Node Core Modules

SD540 Server-Side Programming

Maharishi University of Management

Masters of Software Development

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

Node Core Modules

Node provides many core libraries, to use the modules:

```
import moduleName from 'node:moduleName'; // always use node namespace
```

Read the API documentation at https://nodejs.org/

`Events` core module

To send a new custom event use **`.emit(eventName, eventValue)`**.

To listen and handle the event use **`.on(eventName, f)`** the function will receive the **`eventValue`** as a parameter.

Using events core module

```
import events from 'node:events';

class MSDStudent extends events {
    constructor(){ super() }

    triggerGraduation(year: number) {
        this.emit('graduation', year);
    }
}

const student = new MSDStudent();
student.on('graduation', (year)=> console.log(`Congrats ${year} graduates.`));
student.triggerGraduation(2024);
```

Using path core Module

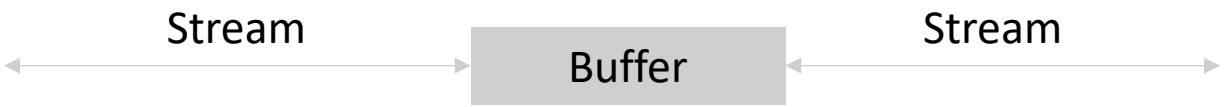
```
import { join } from 'node:path';

const path = join('this', 'is', 'a', 'path');
const pathToFile = join(__dirname, 'file.txt');
```

Buffers and Streams

Buffer: A chunk of memory, limited in size, and cannot be resized.

Stream: a sequence of data made available over time. Pieces of data that eventually combine into a whole.



Example: If you stream a movie over the internet, you are processing the data as being received. there will be a buffer inside your browser that receive the data from the server, process it when enough data is available (few seconds of pictures) , and stream it out to your monitor.

Buffer is a Super Data Type

`Buffer` is a Node primitive (similar to boolean, string, number..etc).

Node uses buffers any time it can, most Node API is built using buffers behind the scenes, for example: reading from the file system, receiving packets over the network.

You can pass buffers to any Node API requiring data to be sent.

Buffers are very helpful when reading files and images and sending streams.

Buffers Encoding

Buffers data are saved in binary, so it can be interpreted in many ways depending on the length of characters. This is why we need to specify the encoding when we read data from Buffers.

Character set vs. Encoding

Character set: A representation of characters as numbers, each character gets a number. Unicode and ASCII are character sets. Where character get a number assigned to them.

Encoding: How characters are stored in binary, the numbers (code points) are converted and stored in binary.

	h	e	l	l	o
Unicode character set	**104**	**101**	**108**	**108**	**111**
UTF-8 encoding	**01101000**	**01100101**	**01101100**	**01101100**	**01101111**

Buffer Example

```
const emptybuffer = Buffer.alloc(8); // allocate a Buffer with 8 bytes
const buffer = Buffer.from('Hello'); // create Buffer of 5 bytes and fill it

console.log(buffer.toString()); // Hello

buffer.write('ipp', 1); // overwrite data in the buffer
console.log(buffer.toString()); // Hippo
```

Read Files

```
import { readFileSync } from 'node:fs';
import { readFile } from 'node:fs/promises';
import { join } from 'node:path';

const pathToFile = join(__dirname, 'file.txt');

const content = readFileSync(pathToFile); // Buffer

const filecontent = await readFile(pathToFile); // Buffer
```

Write Files

```
import { writeFileSync } from 'node:fs';
import { writeFile } from 'node:fs/promises';
import { join } from 'node:path';

const pathToFile = join(__dirname, 'file.txt');

writeFileSync(pathToFile, 'Hello');

await writeFile(pathToFile, 'Hello');
```

Delete Files

```
import { unlink } from 'node:fs/promises';
import { join } from 'node:path';

const pathToFile = join(__dirname, 'file.txt');

await unlink(pathToFile);
```

Revisiting the Event Loop

```
import { readFile } from 'node:fs/promises';
import { join } from 'node:path';

const pathToFile = join(__dirname, 'app.ts');

(async function () {
    const filecontent = await readFile(pathToFile);
    setTimeout(() => { console.log('timeout') }, 0);
    setImmediate(() => { console.log('immediate') });
    process.nextTick(() => console.log('nexttick'));
})();
```

Based on your understanding to the Event Loop, what is the output of the code?

Streams

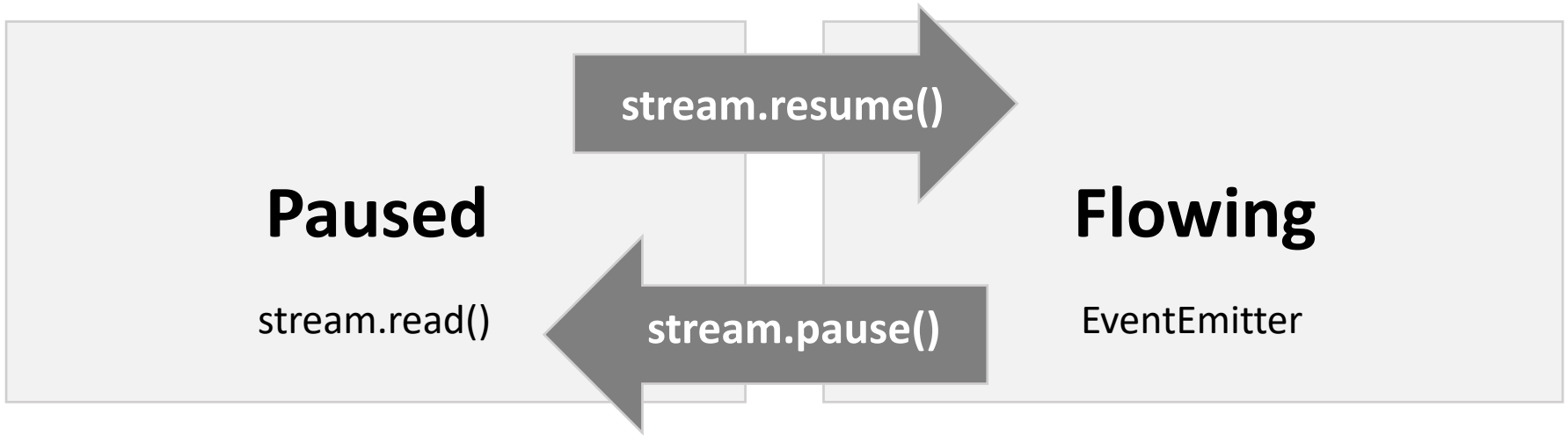
Collection of data unavailable all at once and doesn't have to fit in memory. Streaming the data means an application processes the data while still receiving it. This is useful for extra-large datasets, like video or database migrations.

Stream types:

- **Readable**
- **Writable**
- **Duplex**
- **Transform**

Streams inherit from `events`. So they have access to `on()` and `emit()`.

Readable Streams



Readable streams have two main modes that affect the way we consume them:
They can be in either **paused mode** or **flowing mode** (pull vs push mode).

All readable streams start in paused mode and they will be changed later to flowing and then can be switched back to paused when needed.

Using Streams

```
import { createReadStream, createWriteStream } from 'node:fs';
import { join } from 'node:path';

const pathToSource = join(__dirname, 'file1.txt');
const pathToDestination = join(__dirname, 'file2.txt');

const readableStream = createReadStream(pathToSource);
const writableStream = createWriteStream(pathToDestination);

readableStream.pipe(writableStream);
```

Compress Files

```
import { createReadStream, createWriteStream } from 'node:fs';
import { createGzip } from 'node:zlib';
import { join } from 'node:path';

const pathToSource = join(__dirname, 'file.txt');
const pathToDestination = join(__dirname, 'file.txt.gz');

const readableStream = createReadStream(pathToSource);
const gzipStream = createGzip(); // Duplex Stream
const writableStream = createWriteStream(pathToDestination);

readableStream.pipe(gzipStream).pipe(writableStream);
```

Create Web Server

```
import { createServer, IncomingMessage, Server, ServerResponse } from 'node:http';

const server: Server = createServer();

server.on('request', function(req: IncomingMessage, res: ServerResponse) {
   res.writeHead(200, {'Content-Type': 'text/plain'}); // Inform client of content type
   res.write('Hello '); // content
   res.write('World!'); // content
   res.end();
});

server.listen(3000, ()=> console.log(`listening to 3000`));

// When the server receives a request, Node creates req and res objects
// and schedules the request-handler function in the Poll queue
```

Node treats TCP Packets as Stream

Stream Large Files

```
import { createServer, IncomingMessage, ServerResponse } from 'node:http';
import { createReadStream } from 'node:fs';

const server = createServer();

server.on('request', function(req: IncomingMessage, res: ServerResponse) {
   createReadStream('/big/file').pipe(res);
});

server.listen(3000, ()=> console.log(`listening to 3000`));
```

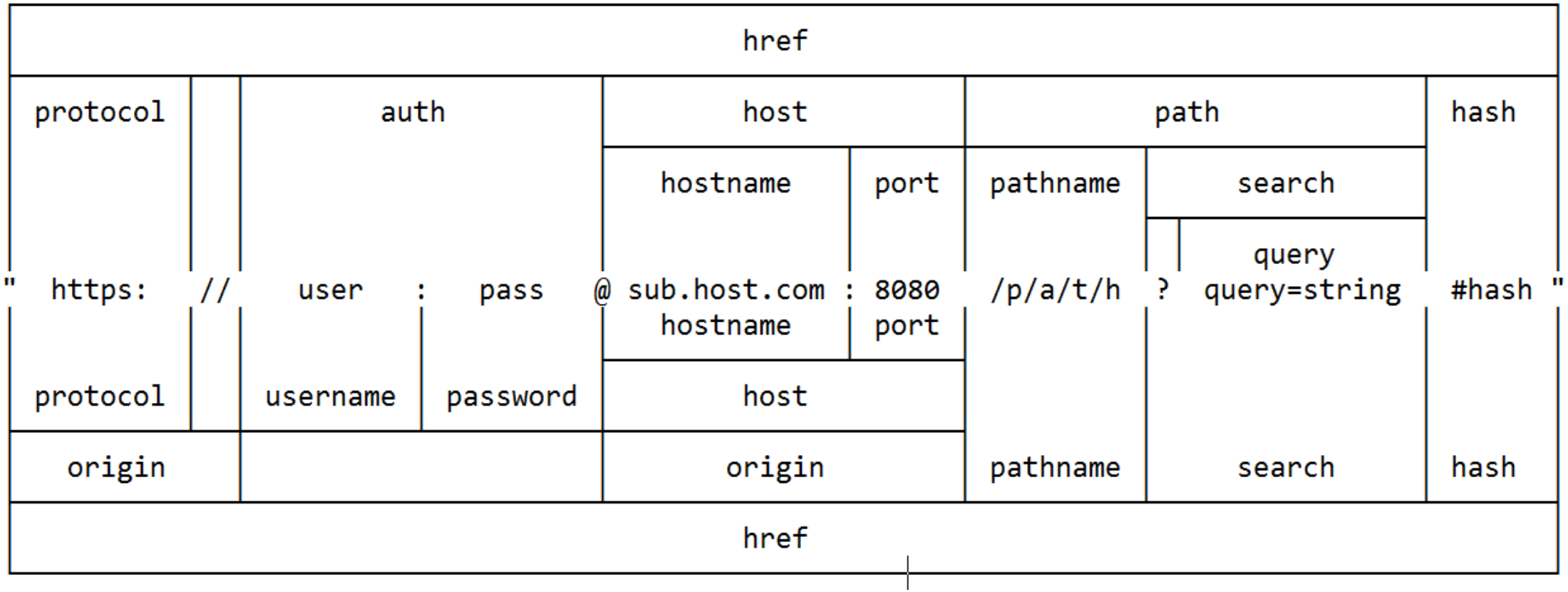
Using `url` and `querystring` core modules

```
import url from 'node:url';
import querystring from 'node:querystring';

const myUrl = url.parse('https://user:pass@host.com:8080/p/a/t/h?name=asaad', true);

const myQuery = querystring.parse('faculty=Asaad%20Saad&course=SD540');
```

URL Object



https://nodejs.org/api/url.html

Routing and sending JSON response

```
function(req, res) {
   if (req.url === '/api') {
       res.writeHead(200, { 'Content-Type': 'application/json' });
       const obj = { firstname: 'Asaad', lastname: 'Saad' };
       res.end(JSON.stringify(obj));
   } else {
       res.writeHead(404);
       res.end();
   }
}
```