Server Side Scalable Application

Unit-2

Call Stack

The **call stack** is a data structure that keeps track of function calls. When a function is invoked, it's **pushed** onto the stack, and when it finishes execution, it's **popped** off.

Example:

```
function greet() {
  console.log("Hello!");
}

function sayName(name) {
  console.log("Name:", name);
}

function main() {
  greet();          // pushed
  sayName("Chirag"); // pushed
}

main(); // pushed
```

Call Stack Execution:

main() → greet() → sayName() → end

Callback Function

A **callback** is a function passed as an argument to another function, usually executed **after** some asynchronous operation completes.

Example (Simple Callback):

```
function fetchData(callback) {
  setTimeout(() => {
    console.log("Data fetched");
    callback();
  }, 2000);
}


fetchData(() => {
  console.log("Callback executed");
});
```

Callback Queue (Task Queue)

The **callback queue** holds messages or callbacks (like setTimeout, setInterval, HTTP request handlers) waiting to be pushed onto the **call stack**. It follows **FIFO (First In, First Out)**.

Event Loop

The **event loop** constantly checks if the call stack is empty. If it is, and the callback queue has any functions, it **pushes them to the call stack** for execution.

How it works:

1. Call Stack empty?
2. Is anything in Callback Queue?
3. Move next task from Queue to Stack
4. Repeat

```
});
```

Execution Flow Example:

```
console.log("Start");


setTimeout(() => {
  console.log("Inside setTimeout");
}, 1000);


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

Output:

```
Start
End
Inside setTimeout
```

Even though setTimeout is written before "End", its callback is pushed to the **callback queue** and executed **after** the current call stack is empty.

2.1.2 Callbacks in Node.js

Node.js heavily uses callbacks for handling asynchronous operations like file I/O, HTTP, databases, etc.

Example 1: Reading a File (Async callback)

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


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

Breakdown:

- fs.readFile is non-blocking
- The callback runs **after file is read** (via event loop)
- Meanwhile, the program can do other tasks

Example 2: Nested Callbacks (Callback Hell)

```
fs.readFile('file1.txt', 'utf8', (err, data1) => {
  fs.readFile('file2.txt', 'utf8', (err, data2) => {
    fs.readFile('file3.txt', 'utf8', (err, data3) => {
      console.log(data1, data2, data3);
    });
  });
});
```

This is called **"Callback Hell"**, often solved using **Promises** or **Async/Await**.

A **Promise** in JavaScript is a modern way to handle asynchronous operations. It provides a cleaner and more structured approach compared to traditional callbacks. In Node.js, promises help avoid issues like **callback hell** and allow chaining operations in a logical sequence. Promises represent the eventual completion (or failure) of an asynchronous operation and its resulting value.

A promise has three states:

1. **Pending** – The operation is still in progress.
2. **Fulfilled** – The operation completed successfully.
3. **Rejected** – The operation failed.

Syntax:

```
const promise = new Promise((resolve, reject) => {
  // Async task here
  if (/* success */) {
    resolve("Success value");
  } else {
    reject("Error message");
  }
});
```

Example:

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      if (success) {
        resolve("Data fetched successfully");
      } else {
        reject("Failed to fetch data");
      }
    }, 1000);
  });
};

fetchData()
  .then(data => {
    console.log(data); // "Data fetched successfully"
  })
  .catch(error => {
    console.error(error); // "Failed to fetch data"
  });
```

Why Use Promises?

- **Improved readability**
- **Error handling with .catch()**
- **Avoid callback nesting**
- **Support chaining**
- **Integration with async/await**

Use Cases in Node.js:

- File operations (fs.promises)
- Network requests
- Database queries
- Any async operation like timers, APIs, etc.

Converting Callback to Promise:

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


const readFilePromise = (filename) => {
  return new Promise((resolve, reject) => {
    fs.readFile(filename, 'utf8', (err, data) => {
      if (err) reject(err);
      else resolve(data);
    });
  });
};
```

```
});

readFilePromise('example.txt')
  .then(data => console.log("File Data:", data))
  .catch(err => console.error("Error:", err));
```

Best Practices:

- Always return promises from functions.
- Handle both .then() and .catch().
- Avoid mixing callbacks and promises.
- Use async/await for better readability when needed.

2.2.1 Promise Chaining

What is Promise Chaining?

Promise chaining is the process of linking multiple .then() calls in sequence. Each .then() gets the result from the previous one and passes its result to the next. It allows complex asynchronous flows to be expressed linearly and clearly.

Basic Syntax:

```
doTask()
  .then(result => doNextTask(result))
  .then(result2 => doAnotherTask(result2))
  .then(finalResult => console.log("Final Result:", finalResult))
  .catch(err => console.error("Error:", err));
```

Example:

```
const firstTask = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve("Step 1 Complete"), 1000);
  });
```

```
};

const secondTask = (msg) => {
  return new Promise(resolve => {
    setTimeout(() => resolve(msg + " -> Step 2 Complete"), 1000);
  });
};

const thirdTask = (msg) => {
  return new Promise(resolve => {
    setTimeout(() => resolve(msg + " -> Step 3 Complete"), 1000);
  });
};

firstTask()
  .then(result => {
    console.log(result);
    return secondTask(result);
  })
  .then(result2 => {
    console.log(result2);
    return thirdTask(result2);
  })
  .then(result3 => {
    console.log(result3);
  })
  .catch(err => {
    console.error("Error occurred:", err);
  });
```

Benefits:

- Clean and linear async logic
- Easier to debug and read
- Handles multiple steps in one flow

Common Mistakes:

- Forgetting to return inside .then()
- Handling errors only at the last step (misses earlier failures)

Real Use Case:

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


fs.readFile('file1.txt', 'utf8')
  .then(data => {
    console.log("File 1:", data);
    return fs.readFile('file2.txt', 'utf8');
  })
  .then(data2 => {
    console.log("File 2:", data2);
    return fs.readFile('file3.txt', 'utf8');
  })
  .then(data3 => {
    console.log("File 3:", data3);
```

```
 })
 .catch(err => {
   console.error("Error reading file:", err);
 });
```

Summary:

Promise chaining allows you to perform a sequence of asynchronous operations, each dependent on the result of the previous one. It’s an effective alternative to nested callbacks and prepares you for working with async/await.

2.2.2 Request Package in Node.js

Introduction

The **request** package was one of the most popular libraries in Node.js for making HTTP requests. Although it's now deprecated, it's still widely used in legacy systems. Alternatives today include axios, node-fetch, and native https.

Despite being deprecated, it's still important to understand because:

- Many older codebases use it.
- Learning it teaches the basics of HTTP requests.

Installation:

```
npm install request
```

Making a GET Request:

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


request('https://jsonplaceholder.typicode.com/posts/1', (error, response, body) => {
 if (!error && response.statusCode == 200) {
   const data = JSON.parse(body);
   console.log(data);
 } else {
   console.error("Error:", error);
 }
});
```

Making a POST Request:

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

const data = {
  title: 'foo',
  body: 'bar',
  userId: 1
};

request.post({
  url: 'https://jsonplaceholder.typicode.com/posts',
  json: data
}, (err, res, body) => {
  if (err) {
    console.error(err);
  } else {
    console.log(body);
  }
});
```

Features:

- Supports all HTTP methods (GET, POST, PUT, DELETE)
- Can send form data and JSON easily

- Handles redirects, cookies, headers, etc.

Limitations:

- **Deprecated**: Not maintained anymore
- **No native Promise support**
- **No built-in retry mechanism**
- Heavy and verbose compared to axios or fetch

Modern Alternative: axios

```
npm install axios


const axios = require('axios');


axios.get('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => {
    console.log(response.data);
  })
  .catch(err => {
    console.error(err);
  });
```

Real Use Case:

In a web app or CLI, you might use request to:

- Fetch weather data
- Call third-party APIs (like payment gateways)
- Scrape web content
- Submit forms or login automatically

Summary:

- request is a simple HTTP client used for years in Node.js.
- It works using callbacks and supports most HTTP features.
- It's deprecated; better options include axios or node-fetch.
- Knowing request helps understand how older projects and callbacks work with HTTP.

2.2.3 Customizing HTTP Requests and Error Handling with HTTP Status Codes

Introduction

In Node.js, making HTTP requests to external APIs or services is a common task. While packages like axios, request, or node-fetch are typically used for these tasks, customizing those requests with headers, body data, authentication, and handling various HTTP status codes is critical for building robust applications.

Also, when **creating your own Node.js server**, it's equally important to understand how to **send appropriate HTTP status codes** and **handle errors** systematically.

What are HTTP Status Codes?

HTTP status codes are 3-digit numbers sent by the server to indicate the result of a request.

Code Range	**Meaning**
1xx	Informational
2xx	Success
3xx	Redirection
4xx	Client Errors
5xx	Server Errors

Common Examples:

- 200 OK: Request successful
- 201 Created: Resource created

- 400 Bad Request: Client sent invalid data
- 401 Unauthorized: Authentication failed
- 404 Not Found: Resource not found
- 500 Internal Server Error: Server crashed or bug

Making a Custom HTTP Request with Headers (Using axios)

```
npm install axios



const axios = require('axios');


axios.get('https://jsonplaceholder.typicode.com/posts/1', {
  headers: {
    'Accept': 'application/json',
    'Authorization': 'Bearer dummy_token'
  }
})
.then(response => {
  console.log("Data:", response.data);
  console.log("Status Code:", response.status);
})
.catch(error => {
  if (error.response) {
    console.log("Error Status:", error.response.status);
    console.log("Message:", error.response.statusText);
  } else {
    console.log("Error:", error.message);
  }
});
```

Creating a Node.js HTTP Server with Custom Error Handling

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


const server = http.createServer((req, res) => {
  if (req.url === "/") {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end("Welcome to the homepage!");
  } else if (req.url === "/about") {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end("About us page");
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end("404 - Page Not Found");
  }
});


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

2.2.4 Introduction to Template Engine (EJS)

What is EJS?

EJS (Embedded JavaScript Templating) is a template engine for Node.js that allows you to write HTML pages that can dynamically include data from the backend. It combines HTML with JavaScript logic like loops and conditionals.

It is commonly used with the **Express.js** web framework to render dynamic HTML pages.

� Installation:

```
npm install express ejs
```

Folder Structure:

```
project/
├── views/
│   ├── index.ejs
├── app.js
```

Setting Up EJS with Express

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

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

app.get('/', (req, res) => {
  const user = { name: "Chirag", age: 30 };
  res.render('index', { user });
});
```

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

---

views/index.ejs:

```
<!DOCTYPE html>
<html>
<head>
  <title>EJS Example</title>
</head>
<body>
  <h1>Welcome <%= user.name %></h1>
  <p>You are <%= user.age %> years old.</p>
</body>
</html>
```

Output on browser:

```
Welcome Chirag
You are 30 years old.
```

EJS Syntax

Purpose	**Syntax**
Output value	<%= variable %>
Logic block	<% if (condition) { %>
Looping	<% items.forEach(...) %>

Loop Example:

```
<ul>
  <% fruits.forEach(fruit => { %>
    <li><%= fruit %></li>
  <% }) %>
</ul>
```

And in Express:

```
app.get('/fruits', (req, res) => {
  const fruits = ['Apple', 'Banana', 'Cherry'];
  res.render('index', { fruits });
});
```

Why Use EJS?

- Works directly with Express
- Allows for clean separation of logic and view
- Easy to integrate partials like header, footer
- No need for browser-side templating frameworks in many cases