session to a file.

> .save ./session.txt

5) .load

Loads a file into the current REPL session.

> .load ./session.txt

6) .exit

Exits the REPL environment.

> .exit

Example Session with Commands:

```
> .help
> const a = 10
> const b = 20
> a + b
30
> .save ./session.txt
> .exit
```

Explanation:

1. **.help**: Displays help information.
2. **const a = 10**: Declares a constant a with the value 10.
3. **const b = 20**: Declares a constant b with the value 20.
4. **a + b**: Evaluates the expression a + b, resulting in 30.
5. **.save ./session.txt**: Saves the session to session.txt.
6. **.exit**: Exits the REPL.

Topic-6 Stopping REPL

To stop the REPL, you can use the .exit command or press Ctrl + C twice.

Using .exit:

```
> .exit
```

Using Ctrl + C:

Press Ctrl + C twice to exit.

```
^C
(To exit, press ^C again or type .exit)
^C
```

Explanation:

1. **> .exit**: Typing .exit will exit the REPL.
2. **^C**: Pressing Ctrl + C once interrupts the current command, displaying a message about how to exit.
3. **(To exit, press ^C again or type .exit)**: Instructions displayed after the first Ctrl + C.
4. **^C**: Pressing Ctrl + C again exits the REPL.

JAMAL MOHAMED COLLEGE (Autonomous)

Department of Computer Applications

VALUE ADDED COURSE

III BCA – NODE JS

Unit-2 Installing Modules using NPM

Topic-1 Installing Modules using NPM

NPM (Node Package Manager) is the default package manager for Node.js and is used to install, update, and manage dependencies in a Node.js project.

To install a module using npm, you can use the following command:

```
bash

npm install <module-
```

Example:

```
bash

npm install express
```

This will install the express module and add it to your node_modules directory.

Topic-2 Attributes of package.json

package.json is a file that holds various metadata relevant to the project and is used to manage the project's dependencies, scripts, version, and other settings.

Common attributes include:

- **name**: The name of your project.
- **version**: The version of your project.
- **description**: A brief description of your project.
- **main**: The entry point of your application.
- **scripts**: Scripts that can be run with npm run <script-name>.
- **dependencies**: A list of dependencies required by your project.
- **devDependencies**: A list of dependencies required only for development.
- **author**: The author of the project.
- **license**: The license under which the project is released.

Example package.json:

json

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "A simple project",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {},
  "author": "John Doe",
  "license": "ISC"
}
```

Topic-3 Uninstalling a Module

To uninstall a module, use the following command:

```
bash

npm uninstall
```

Example:

bash

npm uninstall express

This will remove the express module from your node_modules directory and update package.json and package-lock.json.

Topic-4 Updating a Module

To update a module to the latest version, use the following command:

```
bash

npm update <module-name>
```

Example:

bash
npm update express

To update all modules, you can use:

bash
npm update

Topic-5 Search a Module

To search for a module, you can use the npm search feature:

```
bash

npm search <search-term>
```

Example:

bash

npm search express

Topic-6 Create a Module

To create a module in Node.js, you need to write a JavaScript file and export the functionalities you want to make available.

Example of a simple module (myModule.js):

```
function greet(name)
 {
  return `Hello, ${name}!`;
}

module.exports = greet;
```

To use this module in another file:

```
const greet = require('./myModule');
console.log(greet('World'));  // Output: Hello, World!
```

Topic-7 Callback Concept: What is a Callback

A callback is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Topic-8 Blocking Code Example

Blocking code executes synchronously, meaning each operation must wait for the previous one to complete before proceeding.

Example:

```
const fs = require('fs');
const data = fs.readFileSync('file.txt');  // Blocking call
console.log(data.toString());
console.log('This message is logged after reading the file');
```

In this example, readFileSync blocks the execution until the file is read completely.

Explanation:-

const fs = require('fs');

const fs: This defines a constant named fs.
require('fs'): This statement imports the built-in Node.js fs (file system) module, which provides functions to interact with the file system, such as reading and writing files.

const fs = require('fs');: This assigns the imported fs module to the constant fs, making the file system functions available through fs.

const data = fs.readFileSync('file.txt'); // Blocking call

const data: This defines a constant named data.

fs.readFileSync('file.txt'): This calls the readFileSync function from the fs module, which reads the content of the file named file.txt synchronously. The function reads the entire contents of the file and returns it as a buffer.

Blocking call: The readFileSync function is a blocking call, meaning the program will wait until the file is completely read before moving on to the next line of code.

const data = fs.readFileSync('file.txt');: This assigns the content of the file.txt file (returned as a buffer) to the constant data.

console.log(data.toString());

console.log: This function prints messages to the console.
data.toString(): The data constant holds a buffer (binary data). The toString() method converts this buffer to a string.

console.log(data.toString());: This prints the string representation of the file's contents to the console.

console.log('This message is logged after reading the file');

console.log: Again, this function prints messages to the console.

'This message is logged after reading the file': This is a string that is passed to console.log to be printed.

console.log('This message is logged after reading the file');: This prints the message "This message is logged after reading the file" to the console. Because fs.readFileSync is a blocking call, this line will only execute after the file has been completely read and its contents logged.

Topic-9 Non-Blocking Code Example

Non-blocking code executes asynchronously, meaning operations can start before the previous ones have completed.

Example:

```
const fs = require('fs');
fs.readFile('file.txt', (err, data) => {  // Non-blocking call
  if (err) throw err;
  console.log(data.toString());
}
);
console.log('This message is logged before reading the file');
```

Line-by-line explanation:

1. fs.readFile('file.txt', (err, data) => { ... }): Asynchronously read the file file.txt. The callback function is called when the reading operation is complete.
2. if (err) throw err;: If an error occurs during the file read operation, throw the error.
3. console.log(data.toString());: If no error, log the contents of the file.
4. console.log('This message is logged before reading the file');: This message is logged immediately after initiating the read operation, demonstrating non-blocking behavior.

JAMAL MOHAMED COLLEGE (Autonomous)

Department of Computer Applications

VALUE ADDED COURSE

III BCA – NODE JS

Unit-3 Lecture Notes: Node.js Event Loop and Event Emitter

Topic-1 Event Loop

The event loop is a core concept in Node.js that allows it to perform non-blocking, asynchronous operations despite the fact that is single-threaded. Understanding the event loop is crucial for building efficient, high-performance applications.

How It Works

Node.js operates on a single-threaded event loop model, which means it handles multiple operations by offloading tasks (such as I/O operations) to the system kernel whenever possible. The kernel is multi-threaded, allowing Node.js to handle many operations concurrently without blocking the main thread.

Phases of the Event Loop

The event loop has several phases that Node.js cycles through:

1. **Timers**: This phase executes callbacks scheduled by setTimeout() and setInterval().
2. **I/O Callbacks**: Executes callbacks for some system operations like TCP errors, but not all I/O operations.
3. **Idle, Prepare**: Internal use only.
4. **Poll**: Retrieves new I/O events; executing I/O related callbacks (excluding close callbacks, the ones scheduled by timers, and setImmediate()).
5. **Check**: Executes callbacks scheduled by setImmediate().
6. **Close Callbacks**: Executes close event callbacks like socket.on('close', ...).

Example

Consider a simple example to understand the event loop phases better:

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

console.log('Start');

setTimeout(() => {
  console.log('Timeout callback');
}, 0);

setImmediate(() => {
  console.log('Immediate callback');
});

fs.readFile(__filename, () => {
  console.log('File read callback');
});

process.nextTick(() => {
  console.log('Next tick callback');
});

console.log('End');
```

Explanation:

1. **Synchronous Code**:
    - console.log('Start'): Executes immediately.
    - console.log('End'): Executes immediately.
2. **Next Tick Queue**:
    - process.nextTick(() => ...): nextTick callbacks are executed after the current operation completes, before the event loop continues.
3. **Timers Phase**:
    - setTimeout(() => ..., 0): Scheduled to run in the next cycle of the event loop.
4. **Check Phase**:

    - setImmediate(() => ...): Scheduled to run after the poll phase.
5. **Poll Phase**:
    - fs.readFile(__filename, () => ...): Poll phase waits for I/O events, here the file read completes and the callback is executed.

The output of the above code would be:

Start
End
Next tick callback
File read callback
Immediate callback
Timeout callback

Key Points

- **Single-threaded**: Node.js uses a single-thread for execution, but can handle multiple I/O operations using the underlying system's capabilities.
- **Non-blocking I/O**: Operations like network requests, file system operations are non-blocking.
- **Callbacks and Promises**: Node.js relies on callbacks, promises, and async/await to handle asynchronous operations.
- **Phases**: The event loop has distinct phases for handling timers, I/O callbacks, and other operations.

Topic-2 Event-Driven Programming

Node.js uses an event-driven programming model. In this model, the flow of the program is determined by events such as user actions, sensor outputs, or message passing. Instead of waiting for operations to complete, the program moves on and continues to respond to new events.

Example

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

```
// Create an HTTP server that listens to server ports and gives a response
const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
});

// The server listens on port 3000
server.listen(3000, '127.0.0.1', () => {
  console.log('Server running at http://127.0.0.1:3000/');
});
```

Explanation:

- http.createServer((req, res) => ...): Creates an HTTP server that responds to incoming requests.
- res.writeHead(200, {'Content-Type': 'text/plain'}): Sets the response header.
- res.end('Hello World\n'): Ends the response and sends 'Hello World' to the client.
- server.listen(3000, '127.0.0.1', ...): The server listens on port 3000.

Topic-3 How Node.js Applications Work

Node.js applications are built on an event-driven, non-blocking I/O model. This architecture enables Node.js to handle many simultaneous connections efficiently, making it ideal for I/O-intensive tasks such as web servers, APIs, and real-time applications.

Key Concepts

1. **Single-Threaded**: Node.js operates on a single-threaded event loop.
2. **Event-Driven**: Node.js applications are built around an event-driven architecture.
3. **Non-Blocking I/O**: Node.js uses non-blocking I/O operations to handle multiple requests efficiently.

Lifecycle of a Node.js Application

1. **Initialization**: Node.js initializes the program, runs any synchronous code, and sets up the environment.
2. **Event Registration**: The application registers event handlers for different events.
3. **Event Loop Execution**: The event loop starts, processing events and executing corresponding event handlers.

Event Loop

The event loop is central to how Node.js handles asynchronous operations. It continuously checks for new events to process, running the appropriate callbacks as events occur.

Example Application

Let's build a simple Node.js HTTP server to illustrate how a Node.js application works.

Step-by-Step Example

1. Import Required Modules

First, we need to import the built-in http module:

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

2. Create the Server

We create an HTTP server using the http.createServer() method. This method takes a callback function that will be executed whenever an HTTP request is received.

```
const server = http.createServer((req, res) => {
  res.statusCode = 200; // Set the response status code to 200 (OK)
  res.setHeader('Content-Type', 'text/plain'); // Set the response content type
  res.end('Hello, World!\n'); // Send the response and close the connection
});
```

3. Listen on a Port

We make the server listen on a specific port (e.g., 3000) and hostname (e.g., '127.0.0.1').

```
const port = 3000;
const hostname = '127.0.0.1';
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Full Example

Here's the complete code for the simple HTTP server:

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

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

Detailed Explanation

- **Importing Modules**: const http = require('http'); imports the built-in HTTP module.
- **Creating the Server**: http.createServer((req, res) => {...}); creates an HTTP server. The callback function handles incoming requests.
    - res.statusCode = 200; sets the HTTP status code to 200 (OK).
    - res.setHeader('Content-Type', 'text/plain'); sets the response content type to plain text.
    - res.end('Hello, World!\n'); sends the response body and ends the response.

- **Listening on a Port**: server.listen(port, hostname, () => {...}); makes the server listen on the specified port and hostname.
    - The callback function logs a message when the server starts listening.

Topic-4 EventEmitter class

The EventEmitter class in Node.js is a core module that facilitates the creation and handling of custom events. It forms the foundation of event-driven programming in Node.js, allowing objects to emit named events that cause listeners to be called.

Importing EventEmitter

To use EventEmitter, you need to import it from the events module:

```
const EventEmitter = require('events');
```

Creating an EventEmitter Instance

You can create an instance of EventEmitter as follows:

```
const myEmitter = new EventEmitter();
```

Basic Usage

Here's a simple example demonstrating how to use EventEmitter:

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();

// Register an event listener
myEmitter.on('event', () => {
  console.log('An event occurred!');
});

// Emit the event
myEmitter.emit('event');
```

Explanation:

- myEmitter.on('event', () => {...}): Registers a listener for the 'event' event.
- myEmitter.emit('event'): Emits the 'event' event, invoking all registered listeners.

Topic-5 Methods of EventEmitter

.on(event, listener)

Registers a listener for the specified event. The listener will be called every time the event is emitted.

```
myEmitter.on('event', (arg1, arg2) => {
  console.log(`Event with arguments: ${arg1}, ${arg2}`);
});

myEmitter.emit('event', 'arg1', 'arg2'); // Output: Event with arguments: arg1, arg2
```

.emit(event, [...args])

Emits the specified event, calling all listeners registered for that event with the supplied arguments.

```
myEmitter.emit('event', 'arg1', 'arg2');
```

.once(event, listener)

Registers a one-time listener for the specified event. The listener is invoked only the first time the event is emitted and then removed.

```
myEmitter.once('eventOnce', () => {
  console.log('This will only be logged once');
});

myEmitter.emit('eventOnce'); // Output: This will only be logged once
myEmitter.emit('eventOnce'); // No output
```

.removeListener(event, listener)

Removes a specific listener from the event. The listener must match exactly the one registered.

```
const listener = () => {
  console.log('This listener will be removed');
};

myEmitter.on('removeEvent', listener);
myEmitter.removeListener('removeEvent', listener);
myEmitter.emit('removeEvent'); // No output
```

.removeAllListeners([event])

Removes all listeners for the specified event. If no event is specified, all listeners for all events are removed.

```
myEmitter.on('anotherEvent', () => {
  console.log('This will not be logged as all listeners are removed');
});

myEmitter.removeAllListeners('anotherEvent');
myEmitter.emit('anotherEvent'); // No output
```

Topic-6 Class Methods

EventEmitter.defaultMaxListeners

This property sets the default maximum number of listeners that can be added for any single event. The default value is 10.

```
EventEmitter.defaultMaxListeners = 15;
```

EventEmitter.listenerCount(emitter, event)

Returns the number of listeners listening to the specified event.

```
const listenerCount = EventEmitter.listenerCount(myEmitter, 'event');
console.log(listenerCount); // Output: Number of listeners for 'event'
```

Topic-7 Events

Common events emitted by EventEmitter objects include:

- **newListener**: Emitted when a new listener is added. The event listener is passed the event name and the listener function.
- **removeListener**: Emitted when a listener is removed. The event listener is passed the event name and the listener function.

```
myEmitter.on('newListener', (event, listener) => {
  console.log(`Added listener for event: ${event}`);
});

myEmitter.on('removeListener', (event, listener) => {
  console.log(`Removed listener for event: ${event}`);
});
```

Example

Here's a complete example that demonstrates the usage of EventEmitter:

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();

myEmitter.on('event', (a, b) => {
  console.log(a, b, this); // Prints arguments and `this` context
});

myEmitter.emit('event', 'a', 'b');

myEmitter.once('eventOnce', () => {
  console.log('This will only be logged once');
});

myEmitter.emit('eventOnce');
myEmitter.emit('eventOnce');

const listener = () => {
  console.log('This listener will be removed');
};

myEmitter.on('removeEvent', listener);
myEmitter.removeListener('removeEvent', listener);
myEmitter.emit('removeEvent');

myEmitter.on('anotherEvent', () => {
  console.log('This will not be logged as all listeners are removed');
});
myEmitter.removeAllListeners('anotherEvent');
myEmitter.emit('anotherEvent');

console.log(EventEmitter.listenerCount(myEmitter, 'event')); // Output: Number
of listeners for 'event'
```

Conclusion

The EventEmitter class in Node.js is a powerful utility for creating and handling custom events, making it a cornerstone of the event-driven architecture of Node.js applications. By understanding how to register, emit, and manage events, you can build scalable and maintainable applications that efficiently handle asynchronous operations.

JAMAL MOHAMED COLLEGE (Autonomous)

Department of Computer Applications

VALUE ADDED COURSE

III BCA – NODE JS

Unit-4 Lecture Notes: Node.js Streams and Web Module

Topic-1 What are Streams?

Streams are a fundamental concept in Node.js, providing an efficient way to handle data that might not be available all at once. Instead of reading or writing data in one go, streams allow you to work with data in chunks.

Types of Streams:

1. **Readable Streams**: Streams from which data can be read (e.g., fs.createReadStream()).
2. **Writable Streams**: Streams to which data can be written (e.g., fs.createWriteStream()).
3. **Duplex Streams**: Streams that are both readable and writable (e.g., net.Socket).
4. **Transform Streams**: Duplex streams that can modify or transform the data as it is written and read (e.g., zlib.createGzip()).

Topic-2 Reading from a Stream

Creating a Readable Stream

javascript

```
const fs = require('fs');
const readableStream = fs.createReadStream('example.txt', 'utf8');

// 'data' event is emitted when a chunk of data is available to read
readableStream.on('data', (chunk) => {
  console.log('New chunk received:', chunk);
});

// 'end' event is emitted when there is no more data to read
readableStream.on('end', () => {
  console.log('No more data to read.');
});

// 'error' event is emitted if an error occurs during reading
readableStream.on('error', (err) => {
  console.error('An error occurred:', err);
});
```

Explanation:

- require('fs'): Import the file system module.
- fs.createReadStream('example.txt', 'utf8'): Create a readable stream for 'example.txt' with 'utf8' encoding.
- readableStream.on('data', ...): Set up a listener for the 'data' event, which is triggered when a chunk of data is available.
- readableStream.on('end', ...): Set up a listener for the 'end' event, which is triggered when the stream has been fully read.
- readableStream.on('error', ...): Set up a listener for the 'error' event, which is triggered if an error occurs.

Topic-3 Writing to a Stream

Creating a Writable Stream

javascript

```
const fs = require('fs');
const writableStream = fs.createWriteStream('example.txt');

// Write data to the stream
writableStream.write('Hello, world!\n', 'utf8');
writableStream.write('Writing more data...\n', 'utf8');

// Signal that no more data will be written to the stream
writableStream.end(() => {
  console.log('Finished writing data.');
});

// Handle errors during writing
writableStream.on('error', (err) => {
  console.error('An error occurred:', err);
});
```

Explanation:

- fs.createWriteStream('example.txt'): Create a writable stream for 'example.txt'.
- writableStream.write('Hello, world!\n', 'utf8'): Write 'Hello, world!\n' to the stream with 'utf8' encoding.
- writableStream.write('Writing more data...\n', 'utf8'): Write another piece of data to the stream.
- writableStream.end(() => ...): Signal that no more data will be written and set up a callback for when the writing is finished.
- writableStream.on('error', ...): Set up a listener for the 'error' event to handle any errors that occur.

Topic-4 Piping the Streams

Piping Data from One Stream to Another

javascript

```
const fs = require('fs');
const readableStream = fs.createReadStream('input.txt');
const writableStream = fs.createWriteStream('output.txt');

// Pipe data from readable stream to writable stream
readableStream.pipe(writableStream);

readableStream.on('end', () => {
  console.log('Finished piping data.');
});
```

Explanation:

- fs.createReadStream('input.txt'): Create a readable stream for 'input.txt'.
- fs.createWriteStream('output.txt'): Create a writable stream for 'output.txt'.
- readableStream.pipe(writableStream): Pipe data from the readable stream to the writable stream.
- readableStream.on('end', ...): Set up a listener for the 'end' event to signal when piping is finished.

Topic-5 Chaining the Streams

Using Transform Streams to Chain Operations

javascript

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

```
// Create a readable stream, a transform stream (gzip), and a writable stream
const readableStream = fs.createReadStream('input.txt');
const writableStream = fs.createWriteStream('output.txt.gz');
const gzip = zlib.createGzip();

// Pipe data through gzip transform stream and then to writable stream
readableStream.pipe(gzip).pipe(writableStream);

writableStream.on('finish', () => {
  console.log('File successfully compressed.');
});
```

Explanation:

- require('zlib'): Import the zlib module for compression.
- zlib.createGzip(): Create a transform stream that compresses data using Gzip.
- readableStream.pipe(gzip).pipe(writableStream): Pipe data from the readable stream, through the gzip transform stream, and into the writable stream.
- writableStream.on('finish', ...): Set up a listener for the 'finish' event to signal when compression and writing are finished.

Web Module

Topic-6 What is a Web Server?

A web server is a software application that handles HTTP requests from clients (such as web browsers) and serves them with responses, typically HTML pages, images, stylesheets, and scripts.

Topic-7 Web Application Architecture

1. **Client**: The user's web browser or any client application.
2. **Server**: The backend application that processes requests, runs business logic, and communicates with databases.
3. **Database**: Stores and retrieves data needed by the server.

Topic-8 Creating a Web Server using Node.js

Basic HTTP Server

javascript

```
const http = require('http');
// Create an HTTP server
const server = http.createServer((req, res) => {
  res.statusCode = 200; // Set the status code to 200 (OK)
  res.setHeader('Content-Type', 'text/plain'); // Set the content type to plain text
  res.end('Hello, World!\n'); // Send a response to the client
});
// Server listens on port 3000 and localhost (127.0.0.1)
server.listen(3000, '127.0.0.1', () => {
  console.log('Server running at http://127.0.0.1:3000/');
});
```

Explanation:

- require('http'): Import the HTTP module.
- http.createServer((req, res) => ...): Create an HTTP server with a request listener function.
- res.statusCode = 200: Set the response status code to 200 (OK).
- res.setHeader('Content-Type', 'text/plain'): Set the response header to indicate plain text content.
- res.end('Hello, World!\n'): Send the response body and end the response.
- server.listen(3000, '127.0.0.1', ...): Make the server listen on port 3000 and localhost (127.0.0.1).

Topic-9 Making a Request to a Node.js Server

Using HTTP Module to Make Requests

javascript

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

// Define options for the HTTP request
const options = {
  hostname: 'www.example.com',
  port: 80,
  path: '/',
  method: 'GET'
};

// Make the HTTP request
const req = http.request(options, (res) => {
  console.log(`STATUS: ${res.statusCode}`); // Log the response status code
  console.log(`HEADERS: ${JSON.stringify(res.headers)}`); // Log the response headers

  // Listen for data chunks from the response
  res.on('data', (chunk) => {
    console.log(`BODY: ${chunk}`);
  });

  // Listen for the end of the response
  res.on('end', () => {
    console.log('No more data in response.');
  });
});

// Handle any errors during the request
req.on('error', (e) => {
  console.error(`Problem with request: ${e.message}`);
});
```

```
req.end(); // End the request
```

Explanation:

- http.request(options, (res) => ...): Make an HTTP request with specified options and a response callback.
- options: An object containing hostname, port, path, and method for the request.
- res.on('data', (chunk) => ...): Listen for data chunks from the response.
- res.on('end', ...): Listen for the end of the response.
- req.on('error', (e) => ...): Handle any errors during the request.
- req.end(): End the request.

Topic-10 Creating a Web Client using Node.js

Using HTTP Module to Create a Web Client

javascript

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

// Define options for the HTTP request
const options = {
  hostname: 'localhost',
  port: 3000,
  path: '/',
  method: 'GET'
};

// Make the HTTP request
const req = http.request(options, (res) => {
  res.setEncoding('utf8'); // Set the encoding for the response data

  // Listen for data chunks from the response
  res.on('data', (chunk) => {
```

```
    console.log(`BODY: ${chunk}`);
  });
});

// Handle any errors during the request
req.on('error', (e) => {
  console.error(`Problem with request: ${e.message}`);
});

req.end(); // End the request
```

Explanation:

- options: An object containing hostname, port, path, and method for the request.
- res.setEncoding('utf8'): Set the encoding for the response data to 'utf8'.
- res.on('data', (chunk) => ...): Listen for data chunks from the response.
- req.on('error', (e) => ...): Handle any errors during the request.
- req.end(): End the request.

Conclusion

Understanding how to work with streams and the web module in Node.js is essential for efficiently handling data and building robust web applications. Streams allow for efficient data processing, while the web module enables the creation of HTTP servers and clients.

JAMAL MOHAMED COLLEGE (Autonomous)

Department of Computer Applications

VALUE ADDED COURSE

III BCA – NODE JS

Unit-1 Lecture Notes: File System

File System - Synchronous vs Asynchronous Operations

Topic-1 Introduction to Node.js File System (fs) Module

The Node.js File System (fs) module allows you to work with the file system on your computer. This module provides both synchronous and asynchronous methods for all operations to work with files and directories.

Key Concepts

1. **Synchronous vs Asynchronous**
2. **Opening a File**
3. **Getting File Information**
4. **Writing to a File**
5. **Reading a File**
6. **Closing a File**
7. **Truncating a File**
8. **Deleting a File**
9. **Creating a Directory**
10. **Reading a Directory**
11. **Removing a Directory**
12. **Methods Reference**

Topic-2 Synchronous vs Asynchronous Operations

Synchronous Operations

- Synchronous methods execute line by line, pausing execution until the operation completes.
- Use these methods when you need to ensure operations are completed before moving on to the next step.
- Example: fs.readFileSync()

Asynchronous Operations

- Asynchronous methods execute without blocking the main thread, allowing other operations to run simultaneously.
- Use these methods to avoid blocking the event loop, especially for I/O operations.
- Example: fs.readFile()

Code Example:

javascript

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

// Synchronous example
const dataSync = fs.readFileSync('example.txt', 'utf8');
console.log('Synchronous read:', dataSync);

// Asynchronous example
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log('Asynchronous read:', data);
});
```

Topic-3 Opening a File

Syntax

- **Asynchronous**: fs.open(path, flags, mode, callback)
- **Synchronous**: fs.openSync(path, flags, mode)

Flags Example:

- 'r' - read
- 'w' - write
- 'a' - append

Code Example:

javascript

```
// Asynchronous
fs.open('example.txt', 'r', (err, fd) => {
  if (err) throw err;
  console.log('File opened successfully:', fd);
});

// Synchronous
const fdSync = fs.openSync('example.txt', 'r');
console.log('File opened successfully:', fdSync);
```

Topic-4 Getting File Information

Syntax

- **Asynchronous**: fs.stat(path, callback)
- **Synchronous**: fs.statSync(path)

Code Example:

javascript

```
// Asynchronous
fs.stat('example.txt', (err, stats) => {
  if (err) throw err;
  console.log('File stats:', stats);
});

// Synchronous
const statsSync = fs.statSync('example.txt');
console.log('File stats:', statsSync);
```

Topic-5 Writing to a File

Syntax

- **Asynchronous**: fs.writeFile(file, data, options, callback)
- **Synchronous**: fs.writeFileSync(file, data, options)

Code Example:

javascript

```
// Asynchronous
fs.writeFile('example.txt', 'Hello, world!', (err) => {
  if (err) throw err;
  console.log('File written successfully');
});

// Synchronous
fs.writeFileSync('example.txt', 'Hello, world!');
console.log('File written successfully');
```

Topic-6 Reading a File

Syntax

- **Asynchronous**: fs.readFile(path, options, callback)
- **Synchronous**: fs.readFileSync(path, options)

Code Example:

javascript

```
// Asynchronous
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log('File content:', data);
});

// Synchronous
const dataSync = fs.readFileSync('example.txt', 'utf8');
console.log('File content:', dataSync);
```

Topic-7 Closing a File

Syntax

- **Asynchronous**: fs.close(fd, callback)
- **Synchronous**: fs.closeSync(fd)

Code Example:

javascript

```
// Asynchronous
fs.open('example.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
    console.log('File closed successfully');
  });
});

// Synchronous
const fdSync = fs.openSync('example.txt', 'r');
fs.closeSync(fdSync);
console.log('File closed successfully');
```

Topic-8 Truncating a File

Syntax

- **Asynchronous**: fs.truncate(path, len, callback)
- **Synchronous**: fs.truncateSync(path, len)

Code Example:

javascript

```
// Asynchronous
fs.truncate('example.txt', 10, (err) => {
  if (err) throw err;
  console.log('File truncated successfully');
```

```
});

// Synchronous
fs.truncateSync('example.txt', 10);
console.log('File truncated successfully');
```

Topic-9 Deleting a File

Syntax

- **Asynchronous**: fs.unlink(path, callback)
- **Synchronous**: fs.unlinkSync(path)

Code Example:

javascript

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

// Synchronous
fs.unlinkSync('example.txt');
console.log('File deleted successfully');
```

Topic-10 Creating a Directory

Syntax

- **Asynchronous**: fs.mkdir(path, options, callback)
- **Synchronous**: fs.mkdirSync(path, options)

Code Example:

javascript

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

// Synchronous
fs.mkdirSync('exampleDir');
console.log('Directory created successfully');
```

Topic-11 Reading a Directory

Syntax

- **Asynchronous**: fs.readdir(path, options, callback)
- **Synchronous**: fs.readdirSync(path, options)

Code Example:

javascript

```
// Asynchronous
fs.readdir('exampleDir', (err, files) => {
  if (err) throw err;
  console.log('Directory contents:', files);
});

// Synchronous
const filesSync = fs.readdirSync('exampleDir');
console.log('Directory contents:', filesSync);
```

Topic-12 Removing a Directory

Syntax

- **Asynchronous**: fs.rmdir(path, callback)
- **Synchronous**: fs.rmdirSync(path)

Code Example:

javascript

```
// Asynchronous
fs.rmdir('exampleDir', (err) => {
  if (err) throw err;
  console.log('Directory removed successfully');
});

// Synchronous
fs.rmdirSync('exampleDir');
console.log('Directory removed successfully');
```

Topic-13 Methods Reference

- **fs.open(path, flags, mode, callback)**
- **fs.openSync(path, flags, mode)**
- **fs.stat(path, callback)**
- **fs.statSync(path)**
- **fs.writeFile(file, data, options, callback)**
- **fs.writeFileSync(file, data, options)**
- **fs.readFile(path, options, callback)**
- **fs.readFileSync(path, options)**
- **fs.close(fd, callback)**
- **fs.closeSync(fd)**
- **fs.truncate(path, len, callback)**
- **fs.truncateSync(path, len)**
- **fs.unlink(path, callback)**
- **fs.unlinkSync(path)**
- **fs.mkdir(path, options, callback)**
- **fs.mkdirSync(path, options)**
- **fs.readdir(path, options, callback)**
- **fs.readdirSync(path, options)**
- **fs.rmdir(path, callback)**
- **fs.rmdirSync(path)**

Conclusion

Understanding the differences between synchronous and asynchronous file operations in Node.js is crucial for building efficient and non-blocking applications. Use synchronous methods for small-scale scripts where execution order is critical and asynchronous methods for larger, performance-sensitive applications to keep the event loop unblocked.