NodeJS Training - Day 2

By : LAKSHMIKANT DESHPANDE (M.Tech)

Events and Streams

- Day 1 Recap
- Understanding Modules & Events
- Event Emitter
- Reading and Writing Streams
- Using Pipe()
- Duplex stream
- Caching and Object Creation
- Simple JavaScript Program Execution with Callbacks, Event Emitters & Loops

Session 1

- Day 1 Recap
    - GitHub Account Creation and Basic Git Commands
    - Javascript Basics
    - Functions
    - JSON
    - Callback Functions
    - Promise
    - Async/Await
    - Event Loop
    - ES 6

Session 1 contd….

- Understand JavaScript’s “this”
- Package.json
- Exception Handling

Session 2

- Modules
- Events & Event Emitter

Session 3

- Events & Event Emitter contd….
- Reading and Writing Streams
- Using Pipe()

Session 4

- Reading and Writing Streams contd….

Understand JavaScript’s “this”

- “**this**” is not assigned a value until an object invokes the function where ***this*** is defined.
- "**this**" is a special keyword in JavaScript that refers to the current execution context or the object on which a function is invoked.
- The value of "**this**" is determined dynamically based on how a function is called, rather than where it is defined.
- "**this**" can have different values in different contexts, leading to sometimes confusing behavior.
- Global Context:
  - Outside of any function or object, "**this**" refers to the global object.
  - In web browsers, the global object is the "**window**" object.
  - In Node.js, the global object is called "**global**."

package.json

- **name**: Specifies the unique name of the package for identification.
- **version**: Indicates the version of the package using semantic versioning.
- **description**: Provides a brief description of the package's purpose.
- **keywords**: Helps in categorizing and searching for the package on npm.
- **homepage**: Points to the URL of the project's homepage.
- **repository**: Specifies the version control repository's type and URL.
- **license**: Indicates the license under which the package is distributed.
- **author**: Gives the name and contact information of the package's author.
- **contributors**: Lists contributors' names and contact information.
- **files**: Determines which files and directories to include when publishing the package.
- **main**: Specifies the entry point of the package.
- **scripts**: Defines custom scripts for various npm commands.
- **dependencies**: Lists packages required for the package to run in production.
- **devDependencies**: Lists development-only packages needed during development.
- **peerDependencies**: Specifies packages that consumers must install explicitly.
- **engines**: Specifies compatible Node.js and npm versions for the package.

Exception/Error Handling

In programming, there can be two types of errors in the code:

- **Syntax Error**: Error in the syntax. For example, if you write consol.log('your result');, the above program throws a syntax error. The spelling of console is a mistake in the above code.
- **Runtime Error**: This type of error occurs during the execution of the program. For example, calling an invalid function or a variable.

Error handling, "try...catch…finally"

```
try {
   ... try to execute the code ...
} catch (err) {
   ... handle errors ...
} finally {
   ... execute always ...
}
```

try…catch contd….

- **try...catch only works for runtime errors**
    - For try...catch to work, the code must be runnable. In other words, it should be valid JavaScript.
    - It won’t work if the code is syntactically wrong

- **try...catch works synchronously**
    - If an exception happens in “scheduled” code, like in setTimeout, then try...catch won’t catch it

try…catch contd….

- **Error object**
  - When an error occurs, JavaScript generates an object containing the details about it. The object is then passed as an argument to catch

- **Throwing our own errors**
  - The throw operator generates an error.

Custom Errors

- Custom errors in JavaScript allows to define your own error types to handle specific situations in your code.
- You can create custom errors by using either the Error constructor or by extending the Error class.

When using the Error constructor approach:

  - Define a function to create your custom error, setting the name, message, and optionally stack properties.
  - Inherit from the Error prototype using Object.create(Error.prototype) and set the constructor property.
  - Throw the custom error using throw new CustomError('Your custom message').
  - Catch the custom error using catch and handle it accordingly.

Modules

- In Node.js, modules are a fundamental part of the architecture and enable code reusability, organization, and encapsulation.
- A module in Node.js is a file containing JavaScript code that is executed in its own scope, rather than in the global scope.
- Node.js uses the CommonJS module system, where each file is treated as a separate module.

Types of Modules in Node.js

Core Modules:

- Built into Node.js and can be used without any additional installation.
- Examples: fs, http, path, os, url.

Local Modules:

- Created locally in your application.
- Typically stored in separate files and directories within your project.

Third-Party Modules:

- Available through the Node Package Manager (NPM).
- Examples: express, lodash, axios.

Core Modules

Core modules are part of the Node.js runtime. Here are some commonly used core modules:

- **fs**: File System operations.
- **http**: HTTP server and client functionality.
- **path**: Utilities for handling file and directory paths.
- **os**: Operating system-related utility methods and properties.
- **url**: URL resolution and parsing.

Modules provide a way to:

- Organize code
- Promote reusability
- Avoid namespace conflicts
- Manage dependencies
- Facilitate collaboration
- Improve testing practices
- Support dynamic loading

Events

- In Node.js, events play a crucial role in handling asynchronous operations and building scalable applications. The event-driven architecture of Node.js is fundamental to its non-blocking, I/O-centric design.
- Events are signals or notifications that indicate the occurrence of certain actions or conditions. In Node.js, the event-driven model allows you to write code that reacts to these signals, such as when data is available, when an error occurs, or when a certain operation completes.
- At the core of event handling in Node.js is the EventEmitter class, which is part of the events module.

Events Summary

EventEmitter Class: Provides the core functionality for event handling in Node.js.

Real-Time Examples:

- **File Monitoring**: Using fs.watchFile to detect file changes.
- **HTTP Requests**: Handling incoming requests asynchronously.
- **Messaging System**: Communicating between different parts of an application.
- **Error Handling**: Emitting and handling errors.
- **Real-Time Data Processing**: Simulating data from sensors.

Event Emitter

The EventEmitter class in Node.js is a fundamental component of the event-driven architecture of Node.js. It allows objects to emit and listen for events, facilitating asynchronous programming by enabling code to respond to various events.

Core Concepts

- **Event Emission**: You can emit named events with associated data.
- **Event Listening**: You can listen for and respond to emitted events.
- **Event Handling**: You can add, remove, and manage listeners for specific events.

Event Loop

The Event Loop is a fundamental concept in Node.js (and JavaScript in general) that allows for non-blocking, asynchronous programming. It enables Node.js to handle multiple operations concurrently, such as I/O operations, network requests, and timers, without blocking the execution of the program.

Streams

Streams are a powerful concept in Node.js that allow you to work with large amounts of data efficiently by processing it in chunks rather than loading it all into memory at once.

This is especially useful for tasks like reading/writing files, handling HTTP requests/responses, or working with any kind of data that is too large to fit in memory.

Types of Streams

1. **Readable Streams**: Streams from which data can be read. For example, fs.createReadStream to read files.
2. **Writable Streams**: Streams to which data can be written. For example, fs.createWriteStream to write files.
3. **Duplex Streams**: Streams that are both readable and writable. For example, TCP sockets.
4. **Transform Streams**: Duplex streams that can modify or transform the data as it is written and read. For example, zlib streams for compression.