

UNIT-I

1. Introduction to Node.js

Ans: Node.js is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside of a web browser. It is built on the Chrome V8 JavaScript engine, which makes it fast and efficient. Node.js was first released in 2009 by Ryan Dahl and has since grown into a popular platform for building server-side applications. Node.js has several features that make it a popular choice for building server-side applications:

- **Asynchronous programming model:** Node.js uses an event-driven, non-blocking I/O model that allows for efficient processing of multiple requests simultaneously without blocking the main thread. This enables the server to handle a large number of requests without slowing down or crashing.
- **Fast and scalable:** Node.js is built on the V8 engine, which is the same engine used by Google Chrome. This makes Node.js very fast and efficient, and it can handle a large number of concurrent connections with ease.
- **Cross-platform:** Node.js is designed to be cross-platform, which means it can run on Windows, macOS, and Linux without any modifications.
- **Large ecosystem:** Node.js has a large and active ecosystem of developers who contribute to a wide range of modules and packages that can be easily installed and used through the npm package manager.
- **Real-time applications:** Node.js is well-suited for building real-time applications such as chatbots, gaming servers, and streaming applications.
- **Easy to learn:** Node.js is built on JavaScript, which is a widely-used language with a large community of developers. This makes it easy for developers to learn and start building applications quickly.
- **Easy to deploy:** Node.js applications can be easily deployed to cloud platforms such as AWS, Google Cloud, and Microsoft Azure, making it easy to scale and manage applications in the cloud.

2. Installing Node.js

Ans: To install Node.js, you need a Node.js Binary installable software on your computer. To download the latest version of Node.js installable archive file, visit <https://nodejs.org/en/> and click on 18.15.0 LTS. After downloading is completed run the .exe file, a Setup Wizard is displayed and click Next. Accept the terms of license agreement and click Next. Choose the location where you want to install and click Next. Select the features you want to be installed and then click Next. Click Install to begin the installation. After the installation click Finish to close the setup wizard. In order to check whether it is installed or not, open cmd and type **node -version**. If you see an output with the version number, then it is successfully installed.

3. io.js and the Node.js foundation

Ans: In late 2014, a group of Node.js developers forked the Node.js project to create io.js. The fork was created to address some of the governance and technical issues that the developers felt were not being adequately addressed in the Node.js project. Io.js was based on the same codebase as Node.js and included many new features and improvements. In 2015, the Node.js Foundation was created as a collaborative project to bring together the developers behind Node.js and io.js. The goal of the foundation was to provide a neutral and open governance structure for the development of Node.js and to foster a more inclusive and diverse community. Today, the Node.js Foundation is responsible for the ongoing development of Node.js, and it is supported by a large and diverse community of developers and organizations.

4. Node.js LTS support

Ans: Node.js has a Long Term Support (LTS) schedule that provides stability and security for enterprise and production applications. The LTS schedule offers regular releases that receive support for a certain period of time. The Node.js LTS release line is typically designated as even-numbered versions, such as Node.js 10, 12, 14, and so on. In contrast, the odd-numbered versions, such as Node.js 11, 13, and 15, are not designated as LTS releases and only receive 6 months of support.

5. Node.js ES6 support

Ans: Node.js has been supporting ES6 (ECMAScript 2015) features since version 4.0, which was released in 2015. Since then, Node.js has been continuously adding support for new ES6 features in subsequent releases. Some of the ES6 features that are supported in Node.js are Classes, let and const keywords etc. Node.js also supports some of the newer ES6 features such as ES6 modules and BigInts, which were introduced in later versions of ECMAScript.

6. JavaScript event driven programming

Ans: JavaScript is an event-driven programming language, which means that the code is executed in response to events triggered by user actions. In event-driven programming, the flow of the program is determined by events and event handlers. When an event occurs, the program executes a specific function, called an event handler that is associated with that event. The event handler then performs a specific action or responds to the event in some way. In JavaScript, events are often associated with user interactions, such as mouse clicks, keyboard inputs. Events can also be triggered by other asynchronous actions, such as data requests or server responses. To handle events in JavaScript, developers can use event listeners, which are functions that are executed when a particular event occurs.

Ex:

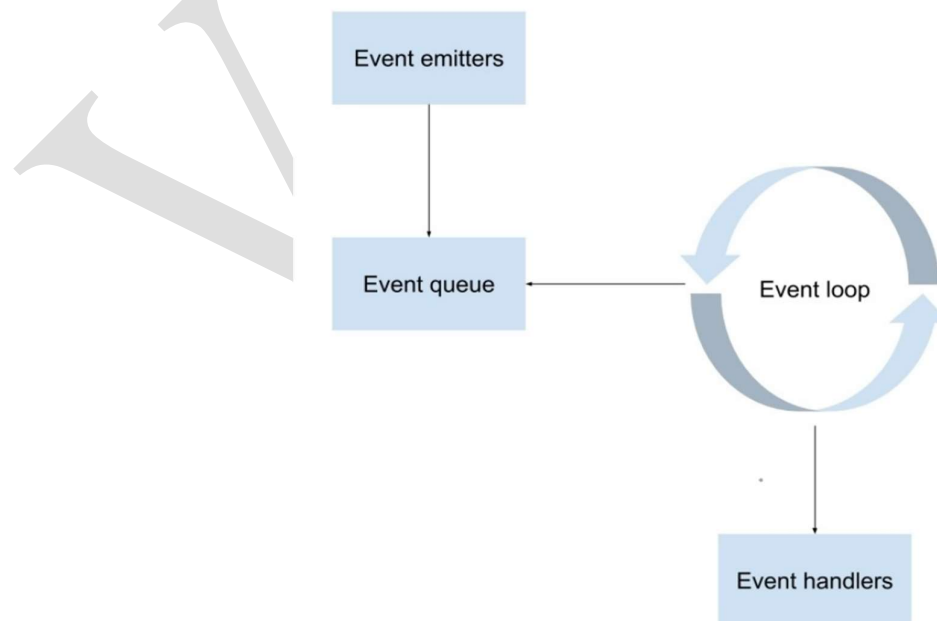
To handle a button click event, `addEventListener()` method is used.

Program:

```
const button = document.querySelector('button');
button.addEventListener('click', function() {
  console.log('Button clicked!');
});
```

7. Node.js event driven programming.

Ans:



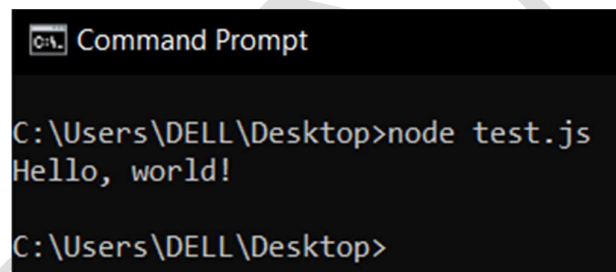
Node.js is a platform built on top of Chrome's V8 JavaScript engine that enables event-driven programming for server-side applications. In Node.js, event-driven programming is implemented using an event loop that listens for and dispatches events. When a Node.js application is started, it enters an event loop, which continuously waits for events to occur. When an event is triggered, such as a new request being received by a web server, Node.js executes the associated event handler function. The event handler then performs some action, such as generating a response to the request, and may also emit new events, which are dispatched to their own event handlers. One of the key features of Node.js is its non-blocking I/O model, which allows multiple I/O operations to be performed simultaneously without blocking the event loop. To handle events in Node.js, we can use Node.js built-in EventEmitter class, which provides a simple and flexible mechanism for implementing event-driven programming. The EventEmitter class provides methods for registering event listeners and emitting events, which can be used to build custom event-driven applications.

Ex: Creation of a custom event emitter that emits a "hello" event when a function is called

Program:

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();
myEmitter.on('hello', () => {
    console.log('Hello, world!');
});
myEmitter.emit('hello');
```

Output:

A screenshot of a Windows Command Prompt window. The title bar says "C:\> Command Prompt". The command prompt shows the following text: "C:\Users\DELL\Desktop>node test.js", followed by the output "Hello, world!", and then the prompt "C:\Users\DELL\Desktop>".

```
C:\> Command Prompt

C:\Users\DELL\Desktop>node test.js
Hello, world!

C:\Users\DELL\Desktop>
```

8. CommonJS modules

Ans: CommonJS is a specification for modular JavaScript programming, which defines a standard way for creating and consuming modules in JavaScript. CommonJS modules were initially developed for use in server-side JavaScript environments like Node.js, but they can also be used in client-side JavaScript applications with the help of module bundlers like webpack. The CommonJS standards specify the following key components when working with modules:

- **require():** A method that is used to load the module into your code.
- **exports:** An object that is used to export functions, objects, or other values from a module.
- **module:** An object that represents the current module being executed and is used to provide metadata information about the module.

9. Node.js core modules

Ans: In Node.js, there are several built-in modules that provide common functionality for various tasks. Here are some of the most commonly used core modules:

- **fs:** This module provides file system-related functionality for reading, writing, and manipulating files and directories.
- **http:** This module provides functionality for creating HTTP servers and clients.
- **https:** This module provides similar functionality to http, but for HTTPS servers and clients.

10. Node.js third-party modules

Ans: In addition to the core modules in Node.js, there are also many third-party modules available through the Node Package Manager (NPM) registry. These modules are typically created and maintained by the Node.js community and provide additional functionality and tools for various tasks. Here are some examples of commonly used third-party modules:

- **express:** This is a popular web framework for Node.js, used for building web applications and APIs. It provides a variety of features for handling HTTP requests and responses, routing, middleware, and more.
- **request:** This module provides a simple and flexible API for making HTTP requests from Node.js.

11. Node.js file modules

Ans: In Node.js, file modules are used to work with files and directories. The core file module in Node.js is the fs (file system) module, which provides methods for reading, writing, and manipulating files and directories. Here are some examples of file modules:

- **path:** This module provides methods for working with file paths and directories.
- **glob:** This module provides a way to search for files using glob patterns, which are a way to match file paths using wildcard characters.

12. Node.js folder modules

Ans: In Node.js, it's possible to require folder modules in the same way as file modules. When requiring a folder module, Node.js will look for a package.json file inside the folder. If the package.json file is found and specifies a main property, Node.js will load the specified file. Otherwise, Node.js will automatically look for an index.js file. Folder modules can be useful for organizing code into logical groups, and they can be particularly helpful for third-party module authors. Node.js modules have proven to be an effective solution for building complex JavaScript applications, and npm has made it easy to find and install third-party modules created by the community.

Ex: Let's say we have a folder named utils, and inside that folder we have a file named math.js that exports some utility functions. To use these utility functions in another file, we can require the utils folder and access the exported functions from the math.js file.

Program:

```
// ./utils/math.js
function add(a, b) {
  return a + b;
}
function subtract(a, b) {
  return a - b;
}
module.exports = {
  add,
  subtract
};

// ./app.js
const math = require('./utils');
console.log(math.add(2, 3));
console.log(math.subtract(5, 3));
```

Output: Without having package.json file

```
Command Prompt
C:\Users\DELL\Desktop>cd utils
C:\Users\DELL\Desktop\utils>node math.js
C:\Users\DELL\Desktop\utils>cd ..
C:\Users\DELL\Desktop>node app.js
node:internal/modules/cjs/loader:959
  throw err;
  ^
Error: Cannot find module './utils'
Require stack:
- C:\Users\DELL\Desktop\app.js
    at Function.Module._resolveFilename (node:internal/modules/cjs/loader:956:15)
    at Function.Module._load (node:internal/modules/cjs/loader:804:27)
    at Module.require (node:internal/modules/cjs/loader:1028:19)
    at require (node:internal/modules/cjs/helpers:102:18)
    at Object.<anonymous> ([C:\Users\DELL\Desktop\app.js:1:14])
    at Module._compile (node:internal/modules/cjs/loader:1126:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1180:10)
    at Module.load (node:internal/modules/cjs/loader:1004:32)
    at Function.Module._load (node:internal/modules/cjs/loader:839:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at code: [MODULE_NOT_FOUND]
    at requireStack: [C:\Users\DELL\Desktop\app.js]
```

Output: After creating package.json file

```
Command Prompt
C:\Users\DELL\Desktop>cd utils
C:\Users\DELL\Desktop\utils>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

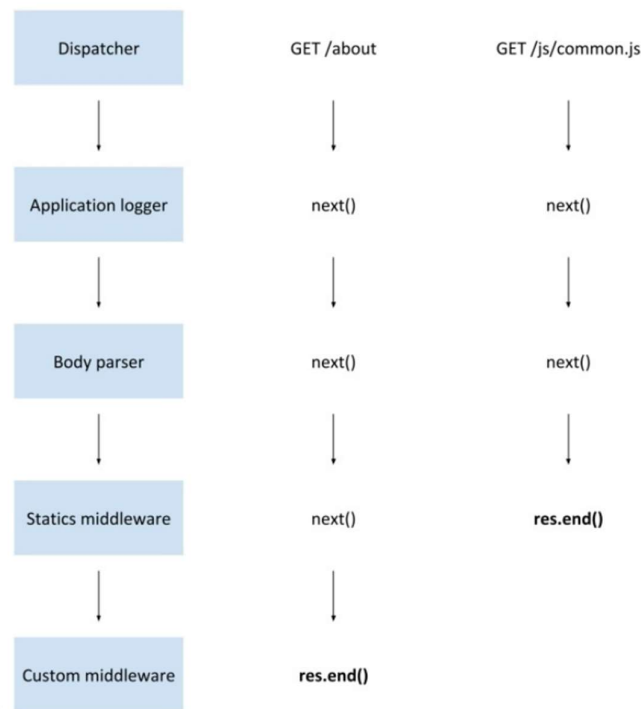
Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (utils)
version: (1.0.0)
description: test json
entry point: (math.js)
test command: test
git repository:
keywords:
author: vinny
license: (ISC)
About to write to C:\Users\DELL\Desktop\utils\package.json:
{
  "name": "utils",
  "version": "1.0.0",
  "description": "test json",
  "main": "math.js",
  "scripts": {
    "test": "test"
  },
  "author": "vinny",
  "license": "ISC"
}

Is this OK? (yes)
C:\Users\DELL\Desktop\utils>node math.js
C:\Users\DELL\Desktop\utils>cd ..
C:\Users\DELL\Desktop>node app.js
5
2
```

13. Meet the Connect module

Ans: Connect is a Node.js module that allows you to intercept HTTP requests and handle them in a modular way. Instead of writing code to manage each HTTP request individually, you can use Connect middleware components to handle different scenarios. Connect middleware components are just functions that get executed when an HTTP request occurs. They can perform some logic, return a response, or call the next middleware component. Connect also includes some pre-built middleware components that can handle common tasks such as logging and serving static files. The Connect application works by using a dispatcher object that decides the order of middleware component execution in a cascading form.



Dispatcher: The dispatcher in Connect handles each HTTP request received by the server and then decides the order of middleware component execution in a cascading form.

Application Logger: It is a pre-built middleware component that logs information about incoming HTTP requests and their corresponding responses.

Body Parser: It is a pre-built middleware component that is used to extract the data from the request body and parse it into a usable format such as JSON, XML, or plain text.

Statics Middleware: It is a pre-built middleware component that allows you to serve static files such as HTML, CSS, and JavaScript files.

Custom Middleware: Custom middleware components are functions that you write to handle specific application logic.

14. Connect middleware

Ans: A Connect middleware is a function in JavaScript that accepts three arguments: req (request), res (response), and next. req holds information about the incoming HTTP request, res allows you to set the response properties for the HTTP response, and next is a callback function that executes the next middleware in the sequence. To use a middleware in your application, you just need to register it using the use() method. This method tells Connect to use the middleware function in its request handling process. For example, if you want to add a middleware that logs the incoming requests, you can define a

function that takes req, res, and next as arguments and logs the incoming request details. Then, you can register this middleware using use() method so that it will be executed for every incoming request.

15. Understanding the order of Connect middleware

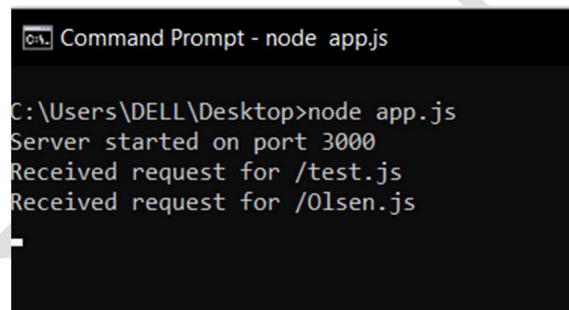
Ans: In each middleware function, you can decide whether to call the next middleware function or stop at the current one. Connect executes middleware functions in a FIFO (First-In-First-Out) manner until there are no more middleware functions to execute or the next middleware function is not called. One of Connect's greatest features is its flexibility when writing applications. You can add as many middleware functions as you need to handle different scenarios.

Ex: Add a logger function to log all requests made to the server in the command line.

Program:

```
const connect = require('connect');
function logger(req, res, next) {
  console.log(`Received request for ${req.url}`);
  next();
}
const app = connect();
app.use(logger);
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

Output:



```
Command Prompt - node app.js
C:\Users\DELL\Desktop>node app.js
Server started on port 3000
Received request for /test.js
Received request for /olsen.js
```

16. Mounting Connect middleware.

Ans: Mounting Connect middleware means adding one or more middleware functions to a specific path in the application. This is useful when you want to apply certain middleware only to a specific section of your application rather than the entire application. To mount middleware in Connect, you can use the use() method and pass in a path as the first argument before the middleware function.

Ex: Apply a logger middleware function only to the /admin path of your application

Program:

```
const connect = require('connect');
function logger(req, res, next) {
  console.log(`Received request for ${req.url}`);
  next();
}
function admin(req, res, next) {
  console.log('Admin section accessed');
  next();
}
const app = connect();
app.use('/admin', logger, admin);
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```