My pendings

Common JS

CommonJS is a module system for JavaScript used primarily in server-side environments, such as Node.js. It allows developers to organize their code into reusable modules, each with its own scope, dependencies, and exports.

Key features of CommonJS modules include:

1. **Module Definition**: CommonJS modules are defined using the `module.exports` object to export values from a module and `require()` function to import values from other modules.

2. **Synchronous Loading**: CommonJS modules are loaded synchronously, meaning that the code within a module is executed only after all of its dependencies have been loaded and evaluated.

3. **Single Instance**: Each module in CommonJS is evaluated and executed only once per application, ensuring that the module's exports are shared across all parts of the application that require it.

4. **Server-Side Focus**: CommonJS was originally designed for server-side JavaScript environments like Node.js, where synchronous module loading and caching are more suitable.

Here's a basic example of a CommonJS module:

```
// myModule.js
const someValue = "Hello, world!";

function greet(name) {
  return `Hello, ${name}!`;
}

module.exports = {
  someValue,
  greet
};
```

```
// main.js
const myModule = require('./myModule');

console.log(myModule.someValue); // Output: Hello, world!
console.log(myModule.greet('John')); // Output: Hello, John!
```

In this example, `myModule.js` exports two values (`someValue` and `greet` function) using the `module.exports` object. In `main.js`, the `require()` function is used to import the module, and its exported values are accessed via the `myModule` object.

CommonJS has been widely adopted in the Node.js ecosystem for organizing and managing dependencies in server-side JavaScript applications. However, it is less commonly used in client-side web development, where asynchronous module loading and other module systems like ECMAScript Modules (ESM) have become more prevalent.

fs.unlink()

`fs.unlink()` is a function provided by the Node.js `fs` (File System) module, which is used to delete (unlink) a file asynchronously. It takes two parameters: the path to the file to be deleted and a callback function that is called once the operation is completed.

Here's the syntax:

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

fs.unlink(path, (err) => {
  if (err) {
    console.error('Error deleting file:', err);
  } else {
    console.log('File deleted successfully');
  }
});
```

Parameters:
- `**path**`: A string representing the path to the file to be deleted.
- `**callback**`: A function that is called once the file deletion operation is completed. It takes one argument, `err`, which is `null` if the operation was successful, or an error object if an error occurred.

It's important to note that `**fs.unlink()**` deletes the specified file directly without moving it to the system's recycle bin or trash. Therefore, be cautious when using this function, as deleted files cannot be easily recovered.

Also, make sure that the file exists before calling `**fs.unlink()**`, as attempting to delete a non-existent file will result in an error. You can check the existence of the file using `**fs.existsSync()**` or handle errors using `**fs.stat()**` or other file system methods.

Core modules in node js

Node.js comes with a set of built-in core modules that provide essential functionalities for developing various types of applications. Some of the core modules in Node.js include:

1. **fs (File System)**: Provides functions for working with the file system, such as reading and writing files, creating and deleting directories, and manipulating file metadata.

2. **http/https**: These modules allow you to create HTTP and HTTPS servers and make HTTP/HTTPS requests. They provide APIs for handling incoming HTTP requests, creating server responses, and configuring server settings.

3. **path**: Offers utilities for working with file paths. It provides methods for resolving, joining, normalizing, and parsing file paths, making it easier to work with file and directory paths cross-platform.

4. **events**: Implements the EventEmitter class, which allows objects to emit and handle events. It serves as the foundation for event-driven programming in Node.js, enabling communication between different parts of an application.

5. **util**: Contains various utility functions and classes that provide helpful functionalities for working with JavaScript objects, including formatting, debugging, and inspecting objects.

6. **os (Operating System)**: Provides information about the operating system on which the Node.js process is running, such as CPU architecture, memory usage, network interfaces, and platform-specific details.

7. **crypto**: Offers cryptographic functionalities, such as generating secure random numbers, hashing data, encrypting and decrypting data, and creating digital signatures.

8. **stream**: Implements the Stream API, which allows you to work with streams of data. Streams provide an efficient way to handle data that can be read from or written to asynchronously, such as files, network connections, or in-memory buffers.

9. **querystring**: Provides utilities for parsing and formatting URL query strings. It allows you to parse query strings into JavaScript objects and stringify JavaScript objects into URL-encoded query strings.

10. **url**: Offers utilities for working with URLs, such as parsing, formatting, resolving, and manipulating URL components.

These are just a few examples of the core modules available in Node.js. Node.js also includes other useful core modules, such as `child_process`, `cluster`, `dns`, `http2`, `net`, `zlib`, and more, each serving specific purposes to facilitate various aspects of application development.

body-parser

`body-parser` is a middleware for handling HTTP POST requests in Node.js. It extracts the entire body portion of an incoming request stream and exposes it on `req.body`.

When you make a POST request to a Node.js server with data encoded in the request body (such as form data or JSON), `body-parser` parses the data and makes it available on the `req.body` object of your request handler.

Here's how you typically use `body-parser` in a Node.js application:

1. Install `body-parser` using npm or yarn:
   ```
   npm install body-parser
   ```

2. Require and use `body-parser` in your Node.js application:

   ```javascript
```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

// Parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }));

// Parse application/json
app.use(bodyParser.json());

// Your route handlers
app.post('/example', (req, res) => {
  console.log(req.body); // Data from the request body
  res.send('Data received and processed');
});

app.listen(3000, () => {
```
```

```
  console.log('Server is running on port 3000');
});
```

In this example:
- We use `body-parser` to parse both `application/x-www-form-urlencoded` and `application/json` request bodies.
- The parsed data is available on `req.body` in the route handlers.

`body-parser` simplifies the process of handling incoming request bodies and parsing them into usable JavaScript objects. It's commonly used in combination with Express.js to build web servers and APIs in Node.js. However, starting from Express version 4.16.0, `express` comes with built-in middleware called `express.json()` and `express.urlencoded()` to parse JSON and URL-encoded request bodies, respectively, reducing the need for `body-parser`.

URL Encoded vs JSON

URL encoded and JSON are two common formats used to send data in HTTP requests and responses, especially in web applications. Here's a comparison of URL encoded and JSON formats:

1. **URL Encoded**:
    - **Format**: URL encoded data consists of key-value pairs separated by `&`, with keys and values separated by `=`. For example: `**key1=value1&key2=value2**`.
    - **Usage**: URL encoded data is typically used when submitting form data via HTML forms or when making POST requests from web browsers.
    - **Supported Data Types**: Supports simple key-value pairs and arrays, but does not natively support nested objects or complex data structures.
    - **Content-Type**: Usually sent with the `**application/x-www-form-urlencoded**` content type.

- **Example**:
  ```
  Content-Type: application/x-www-form-urlencoded

  name=John&age=30
  ```

2. **JSON (JavaScript Object Notation)**:
  - **Format**: JSON is a lightweight data interchange format that uses a human-readable and easy-to-write text format. It consists of key-value pairs, where keys are strings and values can be strings, numbers, arrays, objects, boolean, or null.