1. Basic HTTP Server:

Task: Create a simple HTTP server that responds with "Hello, World!" when the root URL (/) is accessed using the GET method.

Challenge: Implement this without any external libraries or frameworks.

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

const myServer=http.createServer((req,res)=>{
    if(req.method==='GET' && req.url=='/'){
        res.writeHead(200,{"Content-Type":"text/plain"})
        res.end('Hello world')
    }
})

const PORT=8000
myServer.listen(PORT,()=>{
    console.log(`Server started at ${PORT}`);
})
```

2.Routing:

Task: Implement basic routing. Handle three routes (/home, /about, /contact) with the GET method and return different HTML content for each route.

Challenge: Serve static HTML content from your file system for each route.

Hints:

Use the fs module to read and serve HTML files.

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

function serveFile(filePath, res) {
    fs.readFile(filePath, (error, data) => {
        if (error) {
            res.writeHead(500, { 'Content-Type': 'text/plain' });
            res.end('Server Error');
        } else {
            res.writeHead(200, { 'Content-Type': 'text/html' });
            res.end(data);
        }
    });
}
const myServer = http.createServer((req, res) => {
    if (req.method === 'GET') {
        if (req.url === '/home') {
            serveFile('home.html', res);
        } else if (req.url === '/about') {
            serveFile('about.html', res);
        } else if (req.url === '/contact') {
            serveFile('contact.html', res);
        } else {
            res.writeHead(404, { 'Content-Type': 'text/plain' });
            res.end('404 Not Found');
        }
    } else {
        res.writeHead(405, { 'Content-Type': 'text/plain' });
        res.end('404 not allowed');
    }
});
const PORT = 8000;
myServer.listen(PORT, () => {
    console.log(`Server started at ${PORT}`);
});
```

3.Handling POST Data:

Task: Create a simple form submission system. Accept form data using the POST method, parse the data, and return it as a response in JSON format.

Challenge: You'll need to manually collect and parse the POST request body since you won't use any body parsers like in Express.

Hints:Use req.on('data') and req.on('end') to collect data from the request body.

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

const server = http.createServer((req, res) => {
    if (req.method == 'POST' && req.url == '/') {
        let body = '';
        req.on('data', chunk => {
            body += chunk.toString();
        });
        req.on('end', () => {
            const parsedData = new URLSearchParams(body);
            const formData = {};
            parsedData.forEach((value, key) => {
                formData[key] = value;
            });
            res.writeHead(200, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify(formData));
        });
    } else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('404 Not Found');
    }
});

const PORT = 8001;
server.listen(PORT, () => {
    console.log('Server is running on port 8000');
});
```

4. Reading and Writing Files:

Task: Create an API that allows users to read and write data to/from a JSON file.

GET /data: Read the JSON file and return its contents.

POST /data: Write new data into the JSON file.

Hints:

Use the fs module's readFile and writeFile methods to interact with files.

Handle file reading and writing asynchronously.

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

const server = http.createServer(async (req, res) => {
    try {
        if (req.method == 'GET' && req.url == '/data') {
            const data = await fs.readFile('data.json', 'utf-8');
            res.writeHead(200, { 'Content-Type': 'application/json' });
            res.end(data);
        }
        else if (req.method == 'POST' && req.url == '/data') {

            let body = '';

            req.on('data', chunk => {
                body += chunk.toString();
            });

            req.on('end', async () => {
                if (!body) {
                    res.writeHead(400, { 'Content-Type': 'text/plain' });
                    res.end('No data');
                    return;
                }

                try {
                    const newData = JSON.parse(body);
                    const existingData = JSON.parse(await fs.readFile('data.json', 'utf-8'));
                    existingData.push(newData);
                    await fs.writeFile('data.json', JSON.stringify(existingData, 2));

                    res.writeHead(201, { 'Content-Type': 'application/json' });
                    res.end(JSON.stringify(newData));
```

```
            } catch (error) {
                console.error('Error parsing JSON:', error);
                res.writeHead(400, { 'Content-Type': 'text/plain' });
                res.end('Bad Request: Invalid JSON');
            }
        });
    }
    else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('404 Not Found');
    }
} catch (error) {
    console.error(error);
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end('500 Internal Server Error');
}
);

onst PORT = 8000;
erver.listen(PORT, () => {
  console.log(`Server is running on ${PORT}`);
);
```

5. Serve Static Files:

Task: Build a simple file server that serves static files (HTML, CSS, JS, images) based on the request URL.

For example, a request to /index.html should return the index.html file from your project folder.

Challenge: Implement content-type handling based on the file extension (HTML, CSS, JS, images).

Hints:

Use fs.readFile to serve files and path.extname to determine the file type.

Set appropriate Content-Type headers (e.g., text/html, text/css, image/jpeg).

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

const getContentType = (ext) => {
    const types = {
        '.html': 'text/html',
        '.css': 'text/css',
        '.js': 'application/javascript',
        '.jpg': 'image/jpeg',
        '.jpeg': 'image/jpeg',
    };
    return types[ext]
};

const server = http.createServer((req, res) => {
    const filePath = path.join(__dirname, req.url == '/' ? 'index.html' : req.url);

    const ext = path.extname(filePath);
    const contentType = getContentType(ext);

    fs.readFile(filePath, (err, data) => {
        if (err) {
            res.writeHead(404, { 'Content-Type': 'text/plain' });
            res.end('404 Not Found');
        }
        else {
            res.writeHead(200, { 'Content-Type': contentType });
            res.end(data);
        }
    });
});

const PORT = 8000;
server.listen(PORT, () => {
    console.log(`Server is running on ${PORT}`);
});
```

6. Create a Basic API with CRUD Operations:

Task: Create a RESTful API for managing a simple resource like "books" (or any other entity).

GET /books: Return a list of books.

POST /books: Add a new book.

PUT /books/:id: Update an existing book.

DELETE /books/:id: Delete a book.

Challenge: Store the books in a JSON file and perform CRUD operations on that file. You can also use an in-memory array if file storage seems too complex.

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

const filePath = path.join(__dirname, 'books.json');

const readBook = () => {
    return new Promise((resolve, reject) => {
        fs.readFile(filePath, 'utf-8', (error, data) => {
            if (error) {
                reject(error);
            }
            else {
                resolve(JSON.parse(data || '[]'));
            }
        });
    });
};

const writeBook = (books) => {
    return new Promise((resolve, reject) => {
        fs.writeFile(filePath, JSON.stringify(books, 2), (error) => {
            if (error) {
                reject(error);
            }
            else {
                resolve();
            }
        });
    });
};

const server = http.createServer(async (req, res) => {
    try {
        const urlParts = req.url.split('/');
        const id = urlParts[2];
        let books = await readBook();
```

```
        if (req.method === 'GET' && req.url === '/books') {
            res.writeHead(200, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify(books));
        }
        else if (req.method === 'POST' && req.url === '/books') {
            let body = '';
            req.on('data', chunk => {
                body += chunk.toString();
            });

            req.on('end', async () => {
                const newBook = JSON.parse(body);
                books.push(newBook);
                await writeBook(books);
                res.writeHead(201, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify(newBook));
            });
        }
        else if (req.method === 'PUT' && req.url.startsWith('/books/')) {
            let body = '';
            req.on('data', chunk => {
                body += chunk.toString();
            });

            req.on('end', async () => {
                const updatedBook = JSON.parse(body);
                const index = books.findIndex(book => book.id == updatedBook.id);

                if (index !== -1) {
                    books[index] = { ...books[index], ...updatedBook };
                    await writeBook(books);
                    res.writeHead(200, { 'Content-Type': 'application/json' });
                    res.end(JSON.stringify(books[index]));
                }
```

```
                else {
                    res.writeHead(404, { 'Content-Type': 'text/plain' });
                    res.end('404 Not Found');
                }
            });
        }
        else if (req.method === 'DELETE' && req.url.startsWith('/books/')) {

            const filteredBooks = books.filter(book => book.id != id);

            if (books.length !== filteredBooks.length) {
                await writeBook(filteredBooks);
                res.writeHead(200, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify({ message: 'Book deleted' }));
            }
            else {
                res.writeHead(404, { 'Content-Type': 'text/plain' });
                res.end('Book not found');
            }
        }
        else {
            res.writeHead(404, { 'Content-Type': 'text/plain' });
            res.end('404 Not Found');
        }
    } catch (error) {
        console.error(error);
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end('Server Error');
    }
});

const PORT = 8000;
server.listen(PORT, () => {
    console.log(`Server is running on ${PORT}`);
});
```

7. Logging Middleware:

Task: Create a simple logging middleware that logs the request method and URL to the console every time a request is made.

Challenge: Implement this manually without any frameworks.

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

function logRequest(req, res, next) {
    console.log(`${req.method} ${req.url}`);
    next();
}

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

const PORT = 8000;
server.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

8. Simple Authentication System:

Task: Create a basic login system using Node.js. Accept a username and password via POST request and verify them against a predefined list of users.

Challenge: Implement basic security practices like hashing passwords (use crypto for hashing).

```
const http = require('http');
const fs = require('fs');
const crypto = require('crypto');
const path = require('path');

const filePath = path.join(__dirname, 'user.json');

const readUsers = () => {
    return new Promise((resolve, reject) => {
        fs.readFile(filePath, 'utf-8', (error, data) => {
            if (error) {
                reject(error);
            } else {
                resolve(JSON.parse(data || '[]'));
            }
        });
    });
};

const writeUsers = (users) => {
    return new Promise((resolve, reject) => {
        fs.writeFile(filePath, JSON.stringify(users,  2), (error) => {
            if (error) {
                reject(error);
            } else {
                resolve();
            }
        });
    });
};

const hashPassword = (password) => {
    return crypto.createHash('sha256').update(password).digest('hex');
};
```

```
const server = http.createServer(async (req, res) => {
    if (req.method === 'POST' && req.url === '/register') {
        let body = '';

        req.on('data', chunk => {
            body += chunk.toString();
        });

        req.on('end', async () => {
            try {
                const { username, password } = JSON.parse(body);

                const users = await readUsers();

                const existingUser = users.find(euser => euser.username === username);

                if (existingUser) {
                    res.writeHead(409, { 'Content-Type': 'application/json' });
                    res.end(JSON.stringify({ message: 'Username already exists' }));
                    return;
                }

                const hashedPassword = hashPassword(password);
                const newUser = { username, password: hashedPassword };

                users.push(newUser);

                await writeUsers(users);

                res.writeHead(201, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify({ message: 'User registered successfully' }));
            } catch (error) {
                res.writeHead(500, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify({ message: 'Server error' }));
            }
        });
```

```
} else if (req.method === 'POST' && req.url === '/login') {
    let body = '';

    req.on('data', chunk => {
        body += chunk.toString();
    });

    req.on('end', async () => {
        try {
            const { username, password } = JSON.parse(body);

            const users = await readUsers();

            const user = users.find(u => u.username === username);

            if (!user) {
                res.writeHead(401, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify({ message: 'Invalid username or password' }));
                return;
            }

            const hashedPassword = hashPassword(password);

            if (hashedPassword === user.password) {
                res.writeHead(200, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify({ message: 'Login successful' }));
            }
                else {
                res.writeHead(401, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify({ message: 'Invalid username or password' }));
            }
        } catch (error) {
            res.writeHead(500, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify({ message: 'Server error' }));
        }
```

```
            } catch (error) {
                res.writeHead(500, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify({ message: 'Server error' }));
            }
        });
    } else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('404 Not Found');
    }
});
const PORT = 8000;
server.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

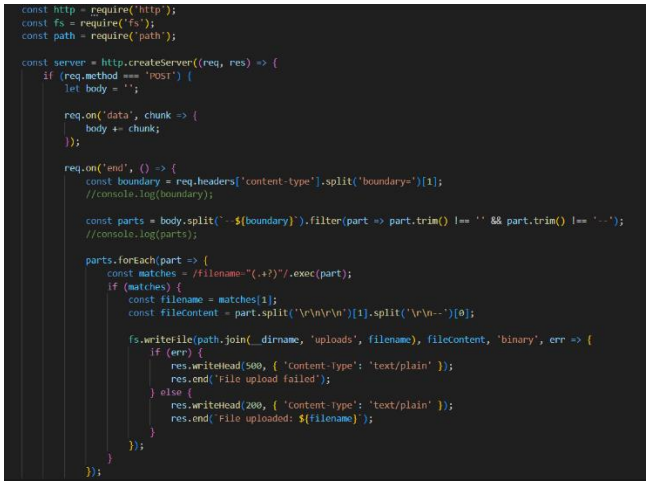
9. File Upload Handling:

Task: Implement a simple file upload system where users can upload files via a POST request. Save the uploaded file to a specific folder on your server.

Challenge: Manually parse the incoming multipart/form-data request and store the file.

Hints:

Use the fs module for saving files and manually handle multipart form data.



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

const server = http.createServer((req, res) => {
    if (req.method === 'POST') {
        let body = '';

        req.on('data', chunk => {
            body += chunk;
        });

        req.on('end', () => {
            const boundary = req.headers['content-type'].split('boundary=')[1];
            //console.log(boundary);

            const parts = body.split(`--${boundary}`).filter(part => part.trim() !== '' && part.trim() !== '--');
            //console.log(parts);

            parts.forEach(part => {
                const matches = /filename="(.+?)"/.exec(part);
                if (matches) {
                    const filename = matches[1];
                    const fileContent = part.split('\r\n\r\n')[1].split('\r\n--')[0];

                    fs.writeFile(path.join(__dirname, 'uploads', filename), fileContent, 'binary', err => {
                        if (err) {
                            res.writeHead(500, { 'Content-Type': 'text/plain' });
                            res.end('File upload failed');
                        } else {
                            res.writeHead(200, { 'Content-Type': 'text/plain' });
                            res.end(`File uploaded: ${filename}`);
                        }
                    });
                }
            });
        });
    } else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('404 Not Found');
    }
});

const PORT=8000
server.listen(PORT, () => {
    console.log('Server listening on port 8000');
});
```

10. Rate Limiting:

Task: Implement a simple rate-limiting system for your API to restrict how often a client can make requests.

Challenge: Track client IP addresses and limit the number of requests they can make to certain routes within a time window (e.g., 10 requests per minute).

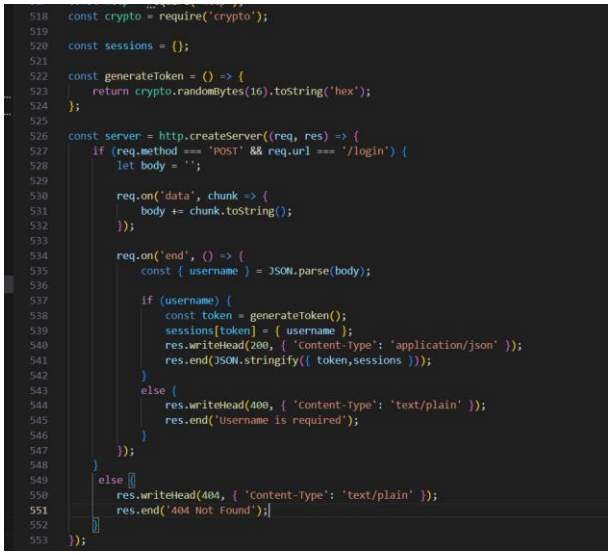
```
const http = require('http');
const rateLimit = require('./ratelimit');

const maxRequests = 10;
const timeWindow = 60 * 1000;

const server = http.createServer((req, res) => {
    if (req.url === '/api') {
        rateLimit(maxRequests, timeWindow)(req, res, () => {
            res.writeHead(200, { 'Content-Type': 'text/plain' });
            res.end('Request received successfully!');
        });
    }
    else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('404 Not Found');
    }
});
const PORT = 8000;
server.listen(PORT, () => {
    console.log(`Server running on ${PORT}`);
});
```

```
const rateLimit = (maxRequests, timeWindow) => {
    const requestCounts = new Map();

    return (req, res, next) => {
        const ip = req.socket.remoteAddress;
        const currentTime = Date.now();

        if (!requestCounts.has(ip)) {
            requestCounts.set(ip, { count: 0, lastRequest: currentTime });
        }

        const requestData = requestCounts.get(ip);

        if (currentTime - requestData.lastRequest > timeWindow) {
            requestData.count = 0;
            requestData.lastRequest = currentTime;
        }

        requestData.count++;
        if (requestData.count > maxRequests) {
            return res.writeHead(429, { 'Content-Type': 'text/plain' })
                      .end('Too many requests, please try again later.');
        }

        next();
    };
};

module.exports = rateLimit;
```

11. Session Management:

Task: Implement a simple session management system where users are assigned session tokens upon logging in, and these tokens are stored in memory.

Challenge: You cannot use any third-party libraries like express-session. Implement token generation using Node's crypto module.



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

const sessions = {};

const generateToken = () => {
    return crypto.randomBytes(16).toString('hex');
};

const server = http.createServer((req, res) => {
    if (req.method === 'POST' && req.url === '/login') {
        let body = '';

        req.on('data', chunk => {
            body += chunk.toString();
        });

        req.on('end', () => {
            const { username } = JSON.parse(body);

            if (username) {
                const token = generateToken();
                sessions[token] = { username };
                res.writeHead(200, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify({ token,sessions }));
            }
            else {
                res.writeHead(400, { 'Content-Type': 'text/plain' });
                res.end('Username is required');
            }
        });
    }
    else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('404 Not Found');
    }
});
```

12. WebSocket Communication:

Task: Implement a simple real-time chat application using WebSockets (without using any WebSocket libraries).

Challenge: Use the native http and net modules to create a WebSocket server for real-time messaging between clients.

Hints:

Implement the WebSocket handshake manually by handling the Upgrade HTTP header.

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

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('WebSocket Server is running');
});

const wss = new WebSocket.Server({ server });

wss.on('connection', (ws) => {
  console.log('A new client connected!');

  ws.on('message', (message) => {
    console.log(`Received message => ${message}`);
    wss.clients.forEach((client) => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(message);
      }
    });
  });

  ws.on('close', () => {
    console.log('A client disconnected');
  });
});

const PORT=8000
server.listen(PORT, () => {
  console.log('Server is listening on port 8000');
});

```

13. Handling File Streams:

Task: Create an API that serves large files (e.g., videos or large text files) using streams.

Allow users to download a file via streaming instead of reading the whole file into memory.

Challenge: Efficiently stream data using fs.createReadStream().

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

const server = http.createServer((req, res) => {
    const filePath = path.join(__dirname, 'files', req.url);

    fs.stat(filePath, (err, stats) => {
        if (err) {
            res.writeHead(404, { 'Content-Type': 'text/plain' });
            res.end('File not found');
            return;
        }

        res.writeHead(200, {
            'Content-Type': 'application/octet-stream',
            'Content-Length': stats.size,
            'Content-Disposition': `attachment; filename="${path.basename(filePath)}"`,
        });
                                            const filePath: string

        const readStream = fs.createReadStream(filePath);
        readStream.pipe(res);

        readStream.on('error', (error) => {
            res.writeHead(500, { 'Content-Type': 'text/plain' });
            res.end('Error reading file');
        });
    });
});

const PORT = 3000;
server.listen(PORT, () => {
    console.log(`Server is running on ${PORT}`);
});
```

14. Event-Driven System:

Task: Implement a custom event-driven system using Node.js's EventEmitter class. Create an event-based notification system where different parts of the application can emit and listen to custom events.

Hints:

Use the events module and EventEmitter to trigger and respond to events.

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

const emitter = new EventEmitter();

emitter.on('registration', (user) => {
    console.log(`Welcome, ${user}!`);
});

emitter.on('loggedIn', (user) => {
    console.log(`${user.name} has logged in.`);
});

function registerUser(user) {
    console.log('Registering user...');
    emitter.emit('registration', user);
}

function loginUser(user) {
    console.log('Logging in user...');
    emitter.emit('loggedIn', user);
}

registerUser('Navya');
loginUser({name:'Navya'});
```

15. File Compression API:

Task: Implement an API that accepts a file via a POST request, compresses it (e.g., into a .zip file), and allows the client to download the compressed file.

Challenge: Use the native zlib module to compress files.

16. Handling Environment Variables:

Task: Implement an API where the database URL and other sensitive information (e.g., API keys) are securely stored in environment variables.

Challenge: Use process.env to load environment variables, and create a .env file for easy management.

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

const envFile = fs.readFileSync('.env', 'utf-8');

envFile.split('/n').forEach(i => {
    const [key, value] = i.split('=');
    process.env[key] = value;
});

const apiKey = process.env.API_KEY;

console.log('API Key:', apiKey);
```

17. Creating Your Own Simple CLI Tool:

Task: Create a simple Node.js command-line interface (CLI) tool. For example, a tool that takes input and outputs the current weather of a given city using an external API (like OpenWeather).

Challenge: Parse command-line arguments using process.argv and handle different commands and options.

18. Simple JSON Web Token (JWT) Authentication:

Task: Implement JWT authentication without using any third-party libraries like jsonwebtoken.

Challenge: Use the native crypto module to generate and verify JWTs manually.

```
const crypto=require('crypto')

function generate(data){
    return Buffer.from(JSON.stringify(data)).toString('base64');
}
function createJWT(payload, secret) {

    const header = { alg: 'HS256', typ: 'JWT' };
    const encodedHeader = generate(header);
    const encodedPayload = generate(payload);

    const signature = crypto
        .createHmac('sha256', secret)
        .update(`${encodedHeader}.${encodedPayload}`)
        .digest('base64');

    return `${encodedHeader}.${encodedPayload}.${signature}`;
}

const secret = 'mysecret';
const payload = { userId: 1, username: 'navya' };

const token = createJWT(payload, secret);
console.log('Generated Token:', token);
```