The http module in Node.js allows you to create and manage HTTP servers and clients. This module provides the foundation for handling HTTP requests and responses, making it useful for building web servers and handling requests in Node.js applications.

Here's an overview of the http module with examples:

1. Creating a Basic HTTP Server

The http.createServer() method creates an HTTP server that listens for requests and responds to them.

javascript
Copy code

```
const http = require('http');
const server = http.createServer((req, res) => {
   res.statusCode = 200; // Set the response status code to 200 (OK)
   res.setHeader('Content-Type', 'text/plain'); // Set the response header
   res.end('Hello, World!'); // Send the response body
});
server.listen(3000, () => {
   console.log('Server running at http://localhost:3000/');
});
```

- **Explanation**:
  - req is the request object, which contains details about the incoming HTTP request.
  - res is the response object, which is used to send data back to the client.
  - res.end() ends the response process, and you can pass a string to it to be sent as the response body.

2. Handling Different HTTP Methods

The server can handle different HTTP methods (like GET, POST, PUT, DELETE) by checking req.method.

javascript
Copy code

```
const http = require('http');
const server = http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/') {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end('<h1>Welcome to the Home Page</h1>');
  } else if (req.method === 'POST' && req.url === '/submit') {
    let body = '';
    req.on('data', chunk => { body += chunk; });
    req.on('end', () => {
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end(`Data received: ${body}`);
    });
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('404 Not Found');
  }
});
server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

- **Explanation**:
  - Different methods (like GET or POST) can be handled with conditional statements.
  - For POST requests, req.on('data') collects incoming data in chunks, and req.on('end') processes the complete data.

3. Sending JSON Responses

Node.js can easily handle JSON responses, which is useful in APIs.

javascript
Copy code

```
const http = require('http');
const server = http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/api') {
    const data = { message: 'Hello, API!' };
    res.writeHead(200, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify(data));
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
```

```
        res.end('404 Not Found');
    }
});
server.listen(3000, () => {
    console.log('Server running at http://localhost:3000/');
});
```

4. Reading Request Data (e.g., POST data)

To read data sent in a POST request, you can use req.on('data') to accumulate data, and req.on('end') to process it once it's fully received.

javascript
Copy code

```
const http = require('http');
const server = http.createServer((req, res) => {
    if (req.method === 'POST' && req.url === '/submit') {
        let body = '';
        req.on('data', chunk => {
            body += chunk.toString();
        });
        req.on('end', () => {
            console.log('Received data:', body);
            res.writeHead(200, { 'Content-Type': 'text/plain' });
            res.end('Data received');
        });
    } else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('404 Not Found');
    }
});
server.listen(3000, () => {
    console.log('Server running at http://localhost:3000/');
});
```

5. Creating an HTTP Client

The http module can also be used to make HTTP requests to other servers.

javascript

Copy code

```
const http = require('http');
const options = {
    hostname: 'jsonplaceholder.typicode.com',
    port: 80,
    path: '/todos/1',
    method: 'GET',
};
const req = http.request(options, (res) => {
    let data = '';

    res.on('data', (chunk) => {
        data += chunk;
    });
    res.on('end', () => {
        console.log('Response:', JSON.parse(data));
    });
});
req.on('error', (error) => {
    console.error('Error:', error);
});
req.end();
```

- **Explanation**:
    - The http.request() method is used to make HTTP requests.
    - You can specify the hostname, port, path, and HTTP method in the options.
    - req.on('data') collects chunks of data, and req.on('end') handles the complete response.

6. Setting Headers in the Response

You can use res.setHeader() to set individual headers or res.writeHead() to set both the status code and headers.

javascript
Copy code

```
const http = require('http');
const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
```

```
        res.setHeader('X-Custom-Header', 'CustomValue');
        res.end('Headers set!');
    });
    server.listen(3000, () => {
        console.log('Server running at http://localhost:3000/');
    });
```

7. Streaming Responses

With Node.js, you can stream large responses instead of sending them all at once.

javascript
Copy code

```
    const http = require('http');
    const fs = require('fs');
    const server = http.createServer((req, res) => {
        if (req.url === '/file') {
            const stream = fs.createReadStream('largefile.txt');
            res.writeHead(200, { 'Content-Type': 'text/plain' });
            stream.pipe(res);
        } else {
            res.writeHead(404, { 'Content-Type': 'text/plain' });
            res.end('404 Not Found');
        }
    });
    server.listen(3000, () => {
        console.log('Server running at http://localhost:3000/');
    });
```

Summary

Feature	Example Methods	Description
Creating a server	http.createServer()	Sets up an HTTP server to handle requests
Handling requests	req.method, req.url	Examines request method and URL to determine responses

Sending responses	res.end(), res.writeHead(), res.setHeader()	Sends data back to the client
Streaming responses	fs.createReadStream().pipe(res)	Streams large files to clients
Making requests	http.request(), http.get()	Sends HTTP requests to external servers
Handling POST data	req.on('data'), req.on('end')	Reads and processes incoming data from POST requests

The http module is powerful for building both HTTP servers and clients, providing flexibility to build web servers, APIs, and handle web requests.