Event Loop

- The event loop is a fundamental concept in Node.js that enables asynchronous, non-blocking I/O operations.
- It's essentially a mechanism that allows Node.js to perform multiple operations concurrently without getting blocked, making it highly efficient for handling I/O-bound tasks.

How the event loop works in Node.js

- **Single-threaded Nature:** Node.js is single-threaded, meaning it operates using a single main thread to execute JavaScript code. This single thread is responsible for handling all incoming requests, executing code, and responding to events.
- **Event-Driven Architecture:** Node.js utilizes an event-driven architecture. Instead of waiting for operations like file I/O, network requests, or database queries to complete synchronously, Node.js delegates these tasks to the operating system and registers callback functions to be executed once the tasks are complete.
- **Event Queue:** Asynchronous operations in Node.js produce events once they are complete. These events are queued in an event queue. The event loop continuously monitors this queue for events and processes them sequentially.
- **Non-Blocking I/O:** While the event loop is processing events, the main thread remains free to handle other tasks. This non-blocking nature allows Node.js to efficiently handle many concurrent connections and I/O operations without getting blocked by long-running tasks.
- **Event Loop Phases:** The event loop in Node.js operates in a series of phases, including timers, pending callbacks, idle/preparation, polling, check, and close callbacks. Each phase has its specific purpose and determines which tasks should be executed next.
- **Timers and Callbacks:** Node.js provides functions like setTimeout() and setInterval() for scheduling code execution after a certain delay. When the specified time elapses, these functions push callback functions into the event queue, where they wait to be processed by the event loop.
- **Concurrency Model:** Node.js achieves concurrency by efficiently switching between different tasks during the event loop. It can handle a large number of concurrent connections and operations without spawning a new thread for each connection, which is more memory-efficient compared to traditional multi-threaded models.

Example : Event Loop Demonstration

App.js

```
// Function to simulate an asynchronous operation
const asyncOperation = (callback) => {
  setTimeout(() => {
    callback('Async operation completed');
  }, 4000); // Simulating a delay of 4 seconds
 };

 // Function to handle the completion of the asynchronous operation
 const handleAsyncOpCompletion = (message) => {
   console.log("status : "+message);
 };

 // Initiate the asynchronous operation
 console.log('Start of script');
 asyncOperation(handleAsyncOpCompletion);
 console.log('End of script');
```

Example : Event Loop Demonstration using setTimeOut() & setImmediate()

App.js

```
// Simulating asynchronous operations with setTimeout
console.log('Start of script');

setTimeout(() => {
  console.log('setTimeout first');
}, 2000);

setTimeout(() => {
  console.log('setTimeout second');
}, 2000);

setImmediate(() => {
  console.log('setImmediate first');
});

setImmediate(() => {
  console.log('setImmediate second');
});

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

The **setTimeout()** callbacks are scheduled with a timeout of 2000 milliseconds, they are pushed to the event queue after the other script statements have been executed.

The **setImmediate()** callbacks are executed immediately in the next iteration of the event loop, after the current one completes.

Example : How the event loop works via involving I/O operations

App.js

```
const fs = require('fs');
console.log('Start of script');
// Asynchronous file read operation
fs.readFile('data.txt', (err, fileData) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  console.log('Data : ', fileData.toString());
});
console.log('End of script');
```

Example : Event loop in Node.js with a complex asynchronous operation

App.js

```
console.log('Start of the script');

// Asynchronous function simulating database query
function studentRecord(callback) {
  setTimeout(() => {
    const data = { id: 17, name: 'Surbhi Gulati', age: 27 };
    callback(null, data);
  }, 2000);
}

// Asynchronous function simulating HTTP request
function sendHttpRequest(callback) {
  setTimeout(() => {
    const response = { status: 200, message: 'Success' };
    callback(null, response);
  }, 1000);
}
```

```
// Fetch data from the database
studentRecord((err, data) => {
  if (err) {
    console.error('Error in fetching student Record:', err);
    return;
  }
  console.log('Student Details :', data);
});

// Send HTTP request
sendHttpRequest((err, response) => {
  if (err) {
    console.error('Error sending HTTP request:', err);
    return;
  }
  console.log('HTTP response details:', response);
});

console.log('End of the script');
```

Points to remember about Event Loop

- The event loop in Node.js is a crucial mechanism that enables asynchronous programming.
- It allows Node.js to efficiently handle I/O operations without blocking the main thread, making it ideal for building high-performance and scalable applications.
- By leveraging the event loop, developers can write non-blocking code that can handle multiple concurrent tasks, such as network requests, file operations, and database queries, efficiently.
- Overall, the event loop is fundamental to Node.js's asynchronous, event-driven architecture, enabling it to handle high concurrency and deliver responsive, performaning applications.

HTTP Verbs

In Node.js, when creating HTTP servers or making HTTP requests, you'll encounter various HTTP methods or verbs. These methods define the action to be performed on the specified resource.

HTTP Methods Overview: HTTP (Hypertext Transfer Protocol) methods, also known as HTTP verbs, define the actions that clients (such as web browsers) can perform on resources identified by a URL. Each HTTP method has a specific purpose and semantic meaning:

- **GET:** Used to request data from a specified resource. It should only retrieve data and should not have any other effect.
- **POST:** Used to submit data to be processed to a specified resource. It often results in a change in state or side effects on the server, such as creating a new resource or updating existing data.
- **PUT:** Similar to POST, but used to replace all current representations of the target resource with the request payload.
- **DELETE:** Used to delete the specified resource.
- **PATCH:** Used to apply partial modifications to a resource.
- **HEAD:** Similar to GET, but requests the headers only, without the response body.
- **OPTIONS:** Used to describe the communication options for the target resource.

Handling HTTP Methods in Node.js

Server-Side Programming: When creating an HTTP server in Node.js using the built-in http module, you define how your server responds to incoming HTTP requests, including different HTTP methods. This is typically done within the callback function passed to http.createServer().

For example:

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

const server = http.createServer((req, res) => {
   // Determine the HTTP method
   switch(req.method) {
      case 'GET':
         // Handle GET request
         break;
      case 'POST':
         // Handle POST request
         break;
      case 'PUT':
         // Handle PUT request
         break;
      case 'DELETE':
         // Handle DELETE request
         break;
      default:
         res.writeHead(405); // Method Not Allowed
         res.end();
         break;
   }
});
server.listen(3000, () => {
   console.log('Server is running on port 3000');
});
```

In the above example, the server responds differently depending on the HTTP method of the incoming request.

Making HTTP Requests: When making HTTP requests from a Node.js application, you specify the HTTP method using libraries such as http, axios, or node-fetch. These libraries provide APIs to perform HTTP requests with various methods.

Using http Module:

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

const options = {
  hostname: 'www.example.com',
  port: 80,
  path: '/resource',
  method: 'GET' // Specify the HTTP method here
};

const req = http.request(options, (res) => {
  // Handle response
});

req.end();
```

Using axios:

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

axios.post('https://www.example.com/resource', { data: 'some data' })
  .then((response) => {
    // Handle response
  })
  .catch((error) => {
    // Handle error
  });
```

In both cases, you specify the HTTP method (GET, POST, etc.) along with other request parameters such as URL, headers, and payload.

Example : handle HTTP methods in a Node.js server using the Express.js framework

Server.js

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

// Middleware to log incoming requests
app.use((req, res, next) => {
   console.log(`${req.method} request received for ${req.url}`);
   next();
});

// Route for GET request
app.get('/resource', (req, res) => {
   res.send('GET request received');
});

// Route for POST request
app.post('/resource', (req, res) => {
   res.send('POST request received');
});

// Route for PUT request
app.put('/resource', (req, res) => {
   res.send('PUT request received');
});

// Route for DELETE request
app.delete('/resource', (req, res) => {
   res.send('DELETE request received');
});

// Start the server
app.listen(PORT, () => {
   console.log(`Server is running on port ${PORT}`);
});
```

Example : handle HTTP methods in a Node.js server using the http module directly, without any additional frameworks.

Server.js

```
const http = require('http');
const PORT = 3000;
// Create a server
const server = http.createServer((req, res) => {
   // Log the incoming request method and URL
   console.log(`${req.method} request received for ${req.url}`);
   // Set response headers
   res.writeHead(200, { 'Content-Type': 'text/plain' });

   // Handle different HTTP methods
   switch (req.method) {
        case 'GET':
                 res.end('Data Received from GET request');
                 break;
              case 'POST':
                 res.end('Data Received from POST request');
                 break;
              case 'PUT':
                 res.end('Data Received from PUT request');
                 break;
              case 'DELETE':
                 res.end('Data Received from DELETE request');
                 break;
              default:
                 res.writeHead(405); // unsupported / wrong HTTP method
                 res.end('Unsupported HTTP method');
                 break;
   }});

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

Example: handle different HTTP methods in a server and how to make HTTP requests using different methods.

Server.js

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

const server = http.createServer((req, res) => {
   // see the log of incoming request method and URL
   console.log(`${req.method} request received for ${req.url}`);

   // set response headers
   res.writeHead(200, { 'Content-Type': 'text/plain' });

   // handle different HTTP methods
   switch(req.method) {
      case 'GET':
         res.end('Data Received from GET request');
         break;
      case 'POST':
         res.end('Data Received from POST request');
         break;
      case 'PUT':
         res.end('Data Received from PUT request');
         break;
      case 'DELETE':
         res.end('Data Received from DELETE request');
         break;
      default:
         res.writeHead(405); // unsupported / wrong HTTP method
         res.end('Unsupported HTTP method');
         break;
   }});

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

Client.js

```
const http = require('http');
// Defining the request
const requestOptions = {
   hostname: 'localhost',
   port: 3000,
   path: '/httpverbs',
   method: 'POST' // Specify the type of HTTP method here
};
// to make the HTTP request
const req = http.request(requestOptions, (res) => {
   let responseData = '';
   // to receive the response data
   res.on('data', (chunk) => {
      responseData += chunk;
   });
   // Process the complete response
   res.on('end', () => {
      console.log(`Response received: ${responseData}`);
   });
});
// Handle errors
req.on('error', (error) => {
   console.error(`Request failed: ${error.message}`);
});
// Send the request
req.end();
```

Steps to execute the above example

- Save both server.js and client.js files in the same directory.
- Open a terminal and navigate to the directory containing the files.
- Run the server script: node server.js.
- Open another terminal window/tab and run the client script: node client.js.

You'll see output in the server terminal indicating requests being received and handled, and in the client terminal showing the responses received from the server.