1. Introduction to Client-Server Architecture

Client-Server Architecture is a model where tasks or workloads are divided between providers of a resource or service (servers) and service requesters (clients). This model is fundamental to modern computing and networking.

- **Client:** A client is any device or application that makes requests for services or resources from a server. Examples include web browsers, mobile apps, and desktop applications.
- **Server:** A server is a machine or software that provides resources, data, services, or programs to clients. Servers typically handle requests from multiple clients simultaneously.

How Requests Are Handled at the Server:

1. **Request Reception:** The server receives a request from a client.
2. **Processing:** The server processes the request, which may involve accessing a database, performing calculations, or other operations.
3. **Response Generation:** After processing, the server generates a response.
4. **Sending Response:** The server sends the response back to the client.

2. Understanding Node.js

Node.js is a runtime built on Chrome's V8 engine. It allows you to run code on the server side.

Advantages:

- **Single Language:** You can use for both client-side and server-side scripting.
- **Non-blocking I/O:** Node.js uses an event-driven, non-blocking I/O model, making it efficient and suitable for I/O-heavy operations.
- **NPM (Node Package Manager):** Provides a vast library of modules and packages that can be easily integrated into projects.
- **Scalability:** It's designed for scalable network applications.

Disadvantages:

- **Single-threaded:** Node.js uses a single-threaded model, which might not be ideal for CPU-intensive operations.
- **Callback Hell:** Complex applications can lead to deeply nested callbacks, although this can be managed with Promises and async/await.
- **Not Ideal for Heavy Computations:** Node.js is less suitable for applications requiring heavy computation due to its single-threaded nature.

3. Installing Node.js

To install Node.js, follow these steps:

1. **Download Node.js:** Go to the official Node.js website and download the installer for your operating system.
2. **Run the Installer:** Follow the installation instructions.

3. **Verify Installation:** Open a terminal or command prompt and run:

bash

.

```
node -v
npm -v
```

This will show the installed versions of Node.js and npm.

4. Creating a Node.js Server

To create a simple Node.js server:

1. **Create a Directory:** Make a new directory for your project.
2. **Initialize the Project:** Run npm init to create a package.json file.
3. **Install Express (Optional):** Express is a popular framework for Node.js. Install it using:

bash

.

```
npm install express
```

4. **Create a Server File:**

.

```
// server.js
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

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

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
```

});

5. **Run the Server:**

bash

.

node server.js

5. Creating Endpoints

Using Express, you can create endpoints easily:

.

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.get('/about', (req, res) => {
  res.send('About Page');
});

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

6. Modules and NPM

Modules in Node.js are reusable pieces of code that can be imported into other files.

- **Creating a Module:**

.

```
// myModule.js
module.exports = function() {
```

```
  return "Hello from my module!";
};
```

- **Using a Module:**

.

```
const myModule = require('./myModule');
console.log(myModule());
```

NPM (Node Package Manager): Manages packages and modules in Node.js. You can install packages using:

bash

.

```
npm install <package-name>
```

7. Handling Static Pages with File Streams

To serve static files (like HTML, CSS, JS) using Express:

.

```
const express = require('express');
const app = express();
const path = require('path');

app.use(express.static(path.join(__dirname, 'public')));

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Put your static files in a public directory.

8. Handling Exceptions

In Node.js, you can handle exceptions using try...catch blocks and handling errors in callback functions or promises.

- **Synchronous Errors:**

.

```
try {
  // Code that might throw an error
} catch (error) {
  console.error('An error occurred:', error);
}
```

- **Asynchronous Errors (Promises):**

.

```
async function doSomething() {
  try {
    // Code that might throw an error
  } catch (error) {
    console.error('An error occurred:', error);
  }
}
```

- **Handling Express Errors:**

.

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

1. Client-Server Architecture

Client-Server Architecture involves:

- **Client:** The client initiates communication. It could be a web browser requesting a webpage, a mobile app fetching data, or even another server making API calls. Clients are typically responsible for the user interface and user interactions.
- **Server:** The server listens for incoming requests from clients, processes them, and sends back the appropriate responses. Servers are responsible for storing data, running applications, and managing resources.

Request-Response Cycle:

1. **Request:** The client sends an HTTP request to the server. This request contains details such as the method (GET, POST, PUT, DELETE), headers, and sometimes a body.

2. **Processing:** The server processes the request. This can include querying a database, performing business logic, or interacting with other services.
3. **Response:** After processing, the server sends an HTTP response back to the client. This response includes a status code, headers, and sometimes a body containing the requested data or information.

2. Understanding Node.js

Node.js is a powerful tool for building server-side applications with . Here are more details:

- **Event-Driven Architecture:** Node.js uses an event-driven model where events (like HTTP requests) are processed by an event loop. This allows Node.js to handle multiple requests concurrently without blocking the main thread.
- **Asynchronous Non-Blocking I/O:** Operations such as reading files or querying databases don't block the execution of other code. This is achieved through callbacks, Promises, or async/await.
- **Single-Threaded with Worker Threads:** Node.js operates on a single thread but uses worker threads for handling CPU-intensive tasks. This helps in maintaining performance for I/O-heavy operations while delegating heavy computations.

3. Installing Node.js

Detailed Steps:

1. **Download Node.js:**
    - Visit the Node.js website and download the version suitable for your operating system (LTS version is recommended for stability).
2. **Run the Installer:**
    - Follow the installation wizard. The installer will also include npm (Node Package Manager), which is essential for managing packages.
3. **Verify Installation:**
    - Open a terminal or command prompt and type:

bash

.

node -v

npm -v

    - This confirms the installation and shows the installed versions of Node.js and npm.

4. Creating a Node.js Server

A basic server with Node.js can be set up as follows:

1. **Create Project Directory:**

bash

.

```
mkdir my-node-server
cd my-node-server
```

2. **Initialize Project:**

bash

.

```
npm init -y
```

- This creates a package.json file with default settings.

3. **Create Server File (server.js):**

.

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

const hostname = '127.0.0.1';
const port = 3000;

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

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

4. **Run Server:**

bash

.

```
node server.js
```

- Navigate to http://127.0.0.1:3000 in your browser to see "Hello World".

5. Creating Endpoints

Endpoints are specific routes in your application where you define how to respond to requests:

.

```
const express = require('express');
const app = express();
const port = 3000;

// Define routes
app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.get('/about', (req, res) => {
  res.send('About Page');
});

// Start server
app.listen(port, () => {
  console.log(`Server running on http://localhost:${port}`);
});
```

Key Points:

- **app.get** defines a route for GET requests.
- **app.listen** starts the server on the specified port.

6. Modules and NPM

Modules:

- **Creating a Module:**

.

```
// myModule.js
module.exports = {
  greet: function() {
```

```
    return "Hello from my module!";
  }
};
```

- **Using a Module:**

.

```
const myModule = require('./myModule');
console.log(myModule.greet());  // Outputs: Hello from my module!
```

NPM:

- **Install a Package:**

bash

.

```
npm install lodash
```

- **Import and Use a Package:**

.

```
const _ = require('lodash');
console.log(_.camelCase('Hello World'));  // Outputs: helloWorld
```

7. Handling Static Pages with File Streams

To serve static files using Express:

1. **Set Up Directory Structure:**
    - Place your static files (HTML, CSS, JS) in a directory called public.
2. **Serve Static Files:**

.

```
const express = require('express');
const path = require('path');
const app = express();
const port = 3000;

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

```
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Key Points:

- express.static middleware serves static files from a specified directory.

8. Handling Exceptions

Error Handling:

- **Synchronous Code:**

```
.
try {
  // Risky code
} catch (error) {
  console.error('Caught an error:', error);
}
```

- **Asynchronous Code:**

```
.
async function riskyOperation() {
  try {
    // Risky code
  } catch (error) {
    console.error('Caught an error:', error);
  }
}
```

- **Express Error Handling Middleware:**

```
.
app.use((err, req, res, next) => {
  console.error(err.stack);
```

```
  res.status(500).send('Something went wrong!');
});
```

Error codes and status codes are fundamental to understanding how web applications and APIs communicate the results of their operations.

HTTP Status Codes

HTTP Status Codes are three-digit codes sent by the server in response to a client's request. They indicate the outcome of the request and are divided into five categories:

1. **1xx (Informational):**
    - **100 Continue:** The initial part of a request has been received and the client should continue with the request.
    - **101 Switching Protocols:** The server is switching protocols as requested by the client.
2. **2xx (Successful):**
    - **200 OK:** The request was successful, and the server is returning the requested data.
    - **201 Created:** The request was successful, and a new resource has been created.
    - **202 Accepted:** The request has been accepted for processing, but the processing has not been completed.
    - **204 No Content:** The request was successful, but there is no content to send in the response.
3. **3xx (Redirection):**
    - **301 Moved Permanently:** The resource has been moved permanently to a new URL.
    - **302 Found (or Temporary Redirect):** The resource has been temporarily moved to a different URL.
    - **304 Not Modified:** The resource has not been modified since the last request.
4. **4xx (Client Error):**
    - **400 Bad Request:** The server could not understand the request due to invalid syntax.
    - **401 Unauthorized:** Authentication is required and has failed or has not been provided.
    - **403 Forbidden:** The server understood the request but refuses to authorize it.
    - **404 Not Found:** The requested resource could not be found.
    - **405 Method Not Allowed:** The request method is not allowed for the requested resource.
    - **408 Request Timeout:** The server timed out waiting for the request.
    - **409 Conflict:** There is a conflict with the current state of the resource.

5. **5xx (Server Error):**
    - **500 Internal Server Error:** The server encountered an unexpected condition that prevented it from fulfilling the request.
    - **501 Not Implemented:** The server does not support the functionality required to fulfill the request.
    - **502 Bad Gateway:** The server received an invalid response from an upstream server.
    - **503 Service Unavailable:** The server is currently unable to handle the request due to temporary overloading or maintenance.
    - **504 Gateway Timeout:** The server did not receive a timely response from an upstream server.

Error Codes in Node.js and Express

In Node.js and Express applications, error codes are often used in conjunction with HTTP status codes to provide more specific information about errors. Here are some common patterns and practices:

1. **Custom Error Handling:**
    - **Creating Custom Errors:**

.

.

```
class CustomError extends Error {
  constructor(statusCode, message) {
    super(message);
    this.statusCode = statusCode;
  }
}
```

    - **Throwing Custom Errors:**

.

.

```
throw new CustomError(404, 'Resource not found');
```

    - **Handling Errors in Express:**

.

.

```
app.use((err, req, res, next) => {
  const statusCode = err.statusCode || 500;
  const message = err.message || 'Internal Server Error';
```

```
  res.status(statusCode).json({ message });
});
```

2. **Common Error Handling Patterns:**
    - **Validation Errors:** Often return 400 Bad Request with details about what went wrong.
    - **Authentication Errors:** Usually return 401 Unauthorized if credentials are missing or incorrect.
    - **Authorization Errors:** Return 403 Forbidden when the user is authenticated but does not have permission.
    - **Resource Not Found:** Return 404 Not Found when the requested resource does not exist.
    - **Server Errors:** Return 500 Internal Server Error for unexpected issues that do not fall into specific categories.

Best Practices for Error Handling

1. **Be Consistent:** Use standard status codes and provide clear, consistent error messages.
2. **Include Error Details:** Where appropriate, include details in the error response to help clients understand and fix the issue.
3. **Log Errors:** Always log errors on the server side for debugging and monitoring purposes.
4. **Graceful Degradation:** Handle errors gracefully to ensure that the user experience is not severely impacted.

Example Express Error Handling:

```
.

.
const express = require('express');
const app = express();


// Define routes
app.get('/', (req, res) => {
  res.send('Hello World!');
});


// Example route that triggers an error
app.get('/error', (req, res, next) => {
  const error = new Error('Something went wrong!');
```

```
  error.statusCode = 500;
  next(error);
});

// Global error handler
app.use((err, req, res, next) => {
  res.status(err.statusCode || 500).json({
    status: 'error',
    statusCode: err.statusCode || 500,
    message: err.message || 'Internal Server Error',
  });
});

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

Node.js

Node.js is a powerful, open-source JavaScript runtime built on Chrome's V8 JavaScript engine. It allows you to execute JavaScript code server-side, enabling the development of scalable network applications.

Key Features:

- **Event-Driven and Asynchronous:** Node.js uses an event-driven, non-blocking I/O model, making it efficient and suitable for handling numerous simultaneous connections.
- **Single-Threaded:** Node.js operates on a single-threaded event loop, which handles multiple operations asynchronously.
- **Built-in Modules:** Node.js includes a set of built-in modules that provide core functionality such as HTTP, file system operations, and more.
- **npm (Node Package Manager):** Node.js comes with npm, a package manager that allows you to install and manage third-party libraries and tools.

Example Usage:

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


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


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

Node Modules

Node Modules are JavaScript libraries and tools that extend Node.js's functionality. They can be:

1. **Core Modules:** Built into Node.js, such as http, fs, and path. They provide essential features without needing to install additional packages.
2. **Local Modules:** Modules that you create in your project. You can use module.exports and require to manage local modules.
3. **Third-Party Modules:** Modules installed from npm. These are shared by the community and can be installed via npm install.

Example of Using a Core Module:

```
const path = require('path');
console.log(path.join('/foo', 'bar', 'baz/asdf', 'quux', '..'));
```

Example of Creating and Using a Local Module:

1. **Creating a Local Module:**

```
// myModule.js
```

```
module.exports = {
  greet: function() {
    return 'Hello from my module!';
  }
};
```

2. **Using the Local Module:**

```
const myModule = require('./myModule');
console.log(myModule.greet()); // Outputs: Hello from my module!
```

Example of Installing and Using a Third-Party Module:

1. **Install a Module:**

bash

```
npm install lodash
```

2. **Use the Module:**

javascript

.

```
const _ = require('lodash');
console.log(_.camelCase('Hello World')); // Outputs: helloWorld
```

The fs Module

The fs (File System) module in Node.js provides an API for interacting with the file system. It allows you to read from and write to files, manage directories, and more.

Key Methods:

1. **Reading Files:**

.

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

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

- **Synchronously:**

.

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

try {
  const data = fs.readFileSync('example.txt', 'utf8');
  console.log('File contents:', data);
} catch (err) {
  console.error('Error reading file:', err);
}
```

2. **Writing Files:**

- **Asynchronously:**

.

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

fs.writeFile('example.txt', 'Hello, world!', (err) => {
  if (err) {
    console.error('Error writing file:', err);
```

```
    return;
  }
  console.log('File has been written');
});
```

- **Synchronously:**

javascript

.

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

try {
  fs.writeFileSync('example.txt', 'Hello, world!');
  console.log('File has been written');
} catch (err) {
  console.error('Error writing file:', err);
}
```

3. **Other File Operations:**

- **Appending to a File:**

javascript

.

```
fs.appendFile('example.txt', 'Appending some text...', (err) => {
  if (err) {
    console.error('Error appending file:', err);
    return;
  }
  console.log('Text has been appended to file');
```

});

- **Deleting a File:**

javascript

.

```
fs.unlink('example.txt', (err) => {
  if (err) {
    console.error('Error deleting file:', err);
    return;
  }
  console.log('File has been deleted');
});
```

- **Creating a Directory:**

javascript

.

```
fs.mkdir('newDirectory', (err) => {
  if (err) {
    console.error('Error creating directory:', err);
    return;
  }
  console.log('Directory created');
});
```

- **Listing Files in a Directory:**

javascript

.

```
fs.readdir('someDirectory', (err, files) => {
```

```
  if (err) {
    console.error('Error reading directory:', err);
    return;
  }
  console.log('Files in directory:', files);
});
```

The fs module supports both callback-based and promise-based APIs (with the fs.promises API), allowing you to work with asynchronous file operations in a more modern way using async/await.

Example with fs.promises:

javascript

.

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

async function readFile() {
  try {
    const data = await fs.readFile('example.txt', 'utf8');
    console.log('File contents:', data);
  } catch (err) {
    console.error('Error reading file:', err);
  }
}

readFile();
```

Streams in Node.js

Streams in Node.js are a powerful way to handle and process large amounts of data efficiently. They allow you to work with data piece by piece, rather than loading it all into memory at once. This is particularly useful for handling large files or real-time data.

Types of Streams:

1. **Readable Streams:**
    - Used for reading data.
    - Example: Reading a file, receiving HTTP request data.

Example of Readable Stream:

javascript

.

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


const readableStream = fs.createReadStream('example.txt', 'utf8');


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


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

2. **Writable Streams:**
    - Used for writing data.
    - Example: Writing to a file, sending HTTP response data.

Example of Writable Stream:

javascript

.

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


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


writableStream.write('Hello, ');
writableStream.write('world!\n');
writableStream.end();
```

3. **Duplex Streams:**
    - Can be used for both reading and writing.
    - Example: TCP sockets.

Example of Duplex Stream:

javascript

.

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


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


server.listen(8080, () => {
```

```
  console.log('Server listening on port 8080');
});
```

4. **Transform Streams:**
    - A type of duplex stream that can modify data as it is read and written.
    - Example: Zlib compression.

Example of Transform Stream:

javascript

.

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

const uppercaseTransform = new Transform({
  transform(chunk, encoding, callback) {
    this.push(chunk.toString().toUpperCase());
    callback();
  }
});

process.stdin.pipe(uppercaseTransform).pipe(process.stdout);
```

Common Methods and Events:

- **Events:**
    - data: Fired when there is data to be read.
    - end: Fired when there is no more data.
    - error: Fired when an error occurs.
    - finish: Fired when all data has been flushed to the underlying system (for writable streams).

- **Methods:**
    - pipe(destination): Pipes the data from a readable stream to a writable stream.
    - unpipe(destination): Stops piping from a readable stream to a writable stream.

Piping Example:

javascript

.

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


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


readableStream.pipe(writableStream);
```

EJS (Embedded JavaScript Templates)

EJS is a templating engine for Node.js that allows you to embed JavaScript code into your HTML. It is used to generate dynamic HTML content on the server side.

Key Features:

- **Template Inheritance:** Allows you to extend and include other templates.
- **JavaScript Logic:** You can use JavaScript expressions and control structures within templates.
- **Simple Syntax:** EJS uses plain HTML with embedded JavaScript.

Basic Syntax:

1. **Embedding Variables:**

html

.

```
<p>Hello, <%= name %>!</p>
```

2. **Using JavaScript Logic:**

html

.

```
<% if (user) { %>
  <p>Welcome, <%= user.name %>!</p>
<% } else { %>
  <p>Please log in.</p>
<% } %>
```

3. **Including Templates:**

html

.

```
<% include header.ejs %>
<p>This is the main content.</p>
<% include footer.ejs %>
```

Example of Using EJS in an Express Application:

1. **Setup:**

bash

.

```
npm install express ejs
```

2. **Server File (server.js):**

javascript

.

```
const express = require('express');
const app = express();


// Set EJS as the view engine
```

```
app.set('view engine', 'ejs');


// Define a route
app.get('/', (req, res) => {
  res.render('index', { name: 'World' });
});


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

3. **EJS Template (views/index.ejs):**

```
<!DOCTYPE html>
<html>
<head>
  <title>My EJS App</title>
</head>
<body>
  <h1>Hello, <%= name %>!</h1>
</body>
</html>
```

In this setup:

- The server uses EJS as the templating engine.
- The res.render method is used to render the index.ejs template with the name variable.