Node.js By Kuldeep Verma

Here's a structured list of the most important chapters and concepts:

1. Introduction to Node.js

- What is Node.js?
- How Node.js works (Single-threaded, Event Loop, Non-blocking I/O)
- Setting up a Node.js environment

2. Core Modules and Global Objects

- `fs` (File System)
- `path`
- `http`
- `url`
- `events`
- `os`
- `crypto`
- Understanding Global objects (`__dirname`, `__filename`, `process`, etc.)

3. Package Management with npm

- Understanding npm (Node Package Manager)
- Installing, updating, and managing packages
- Creating and publishing your own packages

4. Modules and Require

- Understanding CommonJS modules
- Importing and exporting modules

- Creating your own modules

5. Asynchronous Programming

- Callbacks
- Promises
- Async/Await
- Handling errors in asynchronous code

6. Building a Web Server

- Creating an HTTP server
- Handling requests and responses
- Routing
- Serving static files

7. Express.js Framework

- Introduction to Express.js
- Setting up an Express.js server
- Middleware (built-in, third-party, and custom)
- Routing in Express.js
- Handling requests and responses
- Template engines (e.g., Pug, EJS)
- Error handling in Express.js

8. Working with Databases

- Introduction to MongoDB (using Mongoose)
- Connecting Node.js to MongoDB
- CRUD operations
- Schema design and validation

- Using MongoDB with Express.js

9. Authentication and Authorization

- Understanding authentication vs. authorization
- Implementing JWT (JSON Web Tokens)
- Session management and cookies
- Secure password hashing (using `bcrypt`)

10. RESTful API Development

- Designing RESTful APIs
- Creating and handling RESTful routes
- Versioning APIs
- Pagination, filtering, and sorting
- Securing APIs with authentication

11. Error Handling and Debugging

- Centralized error handling
- Logging errors
- Debugging Node.js applications
- Using tools like `nodemon`, `node-inspect`

12. Real-Time Communication with WebSockets

- Understanding WebSockets
- Using `Socket.io` for real-time communication
- Building a basic chat application

13. Testing and Test-Driven Development (TDD)

- Unit testing with `Mocha` and `Chai`
- Integration testing

- Mocking and stubbing in tests
- End-to-end testing

14. Security Best Practices

- Protecting against common security threats (e.g., SQL Injection, XSS, CSRF)
- Implementing HTTPS in Node.js
- Securely storing sensitive information

15. Deploying Node.js Applications

- Preparing your application for production
- Using environments and configuration management
- Deploying on platforms like Heroku, AWS, or DigitalOcean
- Understanding PM2 (Process Manager) for Node.js
- Using Docker for containerization

16. Performance Optimization

- Understanding the event loop and how to avoid blocking it
- Clustering for handling more traffic
- Caching strategies (using Redis, etc.)
- Load balancing and scaling

17. Understanding Streams and Buffers

- Working with streams (Readable, Writable, Duplex, Transform)
- Piping streams
- Handling large files and data streams efficiently

18. Event-Driven Programming

- Understanding the EventEmitter class
- Building custom event emitters

- Using events in real-world applications

Introduction to Node.js

What is Node.js?

Node.js is an open-source, cross-platform JavaScript runtime environment that allows you to run JavaScript code outside of a web browser. It was created by Ryan Dahl in 2009 and has since become a cornerstone of modern web development. Node.js is built on Chrome's V8 JavaScript engine, which is known for its speed and efficiency in executing JavaScript.

In traditional web development, JavaScript is typically used for client-side scripting. However, Node.js brings JavaScript to the server-side, enabling developers to build scalable, high-performance applications using the same language for both client and server code.

Key Features of Node.js:

1. **Asynchronous and Event-Driven:** Node.js uses a non-blocking, event-driven architecture, making it ideal for handling multiple connections and performing I/O operations without locking the main thread.
2. **Single-Threaded:** Despite being single-threaded, Node.js can handle many concurrent connections efficiently due to its event loop and non-blocking nature.
3. **Scalability:** Node.js applications can handle a large number of simultaneous connections with high throughput, making it suitable for building scalable network applications.
4. **Rich Ecosystem:** Node.js has a vast ecosystem of libraries and modules, accessible through npm (Node Package Manager), which makes development faster and easier.
5. **Cross-Platform:** Node.js is cross-platform, meaning it can run on Windows, macOS, and various Linux distributions, making it versatile and widely adopted.

How Node.js Works:

At its core, Node.js operates on a single-threaded, event-driven architecture. This may seem counterintuitive when compared to traditional multi-threaded server architectures, but it's what makes Node.js both powerful and efficient for I/O-bound tasks.

1. Single-Threaded Architecture:

- Unlike traditional web servers that spawn a new thread or process for each incoming connection, Node.js runs on a single thread. This thread is responsible for handling all the incoming requests. The advantage of this approach is that it avoids the overhead associated with thread creation and context switching, making Node.js lightweight.
- Example Code:

  ```
  javascript
  Copy code
  // Simple Node.js server
  const http = require('http');

  const server = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello World!\n');
  });

  server.listen(3000, '127.0.0.1', () => {
    console.log('Server running at http://127.0.0.1:3000/');
  });
  ```

  In this example, the server listens on port 3000 and responds with "Hello World!" for any incoming request. The key point is that the server handles all requests on a single thread.

2. Event Loop:

- The event loop is the heart of Node.js. It’s a mechanism that continuously monitors the event queue for tasks to execute. When a request comes in, Node.js offloads any potentially blocking operations (like I/O tasks) to the event loop. Once the operation completes, a callback is triggered to handle the response. This allows Node.js to handle multiple connections concurrently without blocking the main thread.

  Event Loop Phases:

  - **Timers:** Executes callbacks scheduled by `setTimeout()` and `setInterval()`.
  - **I/O Callbacks:** Executes I/O-related callbacks.
  - **Idle, Prepare:** Internal use only.
  - **Poll:** Retrieves new I/O events; the event loop waits here for new events.
  - **Check:** Executes callbacks scheduled by `setImmediate()`.
  - **Close Callbacks:** Executes `close` event callbacks (e.g., socket closures).

  Example:

  ```
  javascript
  Copy code
  console.log('Start');

  setTimeout(() => {
    console.log('Timer 1 finished');
  }, 0);

  setImmediate(() => {
    console.log('Immediate 1 finished');
  });

  console.log('End');
  ```

  Output:

```
sql
Copy code
Start
End
Immediate 1 finished
Timer 1 finished
```

The event loop processes `setImmediate()` callbacks before `setTimeout()` callbacks with a delay of 0, which might be surprising but demonstrates the asynchronous nature of Node.js.

3. Non-blocking I/O:

- Node.js uses non-blocking I/O operations, allowing it to handle multiple requests concurrently without blocking the main thread. For example, if a server needs to read data from a file, Node.js doesn't wait for the file to be read before handling other tasks. Instead, it initiates the read operation, registers a callback, and continues processing other events. Once the file is read, the callback is invoked, and the data is handled.

  Example:

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

console.log('Start reading file...');

fs.readFile('example.txt', 'utf-8', (err, data) => {
  if (err) throw err;
  console.log('File content:', data);
});

console.log('After reading file');
```

Output:

```
mathematica
Copy code
Start reading file...
After reading file
File content: (file contents here)
```

In this example, `fs.readFile` is non-blocking. The rest of the code executes while the file is being read, and the file's content is logged only after the read operation completes.

Setting up a Node.js Environment

To get started with Node.js, you need to set up a development environment on your machine. This involves installing Node.js and its package manager, npm.

1. Installing Node.js:

- **Windows/Mac/Linux:**
  - Go to the official Node.js website and download the installer for your operating system.
  - Run the installer and follow the instructions. This will install both Node.js and npm.
- **Using a Version Manager:**
  - For better control over Node.js versions, you can use a version manager like `nvm` (Node Version Manager) for macOS/Linux or `nvm-windows` for Windows.

Installing nvm on macOS/Linux:

```
bash
Copy code
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.3
```

```
9.3/install.sh | bash
```

- After installation, you can install Node.js using:

```
bash
Copy code
nvm install node
```

Installing nvm-windows:

- Download the nvm-setup.zip from the nvm-windows GitHub page.
- Extract and run the installer.

- After installation, you can verify the installation with:

```
bash
Copy code
node -v
npm -v
```

This will display the versions of Node.js and npm that have been installed.

2. Writing and Running Your First Node.js Script:

- Create a new file, `app.js`, and add the following code:

```
javascript
Copy code
console.log('Hello, Node.js!');
```

- Run the script in your terminal:

```
bash
Copy code
node app.js
```

You should see the output:

```
Copy code
Hello, Node.js!
```

3. Installing Packages with npm:

- npm (Node Package Manager) is the default package manager for Node.js. It allows you to install, share, and manage packages.
- Example: Installing the popular `express` package:

```
bash
Copy code
npm init -y  # Initialize a new Node.js project with defau
lt settings
npm install express
```

This command installs the `express` package locally in your project and adds it to the `package.json` file.

4. Basic Project Structure:

- A typical Node.js project might look like this:

```
perl
Copy code
my-node-app/
├── node_modules/       # Installed packages
├── public/             # Static files (images, CSS, etc.)
├── src/                # Application source code
│   ├── controllers/    # Route handlers
│   ├── models/         # Data models
│   └── routes/         # Application routes
├── app.js              # Main application file
├── package.json        # Project metadata and dependencie
s
└── README.md           # Project documentation
```

Core Modules and Global Objects in Node.js

Node.js comes with several built-in modules that provide essential functionalities to develop various types of applications. These modules can be accessed using the `require` function. Below, we will cover some of the most commonly used core modules, their features, and practical use cases.

1. `fs` (File System) Module

The `fs` module in Node.js is used for handling the file system. It allows you to interact with the file system on your computer, including reading, writing, updating, deleting, and managing files and directories.

Key Features:

- **Synchronous and Asynchronous Methods:** The `fs` module provides both synchronous (blocking) and asynchronous (non-blocking) methods. The asynchronous methods are preferred in production environments as they do not block the event loop.
- **Handling Files and Directories:** You can perform various operations like creating, reading, updating, and deleting files and directories.

Basic Example:

- **Reading a File (Asynchronous):**

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

fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  console.log('File content:', data);
});
```

This code reads the content of `example.txt` asynchronously. If the file does not exist or another error occurs, the error is logged to the console.

- **Writing to a File (Asynchronous):**

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

const content = 'This is some content to write to the fil
e.';

fs.writeFile('example.txt', content, (err) => {
  if (err) {
    console.error('Error writing to file:', err);
    return;
  }
  console.log('File has been written');
});
```

This example writes the specified content to `example.txt`. If the file doesn't exist, it is created; if it exists, the content is overwritten.

Industry-Relevant Use Case:

- **Log Management:** In a production environment, applications often need to log errors, access requests, or other critical information to files. Using the `fs` module, developers can write logs to files asynchronously, ensuring that the logging process does not block the main application flow.

  Example:

  ```
  javascript
  Copy code
  const fs = require('fs');
  const path = require('path');

  function logError(message) {
    const logMessage = `${new Date().toISOString()} - ERROR:
  ${message}\n`;
    fs.appendFile(path.join(__dirname, 'error.log'), logMess
  age, (err) => {
      if (err) {
        console.error('Failed to write to log file:', err);
      }
    });
  }

  // Usage
  logError('An unexpected error occurred');
  ```

---

2. `path` Module

The `path` module provides utilities for working with file and directory paths. This is especially useful for ensuring that your code works across different operating systems, as different OSs have different conventions for file paths.

Key Features:

- **Path Normalization:** The `path` module can normalize file paths, which is important when working with relative paths.
- **Path Joining:** It allows you to safely concatenate multiple path segments into a single path.
- **Platform Compatibility:** Ensures compatibility across different platforms (e.g., Windows vs. Unix-based systems).

Basic Example:

- **Joining Paths:**

```
javascript
Copy code
const path = require('path');

const fullPath = path.join(__dirname, 'files', 'example.tx
t');
console.log(fullPath);
```

  This code joins the current directory (`__dirname`) with the `files` directory and the `example.txt` file, ensuring that the path is correctly formatted for the operating system.

- **Extracting File Information:**

```
javascript
Copy code
const path = require('path');

const filePath = '/users/kuldeep/documents/example.txt';

console.log('Directory:', path.dirname(filePath));   // /u
sers/kuldeep/documents
console.log('File name:', path.basename(filePath));  // ex
```

```
ample.txt
console.log('Extension:', path.extname(filePath));   // .t
xt
```

This example extracts the directory, file name, and file extension from a given file path.

Industry-Relevant Use Case:

- **File Upload Management:** In a web application, when users upload files, you often need to save those files in a specific directory with a unique name to avoid conflicts. The `path` module helps manage these file paths reliably.

  Example:

```
javascript
Copy code
const path = require('path');
const fs = require('fs');

function saveUploadedFile(file, userId) {
  const uploadDir = path.join(__dirname, 'uploads', userI
d);

  if (!fs.existsSync(uploadDir)) {
    fs.mkdirSync(uploadDir, { recursive: true });
  }

  const fileName = `${Date.now()}-${path.basename(file.ori
ginalname)}`;
  const filePath = path.join(uploadDir, fileName);

  fs.writeFileSync(filePath, file.buffer);  // Sync for si
mplicity
  console.log('File saved at:', filePath);
}
```

```
// Usage: Assume `file` is an object representing the uplo
aded file
saveUploadedFile({ originalname: 'photo.jpg', buffer: Buff
er.from('...') }, 'user123');
```

This code saves an uploaded file to a directory named after the user ID, ensuring that each user's files are stored separately.

---

3. `http` Module

The `http` module allows you to create an HTTP server in Node.js. This module is fundamental for building web servers and RESTful APIs.

Key Features:

- **Creating Servers:** You can create an HTTP server that listens for requests on a specific port and responds to them.
- **Handling Requests and Responses:** The module provides methods for handling incoming HTTP requests and sending responses back to the client.
- **Routing:** Although basic, you can implement simple routing directly with the `http` module.

Basic Example:

- **Creating a Simple HTTP Server:**

```
javascript
Copy code
const http = require('http');

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});
```

```
server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

This example creates a basic HTTP server that responds with "Hello, World!" to every request.

Industry-Relevant Use Case:

- **Building a RESTful API:** The `http` module is often used to build RESTful APIs, which are essential for modern web applications. For example, you can handle different routes (endpoints) and HTTP methods (GET, POST, etc.) to provide an API for frontend applications or third-party clients.

  Example:

```
javascript
Copy code
const http = require('http');

const server = http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/api/users') {
    res.writeHead(200, { 'Content-Type': 'application/jso
n' });
    res.end(JSON.stringify([{ id: 1, name: 'John Doe' }, {
id: 2, name: 'Jane Doe' }]));
  } else if (req.method === 'POST' && req.url === '/api/us
ers') {
    let body = '';
    req.on('data', chunk => {
      body += chunk;
    });
    req.on('end', () => {
      const newUser = JSON.parse(body);
      newUser.id = Date.now(); // Simplified unique ID gen
eration
```

```
      res.writeHead(201, { 'Content-Type': 'application/js
on' });
      res.end(JSON.stringify(newUser));
    });
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});

server.listen(3000, () => {
  console.log('API server running at http://localhost:300
0/');
});
```

This example creates a simple RESTful API with two endpoints: one for fetching a list of users and another for creating a new user.

---

4. `url` Module

The `url` module provides utilities for URL resolution and parsing. It's particularly useful when working with URLs in HTTP servers or clients.

Key Features:

- **URL Parsing:** The `url` module can parse URLs into their constituent parts (e.g., protocol, host, pathname, query string).
- **URL Resolution:** It allows you to resolve relative URLs to absolute ones.

Basic Example:

- **Parsing a URL:**

```
javascript
Copy code
const url = require('url');
const myURL = new URL('https://www.example.com:8080/path/n
ame?search=test#hash');
console.log('Protocol:', myURL.protocol);  // https:
console.log('Host:', myURL.host);          // www.example.
com:8080
console.log('Pathname:', myURL.pathname);  // /path/name
console.log('Search Params:', myURL.searchParams);  // sea
rch=test
console.log('Hash:', myURL.hash);          // #hash
```

This code demonstrates how to parse different elements of a URL, such as the protocol, host, pathname, query parameters, and hash. The `URL` class from the `url` module is used to create an object representing the URL, which can then be easily manipulated or analyzed.

Industry-Relevant Use Case:

- **Query String Handling in APIs:** In web applications, particularly APIs, it's common to receive query parameters in the URL that influence the response. The `url` module can be used to parse these parameters and process them accordingly.

  Example:

  ```
  javascript
  Copy code
  const http = require('http');
  const url = require('url');

  const server = http.createServer((req, res) => {
    const parsedUrl = url.parse(req.url, true); // `true`
  parses the query string into an object
    const queryParams = parsedUrl.query;

    if (req.method === 'GET' && parsedUrl.pathname === '/
  api/search') {
  ```

```
  const searchTerm = queryParams.q || 'default search';
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({ searchTerm, results: `Results for ${searchTerm}` }));
 } else {
  res.writeHead(404, { 'Content-Type': 'text/plain' });
  res.end('Not Found');
 }
});

server.listen(3000, () => {
 console.log('Server running at http://localhost:3000/');
});
```

In this example, the server listens for GET requests on the `/api/search` endpoint and parses the query parameter `q` to perform a search operation.

---

5. `events` Module

The `events` module provides a way to work with events in Node.js. It is a core part of the Node.js event-driven architecture, allowing objects (called "emitters") to emit named events that other objects (called "listeners") can respond to.

Key Features:

- **EventEmitter Class:** The core of the `events` module is the `EventEmitter` class, which allows objects to emit and listen to events.
- **Custom Events:** You can define custom events that are specific to your application logic.

Basic Example:

- **Creating and Using an EventEmitter:**

  ```
  javascript
  Copy code
  const EventEmitter = require('events');

  class MyEmitter extends EventEmitter {}

  const myEmitter = new MyEmitter();

  // Registering an event listener
  myEmitter.on('event', () => {
    console.log('An event occurred!');
  });

  // Emitting the event
  myEmitter.emit('event');
  ```

  This code creates a custom event emitter and registers a listener for the `event` event. When the event is emitted using `emit`, the listener is invoked, logging a message to the console.

Industry-Relevant Use Case:

- **Custom Event Handling in Complex Applications:** In large-scale Node.js applications, particularly those involving microservices or complex workflows, custom events can be used to decouple components and handle asynchronous operations.

  Example:

  ```
  javascript
  Copy code
  const EventEmitter = require('events');
  ```

```
class OrderProcessor extends EventEmitter {
  process(order) {
    console.log('Processing order:', order.id);
    // Simulate order processing
    setTimeout(() => {
      this.emit('orderProcessed', { orderId: order.id, status: 'completed' });
    }, 1000);
  }
}

const processor = new OrderProcessor();

// Listen for the custom event
processor.on('orderProcessed', (result) => {
  console.log(`Order ${result.orderId} has been processed with status: ${result.status}`);
});

// Start processing an order
processor.process({ id: 12345 });
```

In this scenario, an `OrderProcessor` class emits a custom `orderProcessed` event when an order is processed, allowing other parts of the application to react accordingly.

---

6. `os` Module

The `os` module provides utilities for interacting with the operating system. It allows you to retrieve information about the OS on which your Node.js application is running, such as the platform, CPU architecture, memory usage, and more.

Key Features:

- **System Information:** Retrieve details like CPU architecture, network interfaces, free memory, uptime, and more.
- **Cross-Platform Compatibility:** The module abstracts OS-specific details, providing a consistent API across different platforms.

Basic Example:

- **Getting System Information:**

  ```
  javascript
  Copy code
  const os = require('os');

  console.log('Operating System:', os.platform());
  // e.g., win32, linux, darwin
  console.log('CPU Architecture:', os.arch());
  // e.g., x64, arm
  console.log('Total Memory:', os.totalmem());
  // Total system memory in bytes
  console.log('Free Memory:', os.freemem());
  // Free system memory in bytes
  console.log('Uptime:', os.uptime(), 'seconds');
  // System uptime in seconds
  ```

  This code retrieves and logs various pieces of information about the operating system, such as its platform, CPU architecture, and memory status.

Industry-Relevant Use Case:

- **System Monitoring and Diagnostics:** In production environments, it's important to monitor the health of the system running your Node.js application. The `os` module can be used to collect metrics and diagnose issues related to resource usage.

  Example:

```
javascript
Copy code
const os = require('os');

function logSystemHealth() {
  console.log('CPU Load:', os.loadavg());  // Average C
PU load over 1, 5, and 15 minutes
  console.log('Free Memory:', os.freemem() / 1024 / 102
4, 'MB');
  console.log('Total Memory:', os.totalmem() / 1024 / 1
024, 'MB');
  console.log('Uptime:', os.uptime(), 'seconds');
  console.log('Running on:', os.platform(), os.release
());
  console.log('Network Interfaces:', os.networkInterfac
es());
}

setInterval(logSystemHealth, 5000);  // Log system heal
th every 5 seconds
```

This script logs system health metrics every 5 seconds, which can be useful for monitoring resource usage in real-time.

---

7. `crypto` Module

The `crypto` module provides cryptographic functionality, including operations like hashing, encryption, and decryption. This module is essential for building secure applications, handling authentication, and ensuring data integrity.

Key Features:

- **Hashing:** Create secure hashes using algorithms like SHA-256, MD5, etc.
- **Encryption/Decryption:** Encrypt and decrypt data using symmetric (AES) and asymmetric (RSA) encryption.

- **Random Bytes:** Generate secure random numbers or bytes, which are critical for creating tokens, session IDs, and other security-sensitive data.

Basic Example:

- **Creating a Hash:**

```
javascript
Copy code
const crypto = require('crypto');

const hash = crypto.createHash('sha256');
hash.update('password123');
const hashedPassword = hash.digest('hex');
console.log('Hashed Password:', hashedPassword);
```

  This code generates a SHA-256 hash of the string "password123", which is a common technique for securely storing passwords.

Industry-Relevant Use Case:

- **Token Generation for Authentication:** In web applications, especially those using token-based authentication (like JWTs), the `crypto` module is often used to generate secure tokens.

  Example:

```
javascript
Copy code
const crypto = require('crypto');

function generateAuthToken() {
  return crypto.randomBytes(32).toString('hex');  // 64
-character hex string
}

// Usage
const authToken = generateAuthToken();
```

```
console.log('Generated Auth Token:', authToken);
```

This function generates a secure, random authentication token, which could be used in session management or API authentication.

Understanding Global Objects in Node.js

In Node.js, there are several global objects that are available in all modules without the need to require them. These objects provide important information about the environment in which your code is running.

1. `__dirname`

The `__dirname` global object represents the directory name of the current module. This is especially useful for resolving file paths relative to the module's location.

Example:

```
javascript
Copy code
console.log('Directory name:', __dirname);
```

This will print the absolute path of the directory in which the current script is located.

Use Case:

- **Serving Static Files:** When building a web server, you might need to serve static files (HTML, CSS, JavaScript). Using `__dirname`, you can construct the absolute path to the directory where the static files are stored.

  ```
  javascript
  Copy code
  const path = require('path');
  const express = require('express');
  const app = express();
  ```

```
app.use(express.static(path.join(__dirname, 'publi
c')));

app.listen(3000, () => {
  console.log('Server running at http://localhost:300
0/');
});
```

In this example, the static files located in the `public` directory relative to the script's location are served by the Express.js server.

2. `__filename`

The `__filename` global object provides the absolute path of the current module (including the file name).

Example:

```
javascript
Copy code
console.log('File name:', __filename);
```

This will print the full path to the current file.

Use Case:

- **Logging or Error Reporting:** When an error occurs, you might want to log the file in which it happened for easier debugging. `__filename` can be used to include the file name in the error logs, making it easier to identify the source of the problem.

Example:

```
javascript
Copy code
try {
  // Simulate an error
  throw new Error('Something went wrong!');
} catch (error) {
  console.error(`Error in file ${__filename}: ${error.mess
age}`);
}
```

In this example, when an error occurs, the error message includes the name of the file where it happened, which can be very useful in debugging and tracing issues in larger codebases.

---

3. `process`

The `process` global object provides information about the current Node.js process and methods to interact with it. This object is critical for handling environment variables, exiting the process, capturing command-line arguments, and managing signals.

Key Features:

- **Environment Variables:** Access environment variables using `process.env`.
- **Command-Line Arguments:** Access arguments passed to the script using `process.argv`.
- **Exit Codes:** Exit the process with a specific code using `process.exit()`.
- **Standard Input/Output:** Interact with the terminal via `process.stdin`, `process.stdout`, and `process.stderr`.

Basic Examples:

- **Accessing Environment Variables:**

  ```
  javascript
  Copy code
  const dbPassword = process.env.DB_PASSWORD;
  console.log('Database Password:', dbPassword);
  ```

This example retrieves a database password from an environment variable, which is a common practice in production to keep sensitive information out of the source code.

- **Handling Command-Line Arguments:**

  ```
  javascript
  Copy code
  const args = process.argv.slice(2); // The first two el
  ements are node and the script name
  console.log('Command-line arguments:', args);
  ```

  This code logs the arguments passed to the Node.js script when it is executed from the command line.

- **Exiting the Process:**

  ```
  javascript
  Copy code
  if (someCriticalError) {
    console.error('A critical error occurred. Exiting the
  process...');
    process.exit(1); // Exit with a non-zero code to indi
  cate failure
  }
  ```

  This code exits the Node.js process with an exit code of `1` if a critical error occurs, signaling failure to the operating system.

Industry-Relevant Use Case:

- **Graceful Shutdowns in Servers:** In a production environment, it's important to handle shutdown signals (like `SIGINT` or `SIGTERM`) gracefully to ensure that ongoing requests are completed and resources are properly released.

  Example:

```
javascript
Copy code
const http = require('http');

const server = http.createServer((req, res) => {
  res.end('Hello, World!');
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:300
0/');
});

process.on('SIGTERM', () => {
  console.log('Received SIGTERM, shutting down graceful
ly...');
  server.close(() => {
    console.log('All connections closed. Exiting no
w.');
    process.exit(0);
  });
});

process.on('SIGINT', () => {
  console.log('Received SIGINT, shutting down gracefull
y...');
  server.close(() => {
    console.log('All connections closed. Exiting no
w.');
    process.exit(0);
  });
});
```

In this example, the server listens for termination signals (`SIGTERM` and `SIGINT`). When one is received, the server stops accepting new connections and waits for existing connections to close before exiting, ensuring that no data is lost or corrupted.

Summary of Core Modules and Global Objects

Understanding and mastering these core modules and global objects in Node.js is essential for building efficient, scalable, and secure applications. They provide the fundamental tools required to interact with the file system, manage web servers, handle URLs, emit and listen to events, access system information, and work with cryptography.

In a real-world context, these modules and objects allow developers to create everything from simple scripts to complex, enterprise-level applications. Whether you are handling file uploads, creating APIs, managing authentication, or optimizing performance, these tools form the backbone of your Node.js development toolkit.

Introduction to the Stream Module in Node.js

The `stream` module is one of the most powerful features of Node.js, enabling the handling of streaming data in an efficient and memory-conscious manner. Streams are an abstraction for working with data that is either being read from a source or written to a destination in a continuous manner, rather than in one large chunk. This is particularly useful for handling large files, working with real-time data, and optimizing performance in I/O operations.

Types of Streams in Node.js

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

1. **Readable Streams**: Used to read data from a source.
2. **Writable Streams**: Used to write data to a destination.

3. **Duplex Streams**: A stream that is both readable and writable (e.g., a network socket).
4. **Transform Streams**: A type of duplex stream where the output is computed based on the input (e.g., data compression).

---

1. Readable Streams

Readable Streams are streams from which data can be read. A common example is reading data from a file or an HTTP request.

Basic Example:

- **Reading from a File:**

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

const readableStream = fs.createReadStream('largefile.txt', { encoding: 'utf8' });

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

readableStream.on('end', () => {
  console.log('Finished reading file.');
});

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

In this example, data is read from `largefile.txt` in chunks, which helps to manage memory more efficiently, especially when dealing with large files.

Industry-Relevant Use Case:

- **Streaming Video Content:** In a video streaming service, you might use readable streams to send video data from a server to a client in real-time, rather than sending the entire file at once.

  Example:

  ```
  javascript
  Copy code
  const http = require('http');
  const fs = require('fs');

  http.createServer((req, res) => {
    const readableStream = fs.createReadStream('video.mp
  4');
    res.writeHead(200, { 'Content-Type': 'video/mp4' });
    readableStream.pipe(res);  // Pipes the video data to
  the response stream
  }).listen(3000, () => {
    console.log('Server running at http://localhost:300
  0/');
  });
  ```

  Here, the video file is read in chunks and streamed directly to the client, allowing the video to start playing almost immediately.

---

2. Writable Streams

Writable Streams are streams to which data can be written. A common example is writing data to a file or sending data over an HTTP response.

Basic Example:

- **Writing to a File:**

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

const writableStream = fs.createWriteStream('output.tx
t');

writableStream.write('Hello, World!\n');
writableStream.write('Streaming data to a file.\n');

writableStream.end(() => {
  console.log('Finished writing to file.');
});

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

This example writes data to `output.txt` in a streaming manner, rather than all at once.

Industry-Relevant Use Case:

- **Logging System:** In a logging system for a production application, you might use writable streams to continuously write logs to a file or a remote logging service without overwhelming the memory.

  Example:

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

const logStream = fs.createWriteStream('app.log', { fla
gs: 'a' });  // 'a' flag opens the file for appending
```

```
function logMessage(message) {
  logStream.write(`${new Date().toISOString()} - ${mess
age}\n`);
}

logMessage('Server started.');
logMessage('Handling user request.');
```

This code appends log messages to `app.log` as they occur, which is typical in long-running server applications.

---

3. Duplex Streams

Duplex Streams are streams that are both readable and writable. A common example is a TCP socket, where data can be sent and received simultaneously.

Basic Example:

- **TCP Socket Communication:**

```
javascript
Copy code
const net = require('net');

const server = net.createServer((socket) => {
  socket.on('data', (data) => {
    console.log('Received:', data.toString());
    socket.write('Hello, client!');
  });

  socket.on('end', () => {
    console.log('Connection ended.');
  });
});
```

```
server.listen(8080, () => {
  console.log('Server listening on port 8080');
});
```

In this example, the server listens for data from a client and can send a response back using the same socket.

Industry-Relevant Use Case:

- **Real-Time Chat Application:** In a chat application, duplex streams can be used to handle bi-directional communication between the server and multiple clients.

  Example:

```
javascript
Copy code
const net = require('net');

const clients = [];

const server = net.createServer((socket) => {
  clients.push(socket);
  socket.on('data', (data) => {
    clients.forEach((client) => {
      if (client !== socket) {
        client.write(data);
      }
    });
  });

  socket.on('end', () => {
    clients.splice(clients.indexOf(socket), 1);
  });
});
```

```
server.listen(8080, () => {
  console.log('Chat server running on port 8080');
});
```

Here, each client can send and receive messages in real-time, making it possible to implement a basic chat server.

---

4. Transform Streams

Transform Streams are a special type of duplex stream where the output is computed based on the input. This is often used for data compression, encryption, or any operation where the data needs to be transformed as it is being read or written.

Basic Example:

- **Data Compression Using zlib:**

```
javascript
Copy code
const fs = require('fs');
const zlib = require('zlib');

const readableStream = fs.createReadStream('input.tx
t');
const writableStream = fs.createWriteStream('input.txt.
gz');
const gzip = zlib.createGzip();

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

In this example, data from `input.txt` is compressed on-the-fly and written to `input.txt.gz`.

Industry-Relevant Use Case:

- **HTTP Compression:** In web servers, you can use transform streams to compress HTTP responses on the fly, reducing the amount of data transferred and speeding up the user's experience.

  Example:

  ```
  javascript
  Copy code
  const http = require('http');
  const zlib = require('zlib');

  http.createServer((req, res) => {
    const gzip = zlib.createGzip();
    res.writeHead(200, { 'Content-Type': 'text/plain', 'C
  ontent-Encoding': 'gzip' });
    gzip.write('Hello, World!');
    gzip.end();
    gzip.pipe(res);
  }).listen(3000, () => {
    console.log('Server running at http://localhost:300
  0/');
  });
  ```

  Here, the server compresses the response using gzip before sending it to the client, which is a common optimization technique used in production web servers.

Advanced Usage: Stream Piping and Chaining

One of the most powerful features of streams in Node.js is the ability to pipe and chain streams together. Piping allows the output of one stream to be fed directly into another stream, creating a data flow.

Example:

- **Piping Multiple Streams Together:**

```
javascript
Copy code
const fs = require('fs');
const zlib = require('zlib');
const crypto = require('crypto');

const readableStream = fs.createReadStream('input.tx
t');
const gzip = zlib.createGzip();
const cipher = crypto.createCipher('aes-256-cbc', 'pass
word');
const writableStream = fs.createWriteStream('output.tx
t.gz.enc');

readableStream.pipe(gzip).pipe(cipher).pipe(writableStr
eam);
```

In this example, data from `input.txt` is first compressed using gzip, then encrypted, and finally written to an output file. This showcases how streams can be combined to handle complex data flows efficiently.

Industry-Relevant Use Case:

- **Secure File Transfer:** For secure file transfers, you might combine compression and encryption to ensure that files are both compact and protected during transmission.

Summary of the Stream Module

The `stream` module in Node.js provides a powerful and flexible way to handle data flows, making it a fundamental tool for building scalable, high-performance applications. Whether you're reading and writing files, handling network communication, or processing real-time data, streams enable efficient and responsive handling of large volumes of data.
