STREAMS IN NODE.JS.

Streams are collections of data — just like arrays or strings.

Streams are objects that let you read data from a source or write data to a destination in continuous fashion.

Streams are one of the fundamental concepts of Node.js. Streams are a type of data-handling methods and are used to read or write input into output sequentially. Streams are used to handle reading/writing files or exchanging information in an efficient way.

Streams are methods used in handling writing/reading files, network communications, or any kind of end-to-end information exchange in an efficient way.

They form the fundamental concepts that power the node.js application.

With streams, data can be exchanged in small parts, which reduces a lot of memory usage.

Unlike the traditional way of reading files into memory all in a program at once before processing its content, which can be an issue if is not enough memory space to contain the files.

However, Streams on the other hand do not keep all the files in the memory all at once before processing the content, it reads chunks of data and processes the contents of the file piece by piece

This pattern of breaking down files into chunks of data becomes more of an advantage to us when working with large amounts of data because we no longer need to worry about memory space being enough to contain the files.

Let’s examine the usage of streams in the YouTube app:

Youtube offers streaming services, with this service you don’t need to download the videos or audio feeds all at once, but you are allowed to watch the videos or listen to the audio immediately, this is possible because your browser can receive the videos and audio as a continuous flow of chunks. If you learn to work with these streams of bits, you'll be able to build performant and valuable applications. When you watch a video on YouTube. You don't have to wait until the full video downloads. Once a small amount buffers, it starts to play, and the rest keeps on downloading as you watch.

The official Node.js documentation defines streams as “A stream is an abstract interface for working with streaming data in Node.js.”

In short, Streams are objects in Node.js that lets the user read data from a source or write data to a destination in a continuous manner.

What makes streams powerful while dealing with large amounts of data is that instead of reading a file into memory all at once, streams actually read chunks of data, processing its content data without keeping it all in memory.

Advantages of Streams over other data handling methods:

* Time Efficient: We don’t have to wait until entire file has been transmitted. We can start processing data as soon as we have it.
* Memory Efficient: We don’t have to load huge amount of data in memory before we start processing.
* Composability feature: With the stream composability feature, we can build microservices in node.js. With composability, we can carry out complex applications that interact and interconnect with data between different pieces of code.
* Used in building applications: With streams, we can create real-world applications such as video streaming apps.

Node.js includes a built-in module called stream which lets us work with streaming data. In this article, we will explain how you can use the stream module with some simple examples. We'll also describe how you can build pipelines gluing different streams together to build performant applications for complex use cases.

Before we dive into building applications, it's important to understand the features provided by the Node.js stream module.

In Node.js, there are four types of streams: −

* Readable − Stream which is used for read operation. We can read data from these streams. Example: fs.createReadStream(). A readable stream can read data from a particular data source, most commonly, from a file system. Other common uses of readable streams in Node.js applications are:
* process.stdin - To read user input via stdin in a terminal application.
* http.IncomingMessage - To read an incoming request's content in an HTTP server or to read the server HTTP response in an HTTP client.
* Writable − Stream which is used for write operation. We can write data to these streams. Example: fs.createWriteStream().

You use writable streams to write data from an application to a specific destination, for example, a file.process.stdout can be used to write data to standard output and is used internally by console.log. Next up are duplex and transform streams, which you can define as 'hybrid' stream types built on readable and writable streams.

* Duplex − Stream which can be used for both read and write operation. Streams that are both, Writable as well as Readable. Example: net.socket.

A duplex stream is a combination of both readable and writable streams. It provides the capability to write data to a particular destination and read data from a source. The most common example of a duplex stream is net.Socket, used to read and write data to and from a socket.

It's important to know that readable and writable sides operate independently from one another in a duplex stream. The data does not flow from one side to the other.

* Transform − A type of duplex stream where the output is computed based on input. Streams that can modify or transform the data as it is written and read. Example: zlib.createDeflate.

A transform stream is slightly similar to a duplex stream, but the readable side is connected to the writable side in a transform stream.

A good example would be the crypto.Cipher class which implements an encryption stream. Using a crypto.Cipher stream, an application can write plain text data into the writable side of a stream and read encrypted ciphertext out of the readable side of the stream. The transformative nature of this type of stream is why they are called 'transform streams'.

Side-note: Another transform stream is stream.PassThrough, which passes data from the writable side to the readable side without any transformation. Though this might sound trivial, Passthrough streams are very useful for building custom stream implementations and pipelines (e.g., creating multiple copies of one stream's data).

Each type of Stream is an EventEmitter instance and throws several events at different instance of times. For example, some of the commonly used events are −

data − This event is fired when there is data is available to read.

end − This event is fired when there is no more data to read.

error − This event is fired when there is any error receiving or writing data.

finish − This event is fired when all the data has been flushed to underlying system.