Node.JS

1. API Creation?

2. **Middleware Creation?**

Middleware in Node.js are functions that are executed in the middle of the request-response cycle. They can perform tasks like logging, authentication, and error handling. Creating middleware in Node.js is simple and straightforward. Here are the steps:

1. Create a new file for your middleware. For example, `myMiddleware.js`.

2. In this file, define a function that takes three arguments: `request`, `response`, and `next`. The `next` function is a callback function that tells Node.js to move on to the next middleware in the stack.

```javascript

function myMiddleware(request, response, next) {

// Middleware code goes here

next();

}

```

3. Add your middleware function to the Express application using the `use()` method. This method adds middleware to the middleware stack.

```javascript

const express = require('express');

const app = express();

app.use(myMiddleware);

```

4. If you have multiple middleware functions, add them in the order you want them to execute.

```javascript

app.use(myMiddleware1);

app.use(myMiddleware2);

app.use(myMiddleware3);

```

That's it! Your middleware functions will now execute in the order you defined them. If a middleware function calls `next()` without any arguments, the next middleware in the stack will be executed. If a middleware function calls `next(err)`, the error handling middleware will be executed.

Here's an example of a middleware function that logs the request method and URL:

```javascript

function logRequest(request, response, next) {

console.log(`${request.method} ${request.url}`);

next();

}

app.use(logRequest);

```

This middleware function will log every incoming request to the console. You can add more middleware functions to perform other tasks like authentication or error handling.

3. **Global in node Js?**

In Node.js, global refers to the global object that is available throughout the application. It is similar to the `window` object in browsers. The global object provides a set of properties and methods that can be accessed from any part of the application.

Here are some examples of properties and methods available on the global object in Node.js:

- `console`: This property provides methods for writing to the console, such as `console.log()` and `console.error()`.

- `process`: This property provides information about the current Node.js process, such as `process.argv` for command-line arguments and `process.env` for environment variables.

- `Buffer`: This property provides methods for working with binary data, such as creating new buffers and converting between different encodings.

- `setTimeout()`, `clearTimeout()`, `setInterval()`, `clearInterval()`: These methods provide a way to execute code asynchronously after a certain amount of time.

- `require()`: This method is used to load modules in Node.js.

While the global object can be useful in some cases, it is generally not recommended to use it extensively. This is because polluting the global namespace with too many variables and functions can lead to naming collisions and other issues. Instead, it is a good practice to use modules and scopes to organize code and reduce dependencies on the global object.

4. **CORS?**

CORS stands for Cross-Origin Resource Sharing, which is a security mechanism implemented by web browsers to prevent web pages from making requests to a different domain than the one that served the web page. This mechanism is enforced by default in most modern web browsers and can cause problems when developing client-server applications that involve multiple domains.

In Node.js, you can enable CORS for your server by using the `cors` middleware. Here are the steps to use the `cors` middleware in your Node.js application:

1. Install the `cors` middleware package using npm:

```

npm install cors

```

2. Import the `cors` middleware in your Node.js application:

```javascript

const cors = require('cors');

```

3. Use the `cors` middleware in your application to enable CORS for all routes:

```javascript

const express = require('express');

const app = express();

app.use(cors());

```

By default, this middleware will allow requests from any origin and with any headers. You can customize the behavior of the `cors` middleware by passing options to it. For example, you can allow requests from a specific origin by passing an `origin` option:

```javascript

app.use(cors({

origin: 'https://example.com'

}));

```

This will allow requests from `https://example.com` and block requests from all other origins.

You can also allow specific HTTP methods and headers by passing them in the `methods` and `allowedHeaders` options:

```javascript

app.use(cors({

methods: ['GET', 'POST'],

allowedHeaders: ['Content-Type']

}));

```

This will allow only `GET` and `POST` requests and the `Content-Type` header.

Enabling CORS can help you avoid cross-domain issues when building client-server applications. However, it is important to use it carefully and configure it appropriately for your specific use case to avoid security vulnerabilities.

5. **Same Origin Policy:-**

The Same Origin Policy is a security feature of web browsers and is not directly related to Node.js. As a server-side technology, Node.js is not subject to the Same Origin Policy.

However, Node.js can be used to build server-side APIs that are consumed by client-side JavaScript applications running in web browsers. In this case, the Same Origin Policy applies to the client-side code, and you may need to take steps to enable cross-origin requests.

To enable cross-origin requests in a Node.js server, you can use the `cors` middleware. This middleware can be installed using npm:

```

npm install cors

```

And then you can use it in your Node.js application like this:

```javascript

const express = require('express');

const cors = require('cors');

const app = express();

// enable CORS for all routes

app.use(cors());

// or enable CORS for a specific origin

app.use(cors({

origin: 'http://example.com'

}));

```

By default, the `cors` middleware allows requests from any origin and with any headers. You can customize this behavior by passing options to the middleware, such as the `origin` option to allow requests from a specific origin.

It's important to note that enabling cross-origin requests can potentially introduce security vulnerabilities if not configured properly. Be sure to understand the implications of allowing cross-origin requests and configure the `cors` middleware appropriately for your specific use case.

6. Buffer?

In Node.js, a buffer is a temporary storage area for data while it is being moved from one place to another. Buffers are used to represent binary data in a way that can be manipulated and processed in a more efficient manner than using strings.

Buffers are instances of the `Buffer` class in Node.js. They can be created using several different methods, including the `Buffer.alloc()` and `Buffer.from()` methods.

Here's an example of creating a new buffer and filling it with data:

```javascript

const buf = Buffer.alloc(10);

buf.write('Hello');

console.log(buf.toString()); // Output: "Hello"

```

In this example, we are creating a new buffer of length 10 using the `Buffer.alloc()` method. We then write the string "Hello" to the buffer using the `write()` method. Finally, we log the contents of the buffer to the console using the `toString()` method.

Buffers can also be created from strings, arrays, or other buffers using the `Buffer.from()` method. Here's an example:

```javascript

const str = 'Hello, world!';

const buf = Buffer.from(str, 'utf8');

console.log(buf.toString()); // Output: "Hello, world!"

```

In this example, we are creating a new buffer from the string "Hello, world!" using the `Buffer.from()` method and specifying the encoding as `utf8`. We then log the contents of the buffer to the console using the `toString()` method.

Buffers can be useful in a variety of situations, such as when reading and writing files, sending and receiving data over a network, or when dealing with binary data formats like images or audio files. However, it's important to be careful when working with buffers, as they can easily lead to security vulnerabilities like buffer overflow attacks if not used correctly.

**NB $=>difference between SOP and CORS**

The Same Origin Policy (SOP) and Cross-Origin Resource Sharing (CORS) are both security mechanisms used to restrict cross-domain HTTP requests, but they operate at different levels of the web architecture.

The Same Origin Policy is a security measure implemented by web browsers to prevent web pages from making requests to a different domain than the one that served the web page. This policy is enforced by the browser and applies to JavaScript code running in the browser. The SOP requires that the protocol, domain, and port number of the origin must match for requests to be allowed.

On the other hand, Cross-Origin Resource Sharing (CORS) is a mechanism that allows web servers to relax the Same Origin Policy and selectively allow cross-domain requests. CORS works by adding HTTP headers to the response from the server, which inform the browser that cross-domain requests are allowed from certain domains.

In other words, SOP is a security feature of web browsers that prevents cross-domain requests by default, while CORS is a mechanism that allows web servers to explicitly allow cross-domain requests.

To summarize, the SOP is a browser security feature that blocks cross-domain requests by default, while CORS is a server-side mechanism that allows specific domains to make cross-domain requests.

7. Stream?

In Node.js, a stream is a mechanism for reading or writing data continuously, chunk by chunk, instead of all at once. Streams provide a powerful abstraction for handling large amounts of data, especially for operations like reading and writing large files, network operations, or processing real-time data like video or audio streams.

There are four types of streams in Node.js:

1. Readable streams: Streams that allow you to read data from a source. For example, reading a file from disk, fetching data from a network, or reading data from a database.

2. Writable streams: Streams that allow you to write data to a destination. For example, writing data to a file, sending data over a network, or inserting data into a database.

3. Duplex streams: Streams that can both read and write data. For example, a socket connection that allows you to send and receive data.

4. Transform streams: Streams that can modify or transform data as it is being read or written. For example, converting data from one format to another or compressing or decompressing data.

Here's an example of using a readable stream to read data from a file and write it to the console:

```javascript

const fs = require('fs');

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

readableStream.on('data', (chunk) => {

console.log(chunk);

});

readableStream.on('end', () => {

console.log('Finished reading file');

});

```

In this example, we are using the `createReadStream()` method from the `fs` module to create a readable stream for reading data from a file. We then listen for the `data` event, which is emitted when data is available to be read from the stream. When the `data` event is emitted, we log the chunk of data to the console. Finally, we listen for the `end` event, which is emitted when the stream has reached the end of the file.

Writing data to a writable stream is similarly easy. Here's an example of using a writable stream to write data to a file:

```javascript

const fs = require('fs');

const writableStream = fs.createWriteStream('output.txt', 'utf8');

writableStream.write('Hello, world!');

writableStream.write('This is a test.');

writableStream.end();

```

In this example, we are using the `createWriteStream()` method from the `fs` module to create a writable stream for writing data to a file. We then write two chunks of data to the stream using the `write()` method. Finally, we call the `end()` method to close the stream.

8. **HTTP methods**

In Node.js, you can handle HTTP methods using the built-in `http` or `https` module, or use a web framework like Express.js, which provides a more convenient way to handle HTTP requests and responses.

Here is an example of using the `http` module to handle HTTP methods:

```javascript

const http = require('http');

const server = http.createServer((req, res) => {

if (req.method === 'GET') {

// handle GET request

res.end('Received a GET request');

} else if (req.method === 'POST') {

// handle POST request

res.end('Received a POST request');

} else {

// handle unsupported request method

res.statusCode = 405; // Method Not Allowed

res.end();

}

});

server.listen(3000, () => {

console.log('Server listening on port 3000');

});

```

This creates an HTTP server that listens on port 3000 and handles GET and POST requests. For other request methods, it sets the HTTP status code to 405 (Method Not Allowed) and ends the response.

Here is an example of handling HTTP methods using Express.js:

```javascript

const express = require('express');

const app = express();

app.get('/', (req, res) => {

res.send('Received a GET request');

});

app.post('/', (req, res) => {

res.send('Received a POST request');

});

app.all('\*', (req, res) => {

res.status(405).end(); // handle unsupported request method

});

app.listen(3000, () => {

console.log('Server listening on port 3000');

});

```

This creates an Express.js application that handles GET and POST requests on the root path ('/') and sets the HTTP status code to 405 for unsupported request methods. The `app.all()` method is used to handle all HTTP methods for the specified path.

Note that these are just simple examples, and there are many more features and options available for handling HTTP methods in Node.js and Express.js.

9. **Application Level Middleware**

In Node.js, application-level middleware is a function that sits in the request-response cycle of an application and has access to the `request` and `response` objects and the `next()` function in the application's routing system. Middleware functions can be used for a variety of tasks, such as logging, authentication, error handling, and more.

Here's an example of an application-level middleware function that logs the URL and HTTP method of each incoming request:

```javascript

function loggerMiddleware(req, res, next) {

console.log(`${req.method} ${req.url}`);

next();

}

```

In this example, we define a function called `loggerMiddleware` that takes three parameters: `req`, `res`, and `next`. `req` and `res` are the request and response objects, respectively, and `next` is a function that tells the application to move on to the next middleware function in the stack.

Inside the middleware function, we log the HTTP method and URL of the incoming request using `console.log()`, and then call `next()` to move on to the next middleware function.

To use this middleware function in an application, we can simply add it to the middleware stack using the `use()` method of the application object, like this:

```javascript

const express = require('express');

const app = express();

app.use(loggerMiddleware);

// ... other middleware functions and route handlers ...

app.listen(3000, () => {

console.log('Server listening on port 3000');

});

```

In this example, we are using the `use()` method to add our `loggerMiddleware` function to the middleware stack for our Express application. The `use()` method takes a function as its parameter, and adds it to the stack in the order it is called.

When a request comes in to our application, it will first go through our `loggerMiddleware` function, which will log the request method and URL to the console, and then call `next()` to move on to the next middleware function in the stack.

Middleware functions can also be used to handle errors and send error responses to clients. For example:

```javascript

function errorHandlerMiddleware(err, req, res, next) {

console.error(err.stack);

res.status(500).send('Internal server error');

}

app.use(errorHandlerMiddleware);

```

In this example, we define an error handler middleware function called `errorHandlerMiddleware` that takes four parameters, including the `err` parameter, which represents the error that occurred. Inside the middleware function, we log the error stack trace to the console using `console.error()`, and then send a 500 Internal Server Error response to the client using the `res.status()` and `res.send()` methods.

We then add the `errorHandlerMiddleware` function to the middleware stack using the `use()` method, so that it will be called whenever an error occurs in our application.

10. **Ruler Level Middleware**

In Node.js, middleware can be applied at different levels of the application stack, including application-level middleware, router-level middleware, and error-handling middleware.

Router-level middleware is a type of middleware that is applied to a specific route or group of routes in an Express.js application. It allows you to perform actions specific to a particular route, such as validation, authentication, logging, or modifying the request or response objects.

To apply router-level middleware in Express.js, you can use the `router.use()` method, which is similar to `app.use()` but only applies to the routes defined on the router.

Here is an example of using router-level middleware in Express.js:

```javascript

const express = require('express');

const app = express();

const router = express.Router();

// define router-level middleware

router.use((req, res, next) => {

console.log('Router-level middleware');

next();

});

router.get('/', (req, res) => {

res.send('Hello, World!');

});

app.use('/api', router); // mount the router on a specific path

app.listen(3000, () => {

console.log('Server listening on port 3000');

});

```

In this example, the `router.use()` method is used to define router-level middleware that logs a message to the console for every request that goes through the router. The `router.get()` method defines a route that sends a response to the client. Finally, the `app.use()` method is used to mount the router on the '/api' path, which means that the router will only handle requests that match the '/api' path.

Note that router-level middleware can also be chained together, just like application-level middleware. For example:

```javascript

router.use((req, res, next) => {

console.log('First router-level middleware');

next();

}).use((req, res, next) => {

console.log('Second router-level middleware');

next();

}).get('/', (req, res) => {

res.send('Hello, World!');

});

```

This defines two router-level middleware functions that are chained together before the route handler. The order in which middleware functions are defined determines the order in which they are executed.

11. **Error handling Middleware**

In Node.js, error handling middleware is used to handle errors that occur during the processing of an HTTP request. This type of middleware is called after all other middleware and routes have been executed, and is designed to catch and handle errors that were not caught by other middleware or routes.

To define an error handling middleware in an Express.js application, you can use the `app.use()` method with a middleware function that has four parameters, like this:

```javascript

app.use((err, req, res, next) => {

// handle the error here

console.error(err);

res.status(500).send('Internal Server Error');

});

```

This error handling middleware will be called whenever an error occurs during the processing of an HTTP request, regardless of the request method or route. The first parameter, `err`, is the error object that was thrown or passed to `next()` by another middleware or route. The second parameter, `req`, is the request object, and the third parameter, `res`, is the response object. The fourth parameter, `next`, is a function that can be called to pass control to the next middleware function.

You can also define error handling middleware that is specific to a certain route or group of routes by mounting it on a router, like this:

```javascript

const router = express.Router();

router.use((err, req, res, next) => {

// handle the error here

console.error(err);

res.status(500).send('Internal Server Error');

});

router.get('/', (req, res) => {

// handle the route here

});

app.use('/api', router);

```

In this example, the error handling middleware is mounted on the router using the `router.use()` method, which means it will only handle errors that occur during the processing of routes defined on the router.

When an error occurs in your application, you can use the `next()` function to pass the error object to the error handling middleware, like this:

```javascript

app.get('/', (req, res, next) => {

const err = new Error('Something went wrong');

next(err);

});

```

In this example, the route handler throws a new `Error` object and passes it to the next middleware using the `next()` function. The error handling middleware will then be called to handle the error.

12. **Build in Middleware**

Node.js provides several built-in middleware functions that can be used in an Express.js application without having to install any additional packages. These middleware functions can be used to add common functionality to your application, such as parsing request bodies, handling cookies and sessions, and serving static files.

Here are some examples of built-in middleware functions in Node.js:

1. `express.json()`: This middleware function is used to parse JSON-encoded request bodies. It adds a `body` property to the request object containing the parsed JSON data. Here's an example of how to use it:

```javascript

const express = require('express');

const app = express();

app.use(express.json());

app.post('/api/users', (req, res) => {

console.log(req.body);

res.send('User created successfully');

});

```

In this example, the `express.json()` middleware function is used to parse JSON data in the request body of the `POST` request to the `/api/users` endpoint.

2. `express.urlencoded()`: This middleware function is used to parse URL-encoded request bodies. It adds a `body` property to the request object containing the parsed data. Here's an example of how to use it:

```javascript

const express = require('express');

const app = express();

app.use(express.urlencoded({ extended: true }));

app.post('/api/users', (req, res) => {

console.log(req.body);

res.send('User created successfully');

});

```

In this example, the `express.urlencoded()` middleware function is used to parse URL-encoded data in the request body of the `POST` request to the `/api/users` endpoint.

3. `express.static()`: This middleware function is used to serve static files, such as images, CSS, and JavaScript files. It takes a directory path as an argument and serves any files in that directory in response to requests that match the requested URL path. Here's an example of how to use it:

```javascript

const express = require('express');

const app = express();

app.use(express.static('public'));

app.get('/', (req, res) => {

res.sendFile(\_\_dirname + '/public/index.html');

});

```

In this example, the `express.static()` middleware function is used to serve static files in the `public` directory. The `GET` request to the root URL path (`/`) is handled by sending the `index.html` file in the `public` directory.

4. `cookie-parser`: This middleware function is used to handle cookies in HTTP requests. It parses the `Cookie` header and adds a `cookies` property to the request object containing an object of key-value pairs representing the cookies. Here's an example of how to use it:

```javascript

const express = require('express');

const app = express();

const cookieParser = require('cookie-parser');

app.use(cookieParser());

app.get('/', (req, res) => {

console.log(req.cookies);

res.send('Cookies received');

});

```

In this example, the `cookie-parser` middleware function is used to parse cookies in the `GET` request to the root URL path (`/`). The cookies are logged to the console and a response is sent to the client.

13. **Third-party Middleware**

Third-party middleware in Node.js are packages that are not included in the core Node.js framework, but are instead developed and maintained by the open-source community. These middleware packages can be easily integrated into your Node.js application using the Node Package Manager (npm).

Here are some examples of popular third-party middleware packages in Node.js:

1. `morgan`: This middleware function is used to log HTTP requests and responses to the console or a log file. It can be used to debug your application and monitor its performance. Here's an example of how to use it:

```javascript

const express = require('express');

const app = express();

const morgan = require('morgan');

app.use(morgan('dev'));

app.get('/', (req, res) => {

res.send('Hello World!');

});

```

In this example, the `morgan` middleware function is used to log HTTP requests and responses in the "dev" format to the console. The `GET` request to the root URL path (`/`) is handled by sending a "Hello World!" response to the client.

2. `helmet`: This middleware function is used to set security-related HTTP headers in HTTP responses to prevent common web vulnerabilities, such as cross-site scripting (XSS) attacks and clickjacking. Here's an example of how to use it:

```javascript

const express = require('express');

const app = express();

const helmet = require('helmet');

app.use(helmet());

app.get('/', (req, res) => {

res.send('Hello World!');

});

```

In this example, the `helmet` middleware function is used to set security-related HTTP headers in the HTTP response to the `GET` request to the root URL path (`/`). The `helmet` middleware function automatically sets the following headers:

- `X-DNS-Prefetch-Control`: Disable DNS prefetching.

- `X-Frame-Options`: Prevent clickjacking by setting the `X-Frame-Options` header to `DENY`.

- `Strict-Transport-Security`: Enable HTTP Strict Transport Security (HSTS).

- `X-Download-Options`: Prevent Internet Explorer from executing downloads in the "trusted" zone.

- `X-Content-Type-Options`: Prevent browsers from interpreting files as a different MIME type.

- `X-XSS-Protection`: Enable the XSS filter in browsers that support it.

3. `cors`: This middleware function is used to handle Cross-Origin Resource Sharing (CORS) in HTTP requests. It allows restricted resources (e.g. fonts, scripts, and APIs) on a web page to be requested from another domain outside the domain from which the resource originated. Here's an example of how to use it:

```javascript

const express = require('express');

const app = express();

const cors = require('cors');

app.use(cors());

app.get('/', (req, res) => {

res.send('Hello World!');

});

```

In this example, the `cors` middleware function is used to handle CORS in the `GET` request to the root URL path (`/`). The `cors` middleware function automatically sets the following headers:

- `Access-Control-Allow-Origin`: Set the origin that is allowed to access the resource.

- `Access-Control-Allow-Methods`: Set the HTTP methods that are allowed to access the resource.

- `Access-Control-Allow-Headers`: Set the headers that are allowed in the HTTP request.

- `Access-Control-Expose-Headers`: Set the headers that are exposed to the browser.

- `Access-Control-Allow-Credentials`: Set whether the resource can be accessed with credentials (e.g. cookies).

14. **MVC Architecture**

MVC (Model-View-Controller) is a common architectural pattern used in web application development. It separates an application's concerns into three distinct components:

1. Model: The model represents the data and logic of the application. It encapsulates data-related operations and is responsible for managing the data, such as retrieving, updating, and deleting data from a database or other data source.

2. View: The view represents the user interface (UI) of the application. It displays data to the user and allows the user to interact with the application.

3. Controller: The controller acts as an intermediary between the model and view. It receives input from the user and processes it to interact with the model and the view. It is responsible for controlling the flow of data between the model and view and for updating the view with the changes made in the model.

In Node.js, you can implement the MVC pattern using a web framework such as Express.js. Here's an example of how to structure an Express.js application using the MVC pattern:

```

app/

|- controllers/

| |- userController.js

| |- postController.js

|

|- models/

| |- userModel.js

| |- postModel.js

|

|- views/

| |- userView.ejs

| |- postView.ejs

|

|- routes/

| |- userRoutes.js

| |- postRoutes.js

|

|- app.js

|- package.json

```

1. `controllers/`: This directory contains the controller files for each model. For example, `userController.js` and `postController.js` are the controllers for the `userModel.js` and `postModel.js` models, respectively. The controller files define the functions that handle the logic for each model.

2. `models/`: This directory contains the model files for each data type. For example, `userModel.js` and `postModel.js` are the models for the user and post data, respectively. The model files define the schema for each data type and the functions to manipulate the data.

3. `views/`: This directory contains the view files for each model. For example, `userView.ejs` and `postView.ejs` are the views for the `userModel.js` and `postModel.js` models, respectively. The view files define the layout and structure of the UI.

4. `routes/`: This directory contains the route files for each model. For example, `userRoutes.js` and `postRoutes.js` are the routes for the `userModel.js` and `postModel.js` models, respectively. The route files define the HTTP endpoints for each model and map them to the controller functions.

5. `app.js`: This is the main application file. It initializes the Express.js application, sets up the middleware and the routes, and starts the server.

Using this directory structure, you can easily organize your code and separate the concerns of your application. You can also easily add new models, views, controllers, and routes to your application without affecting the existing code.

15.  **View engine**

A view engine is a module in Node.js that allows you to generate dynamic HTML pages using a template language. There are several popular view engines available for Node.js, including:

1. EJS (Embedded JavaScript): EJS is a simple templating language that enables you to generate HTML markup with plain JavaScript code. It is easy to learn and use, and it has a large community of users.

2. Pug (formerly Jade): Pug is a popular view engine that uses indentation and tags to generate HTML. It is a concise language and allows for easy creation of reusable components.

3. Handlebars: Handlebars is a popular view engine that allows you to create HTML templates with placeholders that can be replaced with dynamic content. It uses a syntax similar to Mustache, a popular templating language.

4. Mustache: Mustache is a simple and flexible view engine that allows you to create HTML templates with placeholders that can be replaced with dynamic content. It is available in many programming languages, including Node.js.

To use a view engine in Node.js, you need to install it using npm and configure it in your application. Here's an example of how to set up the EJS view engine in an Express.js application:

1. Install EJS using npm:

```

npm install ejs --save

```

2. Configure the view engine in your Express.js application:

```javascript

const express = require('express');

const app = express();

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

```

3. Create an EJS file in the views directory:

```html

<!-- views/index.ejs -->

<!DOCTYPE html>

<html>

<head>

<title><%= title %></title>

</head>

<body>

<h1><%= message %></h1>

</body>

</html>

```

4. Render the EJS file in your route handler:

```javascript

app.get('/', (req, res) => {

res.render('index', { title: 'Home', message: 'Welcome to my website!' });

});

```

In this example, the `res.render()` method renders the `views/index.ejs` file with the given data object. The `<%= %>` syntax is used to insert dynamic data into the HTML page.

16. **Express**

Express is a popular web application framework for Node.js that simplifies the process of creating web applications and APIs. It provides a set of tools and features that allow developers to build robust and scalable applications quickly and easily.

Some of the key features of Express include:

1. Routing: Express provides a simple and flexible mechanism for defining routes for your application. You can easily map HTTP requests to specific functions or controllers based on the request method and URL.

2. Middleware: Express supports a wide range of middleware functions that can be used to add functionality to your application. Middleware functions can be used for tasks such as authentication, logging, error handling, and more.

3. Templating: Express supports a variety of templating engines, such as EJS, Handlebars, and Jade, which allow you to generate dynamic HTML pages.

4. Error handling: Express provides a built-in error handling middleware that can be used to catch and handle errors in your application.

5. Static file serving: Express can serve static files, such as HTML, CSS, and JavaScript, directly from the file system without the need for a separate server.

6. View rendering: Express allows you to render dynamic views using template engines, such as EJS and Handlebars.

To use Express in your Node.js application, you first need to install it using npm:

```

npm install express --save

```

Once you have installed Express, you can create a new application by requiring it and creating an instance of the `express()` function:

```javascript

const express = require('express');

const app = express();

```

You can then define routes using the `app.get()`, `app.post()`, `app.put()`, `app.delete()`, and other methods:

```javascript

app.get('/', (req, res) => {

res.send('Hello, World!');

});

```

You can also use middleware functions to add functionality to your application. For example, you can use the `express.json()` middleware to parse JSON data from incoming requests:

```javascript

app.use(express.json());

```

Express provides a wide range of middleware functions and features that can be used to build powerful and scalable web applications and APIs.

17. **Dynamic routing in node**

Dynamic routing in Node.js allows you to create routes that can handle variable values in the URL. This is useful when you want to create a web application that needs to handle multiple resources or entities.

Here's an example of how to create a dynamic route in Node.js using the `Express` framework:

```javascript

const express = require('express');

const app = express();

// Dynamic route for a user with a specific ID

app.get('/users/:id', (req, res) => {

const userId = req.params.id;

res.send(`User with ID ${userId} was requested`);

});

app.listen(3000, () => {

console.log('Server started on port 3000');

});

```

In this example, we define a dynamic route using the `:` character to indicate that this part of the URL can vary. When a request is made to a URL that matches this pattern, the `Express` framework will extract the value of the dynamic part of the URL and make it available in the `req.params` object. In this example, the dynamic part of the URL is `:id`, which can be accessed as `req.params.id`.

When a client makes a GET request to `/users/42`, the server will extract the value `42` from the URL and use it to generate a response, which will be sent back to the client as a plain text message: "User with ID 42 was requested".

Dynamic routing in Node.js can be used to create powerful and flexible web applications that can handle a wide range of resources and entities.

18. Pending Topics:

-use different routes follow ES6try to use

crypto module instead of uuidres.

-write vs res.send options methodCORS

-types of middlewares

19. What is node js?

Node. js is an open-source, cross-platform JavaScript runtime environment and library for running web applications outside the client's browser. Ryan Dahl developed it in 2009, and its latest iteration, version 15.14, was released in April 2021. Developers use Node.

20. http

The http module in Node.js is used for creating HTTP servers and making HTTP requests. It provides a way to communicate with web servers through HTTP protocol.

With the http module, you can create an HTTP server that listens for requests on a specific port and sends responses back to clients. It also provides methods for making HTTP requests to other servers, such as GET, POST, PUT, DELETE, etc.

Here are some common use cases of the http module:

Creating an HTTP server to handle incoming requests and return responses to clients. Sending HTTP requests to other servers to retrieve data or update resources. Implementing a web API using HTTP methods and status codes. Serving static files or resources over HTTP. In addition to the http module, Node.js also provides the https module for secure communication over SSL/TLS.

21. HTTP methods

http methods

HTTP methods are standardized ways of making requests to a web server. HTTP methods define the purpose of the request and how the server should respond to it. The most commonly used HTTP methods are:

GET: Used to request data from a specified resource.

POST: Used to submit an entity to a specified resource, often causing a change in state or side effects on the server.

PUT: Used to replace an entity that exists on the server with a new entity.

DELETE: Used to delete an entity that exists on the server.

PATCH: Used to update a part of an entity on the server.

HEAD: Used to retrieve only the headers of the response that would be obtained if the corresponding GET request was made.

OPTIONS: Used to describe the communication options for the resource.

TRACE: Used to retrieve a diagnostic trace of the request-response cycle.

CONNECT: Used to establish a network connection to a resource, usually a server using SSL/TLS.

It's worth noting that not all web servers and APIs support all of these methods, and some may have additional methods or use different names for the same methods.

22. Http method -get diff post

HTTP (Hypertext Transfer Protocol) defines a set of request methods (also referred to as HTTP verbs) that indicate the desired action to be performed for a given resource. The most commonly used HTTP methods are:

GET - Retrieve data or information from a specified resource

POST - Submit data or information to be processed to a specified resource

PUT - Update data or information on a specified resource

DELETE - Delete a specified resource

PATCH - Update a part of the data or information on a specified resource

HEAD - Retrieve only the header section of a specified resource

OPTIONS - Describe the communication options for the target resource

TRACE - Perform a message loop-back test along the path to the target resource

23. Patch & put

In Node.js, PATCH and PUT are two different HTTP methods that can be used for updating resources on a server.

PATCH: The PATCH method is used to partially update an existing resource. In other words, if you only need to update a specific field or property of a resource, you can use the PATCH method to make the change.

PUT: The PUT method is used to completely replace an existing resource with a new one. In other words, if you need to update the entire resource, you can use the PUT method to replace the old resource with a new one.To handle

24. http status code

HTTP status codes are three-digit numbers returned by a server in response to a client's request. They indicate the status of the requested resource and provide information about how the server has handled the request. The first digit of the code indicates a general class of response, while the second and third digits provide additional detail.

Some common HTTP status codes include:

200 OK: The request was successful, and the server returned the requested data.

201 Created: The server successfully created a new resource in response to the request.

204 No Content: The server successfully processed the request but did not return any content.

400 Bad Request: The server could not understand the request due to invalid syntax.

401 Unauthorized: The request requires authentication, and the client is not authenticated.

403 Forbidden: The client does not have permission to access the requested resource.

404 Not Found: The server could not find the requested resource.

500 Internal Server Error: The server encountered an error while processing the request.

502 Bad Gateway: The server acting as a gateway or proxy received an invalid response from the upstream server.

503 Service Unavailable: The server is currently unable to handle the request due to maintenance or overload.

There are many other status codes, each with its own specific meaning. By understanding

25. Option method

The OPTIONS method is an HTTP method that is used to retrieve information about the communication options available for a target resource. When a client sends an OPTIONS request to a server, the server should return a response that lists the HTTP methods, headers, and other communication options that are supported by the target resource.

The OPTIONS method can be used by clients to determine which HTTP methods are supported by the server for a particular resource, which headers are accepted by the server, and which authentication schemes are supported. This information can be useful for clients that need to interact with the server, as it can help them make informed decisions about how to structure their requests.

In Node.js, you can handle OPTIONS requests using middleware functions in a similar way to handling other HTTP methods. When an OPTIONS request is received, you can use the res.set() method to set the response headers that list the supported HTTP methods, headers, and other

26. Middle wares

In web development, a middleware is a function that is executed in the request-response cycle of an application. Middleware functions have access to the request and response objects, and can perform certain actions on them before passing control to the next middleware function in the chain.

In the context of Node.js, middleware functions can be used in frameworks like Express.js to perform tasks such as logging, error handling, authentication, and request parsing. Middleware functions can be added to the request pipeline using the app.use() method, and can be chained together to create a series of actions that are executed in order.

Middleware functions can also be written by developers to perform custom tasks that are specific to their application. They can be created as standalone functions, or as part of a larger module or library.

27. Middle wares types

There are various types of middleware that can be used in web development. Here are some of the most common ones:

**1.Logging middleware:** This type of middleware is used to log incoming requests and outgoing responses. It can be used to track the performance of the application, as well as to debug any issues that arise.

**2.Error handling middleware:** This type of middleware is used to handle errors that occur during the request-response cycle. It can be used to catch errors, log them, and return an appropriate error response to the client.

**3.Authentication middleware :**This type of middleware is used to authenticate incoming requests. It can be used to check if the request is coming from an authorized user, and to enforce access controls.

**4.Parsing middleware:** This type of middleware is used to parse incoming requests. It can be used to extract data from the request body, query parameters, or headers.

Compression middleware: This type of middleware is used to compress outgoing responses. It can be used to reduce the size of the response, and to improve the performance of the application.

**5.Caching middleware:** This type of middleware is used to cache responses. It can be used to improve the performance of the application by reducing the number of requests that need to be processed.

These are just a few examples of the types of middleware that can be used in web development. There are many other types of middleware that can be used to perform a wide range of tasks.

28. Set-immediate

Set-Immediate is a Node.js-specific function that allows a call back function to be executed immediately after the current poll phase finishes. It is similar to set-Timeout but with a few key differences.

Set-Timeout schedules a call back to be executed after a specified delay, while set-Immediate schedules a call back to be executed at the next available opportunity after the current operation completes. This makes set-Immediate more efficient than set-Timeout in some cases, especially when dealing with large amounts of I/O.

set-Immediate call-backs are also prioritized over set Timeout call-backs, so if both are queued to run at the same time, the set Immediate call back will be executed first.

29. Error middle ware

30. Core modules

core modules

Core modules refer to modules that are available within the Node.js environment without the need for any additional installation or configuration. They are part of the Node.js package and provide basic functionalities that are essential for building any application.

Some examples of core modules include:

fs: provides file system related functionalities.

http: provides an HTTP server and client.

url: provides URL parsing and formatting functionalities.

path: provides path-related functionalities such as resolving and normalizing paths.

os: provides information about the operating system on which Node.js is running.

To use a core module in a Node.js application, you just need to require it using the require() function. For example, to use the fs module, you can do:

JavaScript

Copy code

const fs = require('fs');

Once you have required a module, you can use its exported functions and variables in your application.

31. Fs modules

The fs module in Node.js provides file system-related functionality. It allows you to read/write files, create and delete files and directories, modify file permissions, and more. Some of the commonly used functions provided by the fs module include:

fs.readFile(): Reads the contents of a file asynchronously and returns the data in a callback function.

fs.writeFile(): Writes data to a file asynchronously and returns a callback function to handle errors.

fs.appendFile(): Appends data to a file asynchronously.

fs.mkdir(): Creates a directory asynchronously.

fs.readdir(): Reads the contents of a directory asynchronously and returns a list of filenames.

fs.unlink(): Deletes a file asynchronously.

fs.rmdir(): Deletes a directory asynchronously.

The fs module is a core module in Node.js, which means it is included in the installation package and can be used without the need for installing

32. Events

In Node.js, the events module is a built-in module that provides an event-driven architecture for building asynchronous applications. It allows developers to create, emit, and listen for custom events.

The key components of the events module are the EventEmitter class and the emit() and on() methods. The EventEmitter class provides the ability to register event listeners and to emit events when a certain action occurs. The emit() method is used to trigger an event, while the on() method is used to register a listener for a particular event.

The `events` module in Node.js is a built-in module that provides an implementation of the observer pattern. It allows objects in Node.js to emit and listen for events.

Here's an example of how to use the `events` module in Node.js:

```javascript

const EventEmitter = require('events');

// Create a new event emitter instance

const myEmitter = new EventEmitter();

// Define an event handler function

const myEventHandler = (arg1, arg2) => {

console.log(`Event was fired with arguments: ${arg1}, ${arg2}`);

};

// Register the event handler function for a specific event

myEmitter.on('my-event', myEventHandler);

// Emit the event with some data

myEmitter.emit('my-event', 'hello', 42);

```

In this example, we create a new instance of the `EventEmitter` class from the `events` module. We then define an event handler function `myEventHandler` that takes two arguments and logs them to the console. We register this function as the handler for an event named `'my-event'` using the `on` method.

Finally, we emit the `'my-event'` event with the arguments `'hello'` and `42` using the `emit` method. When the event is emitted, the registered event handler function will be called with the arguments passed to `emit`.

The `events` module can be used to build event-driven applications in Node.js, where objects can emit events to signal that something has happened, and other objects can listen for those events and respond accordingly.

33. packages.js dependencies

package.js dependencies

I assume you're referring to a package.json file, which is a configuration file used in Node.js projects to define the metadata and dependencies for the project.

In the dependencies property of a package.json file, you can list the modules that your project requires to run properly in production. These dependencies can be installed using the npm install command.

34. **Routing**

**Routing in Node.js refers to the process of defining URL patterns that map to request handlers, allowing the server to respond to different requests with different responses.**

**Here's an example of how to set up routing in Node.js using the `Express` framework:**

**```**

**const express = require('express');**

**const app = express();**

**// Define a route for the homepage**

**app.get('/', function(req, res) {**

**res.send('Hello, World!');**

**});**

**// Define a route for a product page**

**app.get('/product/:id', function(req, res) {**

**const productId = req.params.id;**

**res.send('Product ID: ' + productId);**

**});**

**// Start the server**

**app.listen(3000, function() {**

**console.log('Server started on port 3000');**

**});**

**```**

**In this example, we use the `Express` framework to create a server and define two routes: one for the homepage and one for a product page. The `app.get` method is used to define a route that responds to GET requests to the specified URL pattern. The first argument to `app.get` is the URL pattern, which can include route parameters like `:id` that can be accessed using the `req.params` object. The second argument is a request handler function that takes a `req` object (representing the request) and a `res` object (representing the response) and sends a response back to the client.**

**When a client makes a GET request to the server, the server will match the URL pattern to a defined route and call the corresponding request handler function. In this example, visiting the homepage URL (`/`) will return the response `'Hello, World!'`, while visiting a product page URL like `/product/123` will return the response `'Product ID: 123'`.**

**Note that routing can become more complex as your application grows, with multiple layers of middleware and more advanced URL patterns. However, the basic principle remains the same: define URL patterns that map to request handlers, and let the server handle the rest.**

35. **query parameters**

Query parameters are a way to pass data to the server through the URL in Node.js. They are a key-value pair that comes after the question mark (?) in the URL and are separated by the ampersand (&) symbol.

Here is an example URL that includes query parameters:

```

http://localhost:3000/users?name=John&age=30

```

In this example, the URL contains two query parameters: `name=John` and `age=30`. These parameters can be accessed by the server through the `req.query` object.

Here's an example of how to handle query parameters in Node.js using the `Express` framework:

```javascript

const express = require('express');

const app = express();

app.get('/users', (req, res) => {

const name = req.query.name;

const age = req.query.age;

res.send(`User's name is ${name} and age is ${age}`);

});

app.listen(3000, () => {

console.log('Server started on port 3000');

});

```

In this example, we define a route for `/users` and handle the query parameters passed in the URL. The `req.query` object contains a property for each query parameter, and their values can be accessed by their names. We then use the response object (`res`) to send a response back to the client.

When a client makes a GET request to `/users?name=John&age=30`, the server will extract the query parameters from the URL and use them to generate a response, which will be sent back to the client as a plain text message: "User's name is John and age is 30".

36. **session and cookies**

const express = require('express');

const session = require('express-session');

const app = express();

// Set up session middleware

app.use(session({

secret: 'mysecretkey',

resave: false,

saveUninitialized: true,

cookie: {

maxAge: 60000 // Set cookie expiration time to 1 minute

}

}));

// Define a route that sets a session variable

app.get('/', function(req, res) {

req.session.username = 'John';

res.send('Session variable set.');

});

// Define a route that retrieves a session variable

app.get('/user', function(req, res) {

const username = req.session.username;

res.send('Username: ' + username);

});

// Start the server

app.listen(3000, function() {

console.log('Server started on port 3000');

});

37. **Can we get data by using the post method?**

Yes, you can retrieve data using the POST method in Node.js. When a client makes a POST request to a Node.js server, the server can access the data sent in the request body using the `req.body` object.

Here's an example of how to handle a POST request and retrieve the data in Node.js using the `body-parser` middleware:

```

const express = require('express');

const bodyParser = require('body-parser');

const app = express();

// Set up body-parser middleware

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

// Handle POST requests

app.post('/data', function(req, res) {

const data = req.body;

console.log(data);

res.send('Data received.');

});

// Start the server

app.listen(3000, function() {

console.log('Server started on port 3000');

});

```

In this example, we use the `body-parser` middleware to parse the request body and make it available in the `req.body` object. We then define a route that handles POST requests to the `/data` URL. When the client makes a POST request with data in the request body, the server logs the data to the console and sends a response indicating that the data was received.

Note that to send data using the POST method, the client needs to include the data in the request body. This can be done using an HTML form with a POST method or by using a tool like `curl` or `Postman` to send a POST request with the data in the request body.

38. **What’s view engine**

A view engine is a software component that is responsible for rendering the output of an application in a format that can be displayed to the user. It takes a template, usually written in a specific language, and combines it with data from the application to produce a view that can be sent to the client.

In web development, a view engine is often used to render HTML templates with data from a web application. The view engine reads the template file and replaces placeholders with data provided by the application, generating the final HTML document that will be sent to the client.

Some popular view engines for Node.js include EJS, Handlebars, Pug, and Mustache. These view engines provide a way to create dynamic templates that can be rendered with different data to generate different outputs.

39. **Working of session**

In Node.js, a session is a way to store data about a user across multiple HTTP requests. Sessions are typically used to store user data such as authentication status, shopping cart contents, or preferences.

Here is a basic overview of how sessions work in Node.js:

1. When a user first visits your website, the server generates a unique session ID and stores it in a cookie in the user's browser. The cookie is typically named something like "sessionID".

2. The server uses the session ID to associate the user's requests with their session data. When the user makes a request, the server retrieves the session data associated with the session ID stored in the cookie.

3. The session data is typically stored in a database or in memory on the server. The data can be any type of data that you want to store, such as an object, an array, or a string.

4. When the user logs out or their session expires, the server deletes the session data from the database or memory.

To implement sessions in Node.js, you can use a middleware like `express-session`. Here is an example of how to use `express-session` to create a session:

```

const express = require('express');

const session = require('express-session');

const app = express();

// Set up session middleware

app.use(session({

secret: 'mysecretkey',

resave: false,

saveUninitialized: true

}));

// Define a route that sets a session variable

app.get('/', function(req, res) {

req.session.username = 'John';

res.send('Session variable set.');

});

// Define a route that retrieves a session variable

app.get('/user', function(req, res) {

const username = req.session.username;

res.send('Username: ' + username);

});

// Start the server

app.listen(3000, function() {

console.log('Server started on port 3000');

});

```

In this example, we use `express-session` middleware to create a session with a secret key, which is used to encrypt the session data. We then define two routes: one that sets a session variable (`username`) and another that retrieves it. When the user visits the `/` route, their `username` session variable is set to `'John'`. When they visit the `/user` route, their `username` session variable is retrieved and displayed in the response.

Note that sessions are an important security feature in web applications, but they can also be a security risk if not implemented correctly. To prevent attacks like session hijacking or session fixation, you should use best practices like setting a secure `HttpOnly` flag on your session cookies and regenerating session IDs on login and logout.

40. **Examples of view engine**

In Node.js, a view engine is a templating language that allows you to generate HTML dynamically based on data. There are several view engines available in Node.js, and here are some popular examples:

1. EJS (Embedded JavaScript)

EJS is a popular view engine in Node.js that allows you to write HTML with embedded JavaScript. It uses a simple syntax that allows you to inject data into your HTML templates easily.

Example of using EJS in Node.js:

```

const express = require('express');

const app = express();

// Set EJS as the view engine

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

// Define a route that renders an EJS template

app.get('/', function(req, res) {

const data = { name: 'John', age: 30 };

res.render('index', { data: data });

});

// Start the server

app.listen(3000, function() {

console.log('Server started on port 3000');

});

```

2. Handlebars

Handlebars is another popular view engine in Node.js that allows you to create templates with dynamic data. It uses a syntax similar to HTML and has a lot of built-in helpers that make it easy to generate complex HTML.

Example of using Handlebars in Node.js:

```

const express = require('express');

const exphbs = require('express-handlebars');

const app = express();

// Set Handlebars as the view engine

app.engine('handlebars', exphbs({ defaultLayout: 'main' }));

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

// Define a route that renders a Handlebars template

app.get('/', function(req, res) {

const data = { name: 'John', age: 30 };

res.render('index', { data: data });

});

// Start the server

app.listen(3000, function() {

console.log('Server started on port 3000');

});

```

3. Pug (formerly Jade)

Pug is a view engine in Node.js that allows you to create templates with a simplified syntax. It uses indentation instead of curly braces to define code blocks, which can make your code more readable.

Example of using Pug in Node.js:

```

const express = require('express');

const app = express();

// Set Pug as the view engine

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

// Define a route that renders a Pug template

app.get('/', function(req, res) {

const data = { name: 'John', age: 30 };

res.render('index', { data: data });

});

// Start the server

app.listen(3000, function() {

console.log('Server started on port 3000');

});

```

These are just a few examples of view engines available in Node.js. There are many other options to choose from, and each has its own strengths and weaknesses. The best view engine for your project will depend on your specific needs and preferences.

41. **Get method – query string**

In Node.js, you can pass data in the URL of a GET request using a query string. A query string is a set of key-value pairs that are added to the end of a URL after a `?` character.

Here's an example of how to pass data in a GET request using a query string:

```

const http = require('http');

const options = {

hostname: 'localhost',

port: 3000,

path: '/users?name=John&age=30',

method: 'GET'

};

const req = http.request(options, (res) => {

console.log(`Status code: ${res.statusCode}`);

res.on('data', (chunk) => {

console.log(`Response body: ${chunk}`);

});

});

req.on('error', (error) => {

console.error(`Error: ${error}`);

});

req.end();

```

In this example, we're passing data in the URL of the GET request using a query string. The query string consists of two key-value pairs separated by an `&` character: `name=John` and `age=30`.

To parse the query string in Node.js, you can use the built-in `querystring` module. Here's an example of how to parse the query string from the previous example:

```

const http = require('http');

const querystring = require('querystring');

const options = {

hostname: 'localhost',

port: 3000,

path: '/users?name=John&age=30',

method: 'GET'

};

const req = http.request(options, (res) => {

console.log(`Status code: ${res.statusCode}`);

res.on('data', (chunk) => {

console.log(`Response body: ${chunk}`);

});

});

req.on('error', (error) => {

console.error(`Error: ${error}`);

});

req.end();

const query = querystring.parse(options.path.split('?')[1]);

console.log(`Name: ${query.name}`);

console.log(`Age: ${query.age}`);

```

In this example, we're using the `querystring` module to parse the query string from the `path` option of the HTTP request. We're splitting the `path` string at the `?` character to get the query string, then using the `parse()` method to convert the query string to an object with key-value pairs. Finally, we're logging the values of the `name` and `age` keys from the parsed query string.

Note that if you're using a web framework such as Express or Koa, query string parsing is often handled automatically for you.

42. **How data passed in post method**

In Node.js, you can pass data in the body of a POST request using the `application/x-www-form-urlencoded` or `application/json` content type.

Here's an example of how to pass data in a POST request using the `application/x-www-form-urlencoded` content type:

```

const http = require('http');

const postData = 'name=John&age=30';

const options = {

hostname: 'localhost',

port: 3000,

path: '/users',

method: 'POST',

headers: {

'Content-Type': 'application/x-www-form-urlencoded',

'Content-Length': Buffer.byteLength(postData)

}

};

const req = http.request(options, (res) => {

console.log(`Status code: ${res.statusCode}`);

res.on('data', (chunk) => {

console.log(`Response body: ${chunk}`);

});

});

req.on('error', (error) => {

console.error(`Error: ${error}`);

});

req.write(postData);

req.end();

```

In this example, we're passing data in the body of the POST request as a URL-encoded string. We set the `Content-Type` header to `application/x-www-form-urlencoded` to indicate the type of data being sent, and set the `Content-Length` header to the length of the data.

Here's an example of how to pass data in a POST request using the `application/json` content type:

```

const http = require('http');

const postData = JSON.stringify({ name: 'John', age: 30 });

const options = {

hostname: 'localhost',

port: 3000,

path: '/users',

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Content-Length': Buffer.byteLength(postData)

}

};

const req = http.request(options, (res) => {

console.log(`Status code: ${res.statusCode}`);

res.on('data', (chunk) => {

console.log(`Response body: ${chunk}`);

});

});

req.on('error', (error) => {

console.error(`Error: ${error}`);

});

req.write(postData);

req.end();

```

In this example, we're passing data in the body of the POST request as a JSON string. We set the `Content-Type` header to `application/json` to indicate the type of data being sent, and set the `Content-Length` header to the length of the data.

In both examples, we're using the built-in `http` module to make the POST request. However, you can also use popular web frameworks such as Express or Koa to handle POST requests and automatically parse the data in the request body.

43. **Storage handling on client-side**

On the client-side, the most common ways to store data are using cookies, local storage, and session storage.

Cookies: Cookies are small data files stored on the user's device by the browser. They are sent back to the server on each subsequent request, allowing the server to identify the user. Cookies can be created, read, and deleted using JavaScript. However, they have some limitations such as size restrictions, security concerns, and the fact that they are not suitable for storing large amounts of data.

Local Storage: Local storage is a type of web storage that allows data to be stored on the user's device indefinitely, even after the browser is closed. It can be used to store data such as user preferences, shopping cart items, and other data that needs to be persistent. Local storage can be accessed using JavaScript, and it is supported by all modern browsers.

Session Storage: Session storage is similar to local storage, but the data is only available during the session, meaning that it is lost when the browser is closed. It can be used to store data that only needs to be available during the user's visit to the website, such as the user's shopping cart contents or login status. Session storage can also be accessed using JavaScript, and it is supported by all modern browsers.

It is important to note that storing sensitive information such as passwords or credit card information in client-side storage is not recommended due to security concerns. It is recommended to store this type of information on the server-side instead.

44. **Patch diff put**

`PATCH`, `PUT`, and `DIFF` are HTTP methods used for updating or modifying resources in a web application.

In Node.js, you can use popular web frameworks such as Express or Koa to handle these HTTP methods.

Here's a brief overview of each method:

1. \*\*PATCH:\*\* The `PATCH` method is used to update a resource with a partial set of changes. This means that you can send a request with only the fields that need to be updated, rather than sending the entire resource. To handle `PATCH` requests in Express, you can use the `app.patch()` method:

```

app.patch('/users/:id', (req, res) => {

// Update the user with the specified ID using the data in req.body

});

```

2. \*\*PUT:\*\* The `PUT` method is used to replace an existing resource with a new one. This means that you must send the entire resource in the request. To handle `PUT` requests in Express, you can use the `app.put()` method:

```

app.put('/users/:id', (req, res) => {

// Replace the user with the specified ID with the data in req.body

});

```

3. \*\*DIFF:\*\* The `DIFF` method is not a standard HTTP method, but it can be used for comparing two resources and returning the differences between them. This can be useful for implementing version control or change tracking in a web application. To handle `DIFF` requests in Express, you can create a custom middleware function that compares the two resources and returns the differences.

```

app.diff('/users/:id', (req, res) => {

// Compare the resource with the specified ID to another resource

// and return the differences in the response

});

```

Overall, these HTTP methods provide different ways of updating or modifying resources in a web application, and can be handled using web frameworks such as Express or Koa.

45. **Body parser**

body-parser is a middleware for parsing the body of incoming requests in Express.js. It allows you to extract data from the request body and make it available in req.body.

body-parser supports various data formats such as JSON and URL-encoded data. It can be used to parse data from a POST, PUT, or DELETE request.

46. **Append file and write files**

In Node.js, to write or append data to a file, you can use the fs module. The fs module provides methods for working with the file system in a synchronous or asynchronous way.

To append data to a file in Node.js, you can use the fs.appendFile() method. This method appends the specified data to the end of the file, creating the file if it does not exist.

Here is an example of using fs.appendFile():

javascript

Copy code

const fs = require('fs');

fs.appendFile('message.txt', 'data to append', (err) => {

if (err) throw err;

console.log('The data was appended to file!');

});

To write data to a file in Node.js, you can use the fs.writeFile() method. This method replaces the file with the specified data, creating the file if it does not exist.

Here is an example of using fs.writeFile():

javascript

Copy code

const fs = require('fs');

fs.writeFile('message.txt', 'Hello Node.js', (err) => {

if (err) throw err;

console.log('The file has been saved!');

});

In both cases, the first argument is the file path, the second argument is the data to write or append to the file, and the third argument is an

47. **Global execution context**

The global execution context is the top-level context in a JavaScript program. It is created when the program starts running and includes any variables, functions, and other elements defined in the global scope.

In a browser environment, the global execution context is associated with the window object, which is the global object in the browser. In a Node.js environment, the global execution context is associated with the global object.

Variables and functions defined in the global scope are accessible from anywhere in the program, including from within other functions and execution contexts. However, it's generally considered good practice to minimize the use of global variables, as they can cause naming conflicts and make it harder to reason about the behavior of the program.

48. **Package . json**

The package.json file is a JSON file that is present at the root of any Node.js project. It contains the metadata about the project, such as the project name, version, author, dependencies, etc.

The package.json file can be created manually or generated by running the npm init command in the terminal. It is usually created at the beginning of a project and updated as the project progresses.

Some of the common fields that are present in a package.json file include:

name: The name of the project

version: The version of the project

description: A short description of the project

main: The entry point of the project

49. **Morgan**

Morgan is a popular logging middleware for Node.js HTTP servers. It logs HTTP requests and responses in a customizable format. Morgan logs can be written to the console or to a file, depending on how it is configured.

Morgan has several predefined logging formats, such as "combined", "common", "dev", "short", and "tiny". Developers can also create custom logging formats that fit their specific needs.

Morgan is a popular middleware for Node.js applications that provides logging functionality for HTTP requests and responses. It can be used with web frameworks such as Express, Connect, and Hapi, and provides several features for customizing the format and destination of logs.

Some of the key features of Morgan include:

HTTP request logging: Morgan logs information about HTTP requests, including the HTTP method, URL, status code, response time, and IP address of the client.

Customizable log format: Morgan provides several predefined log formats, as well as the ability to define a custom log format using tokens. Tokens are placeholders that can be used to include dynamic information in the log output.

Support for multiple destinations: Morgan can log output to a file, console, or stream, and can also be configured to use a rotating log file to prevent log files from growing too large.

Middleware integration: Morgan can be used as middleware in an Express or Connect application, making it easy to add logging to an existing application.

Here's an example of how to use Morgan with an Express application:

const express = require('express');

const morgan = require('morgan');

const app = express();

app.use(morgan('combined'));

app.get('/', (req, res) => {

res.send('Hello, world!');

});

app.listen(3000, () => {

console.log('Server listening on port 3000');

});

50. **NPM**

npm stands for "Node Package Manager". It is a command-line tool for managing packages and dependencies for Node.js applications. It allows developers to easily download and install packages and libraries from the npm registry, as well as publish their own packages for others to use.

npm is pre-installed with Node.js, and can be accessed through the command line by typing "npm”

51. **PM2**

PM2 is a process manager for Node.js applications that allows you to keep your application running continuously, even after it crashes or is shut down. It provides features like automatic restarts, load balancing, and advanced monitoring to ensure that your application runs smoothly in production.

With PM2, you can start, stop, and restart your application easily,

PM2 is a popular process manager for Node.js applications that provides several features for monitoring and managing Node.js processes. Some of the key features of PM2 include:

Process management : PM2 can start, stop, and restart Node.js processes, as well as manage multiple processes in a cluster.

Logging : PM2 can log output from Node.js processes to files, streams, or a centralized logging service.

Monitoring : PM2 can monitor the health and performance of Node.js processes, including CPU and memory usage, and provide alerts when thresholds are exceeded.

Automatic restarts : PM2 can automatically restart Node.js processes if they crash or become unresponsive.

Deployment : PM2 can be used to deploy Node.js applications to production environments, including support for zero-downtime deployments and rolling restarts.

Here's an example of how to use PM2 to start a Node.js process:

Install PM2 using npm: npm install pm2 -g.

Navigate to the directory of your Node.js application.

Start the Node.js process using PM2: pm2 start app.js.

View the status of the process using PM2: pm2 status.

52. **Error handling in express**

In Express, error handling can be done using middleware functions that are designed to catch errors and pass them to the next error handling middleware. The error handling middleware should be added after all other middleware functions.

Here's an example of how to implement error handling in Express:

app.use((err, req, res, next) => {

console.error(err.stack);

res.status(500).send('Something broke!');

});

In the above example, we define a middleware function that takes four parameters. The first parameter is the error object that is thrown by the application, while the other three parameters are the usual req, res, and next objects.

If an error is thrown in any of the routes or middleware functions, this error handling middleware will catch it and return an error response to the client.

Alternatively, you can also use the try...catch block to handle errors in a specific route or middleware function,

53. **Types of error**

In Node.js, errors can occur in different parts of an application, including the core Node.js modules, third-party modules, and application code. Here are some of the most common types of errors that can occur in Node.js:

Syntax errors: These occur when there is a problem with the syntax of the code, such as a missing or misplaced character or keyword. Syntax errors are usually detected by the Node.js compiler when the code is parsed, and they prevent the code from running.

Runtime errors: These occur when a program is running and encounters an error that prevents it from completing its task. Common types of runtime errors in Node.js include 1. reference errors (e.g., trying to access a variable that does not exist),

2.type errors (e.g., trying to use a variable of the wrong type),

3.range errors (e.g., trying to access an array element that does not exist).

Asynchronous errors: These occur when working with asynchronous functions that use callbacks or promises. Asynchronous errors can occur when a callback or promise is not properly handled, when an error is thrown inside a callback or promise, or when a callback or promise is not called at all.

System errors: These occur when there is a problem with the underlying operating system or hardware, such as a disk or network failure. System errors in Node.js are usually represented as error objects that provide information about the error, such as the error code and message.

It's important to handle errors properly in Node.js applications to prevent them from crashing or causing unexpected behavior. Some best practices for error handling in Node.js include using try-catch blocks to handle synchronous errors, using error callbacks or promises to handle asynchronous errors, logging errors to a file or console, and providing meaningful error messages to users.

54. **App.use**

app.use() is a method in Express.js that is used to mount middleware functions to a specified path in the application's request-response cycle. Middleware functions are functions that have access to the request object (req), response object (res), and the next middleware function in the application's request-response cycle (next) and are used to perform specific actions on the incoming request, modify the response, and/or pass control to the next middleware function in the cycle.

The app.use() method can be used to mount middleware functions for all HTTP request methods or for specific HTTP request methods such as GET, POST, PUT, and DELETE.

55. **Npm I express**

To install the Express framework in a Node.js project using npm, you can follow these steps:

Open a command prompt or terminal window.

Navigate to the directory of your Node.js project.

Run the following command to install Express:

npm install express

This will download and install the latest version of the Express framework and all its dependencies.

After the installation is complete, you can create a new JavaScript file and add the following code to create an Express application

const express = require('express');

const app = express();

app.get('/', (req, res) => {

res.send('Hello World!');

});

app.listen(3000, () => {

console.log('Server started on port 3000');

});

56. **json methods**

In Node.js, working with JSON data is a common task. JSON (JavaScript Object Notation) is a lightweight data interchange format that is widely used for data exchange between web clients and servers.

Node.js provides several built-in methods for working with JSON data:

1. JSON.parse() - This method is used to parse a JSON string and convert it into a JavaScript object.
2. JSON.stringify() - This method is used to convert a JavaScript object into a JSON string.
3. JSON.parse() with a reviver function - This method is used to parse a JSON string and convert it into a JavaScript object, with the ability to modify the object during parsing.

57. **URL Encode**

URL encoding is a process of converting characters or strings to a format that can be transmitted over the Internet. It is used to encode special characters such as space, question mark, ampersand, and other characters that are not allowed in a URL.

In Node.js, the querystring module provides functions to handle URL encoding and decoding. The querystring.stringify() method is used to convert a JavaScript object to a URL-encoded string. The querystring.parse() method is used to parse a URL-encoded string into a JavaScript object.

58. **Express, static**

In Node.js, Express is a popular web framework that provides a simple and flexible way to build web applications. One of the key features of Express is its ability to serve static files, such as HTML, CSS, JavaScript, and images.

To serve static files in Express, you can use the built-in express.static() middleware function. The express.static() middleware function takes a directory path as an argument and serves any files in that directory as static files.

const express = require('express');

const app = express();

// Serve static files from the "public" directory

app.use(express.static('public'));

59. **cors policy**

CORS (Cross-Origin Resource Sharing) is a security mechanism that is enforced by web browsers to restrict web page sc ripts from making requests to a different domain than the one that served the web page. This policy prevents malicious scripts from running on a web page and making unauthorized requests to other domains.

The CORS policy is implemented by the browser and can be configured on the server. By default, servers do not allow cross-origin requests. However, it is possible to configure the server to allow specific domains to make cross-origin requests.

To allow cross-origin requests, the server must include the appropriate CORS headers in its response. The most commonly used headers are:

Access-Control-Allow-Origin: Specifies the domains that are allowed to make cross-origin requests.

Access-Control-Allow-Headers: Specifies the HTTP headers that are allowed in the request.

Access-Control-Allow-Methods: Specifies the HTTP methods that are allowed in the request.

60. **thread pool**

A thread pool is a collection of threads that are created to perform a set of tasks concurrently. In Node.js, a thread pool is used to perform I/O operations and other computationally intensive tasks asynchronously.

Node.js is built on top of the V8 JavaScript engine, which is single-threaded by design. However, Node.js provides a mechanism to run JavaScript code in multiple threads using the Worker API. The Worker API allows you to spawn new threads in Node.js and run JavaScript code in those threads.

To create a thread pool in Node.js, you can use the built-in worker\_threads module. The worker\_threads module provides the Worker class, which you can use to create new threads

61. event queue

The event queue is a data structure in JavaScript that holds events that are waiting to be processed by the event loop. It's a part of the event-driven programming model, where events trigger callbacks or handlers that are executed in response.

In a web browser environment, the event queue handles events such as mouse clicks, keyboard presses, and network requests. When an event is triggered, it's added to the event queue and waits for the event loop to process it. The event loop constantly checks the event queue and processes the events in order, one by one.

In Node.js

62. API

API stands for Application Programming Interface. It is a set of protocols, routines, and tools for building software applications that specify how different software components should interact with each other. APIs are used to enable communication between different software systems, allowing developers to create software applications that can leverage functionality provided by other applications or services. APIs can be used to access data, perform specific tasks, or provide access to other software functionality, among other things. APIs can be built using a variety of programming languages and protocols, such as HTTP, REST, and SOAP.

63. Express

Express is a popular web application framework for Node.js. It provides a set of tools and features for building web applications and APIs, such as routing, middleware, view rendering, and more. Express is designed to be minimal and flexible, allowing developers to create web applications in their own way. It also has a large and active community, with many third-party modules and resources available for use. Express can be used in a wide variety of projects, from small personal projects to large-scale enterprise applications.

64. set,put,delete,post,curd

CRUD stands for Create, Read, Update, and Delete, which are the four basic functions of persistent storage in computer applications. It is a set of actions that can be performed on data, where each letter represents a particular operation:

Create: This operation is used to create new data in the system.

Read: This operation is used to read or retrieve data from the system.

Update: This operation is used to modify or update existing data in the system.

Delete: This operation is used to remove or delete data from the system.

CRUD is a fundamental concept in computer programming and is used in many software systems, including web applications

65. **handle concurrency**

Node.js is designed to handle concurrency using a non-blocking, event-driven architecture. This means that Node.js is able to handle multiple requests and tasks simultaneously without blocking the execution of other requests or tasks.

At the core of Node.js is an event loop, which is a single thread that listens for events and executes callbacks when they are triggered. When a request is received, Node.js adds it to the event queue and continues processing other requests or tasks. When the event loop is available, it dequeues the next event and triggers the associated callback.

Node.js uses an asynchronous, non-blocking I/O model to handle I/O operations, such as reading and writing to files and network sockets. When an I/O operation is initiated, Node.js does not block the execution of other requests or tasks. Instead, it adds the operation to a queue and continues processing other requests or tasks. When the operation is completed, Node.js triggers the associated callback, which can then be executed.

Node.js also supports the use of worker threads, which allow developers to execute CPU-intensive tasks in parallel with the event loop. Worker threads allow Node.js to take advantage of multi-core CPUs and distribute tasks across multiple threads.

Overall, Node.js's non-blocking, event-driven architecture and support for worker threads enable it to handle concurrency efficiently and effectively.

66. **req.params and req.query in node js**

In Node.js with Express, `req.params` and `req.query` are properties of the `Request` object that are used to extract data from HTTP requests.

`req.params` is an object that contains the named parameters of the URL path. These parameters are defined in the route path and are denoted by a colon `:` followed by the parameter name. For example, consider the following route:

```javascript

app.get('/users/:userId', (req, res) => {

const userId = req.params.userId;

// Do something with the userId...

});

```

In this example, `req.params` is an object that contains a `userId` property, which corresponds to the value of the `userId` parameter in the URL path. For example, if the request URL is `/users/123`, then `req.params` would be `{ userId: '123' }`.

`req.query` is an object that contains the query parameters of the URL. Query parameters are appended to the URL after a question mark `?` and are in the form `key=value`. Multiple query parameters are separated by an ampersand `&`. For example, consider the following URL:

```

http://example.com/search?q=nodejs&page=2

```

In this example, `req.query` is an object that contains a `q` property with the value `'nodejs'` and a `page` property with the value `'2'`. You can access query parameters using `req.query.key`.

```javascript

app.get('/search', (req, res) => {

const query = req.query.q;

const page = req.query.page;

// Do something with the query and page...

});

```

In this example, `req.query` is an object that contains a `q` property with the value `'nodejs'` and a `page` property with the value `'2'`. You can access query parameters using `req.query.key`.

**67. res.send and res.write and its differences**

In Node.js with Express, `res.send()` and `res.write()` are methods of the `Response` object that are used to send data back to the client in an HTTP response. The main difference between `res.send()` and `res.write()` is the way they handle the response data.

`res.send()` is used to send a complete HTTP response to the client and it can be called only once per request. `res.send()` is typically used to send data that does not need to be streamed to the client. It sets the `Content-Type` header based on the type of data being sent and also sets the `Content-Length` header automatically.

```javascript

app.get('/', (req, res) => {

res.send('Hello, World!');

});

```

In this example, `res.send()` sends the string `'Hello, World!'` as the response body to the client.

`res.write()` is used to send partial HTTP responses to the client and can be called multiple times per request. `res.write()` is typically used to send large amounts of data that need to be streamed to the client. `res.write()` does not set the `Content-Type` or `Content-Length` headers automatically, so these headers must be set manually.

```javascript

app.get('/stream', (req, res) => {

res.write('Data 1\n');

res.write('Data 2\n');

res.write('Data 3\n');

res.end();

});

```

In this example, `res.write()` is used to send three lines of text to the client as separate responses, followed by a call to `res.end()` to signal the end of the response.

In summary, `res.send()` is used to send a complete HTTP response to the client and can be called only once per request, while `res.write()` is used to send partial HTTP responses to the client and can be called multiple times per request.

68. http options in nodejs

In Node.js with the built-in `http` module, the `OPTIONS` HTTP method can be handled by creating a server and using the `http.Server` class.

The `OPTIONS` method is used to determine which HTTP methods are supported by the server for a specific URL. The server should return a response that includes the `Allow` header with a comma-separated list of HTTP methods that are allowed for the requested resource.

Here is an example of how to handle `OPTIONS` requests using the `http` module:

```javascript

const http = require('http');

const server = http.createServer((req, res) => {

if (req.method === 'OPTIONS') {

res.writeHead(200, {

'Access-Control-Allow-Origin': '\*',

'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, OPTIONS',

'Access-Control-Allow-Headers': 'Content-Type, Authorization',

'Content-Length': 0

});

res.end();

} else if (req.method === 'GET') {

// handle GET request

res.end('Hello, World!');

} else {

res.statusCode = 405; // Method Not Allowed

res.end();

}

});

server.listen(3000, () => {

console.log('Server started on port 3000');

});

```

In this example, the server checks if the request method is `OPTIONS`, and if so, sets the appropriate headers in the response and sends an empty response body with a `200` status code. If the request method is not `OPTIONS`, the server handles the request normally, in this case by sending a simple "Hello, World!" response for a `GET` request. If the request method is not `OPTIONS` or `GET`, the server returns a `405` status code, indicating that the requested method is not allowed for the requested resource.

Pending topics:-

**6-May-2023**

- form validation

- create a file

- how nodeJs handles concurrency

- buffer

- error middleware

- morgan

- http options

- Explore CORS, same origin policy, dynamic routing, chained route

- req.params

- res.send vs res.write

**11-May-2023**

- Create a file

- Thread pool?

- Coding standards (const-> let)

- DevDependencies in package.json

- Why package-lock.json