Node.js Documentation

Module 7: Streams

- ### What are Streams

  Streams are a fundamental concept in Node.js for handling I/O operations efficiently. They provide a way to read and write data in a continuous, sequential manner, which is particularly useful for working with large data sets, such as files, network requests, and real-time communication. Streams are an instance of the `EventEmitter` class and allow you to process data piece by piece (chunks), rather than loading the entire data into memory at once.

  #### Types of Streams

  Node.js provides four main types of streams:

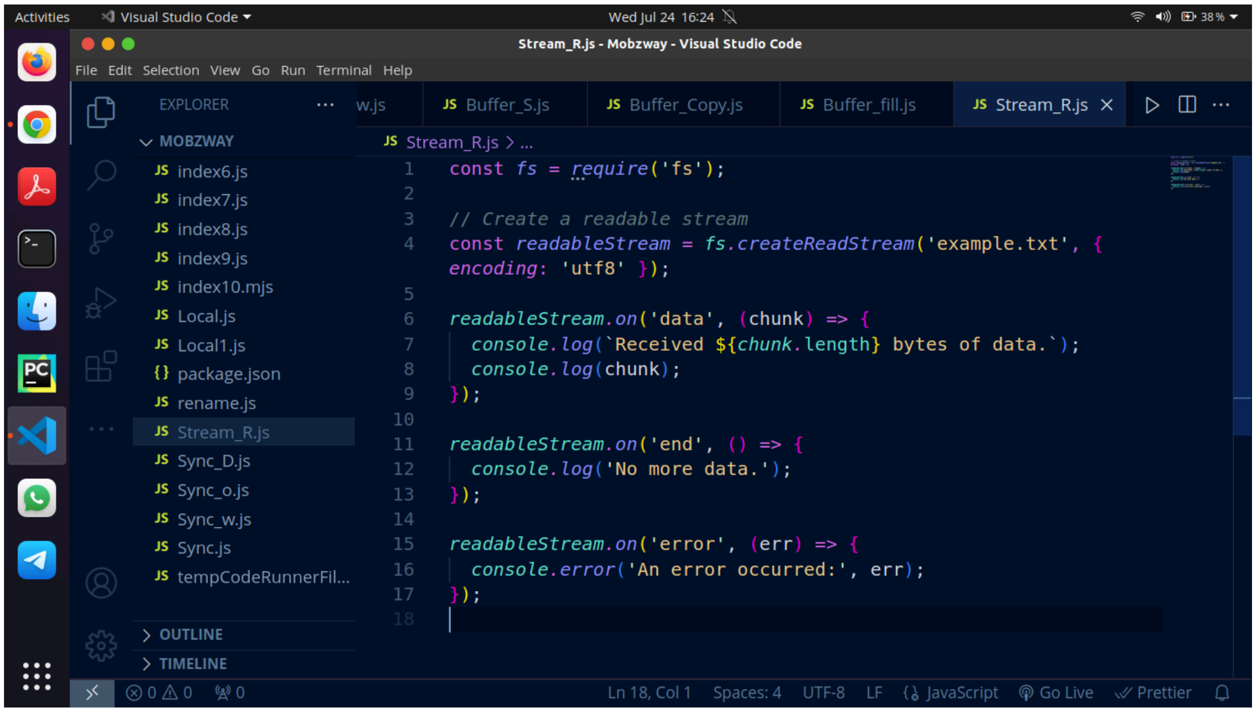
  1. **Readable Streams**:
     - Used for reading data.
     - Examples: `fs.createReadStream`, `http.IncomingMessage` (for HTTP requests).
  2. **Writable Streams**:
     - Used for writing data.
     - Examples: `fs.createWriteStream`, `http.ServerResponse` (for HTTP responses).
  3. **Duplex Streams**:
     - Both readable and writable.
     - Examples: `net.Socket`, `zlib.createDeflate`.
  4. **Transform Streams**:
     - A type of duplex stream where the output is computed based on the input.
     - Examples: `zlib.createGzip`, `crypto.createCipher`.

Read and Write Stream API

The Read and Write Stream API in Node.js provides a way to handle reading from and writing to streams in a more efficient manner, especially for large amounts of data. Here's a comprehensive guide on how to use the Read and Write Stream API with examples.

Readable Streams

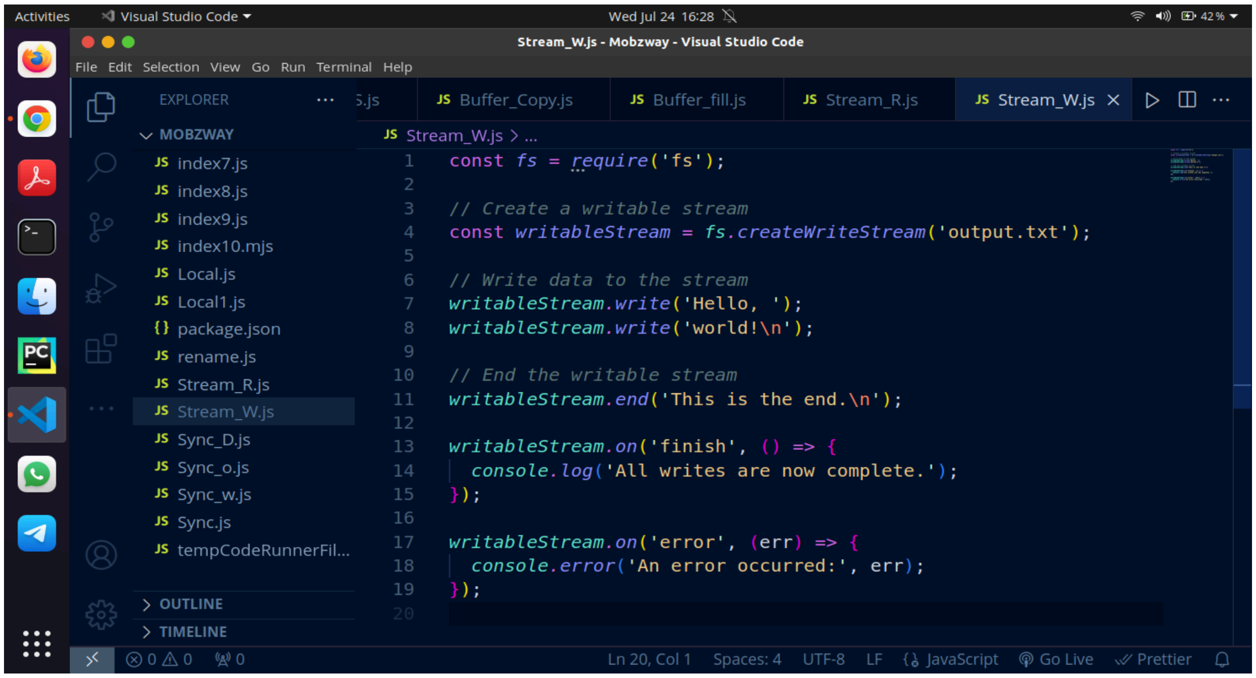
Readable streams are used to read data from a source in a sequential manner.



```
const fs = require('fs');

// Create a readable stream
const readableStream = fs.createReadStream('example.txt', { encoding: 'utf8' });

readableStream.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
  console.log(chunk);
});

readableStream.on('end', () => {
  console.log('No more data.');
});

readableStream.on('error', (err) => {
  console.error('An error occurred:', err);
});
```

Writable Streams

Writable streams are used to write data to a destination sequentially



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

// Create a writable stream
const writableStream = fs.createWriteStream('output.txt');

// Write data to the stream
writableStream.write('Hello, ');
writableStream.write('world!\n');

// End the writable stream
writableStream.end('This is the end.\n');

writableStream.on('finish', () => {
  console.log('All writes are now complete.');
});

writableStream.on('error', (err) => {
  console.error('An error occurred:', err);
});
```

- ## Flow Control

Flow control in Node.js streams is essential for managing the rate at which data is read from a source and written to a destination, ensuring that neither the readable stream nor the writable stream becomes overwhelmed. This is especially important when dealing with large data sets or when the speed of the source and destination are mismatched.

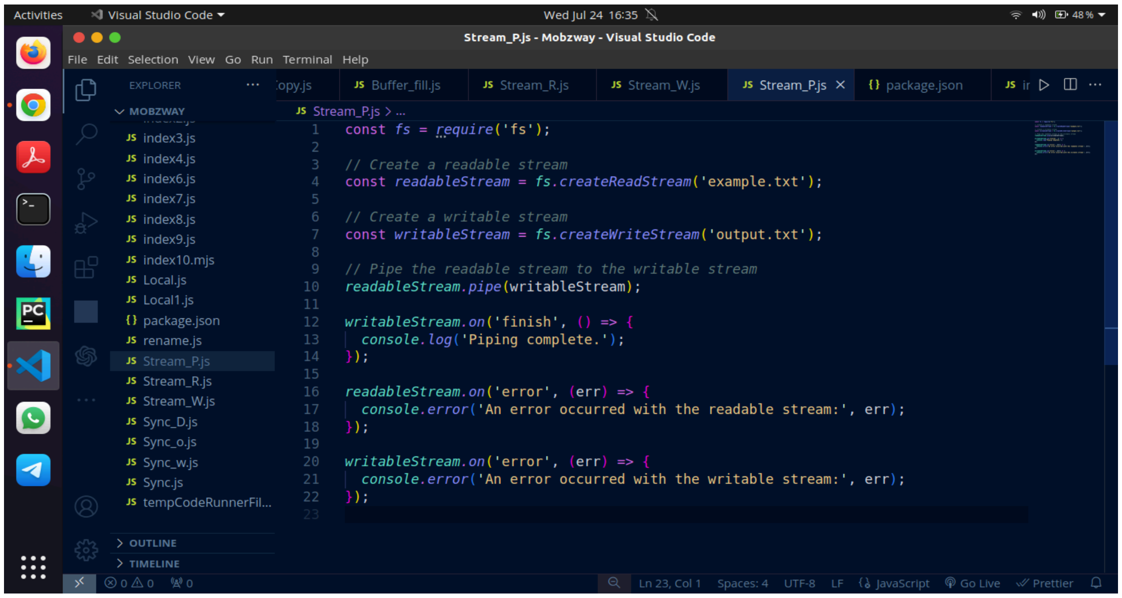
Understanding Flow Control

When working with streams, flow control mechanisms help balance the flow of data. Key aspects include:

1. **Readable Stream**: Emits data as it is read from the source.
2. **Writable Stream**: Writes data to the destination at its own pace.
3. **Backpressure**: Occurs when the writable stream can't handle the rate at which the readable stream is providing data.

- ## Piping

Piping streams is a powerful feature that allows you to connect a readable stream to a writable stream. This automatically handles the flow of data from the source to the destination.



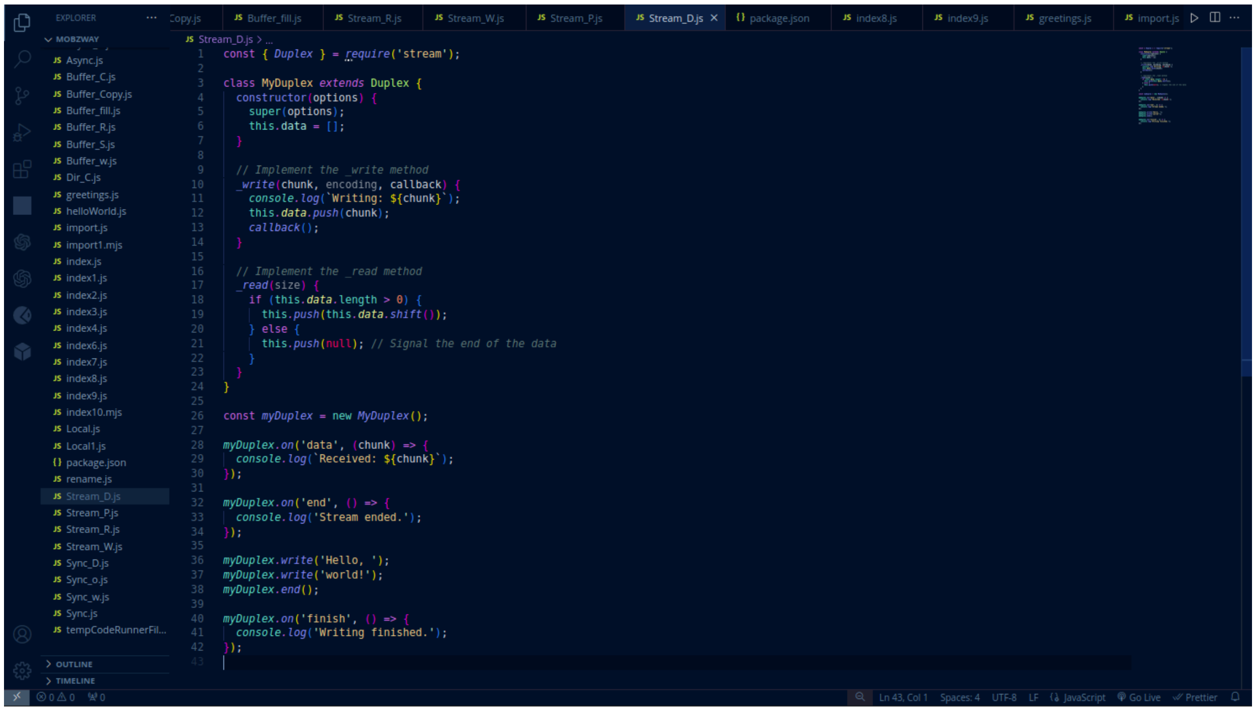
```javascript
const fs = require('fs');

// Create a readable stream
const readableStream = fs.createReadStream('example.txt');

// Create a writable stream
const writableStream = fs.createWriteStream('output.txt');

// Pipe the readable stream to the writable stream
readableStream.pipe(writableStream);

writableStream.on('finish', () => {
  console.log('Piping complete.');
});

readableStream.on('error', (err) => {
  console.error('An error occurred with the readable stream:', err);
});

writableStream.on('error', (err) => {
  console.error('An error occurred with the writable stream:', err);
});
```

- ## Duplex Stream

A duplex stream in Node.js is a type of stream that implements both the Readable and Writable interfaces. This means that it can read and write data, making it suitable for use cases where you need to perform both operations, such as in network protocols, file handling, or transformation streams.

Creating a Duplex Stream

To create a duplex stream, you can extend the `stream.Duplex` class and implement the `_read` and `_write` methods. Here is an example of how to create a custom duplex stream.

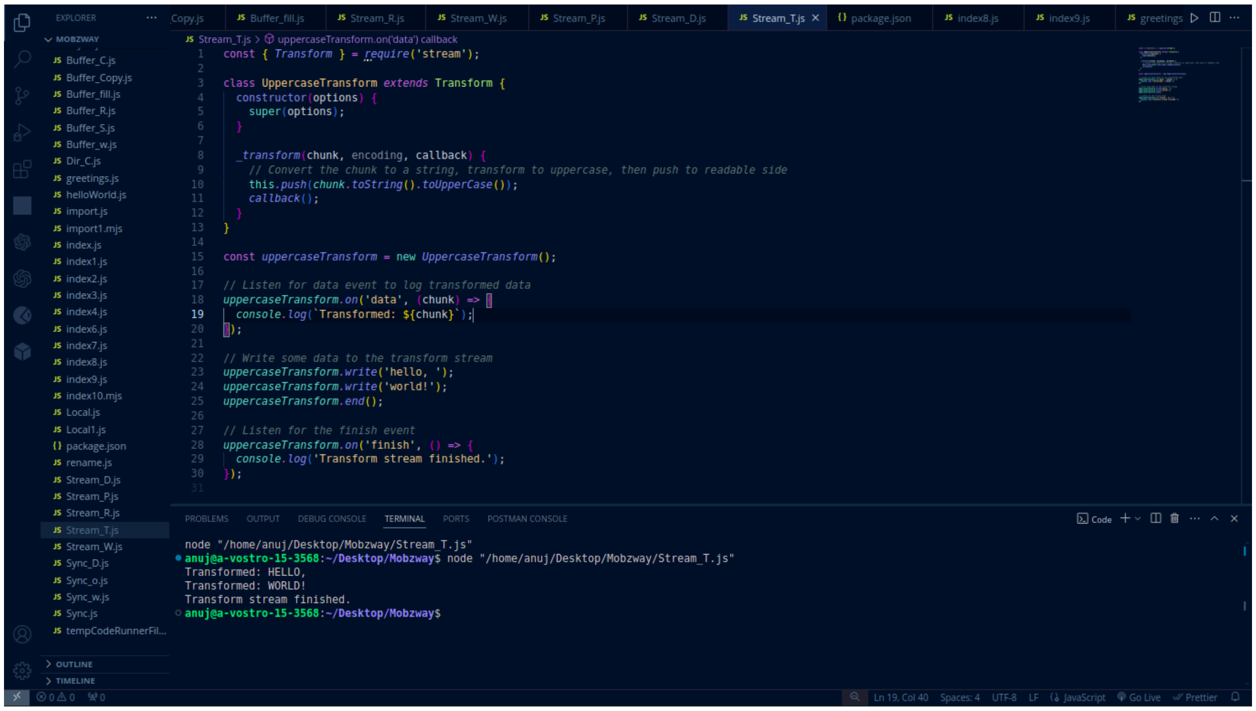


```javascript
const { Duplex } = require('stream');

class MyDuplex extends Duplex {
  constructor(options) {
    super(options);
    this.data = [];
  }

  // Implement the _write method
  _write(chunk, encoding, callback) {
    console.log(`Writing: ${chunk}`);
    this.data.push(chunk);
    callback();
  }

  // Implement the _read method
  _read(size) {
    if (this.data.length > 0) {
      this.push(this.data.shift());
    } else {
      this.push(null); // Signal the end of the data
    }
  }
}

const myDuplex = new MyDuplex();

myDuplex.on('data', (chunk) => {
  console.log(`Received: ${chunk}`);
});

myDuplex.on('end', () => {
  console.log('Stream ended.');
});

myDuplex.write('Hello, ');
myDuplex.write('world!');
myDuplex.end();

myDuplex.on('finish', () => {
  console.log('Writing finished.');
});
```

- ## Transform Stream

  A transform stream is a type of duplex stream where the output is computed based on the input. This allows you to transform data as it is being read and written. Node.js provides a `Transform` class in the `stream` module for creating such streams.

  Here's a simple example of a transform stream that converts all input text to uppercase.



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

class UppercaseTransform extends Transform {
  constructor(options) {
    super(options);
  }

  _transform(chunk, encoding, callback) {
    // Convert the chunk to a string, transform to uppercase, then push to readable side
    this.push(chunk.toString().toUpperCase());
    callback();
  }
}

const uppercaseTransform = new UppercaseTransform();

// Listen for data event to log transformed data
uppercaseTransform.on('data', (chunk) => {
  console.log(`Transformed: ${chunk}`);
});

// Write some data to the transform stream
uppercaseTransform.write('hello, ');
uppercaseTransform.write('world!');
uppercaseTransform.end();

// Listen for the finish event
uppercaseTransform.on('finish', () => {
  console.log('Transform stream finished.');
});
```

```
node "/home/anuj/Desktop/Mobzway/Stream_T.js"
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/Stream_T.js"
Transformed: HELLO,
Transformed: WORLD!
Transform stream finished.
anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```