4 Event Emitters and Streams

- Introduction to Event Emitters in Node.js
- Working with Streams (Readable, Writable, Duplex, and Transform)
- Implementing file operations using Streams

Introduction to Event Emitters in Node.js

In Node.js, events are a core concept, and the **EventEmitter** class, which is part of the events module, provides a way to handle these events. Event-driven programming is a key part of Node.js, allowing you to execute code when specific events occur, making the system reactive and non-blocking.

What is an Event Emitter?

An **EventEmitter** is a class in Node.js that allows objects to emit and listen for events. It forms the backbone of Node's asynchronous, event-driven architecture. Objects that emit events are instances of EventEmitter.

Key concepts:

- **Emit**: The act of emitting or triggering an event.
- **Listen**: Reacting to an event by executing a callback function.

Basic Example of Event Emitter

First, let's see a simple example of using an EventEmitter in Node.js:

Step 1: Import the events module

You need to require the events module and create an instance of the EventEmitter class:

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();
console.log(eventEmitter)
```

Step 2: Set up an Event Listener

To react to an event, you need to listen for it using the on() method. This method binds a callback function to an event:

```
eventEmitter.on('greet', (name) => {
   console.log(`Hello, ${name}!`);
});
```

In this example, we set up a listener for the greet event, and when the event is emitted, it will call the function and pass the name argument.

Step 3: Emit an Event

You can trigger an event using the emit() method:

```
eventEmitter.emit('greet', 'Hema');
```

When emit('greet',’Hema’) is called, it will trigger the greet event and pass 'John' to the listener function, outputting:

Hello, Hema!

Real-World Example: Custom Event Emitter

In real-world applications, EventEmitters are often used to handle asynchronous events, such as when a file finishes downloading, a database query completes, or a user action triggers some functionality.

Here's a more practical example where an event emitter handles file reading asynchronously.

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();
console.log(eventEmitter)

// eventEmitter.on('greet', (name) => {
//     console.log(`Hello, ${name}!`);
// });
// eventEmitter.emit('greet', 'Hema');
const fs = require('fs');

// Set up an event listener for 'fileRead'
eventEmitter.on('fileRead', (fileName) => {
    console.log(`Finished reading the file: ${fileName}`);
});

// Read a file asynchronously and emit an event once it's done
fs.readFile('example.txt', 'utf-8', (err, data) => {
    if (err) {
        console.error('Error reading the file:', err);
    } else {
        console.log(data);
        eventEmitter.emit('fileRead', 'example.txt');
    }
});
```

Key Methods in EventEmitter

1. **on(eventName, listener)**: Registers a listener for the specified event. The listener is called every time the event is emitted.

emit(eventName, [...args]): Triggers an event and calls all the listeners attached to that event

once(eventName, listener):

Example: Using once() for One-Time Events

The once() method is useful when you need to react to an event just once. For example, you might want to notify a user only once when they log in:

```
eventEmitter.once('login', (username) => {
   console.log(`Welcome, ${username}!`);
});

eventEmitter.emit('login', 'Hema');  // Output: Welcome, Hema!
eventEmitter.emit('login', 'Mahesh');    // No output, as listener is only executed once
```

removeListener(eventName, listener):

```
// Define the event listener
function greetListener(name) {
   console.log(`Hello, ${name}!`);
}

// Attach the listener to the 'greet' event
eventEmitter.on('greet', greetListener);

// Emit the 'greet' event
eventEmitter.emit('greet', 'Mahesh'); // Output: Hello, Mahesh!

// Remove the specific 'greetListener' for the 'greet' event
eventEmitter.removeListener('greet', greetListener);

// Try emitting the 'greet' event again
eventEmitter.emit('greet', 'Hema'); // No output, as the listener has been removed
```

We first define a greetListener function and attach it to the greet event using eventEmitter.on().
After emitting the event once (which outputs "Hello, Mahesh!"), we remove the greetListener using removeListener().
When we emit the event again, nothing happens because the listener has been removed.

removeAllListeners([eventName]):

```
function greetListener1(name) {
   console.log(`Hello from listener 1, ${name}!`);
}

function greetListener2(name) {
   console.log(`Hello from listener 2, ${name}!`);
}

// Attach both listeners to the 'greet' event
eventEmitter.on('greet', greetListener1);
eventEmitter.on('greet', greetListener2);

// Emit the 'greet' event
eventEmitter.emit('greet', 'Mahesh');
// Output:
// Hello from listener 1, Mahesh!
// Hello from listener 2, Mahesh!

// Remove all listeners for the 'greet' event
eventEmitter.removeAllListeners('greet');

// Try emitting the 'greet' event again
eventEmitter.emit('greet', 'Hema'); // No output, as all listeners for 'greet' are removed
```

Working with Streams in Node.js

Streams in Node.js are a powerful way to handle reading and writing data asynchronously and efficiently. They allow you to process data in chunks, which is especially useful for handling large files, network requests, or real-time data. Streams can be classified into four types:

1. **Readable Streams**: Used for reading data.
2. **Writable Streams**: Used for writing data.
3. **Duplex Streams**: Can be both readable and writable.
4. **Transform Streams**: A type of duplex stream that modifies the data as it is read or written.

Each of these stream types is built on the EventEmitter class, meaning they emit events such as data, end, error, finish, etc.

Readable Streams

A **Readable Stream** is used to read data in chunks from a source (like a file or an HTTP response) instead of reading it all at once.

Example: Reading a file using a readable stream

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

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

// Listen for data events to receive chunks of data
```

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

// Listen for the end event, which indicates that the stream has finished reading
readableStream.on('end', () => {
  console.log('Finished reading file');
});

// Listen for error events
readableStream.on('error', (err) => {
  console.error('Error reading file:', err);
});
```

In this example:

- We use fs.createReadStream() to create a readable stream from the example.txt file.
- The data event is triggered when a chunk of data is available.
- The end event is triggered when the stream has no more data to read.

Writable Streams

A **Writable Stream** is used to write data in chunks to a destination (like a file or an HTTP response).

Example: Writing to a file using a writable stream

```
-----------------------------------------------------------------writableStream

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

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

// Write data to the stream
writableStream.write('Hello, world!\n');
writableStream.write('This is a writable stream example.\n');

// Signal the end of writing
writableStream.end();

// Listen for the finish event to know when writing is complete
writableStream.on('finish', () => {
  console.log('Finished writing to file');
});

// Listen for error events
writableStream.on('error', (err) => {
  console.error('Error writing to file:', err);
});
```

In this example:

- We create a writable stream using fs.createWriteStream() to write to the file output.txt.
- The finish event is triggered when the stream has finished writing all the data.

Duplex Streams

A **Duplex Stream** is both readable and writable. It allows you to both read from and write to the stream.

Example: Using a Duplex stream (Custom Implementation)

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

// Create a duplex stream (both readable and writable)
const duplexStream = new Duplex({
  write(chunk, encoding, callback) {
    console.log(`Writing: ${chunk}`);
    callback();
  },
  read(size) {
    this.push('This is data from the readable side\n');
    this.push(null); // Signals the end of the stream
  }
});

// Writing to the duplex stream
duplexStream.write('Hello from the writable side\n');
duplexStream.end();

// Reading from the duplex stream
duplexStream.on('data', (chunk) => {
  console.log(`Readable received: ${chunk}`);
});
```

In this example:

- We define a custom duplex stream that can read and write.
- The write function handles writing, and the read function pushes data into the stream for reading.
- This custom duplex stream writes and reads data as demonstrated in the output.

Transform Streams

A **Transform Stream** is a type of duplex stream where the output is a transformation of the input. These are typically used for things like compression, encryption, or other data transformations.

Example: Creating a Transform stream (uppercase conversion)

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

// Create a transform stream to convert input to uppercase
const transformStream = new Transform({
    transform(chunk, encoding, callback) {
        const upperCaseChunk = chunk.toString().toUpperCase();
        this.push(upperCaseChunk);
        callback();
    }
});

// Using the transform stream
process.stdin.pipe(transformStream).pipe(process.stdout);
```

Piping Streams

Streams in Node.js can be connected (or **piped**) together, allowing data to flow from one stream to another.

Example: Piping a readable stream to a writable stream

```

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

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

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

// Listen for the finish event
writableStream.on('finish', () => {
    console.log('File has been copied');
});
```

Interview Questions and Answers:

What is an EventEmitter in Node.js?

Answer: EventEmitter is a core module in Node.js that facilitates communication between objects in an event-driven architecture. It allows objects to emit events and for other objects to listen for these events, making Node.js efficient for handling asynchronous operations.

How do you create and use an EventEmitter in Node.js?

Answer: To use EventEmitter, you first need to import the events module, create an instance of the EventEmitter class, and then use its methods (on(), emit(), etc.) to handle events.

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();

// Define an event listener
eventEmitter.on('greet', (name) => {
   console.log(`Hello, ${name}!`);
});

// Emit an event
eventEmitter.emit('greet', 'Mahesh');
```

n this example, the event greet is listened to by on(), and when emit('greet') is called, the listener is executed with the argument ‘Mahesh’.

What are the key methods available in the EventEmitter class?

Answer: Key methods include:

- **on(eventName, listener)**: Registers a listener for the specified event.
- **emit(eventName, [...args])**: Emits an event, invoking all listeners attached to that event.
- **once(eventName, listener)**: Registers a listener that is invoked only the first time the event is emitted, then it’s automatically removed.
- **removeListener(eventName, listener)**: Removes a specific listener for an event.
- **removeAllListeners([eventName])**: Removes all listeners for the specified event, or all events if no event name is provided.

What is the difference between on() and once() in EventEmitter?

Answer:

- **on()**: Adds a listener for an event, and the listener will be called every time the event is emitted.

- **once()**: Adds a listener for an event that will only be invoked the first time the event is emitted. After that, the listener is automatically removed and won't be called again.

```
eventEmitter.once('greet', () => {
   console.log('Hello, this will only run once.');
});
```

How do you remove an event listener in Node.js?

Answer: You can remove a specific listener for an event using the removeListener() method. You need to pass the exact function reference that was used to add the listener.

```
const greet = () => console.log('Hello!');

// Add the listener
eventEmitter.on('greet', greet);

// Emit the event
eventEmitter.emit('greet'); // Output: Hello!

// Remove the listener
eventEmitter.removeListener('greet', greet);

// Emit the event again (nothing happens)
eventEmitter.emit('greet');
```

What are some common events emitted by core modules in Node.js that inherit from EventEmitter?

Answer: Several core Node.js modules extend the EventEmitter class, emitting their own events:

- **http.Server**: Emits events like request, connection, close.
- **fs.ReadStream and fs.WriteStream**: Emit events like open, data, end, close, error.
- **net.Socket**: Emits events like connect, data, end, timeout, close, error.

Example (HTTP Server):

```
const http = require('http');
const server = http.createServer();

server.on('request', (req, res) => {
   res.end('Hello World!');
});

server.listen(3000, () => console.log('Server running on port 3000'));
```

Can you explain the purpose of the error event in EventEmitter?

Answer: The error event is a special event in EventEmitter. When an error occurs in asynchronous operations, you can emit the error event. If there is no listener for the error event, Node.js will throw an exception and crash the process. Therefore, it's important to always attach a listener for error.

Example:

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

// Emitting an error event
eventEmitter.emit('error', new Error('Something went wrong!'));
```

How do you emit an event with multiple arguments?

Answer: You can pass multiple arguments to emit(), and these arguments will be passed to the listener function.

```
eventEmitter.on('data', (arg1, arg2) => {
   console.log(`Received: ${arg1} and ${arg2}`);
});

// Emit event with two arguments
eventEmitter.emit('data', 'Hello', 'World');
```

How do you handle memory leaks caused by too many event listeners in Node.js?

Answer: Node.js warns you if you add more than 10 listeners to an event, which can potentially cause a memory leak. To avoid this:

- Use removeListener() or removeAllListeners() to clear unnecessary listeners.
- You can increase the limit using setMaxListeners().

Example:

```
eventEmitter.setMaxListeners(20); // Increase limit from 10 to 20
```

What is the difference between removeListener() and removeAllListeners()?

Answer:

- **removeListener(eventName, listener)**: Removes a specific listener for the given event. You must pass the exact listener function reference that was added.
- **removeAllListeners([eventName])**: Removes all listeners for the given event. If no event name is provided, it removes all listeners for all events.

```
// Removing a specific listener
eventEmitter.removeListener('greet', listenerFunction);

// Removing all listeners for an event
eventEmitter.removeAllListeners('greet');

// Removing all listeners for all events
eventEmitter.removeAllListeners();
```

Streams:

What are streams in Node.js?

Answer:
Streams in Node.js are a way to handle reading and writing data in chunks, rather than all at once. They are especially useful when dealing with large files or data, allowing for more efficient memory usage. Streams are instances of EventEmitter and work asynchronously. They are used for handling data from various sources, such as files, network requests, or standard input/output.

What are the different types of streams in Node.js?

Answer: There are four types of streams in Node.js:

1. **Readable Streams**: For reading data from a source, like files.
2. **Writable Streams**: For writing data to a destination, like a file or network.
3. **Duplex Streams**: For both reading and writing data, like a network socket.
4. **Transform Streams**: A type of duplex stream that allows transforming or modifying data as it is read or written (e.g., compressing or encrypting data).

What is the purpose of the pipe() method in Node.js streams?

Answer:
The pipe() method is used to connect a readable stream to a writable stream, allowing data to flow between them. It simplifies the process of reading data from a source and writing it to a destination, such as copying data from one file to another.

Example:

```
const fs = require('fs');
const readableStream = fs.createReadStream('input.txt');
const writableStream = fs.createWriteStream('output.txt');

// Piping data from readable stream to writable stream
```

```
readableStream.pipe(writableStream);
```

In this example, pipe() transfers data from input.txt to output.txt.

What is the difference between readable.pipe() and readable.on('data', ...) in Node.js?

Answer:

- **readable.pipe()**: Automatically manages the flow of data from a readable stream to a writable stream. It handles backpressure, meaning it ensures that the writable stream does not get overwhelmed by too much data at once.
- **readable.on('data', ...)**: Allows you to manually listen for chunks of data from a readable stream, but you need to handle writing data manually. Using on('data') does not handle backpressure, so the writable stream could get overloaded if not managed properly.

What is backpressure in Node.js streams, and how is it managed?

Answer:
Backpressure occurs when the writable stream is unable to process the incoming data as quickly as the readable stream provides it. This can lead to memory issues as the data gets buffered. Node.js handles backpressure by pausing the readable stream until the writable stream is ready to handle more data.

The pipe() method automatically manages backpressure, ensuring that data flows smoothly between the streams without overwhelming the writable stream.

Can you explain the data, end, error, and finish events in Node.js streams?

Answer:

- **data**: Emitted when a readable stream has a chunk of data available to be consumed.
- **end**: Emitted when there is no more data to read from the readable stream.
- **error**: Emitted when an error occurs while reading or writing to the stream.
- **finish**: Emitted when all data has been written to a writable stream.

Example:

```
const fs = require('fs');
const readableStream = fs.createReadStream('input.txt');

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

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

```
readableStream.on('error', (err) => {
   console.error('Error reading file:', err);
});
```

What is a Transform stream in Node.js, and how is it different from Duplex streams?

Answer:
A **Transform stream** is a type of duplex stream where the input is transformed before it is passed to the output. Transform streams are both readable and writable, but they modify the data while it's being processed. Examples of transform streams include compression, encryption, and data parsing.

Duplex streams, on the other hand, allow for independent reading and writing without any modification of the data.

Example (Transform Stream):

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

// Create a transform stream to convert input to uppercase
const transformStream = new Transform({
   transform(chunk, encoding, callback) {
      const upperCaseChunk = chunk.toString().toUpperCase();
      this.push(upperCaseChunk);
      callback();
   }
});

// Using the transform stream to convert input to uppercase
process.stdin.pipe(transformStream).pipe(process.stdout);
```

How can you implement a custom readable or writable stream in Node.js?

Answer: You can implement a custom stream by extending the Readable or Writable class and overriding the _read() or _write() methods, respectively.

Example (Custom Readable Stream):

```
const { Readable } = require('stream');

class MyReadableStream extends Readable {
   constructor() {
      super();
      this.index = 0;
   }
```

```
  _read(size) {
    this.index++;
    if (this.index > 5) {
      this.push(null); // End the stream
    } else {
      this.push(`Chunk ${this.index}\n`);
    }
  }
}

const myReadableStream = new MyReadableStream();
myReadableStream.pipe(process.stdout);
```

Example (Custom Writable Stream):

```
const { Writable } = require('stream');

class MyWritableStream extends Writable {
  _write(chunk, encoding, callback) {
    console.log(`Writing: ${chunk.toString()}`);
    callback();
  }
}

const myWritableStream = new MyWritableStream();
process.stdin.pipe(myWritableStream);
```

How do you handle errors in streams?

Answer: Errors in streams can be handled by listening for the error event. It is important to always attach an error listener to streams to prevent unhandled exceptions.

```
const fs = require('fs');
const readableStream = fs.createReadStream('nonexistent-file.txt');

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

Without handling the error event, an uncaught exception would be thrown, which could crash the application.

What is the purpose of the readable.push() method in Node.js?

Answer: The readable.push() method is used in custom readable streams to push data into the internal buffer of the stream. It can be used to push chunks of data or to signal the end of the stream by passing null.

```
const { Readable } = require('stream');

class MyReadableStream extends Readable {
  constructor() {
    super();
    this.index = 0;
  }

  _read(size) {
    this.index++;
    if (this.index > 3) {
      this.push(null); // No more data, end the stream
    } else {
      this.push(`Chunk ${this.index}\n`);
    }
  }
}

const myStream = new MyReadableStream();
myStream.pipe(process.stdout);
```

How do you close a stream in Node.js?

Answer: Streams can be closed manually by calling the end() method on writable streams, or automatically when there is no more data to be read (for readable streams). The end() method signals that no more data will be written.

```
const fs = require('fs');
const writableStream = fs.createWriteStream('output.txt');

writableStream.write('Hello, World!\n');
writableStream.end('Goodbye, World!\n'); // This closes the stream

writableStream.on('finish', () => {
  console.log('Stream is finished writing');
});
```

In this example, calling writableStream.end() closes the writable stream after writing the final data.

What is the cork() and uncork() method in writable streams, and why are they used?

Answer: The cork() method in writable streams is used to buffer all writes until uncork() is called. This can be useful when you want to batch multiple writes together for performance reasons, ensuring they are flushed to the destination in a single operation rather than one at a time.

Example:

```
const fs = require('fs');
const writableStream = fs.createWriteStream('output.txt');

writableStream.cork(); // Buffers writes

writableStream.write('Line 1\n');
writableStream.write('Line 2\n');

// Flush the buffer
writableStream.uncork();

writableStream.end();
```

How does readable.pause() and readable.resume() work in Node.js streams?

Answer:

- **readable.pause()**: Pauses the flow of data from a readable stream, temporarily stopping the data event from being emitted.
- **readable.resume()**: Resumes the flow of data, restarting the `data

` event.

Example:

```
const fs = require('fs');
const readableStream = fs.createReadStream('input.txt');

readableStream.on('data', (chunk) => {
  console.log(`Received: ${chunk}`);
  readableStream.pause(); // Pause the stream
  setTimeout(() => {
    readableStream.resume(); // Resume after 1 second
  }, 1000);
});
```

In this example, the stream is paused after each chunk is received, then resumes after a 1-second delay.