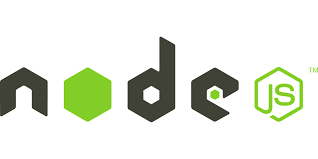
**Study on Streams (Node js)**



Submitted by :

Jayakumar Mhk

Karan

K Chandana Supriya

**Under the guidance of**

Mr. Naga Venkat Samurai

**Table of Contents**

| S.No | Title | Pg. No. |
| --- | --- | --- |
| 1. | Introduction |  |
| 2. | Read and Write Streams |  |
| 3. | Duplex Stream |  |
| 4. | Conclusion |  |
| 5. | Appendix |  |

## 

## **Introduction**

## **What are Streams?**

A stream is an abstract interface that lets you perform specific tasks continuously. A stream is an ***EventEmitter*** that implements different methods. A user can use streams to perform a variety of functions like reading, write, read & write, and transform functions.

In a Node.js environment, streams are used to work with streaming data. It provides users with an API that helps them in creating the streaming interface. The data here is received in parts and is read in parts as well.

**Types of Streams**

Streams are capable of performing various functions depending on the category under which they fall. We can divide streams in the following categories –

1. Readable Streams
2. Writable Streams
3. Duplex

**Readable and Writable Streams**

****

**Readable Streams**

A readable stream is the one from which you can read data. They come in two variants or two different reading modes – *Paused* and *Flowing*. All the readable streams run in pause mode unless you mingle with them. It means users have to request to get an output from the stream. Flowing mode ensures that the data flows continuously. You can choose to either consume or ignore the output that you see on the screen.

The *fs.createReadStream* function is used to create a readable stream or you can *read()* continuously until all of the data finishes reading. To make the stream flow, you will need an additional bit of code. Here is an example of reading stream –

var fs = require('fs');

var readableStream = fs.createReadStream('file.txt');

var data = '';

var chunk;

readableStream.on('readable', function() {

while ((chunk=readableStream.read()) != null) {

data += chunk;

}

});

readableStream.on('end', function()

{ console.log(data)

});

In the above example, the *read()* function will read data from the internal buffer and return it to the user. As soon as there is no data to read further, it will terminate the loop and return *null.*

While analyzing the readable stream, we realize that the *data* event and the *end* event are its most important events. The data event gets emitted whenever the stream sends a pile of data to the user. The end event comes into the scene when there is no more data left to be consumed by the stream.

**Writable Streams**

Another *EventEmitter*, writable streams allow users to write on the chosen destination. We use *write()* function to initiate a writable stream. The API here is simpler and prefers the use of methods instead of events. It is very easy for users to learn a writable stream. Here is a basic example of writable streams –

var fs = require('fs');

var readableStream = fs.createReadStream('file1.txt');

var writableStream = fs.createWriteStream('file2.txt'); readableStream.setEncoding('utf8');

readableStream.on('data',

function(chunk) {

writableStream.write(chunk);

});

The above example ,we are using a readable stream to read the inputs, and then the *write()* stream writes it on the destination. You will get a Boolean as soon as the function is successful. If the return is true, the process is complete. In case there is any discrepancy, the function will return false.

Two significant events are usually attached to a writable stream – *drain* and *finish* events. The drain event is an indicator that the stream is capable of receiving more data. Whereas the finish event signifies that the underlying system has received the entire data.

**Duplex Streams**

The first two streams are good at performing individual functions. With the duplex streams, you can perform both of their functions collectively without the burden of extra coding. It’s almost like the child inheriting the genes of both the mother and the father. Mostly a duplex stream consists of two individual streams, one of which is for flowing in, and the other one is for flowing out. Below is an example of a basic duplex stream –

net.createServer(socket => {

socket.pipe(socket)

}).listen(8001);

**Pipes**

Pipes can **be used to connect multiple streams together**. One of the most common example is to pipe the read and write stream together for the transfer of data from one file to the other. Node. js is often also tagged as an event driven framework, and it's very easy to define events in Node.

In the given example, the *socket* has been piped to itself, which in turn will ensure the creation of a duplex stream. The first *socket* is a readable stream, whereas the next one is a writable stream. Whenever you run the function, the netcat will try to send some data to the server. On the other hand, the server will try to write the data received.

### 

### 

### **Benefits of Streams**

Being used worldwide, there has to be some benefits attached to streams. Other than the fact that even a beginner can implement them, here are some other benefits of using streams –

**Time Efficiency**

What is the benefit of a chain? It ensures that the person at the back is traveling along with the person at the front. Regarding the stream environment, due to piping, output data of a stream is transferred as an input of another stream. It ensures the timely processing of massive data due to constant flowing. Piping allows us to process multiple stages at the same time, thereby reducing unnecessary time wastage.

**Spatial Efficiency**

What do you do when you have a small buffer but a larger input file? You create a stream to display the data as soon as possible to ensure that the buffer remains free for the next lot. Suppose you want to read a file that is around 35 MB in size and display output. But the buffer is limited to 25 MB. What do you do in such a situation?

To avert the crisis, create a readable stream. What will it do? As soon as a part of data is read from your input, you can get the pile on to the buffer, display it, and clear it off to make space for the next lot. It will ensure that the data is not being leaked and fully processed.

**Composed**

Due to the piping ability of the streams, you can always know that the data stay composed in nature. In spite of a heavy code, streams will still connect to each other due to piping. It means that one input is piped to output and so on. You can also use duplex and transform streams to read and write at the same time.

#### **Conclusion**

Streams are an integral part of Node.js and have helped to simplify codes for the developers. With the help of streams, developers can now build a code in far less time than earlier. With so many other environments available that do the same thing, streams are the reason why most people have stayed on Node.js. This article should have given you a fair idea of what streams are and how they operate.