Understanding Streams in Node.js

Streams are one of the fundamental concepts that power Node.js applications. They are data-handling method and are used to read or write input into output sequentially.

What makes streams unique, is that instead of a program reading a file into memory **all at once** like in the traditional way, 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**, for example, 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. That's where streams come to the rescue!

> Consider this scenario, where we have a GET API on our server and a mp4 video file. This server, is not implementing stream, and when the API is called, it reads the entire mp4 file and sends it as response. So, in this case, if there are 5 parallel API calls, then the server would read the file 5 times and the memory consumption would be 5x of the size of the mp4 file.

Why streams

Streams basically provide two major advantages compared to other data handling methods:

1. **Memory efficiency:** you don't need to load large amounts of data in memory before you are able to process it
2. **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 payload has been transmitted

There are 4 types of streams in Node.js:

1. **Writable:** streams to which we can write data. For example, `fs.createWriteStream()` lets us write data to a file using streams.
2. **Readable:** streams from which data can be read. For example: `fs.createReadStream()` lets us read the contents of a file.
3. **Duplex:** streams that are both Readable and Writable. For example, `net.Socket`
4. **Transform:** streams that can modify or transform the data as it is written and read. For example, in the instance of file-compression, you can write compressed data and read decompressed data to and from a file.

Readable Streams

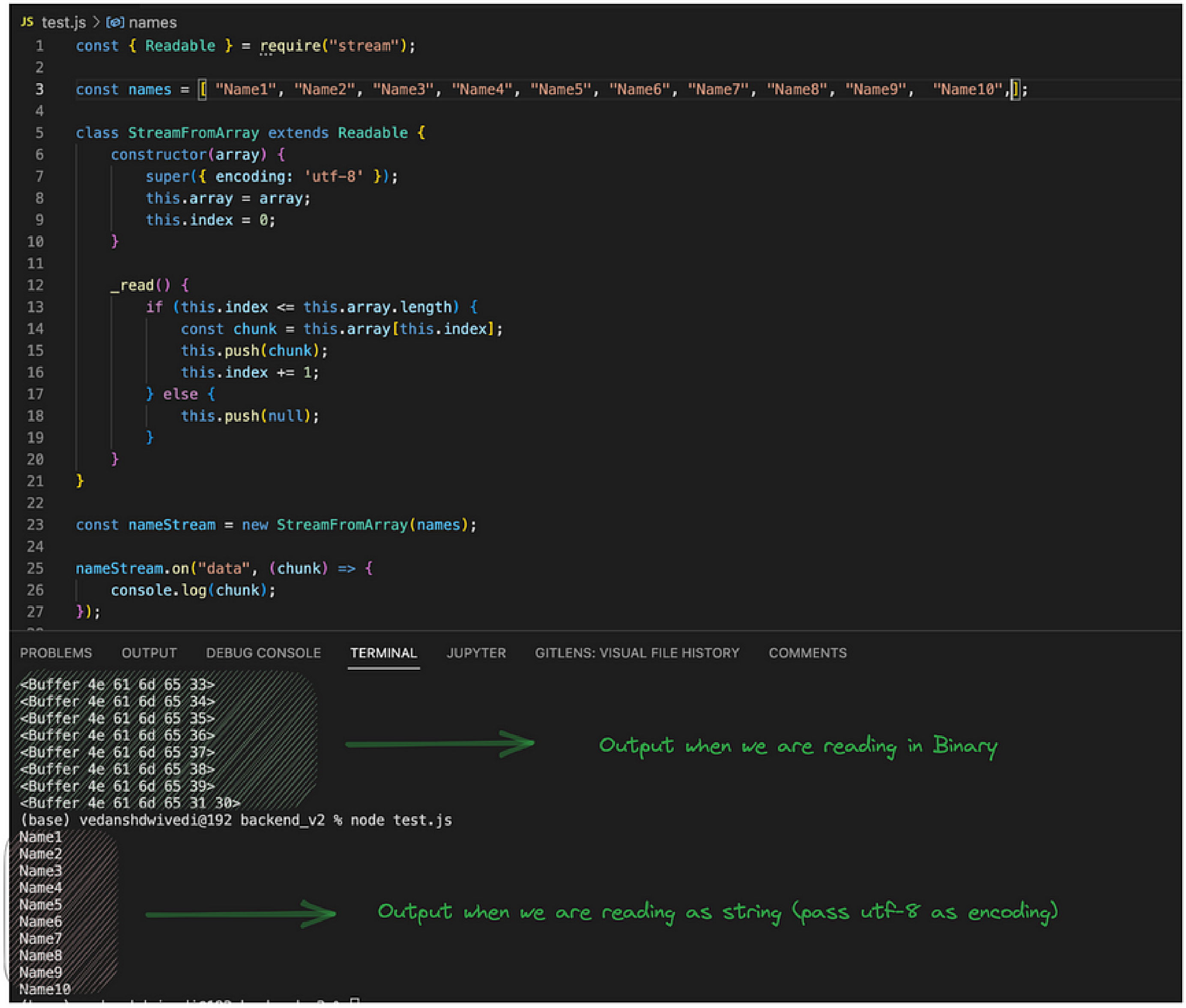
A readable stream, reads data from a source and then feeds it to a pipeline bit by bit. For instance, let’s create a read stream that can read from an array, and then pass it chunk by chunk.
Streams make use of event-emitters, which means they raise events, so we can listen for events.

- On a read stream, we want to listen to **‘data**’ events. So, basically whenever a data event is raised, a small chunk of data is passed to the callback function.
- Another important readStream event is **‘end’** , which is raised when the stream has finished reading.
- When we encounter any error while handling the data etc, an **‘error’** event is emitted, with an error object.

Also, streams can read data in the following manner:

1. Binary: Reads data as binary. This is the default option.
2. String: Reads data as string. To enable this, we need to pass the **encoding: ‘utf-8’** as the option
3. Object: Reads data as objects. To enable this, we need to pass the **objectMode: true** as the option

This code snippet shows the implementation of streams in the two different modes (Here, to implement the StreamFromArray, we have created a class and extended the stream.Readable class according to our need, which can be skipped, the main idea was to show the use of readable streams. Most of the time, we don't need this custom implementations, we can implement the already existing stream types).



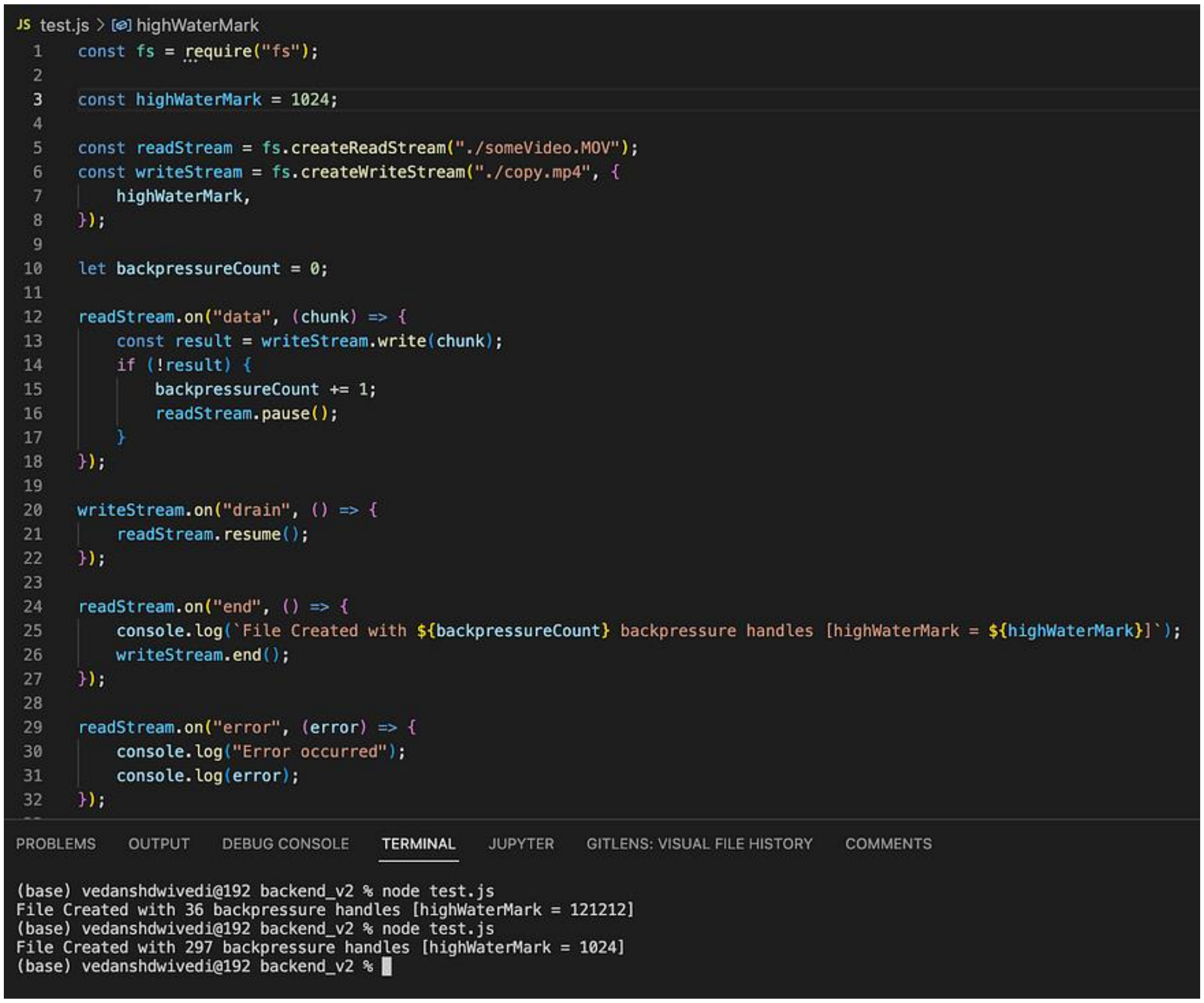
```javascript
const { Readable } = require("stream");

const names = [ "Name1", "Name2", "Name3", "Name4", "Name5", "Name6", "Name7", "Name8", "Name9",  "Name10",];

class StreamFromArray extends Readable {
    constructor(array) {
        super({ encoding: 'utf-8' });
        this.array = array;
        this.index = 0;
    }

    _read() {
        if (this.index <= this.array.length) {
            const chunk = this.array[this.index];
            this.push(chunk);
            this.index += 1;
        } else {
            this.push(null);
        }
    }
}

const nameStream = new StreamFromArray(names);

nameStream.on("data", (chunk) => {
    console.log(chunk);
});
```

```
<Buffer 4e 61 6d 65 33>
<Buffer 4e 61 6d 65 34>
<Buffer 4e 61 6d 65 35>
<Buffer 4e 61 6d 65 36>
<Buffer 4e 61 6d 65 37>
<Buffer 4e 61 6d 65 38>
<Buffer 4e 61 6d 65 39>
<Buffer 4e 61 6d 65 31 30>
(base) vedanshdwivedi@192 backend_v2 % node test.js
Name1
Name2
Name3
Name4
Name5
Name6
Name7
Name8
Name9
Name10
```

Output when we are reading in Binary
Output when we are reading as string (pass utf-8 as encoding)

Writable Streams

Writable Streams are like the counter-parts of readable streams. Just like how we read data in chunks, we can also write data in chunks. Writable Streams represent a destination for the incoming data. Writable stream enables us to read data from a readable source and do something with that data. Also, just like the readableStreams, writableStreams are also implemented in numerous places.

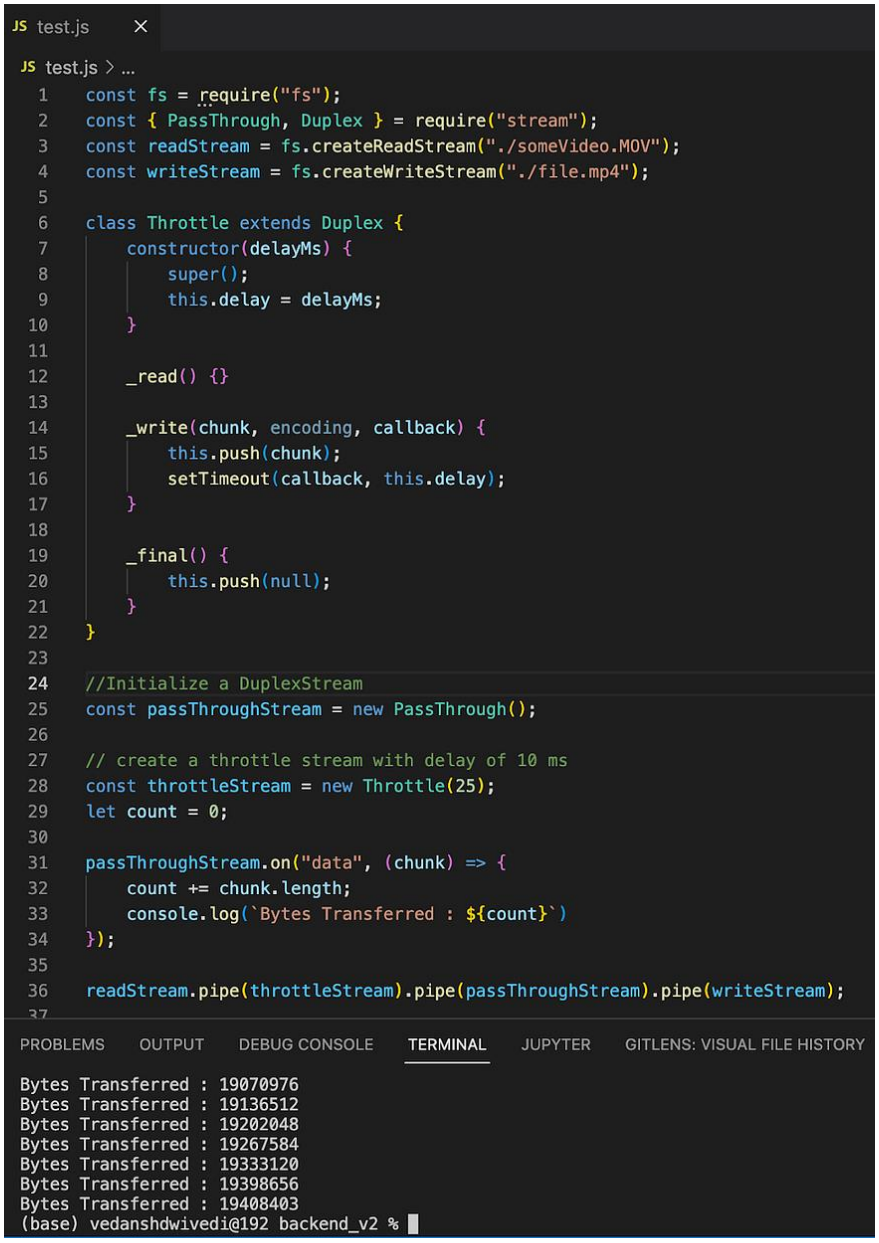


```javascript
const fs = require("fs");

const highWaterMark = 1024;

const readStream = fs.createReadStream("./someVideo.MOV");
const writeStream = fs.createWriteStream("./copy.mp4", {
    highWaterMark,
});

let backpressureCount = 0;

readStream.on("data", (chunk) => {
    const result = writeStream.write(chunk);
    if (!result) {
        backpressureCount += 1;
        readStream.pause();
    }
});

writeStream.on("drain", () => {
    readStream.resume();
});

readStream.on("end", () => {
    console.log(`File Created with ${backpressureCount} backpressure handles [highWaterMark = ${highWaterMark}]`);
    writeStream.end();
});

readStream.on("error", (error) => {
    console.log("Error occurred");
    console.log(error);
});
```

```
(base) vedanshdwivedi@192 backend_v2 % node test.js
File Created with 36 backpressure handles [highWaterMark = 121212]
(base) vedanshdwivedi@192 backend_v2 % node test.js
File Created with 297 backpressure handles [highWaterMark = 1024]
(base) vedanshdwivedi@192 backend_v2 %
```

Duplex Stream

A duplex stream is a stream which implements both readable and writable. Readable stream will pipe data into the stream and the duplex stream can also write that data

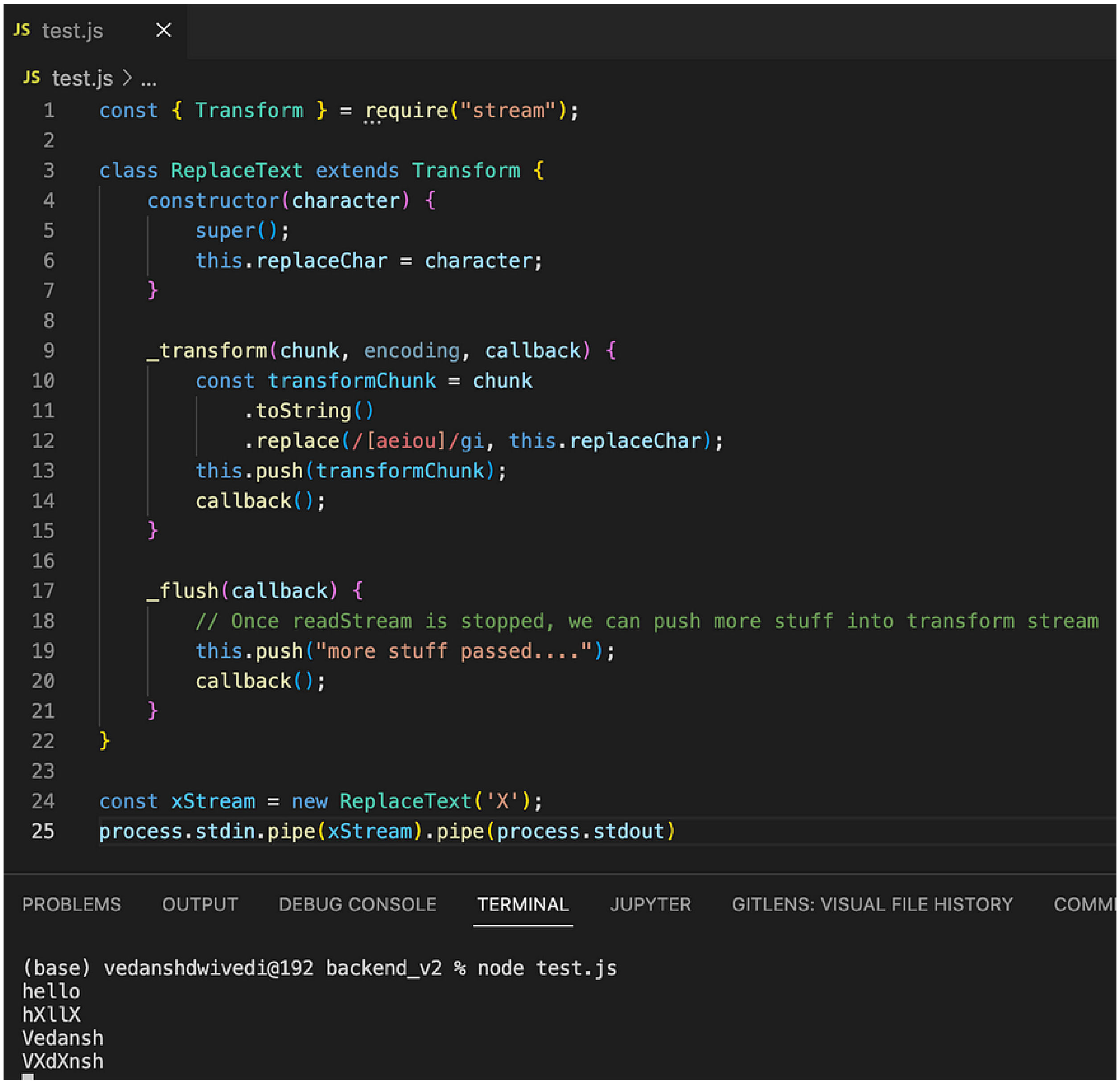


```javascript
const fs = require("fs");
const { PassThrough, Duplex } = require("stream");
const readStream = fs.createReadStream("./someVideo.MOV");
const writeStream = fs.createWriteStream("./file.mp4");

class Throttle extends Duplex {
    constructor(delayMs) {
        super();
        this.delay = delayMs;
    }

    _read() {}

    _write(chunk, encoding, callback) {
        this.push(chunk);
        setTimeout(callback, this.delay);
    }

    _final() {
        this.push(null);
    }
}

//Initialize a DuplexStream
const passThroughStream = new PassThrough();

// create a throttle stream with delay of 10 ms
const throttleStream = new Throttle(25);
let count = 0;

passThroughStream.on("data", (chunk) => {
    count += chunk.length;
    console.log(`Bytes Transferred : ${count}`)
});

readStream.pipe(throttleStream).pipe(passThroughStream).pipe(writeStream);
```

```
Bytes Transferred : 19070976
Bytes Transferred : 19136512
Bytes Transferred : 19202048
Bytes Transferred : 19267584
Bytes Transferred : 19333120
Bytes Transferred : 19398656
Bytes Transferred : 19408403
(base) vedanshdwivedi@192 backend_v2 %
```

Transform Streams

Transform streams are special kind of duplex streams. Instead of simply passing the data from the read stream to the write stream, transform streams change the data.

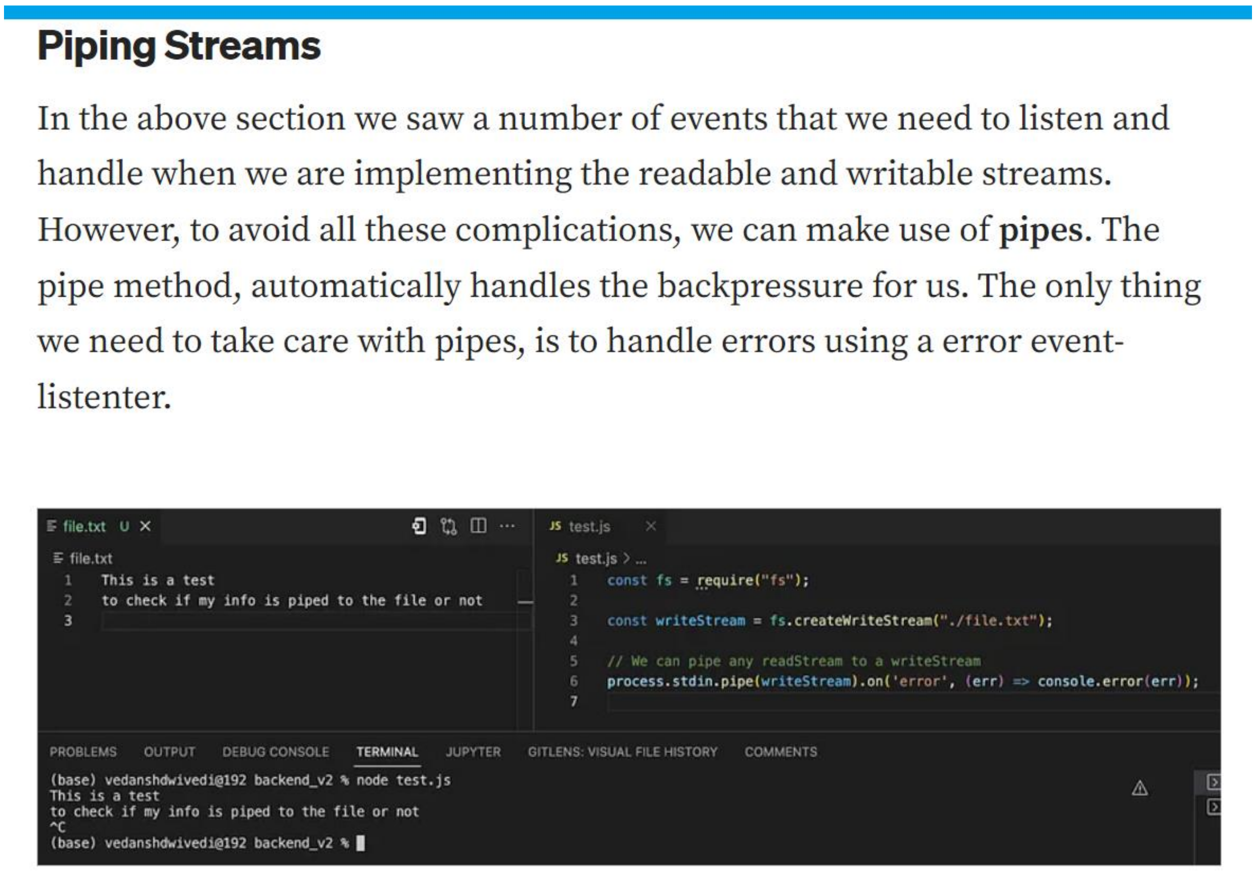


```javascript
const { Transform } = require("stream");

class ReplaceText extends Transform {
    constructor(character) {
        super();
        this.replaceChar = character;
    }

    _transform(chunk, encoding, callback) {
        const transformChunk = chunk
            .toString()
            .replace(/[aeiou]/gi, this.replaceChar);
        this.push(transformChunk);
        callback();
    }

    _flush(callback) {
        // Once readStream is stopped, we can push more stuff into transform stream
        this.push("more stuff passed....");
        callback();
    }
}

const xStream = new ReplaceText('X');
process.stdin.pipe(xStream).pipe(process.stdout)
```

```
(base) vedanshdwivedi@192 backend_v2 % node test.js
hello
hXllX
Vedansh
VXdXnsh
```

pipeline()

Piping is a mechanism where we provide the output of one stream as the input to another stream. It is normally used to get data from one stream and to pass the output of that stream to another stream. There is no limit on piping operations. In other words, piping is used to process streamed data in multiple steps.



Piping Streams

In the above section we saw a number of events that we need to listen and handle when we are implementing the readable and writable streams. However, to avoid all these complications, we can make use of **pipes**. The pipe method, automatically handles the backpressure for us. The only thing we need to take care with pipes, is to handle errors using a error event-listenter.

```
This is a test
to check if my info is piped to the file or not
```

```js
const fs = require("fs");

const writeStream = fs.createWriteStream("./file.txt");

// We can pipe any readStream to a writeStream
process.stdin.pipe(writeStream).on('error', (err) => console.error(err));
```

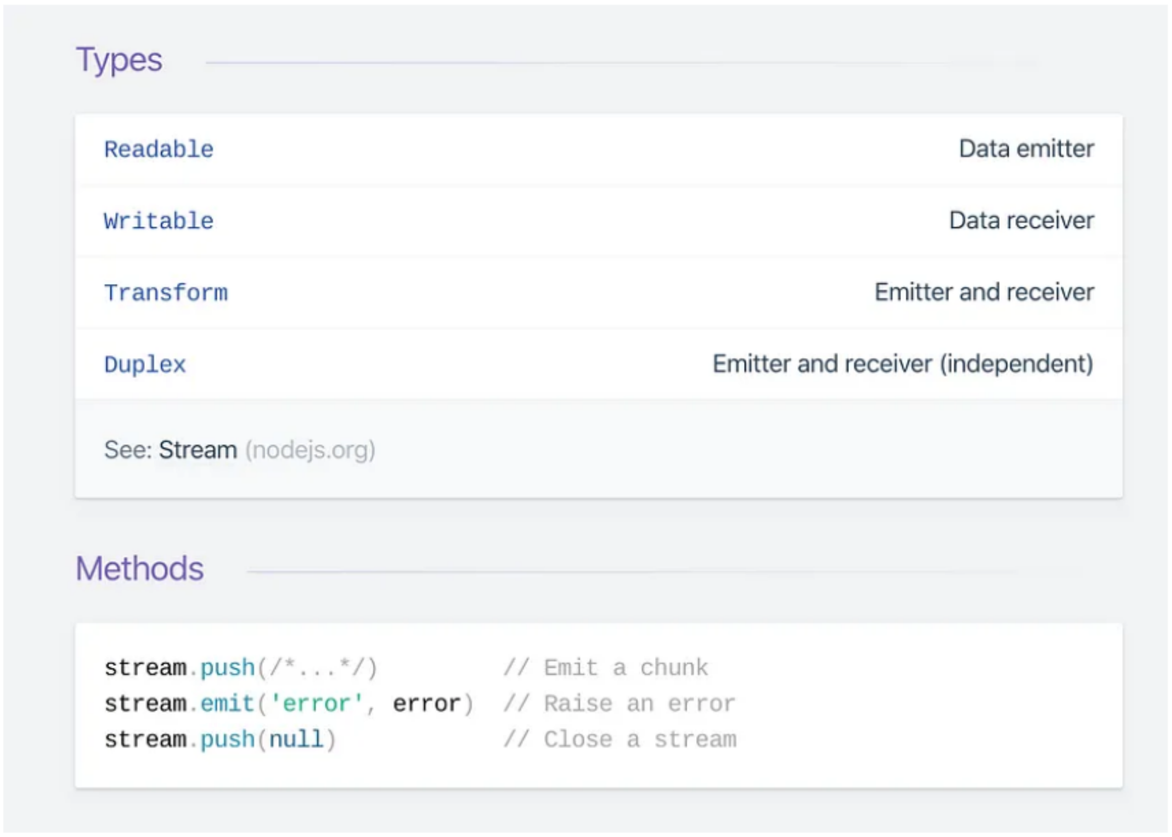
```
(base) vedanshdwivedi@192 backend_v2 % node test.js
This is a test
to check if my info is piped to the file or not
^C
(base) vedanshdwivedi@192 backend_v2 %
```

Streams-powered Node APIs

Due to their advantages, many Node.js core modules provide native stream handling capabilities, most notably:

- `net.Socket` is the main node api that is stream are based on, which underlies most of the following APIs
- `process.stdin` returns a stream connected to stdin
- `process.stdout` returns a stream connected to stdout
- `process.stderr` returns a stream connected to stderr
- `fs.createReadStream()` creates a readable stream to a file
- `fs.createWriteStream()` creates a writable stream to a file
- `net.connect()` initiates a stream-based connection
- `http.request()` returns an instance of the http.ClientRequest class, which is a writable stream
- `zlib.createGzip()` compress data using gzip (a compression algorithm) into a stream
- `zlib.createGunzip()` decompress a gzip stream.
- `zlib.createDeflate()` compress data using deflate (a compression algorithm) into a stream
- `zlib.createInflate()` decompress a deflate stream

Streams Cheat Sheet:



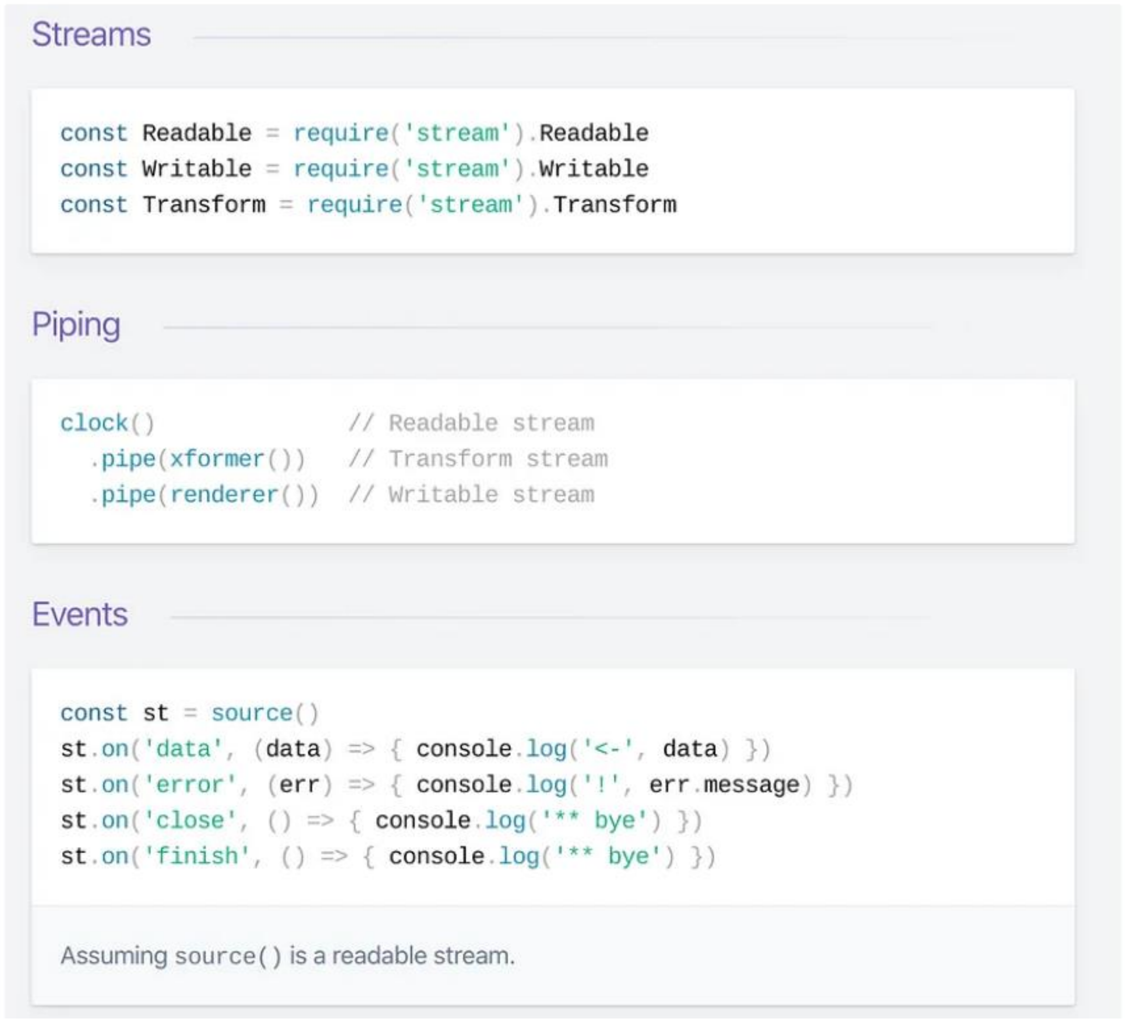
Types

`Readable`	Data emitter
`Writable`	Data receiver
`Transform`	Emitter and receiver
`Duplex`	Emitter and receiver (independent)

See: Stream (nodejs.org)

Methods

```
stream.push(/*...*/)         // Emit a chunk
stream.emit('error', error)  // Raise an error
stream.push(null)            // Close a stream
```



Streams

```
const Readable = require('stream').Readable
const Writable = require('stream').Writable
const Transform = require('stream').Transform
```

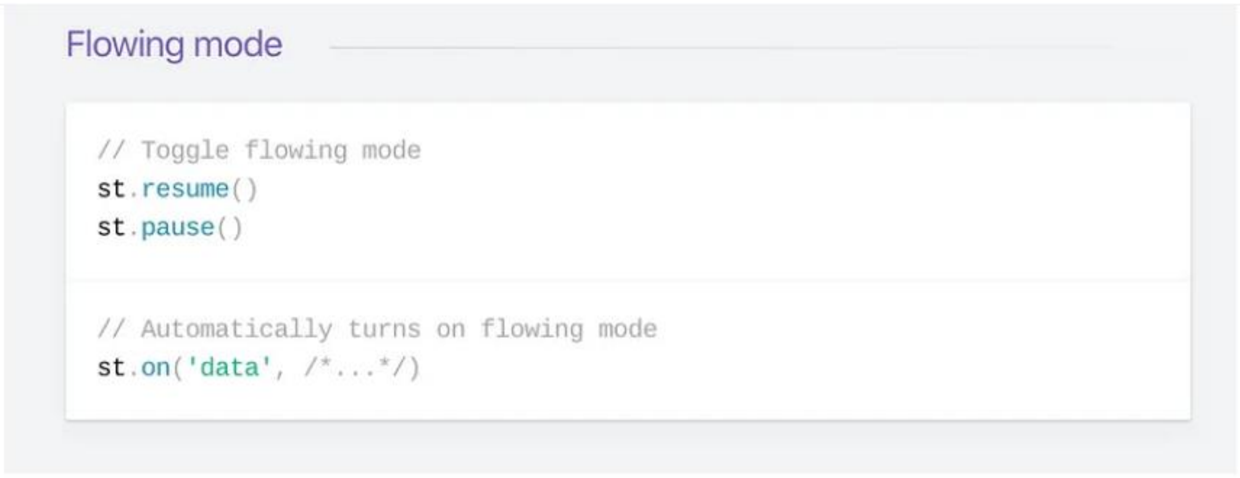
Piping

```
clock()              // Readable stream
  .pipe(xformer())   // Transform stream
  .pipe(renderer())  // Writable stream
```

Events

```
const st = source()
st.on('data', (data) => { console.log('<-', data) })
st.on('error', (err) => { console.log('!', err.message) })
st.on('close', () => { console.log('** bye') })
st.on('finish', () => { console.log('** bye') })
```

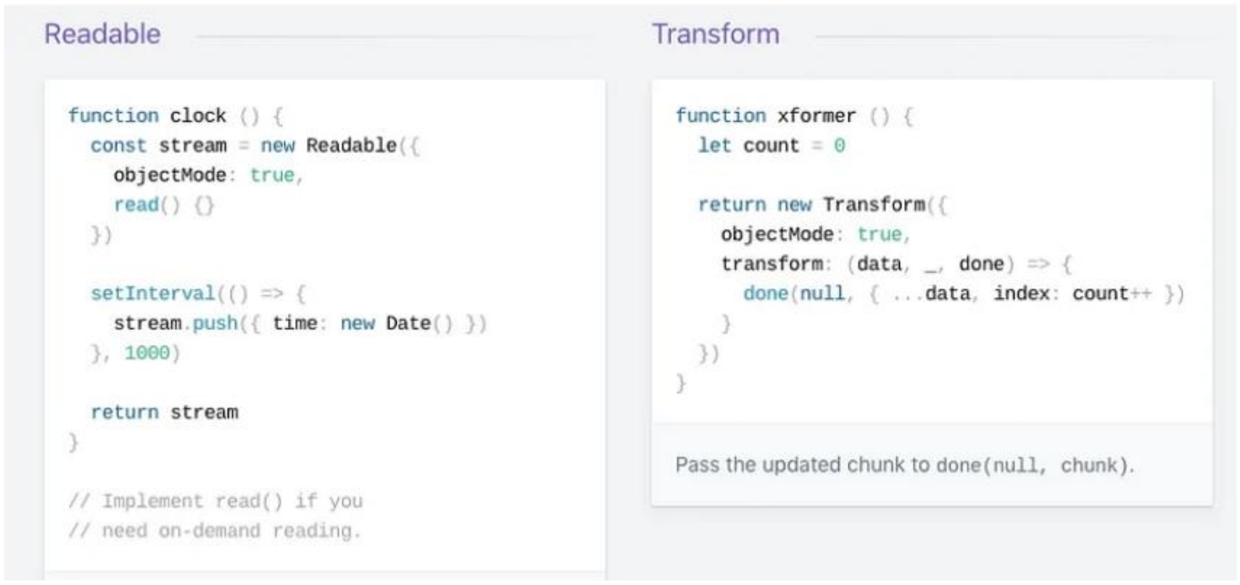
Assuming `source()` is a readable stream.



Flowing mode

```
// Toggle flowing mode
st.resume()
st.pause()
```

```
// Automatically turns on flowing mode
st.on('data', /*...*/)
```



Readable

```
function clock () {
  const stream = new Readable({
    objectMode: true,
    read() {}
  })

  setInterval(() => {
    stream.push({ time: new Date() })
  }, 1000)

  return stream
}

// Implement read() if you
// need on-demand reading.
```

Transform

```
function xformer () {
  let count = 0

  return new Transform({
    objectMode: true,
    transform: (data, _, done) => {
      done(null, { ...data, index: count++ })
    }
  })
}
```

Pass the updated chunk to `done(null, chunk)`.

Here are some important events related to writable streams:

- `error` – Emitted to indicate that an error has occurred while writing/piping.
- `pipeline` – When a readable stream is piped into a writable stream, this event is emitted by the writable stream.
- `unpipe` – Emitted when you call unpipe on the readable stream and stop it from piping into the destination stream.

Conclusion

This was all about the basics of streams. Streams, pipes, and chaining are the core and most powerful features in Node.js. Streams can indeed help you write neat and performant code to perform I/O.

Also, there is a Node.js strategic initiative worth looking to, called BOB, aiming to improve Node.js streaming data interfaces, both within Node.js core internally, and hopefully also as future public APIs.