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.

- **Usage**: JSON is commonly used for sending and receiving data between client and server in web APIs (RESTful APIs). It's also frequently used for configuration files and storing structured data.
- **Supported Data Types**: Supports complex data structures, including nested objects and arrays.
- **Content-Type**: Usually sent with the `application/json` content type.
- **Example**:
  ```
  Content-Type: application/json

  {
    "name": "John",
    "age": 30
  }
  ```

Choosing Between URL Encoded and JSON:
- **URL Encoded**:
  - Suitable for simple data structures, such as form submissions.
  - Useful when interacting with web forms or APIs that expect URL encoded data.
- **JSON**:
  - Preferred for more complex data structures or when working with APIs that support JSON.
  - Provides better support for nested objects and arrays.
  - Offers a more standardized and widely adopted format for data interchange.

Ultimately, the choice between URL encoded and JSON depends on the requirements of your application and the expectations of the API or service you are interacting with.

Session management

Session management in Node.js involves maintaining stateful information about a user's interaction with a web application across multiple requests. This typically includes storing user-specific data, such as authentication status, user preferences, shopping cart contents, etc., for the duration of a user's session.

Here's a basic overview of session management in Node.js:

1. **Session Creation**: When a user visits a web application, a unique session identifier (session ID) is generated for that user. This session ID is typically stored in a cookie in the user's browser.

2. **Storing Session Data**: Session data, such as user authentication status, user preferences, etc., is stored on the server-side. This data is associated with the session ID and can be accessed and modified as needed during the user's session.

3. **Retrieving Session Data**: On subsequent requests from the same user, the session ID stored in the cookie is sent to the server. The server uses this session ID to retrieve the corresponding session data from its storage.

4. **Updating Session Data**: Session data can be updated during the user's session to reflect changes in the user's state or interactions with the web application.

5. **Session Expiry**: Sessions typically have a timeout period after which they expire and are no longer valid. This helps prevent sessions from persisting indefinitely, reducing the risk of security vulnerabilities.

Node.js provides various mechanisms for implementing session management. Some common approaches include:

- **Using Middleware**: Express.js, a popular web framework for Node.js, provides middleware like `express-session` for session management. This middleware handles session creation, storage, and retrieval automatically, allowing you to focus on your application logic.

- **Using Database**: Session data can be stored in a database, such as MongoDB or Redis, with the session ID acting as the key to retrieve the data. This allows for scalability and persistence across server restarts.

- **Using Memory Store**: For smaller applications or development purposes, session data can be stored in memory on the server-side. However, this approach is not suitable for production environments with multiple server instances or server restarts.

It's essential to consider security aspects, such as session hijacking and session fixation, when implementing session management. This includes using secure cookies, encrypting session data, and implementing measures to prevent session-related attacks. Additionally, compliance with privacy regulations, such as GDPR, may require careful handling of session data, including user consent and data retention policies.

Session vs cookie

Session and cookie are both mechanisms used in web development for managing user state, but they serve different purposes and have distinct characteristics:

1. **Cookie**:
   - A cookie is a small piece of data sent from a web server and stored in the user's browser. It's typically used to store user-specific information locally on the client side.
   - Cookies are sent with every HTTP request to the server, including subsequent requests to the same website. This allows the server to recognize returning users and maintain some level of statefulness.

- Cookies can have an expiration time, after which they are automatically deleted from the user's browser. They can also be set with flags like `httpOnly` and `secure` for additional security.
- Cookies are often used for purposes like session management, user authentication, tracking user behavior, and personalization.

2. **Session**:
- A session is a server-side storage mechanism for maintaining stateful information about a user's interaction with a web application across multiple requests.
- Sessions typically involve creating a unique session identifier (session ID) for each user, which is sent to the client, usually as a cookie. However, the session data itself is stored on the server side.
- Unlike cookies, session data is not directly accessible or modifiable by the client. Instead, the client sends the session ID with each request, allowing the server to retrieve the corresponding session data.
- Sessions are often used for purposes like user authentication, storing user preferences, maintaining shopping cart contents, and managing temporary data during a user's visit to a website.

In summary, the main differences between session and cookie are:

- **Location of Data**: Cookies store data on the client side (in the user's browser), while session data is stored on the server side.
- **Access and Security**: Session data is more secure because it is stored on the server side and is not directly accessible or modifiable by the client. Cookies, on the other hand, are accessible to and modifiable by the client.
- **Persistence**: Cookies can have an expiration time, but they persist until they are deleted or expired. Session data typically expires after a certain period of inactivity or when the user logs out.
- **Usage**: Cookies are often used for tracking user behavior, personalization, and session management. Sessions are primarily used for maintaining user state and managing user sessions on the server side.

Backup and restore utilities in Mongodb

MongoDB provides several backup and restore utilities to help manage data backups and recovery. These utilities offer different levels of granularity and flexibility depending on your backup requirements. Some of the commonly used backup and restore utilities in MongoDB include:

1. **mongodump**:
- `mongodump` is a utility used to create binary dump of the contents of a MongoDB database. It can be used to backup entire databases, collections, or specific documents.
- `mongodump` produces a binary dump of the data in BSON format, which can be easily restored using the `mongorestore` utility.

- **Example usage:**
  ```
  **mongodump --db mydatabase --out /backup/directory**
  ```

2. **mongorestore**:
   - `mongorestore` is the counterpart to `mongodump` and is used to restore data from binary dumps created with `mongodump`.
   - It reads BSON data files from a backup directory and restores them to the specified MongoDB database and/or collection.
   - **Example usage:**
  ```
  **mongorestore --db mydatabase /backup/directory/mydatabase**
  ```

3. **Snapshot Backups**:
   - MongoDB supports filesystem-level snapshot backups, which create a point-in-time snapshot of the entire MongoDB data directory.
   - These snapshots can be taken using filesystem-specific tools or backup software that supports snapshot functionality.
   - Snapshot backups provide a consistent view of the data at a specific point in time and can be used for both backup and disaster recovery purposes.
   - It's important to ensure that the MongoDB instance is quiesced (i.e., no write operations are in progress) before taking a filesystem snapshot to avoid data corruption.

4. **MongoDB Cloud Backup Service**:
   - MongoDB offers a Cloud Backup service that provides managed backup and restore capabilities for MongoDB Atlas clusters.
   - It allows you to schedule automated backups of your Atlas clusters and restore data to any point in time within the backup retention period.
   - The Cloud Backup service offers features like point-in-time recovery, continuous backups, and cross-region data redundancy to ensure data durability and availability.

These are some of the primary backup and restore utilities and strategies available in MongoDB. The choice of utility or strategy depends on factors like backup requirements, data size, performance considerations, and infrastructure setup. It's important to have a robust backup and recovery plan in place to protect against data loss and ensure business continuity.