
# Streams

# The stream module

- Streams are the objects that facilitate reading data from a source and writing it to a destination.

- There are four types of streams in Node.js:
  - Readable: This stream is used for read operations.
  - Writable: This stream is used for write operations.
  - Duplex: This stream can be used for both read and write operations.
  - Transform: It is type of duplex stream where the output is computed according to input.

# Advantages of streams

- Streams are a way to handle reading/writing file operations, network communications, or any kind of end-to-end information exchange in an efficient way.
- Instead of a program loading the entire file contents into memory at once, streams read chunks of data piece by piece, processing its content without keeping it all in memory.
- This makes streams really powerful when working with large amounts of data
- A file size can be larger than your free memory space, making it impossible to read the whole file into the memory in order to process it, but you can achieve this with streams.

# Advantages of streams

- For example - "streaming" services such as YouTube or Netflix. These services don't make you download the video and audio feed all at once. Instead, your browser receives the video as a continuous flow of chunks, allowing the recipients to start watching and/or listening almost immediately.
- Streams basically provide two major advantages compared to other data handling methods:
  - Memory efficiency: You don't need to load large amounts of data in memory before you are able to process it.
  - Time efficiency: It takes significantly less time to start processing data as soon as you have it, rather than having to wait with processing until the entire data has been transmitted.

# The stream module

- Each type of stream is an EventEmitter instance and fires several events at different times.
- Following are some commonly used events:
  - data: This event is fired when there is data available to read.
  - end: This event is fired when there is no more data available to read.
  - error: This event is fired when there is any error receiving or writing data.
  - finish: This event is fired when all data has been flushed to underlying system.

# fs.createReadStream

```
fs.createReadStream (path, [options])
```

- Returns a readable stream object for file at path.
- You may then perform stream operations on the returned object, such as pipe().

The following options are available:

flags: File mode argument as a string. Defaults to 'r'.

encoding: One of utf8, ascii, or base64. Defaults to no encoding.

fd: One may set path to null, instead passing the call a file descriptor.

mode: Octal representation of file mode, defaulting to '0666'.

highWaterMark: The chunk size in bytes. Default is 64kb.

# fs.createWriteStream

```
fs.createWriteStream (path, [options])
```

- Returns a writable stream object for file at path.

The following options are available:

flags: File mode argument as a string. Defaults to 'w'.

encoding: One of utf8, ascii, or base64. Defaults to no encoding.

mode: Octal representation of file mode, defaulting to '0666'.

start: An offset indicating the position in the file where writing should begin.

# Reading from stream

```
var fs = require("fs");
var data = '';

// Create a readable stream
var readerStream = fs.createReadStream('input-read.txt');

// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error
readerStream.on('data', function (chunk) {
    data += chunk;
});
readerStream.on('end', function () {
    console.log(data);
});
readerStream.on('error', function (err) {
    console.log(err.stack);
});
console.log("Program Completed Successfully");
```

# Writing to stream

```
var fs = require("fs");

var data = 'This data will be written to a file using write
stream.';

// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');

// Write the data to stream with encoding to be utf8
writerStream.write(data, 'UTF8');

// Mark the end of file
writerStream.end();

// Handle stream events --> finish, and error
writerStream.on('finish', function () {
    console.log("Write completed.");
});
writerStream.on('error', function (err) {
    console.log(err.stack);
});
console.log("Program Completed Successfully");
```

# Piping streams

- Piping is a mechanism where output of one stream is used as an input to another stream.
- There is no limit on piping operation.

```
var fs = require("fs");

// Create a readable stream
var readerStream = fs.createReadStream('input-pipe.txt');

// Create a writable stream
var writerStream = fs.createWriteStream('output-pipe.txt');

// Pipe the read and write operations
// read input-pipe.txt and write data to output-pipe.txt
readerStream.pipe(writerStream);

console.log("Program Ended")
```

# Chaining streams

- Chaining stream is a mechanism of creating a chain of multiple stream operations by connecting output of one stream to another stream.
- It is generally used with piping operation.
- Below is an example of piping and chaining to compress a file and then decompress the same file.

# Compress / Decompress

```
var fs = require("fs");

var zlib = require('zlib');

// Compress the file input.txt to input.txt.gz
fs.createReadStream('input-compress.txt')
    .pipe(zlib.createGzip())
    .pipe(fs.createWriteStream('input-compress.txt.gz'));

console.log("File Compressed.");
```

```
var fs = require("fs");

var zlib = require('zlib');

// Decompress the file input.txt.gz to input.txt
fs.createReadStream('input-compress.txt.gz')
    .pipe(zlib.createGunzip())
    .pipe(fs.createWriteStream('input-decompress.txt'));

console.log("File Decompressed.");
```

# Practice

Write Node.js application(s) –

1) To demonstrate read/write operations using streams.

2) To demonstrate all the methods of readline module.

3) To swap contents of two files. Accept files names from the user using readline.

4) To copy 'n' lines from a file to another file. Accept the number of lines to be copied from the user using readline.

5) To demonstrate piping, chaining in streams using compress/decompress.

# Practice

Write Node.js application(s) –

6) To copy contents from one file to another file in the reverse order of words. Note: Program should work for single as well as multi-line file.

7) To accept a file name and contents from the user. If the file already exists, prompt the user to confirm if the file should be overwritten. If yes, then write the contents to the specified file. If not, then inform the user that file has not been modified.