UNIT II

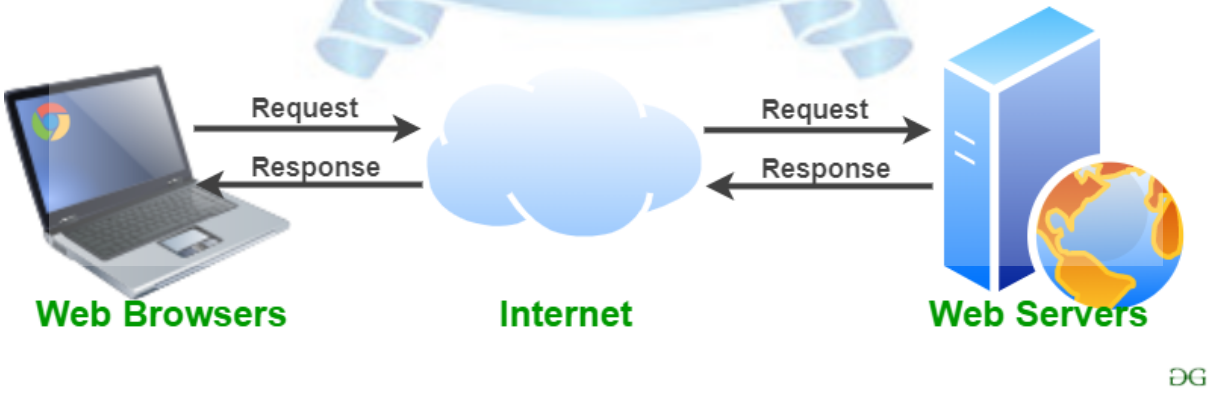
SERVER SIDE PROGRAMMING WITH NODE JS

2.1 INRODUCTION TO WEB SERVER

Web Server:

Web server is a program which processes the network requests of the users and serves them with files that create web pages. This exchange takes place using Hypertext Transfer Protocol (HTTP).

Basically, web servers are computers used to store HTTP files which makes a website and when a client requests a certain website, it delivers the requested website to the client. For example, you want to open Facebook on your laptop and enter the URL in the search bar of google. Now, the laptop will send an HTTP request to view the facebook webpage to another computer known as the webserver. This computer (webserver) contains all the files (usually in HTTP format) which make up the website like text, images, gif files, etc. After processing the request, the webserver will send the requested website-related files to your computer and then you can reach the website.



There are many web servers available in the market both free and paid. Some of them are described below:

- **Apache HTTP server:** It is the most popular web server and about 60 percent of the world's web server machines run this web server. The Apache HTTP web server was developed by the Apache Software Foundation. It is an open-source software which means that we can access and make changes to its code and mold it according to our preference. The Apache Web Server can be installed and operated easily on almost all operating systems like Linux, MacOS, Windows, etc.

- Microsoft Internet Information Services (IIS): IIS (Internet Information Services) is a high performing web server developed by Microsoft. It is strongly united with the operating system and is therefore relatively easier to administer. It is developed by Microsoft, it has a good customer support system which is easier to access if we encounter any issue with the server. It has all the features of the Apache HTTP Server except that it is not an open-source software and therefore its code is inaccessible which means that we cannot make changes in the code to suit our needs. It can be easily installed in any Windows device.

- **Lighttpd:** Lighttpd is pronounced as 'Lightly'. It currently runs about 0.1 percent of the world's websites. Lighttpd has a small CPU load and is therefore comparatively easier to run. It has a low memory footprint and hence in comparison to the other web servers, requires less memory space to run which is always an advantage. It also has speed optimizations which means that we can optimize or change its speed according to our requirements. It is an open-source software which means that we can access its code and add changes to it according to our needs and then upload our own module (the changed code).

2.2 JAVASCRIPT IN THE DESKTOP WITH NODE JS

What is Node.JS?

Node.js is an open-source, cross-platform JavaScript **runtime environment** that allows developers to run JavaScript code on the server side. Created by Ryan Dahl in 2009, Node.js has revolutionized server-side programming by offering an efficient, event-driven, and non-blocking I/O model.

It's a powerful tool used for various types of projects. Let's explore some key aspects:

- **JavaScript Runtime**: Node.js runs on the **V8 JavaScript engine**, which is also the core engine behind Google Chrome.
- **Single Process Model**: A Node.js application operates within a **single process**, avoiding the need to create a new thread for every request.
- **Asynchronous I/O**: Node.js provides a set of **asynchronous I/O primitives** in its standard library. These primitives prevent JavaScript code from blocking, making non-blocking behavior the norm.
- **Concurrency Handling**: Node.js efficiently handles **thousands of concurrent connections** using a single server. It avoids the complexities of managing thread concurrency, which can lead to bugs.

- **JavaScript Everywhere**: Frontend developers familiar with JavaScript can seamlessly transition to writing server-side code using Node.js.
- **ECMAScript Standards**: Node.js supports the latest ECMAScript standards. You can choose the version you want to use, independent of users' browser updates.

Why Node.JS?

Node.js is used to build back-end services like **APIs** like Web App, Mobile App or Web Server. A Web Server will open a file on the server and return the content to the client. It's used in production by large companies such as **Paypal**, **Uber**, **Netflix**, **Walmart**, and so on.

Reasons to Choose Node.js

- **Easy to Get Started**: Node.js is beginner-friendly and ideal for prototyping and agile development.
- **Scalability**: It scales both horizontally and vertically.
- **Real-Time Web Apps**: Node.js excels in real-time synchronization.
- **Fast Suite**: It handles operations quickly (e.g., database access, network connections).
- **Unified Language**: JavaScript everywhere—frontend and backend.
- **Rich Ecosystem**: Node.js boasts a large open-source library and supports asynchronous, non-blocking programming.

PHP and ASP handling file requests:

Send Task -> Waits -> Returns -> Ready for Next Task

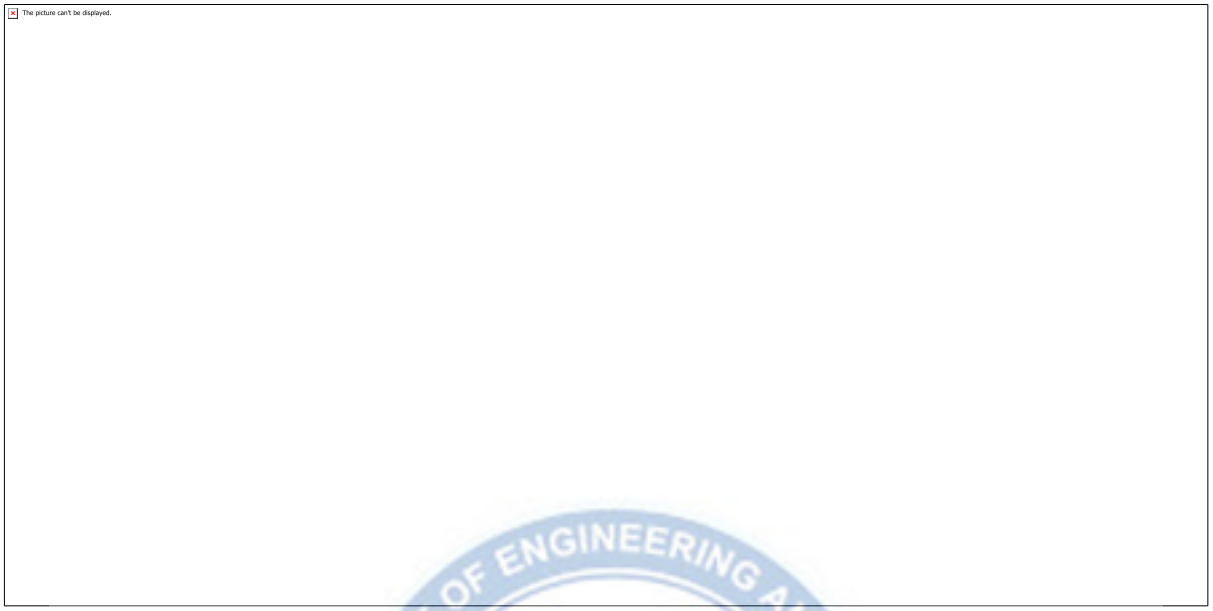
Node.js handling file request:

Send Task -> Returns -> Ready for Next Task

Node.js takes requests from users, processes those requests, and returns responses to the corresponding users, there is no Wait for open and read file phase in Node.js.

Basic Concepts of Node.JS

The following diagram depicts some important parts of Node.js that are useful and help us understand it better.



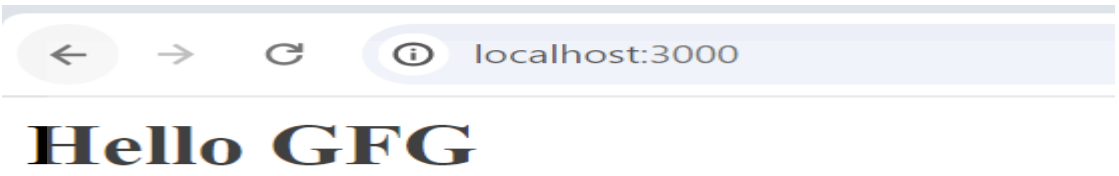
Node.js Example to Create Web Server

It is the basic code example to create node.js server.

```
// Importing the http module
const http = require('http');

// Creating a server
const server = http.createServer((req, res) => {
    // Setting the content type to HTML
    res.writeHead(200, {
        'Content-Type': 'text/html'
    });

    // Sending the HTML response
    res.end('<h1>Hello GFG</h1>');
});

// Listening on port 3000
```

```
const PORT = 3000;
server.listen(PORT, () => {
    console.log(`Server running at http://localhost:${PORT}/`);
});
```

Output:



Example of Node.js Server Output

Code Explaination:

- We use the **HTTP module** to create an **HTTP server.**
- The server listens on the specified **port** and **hostname**.
- When a new request arrives, the callback function handles it by setting the response status, headers, and content.

How Node.JS Works?

Node.js accepts the request from the clients and sends the response, while working with the request node.js handles them with a single thread. To operate I/O operations or requests node.js use the concept of threads. Thread is a sequence of instructions that the server needs to perform. It runs parallel on the server to provide the information to multiple clients. Node.js is an event loop single-threaded language. It can handle concurrent requests with a single thread without blocking it for one request.

Advantages of Node.JS

- **Easy Scalability:** Easily scalable the application in both horizontal and vertical directions.

- **Real-time web apps:** Node.js is much more preferable because of faster synchronization. Also, the event loop avoids HTTP overloaded for Node.js development.
- **Fast Suite:** NodeJS acts like a fast suite and all the operations can be done quickly like reading or writing in the database, network connection, or file system
- **Easy to learn and code:** NodeJS is easy to learn and code because it uses JavaScript.
- **Advantage of Caching:** It provides the caching of a single module. Whenever there is any request for the first module, it gets cached in the application memory, so you don't need to re-execute the code.

What is Node.JS file?

Node.js files contain tasks that handle file operations like **creating**, **reading**, **deleting**, etc., Node.js provides an inbuilt module called FS (File System).

Application of Node.JS

Node.js is suitable for various applications, including:

- Real-time chats
- Complex single-page applications
- Real-time collaboration tools
- Streaming apps
- JSON APIs

Common Use Cases of Node.JS

Node.js is versatile and finds applications in various domains:

1. **Web Servers**: Node.js excels at building lightweight and efficient web servers. Its non-blocking I/O model makes it ideal for handling concurrent connections.
2. **APIs and Microservices**: Many companies use Node.js to create RESTful APIs and microservices. Express.js simplifies API development.
3. **Real-Time Applications**: Node.js shines in real-time scenarios like chat applications, live notifications, and collaborative tools. Socket.io facilitates real-time communication.

4. **Single-Page Applications (SPAs)**: SPAs benefit from Node.js for server-side rendering (SSR) and handling API requests.
5. **Streaming Services**: Node.js is well-suited for streaming data, whether it's video, audio, or real-time analytics.

Node.JS Ecosystem

Node.js has a vibrant ecosystem with a plethora of libraries, frameworks, and tools. Here are some key components:

1. **npm (Node Package Manager)**: npm is the default package manager for Node.js. It allows developers to install, manage, and share reusable code packages (called modules). You can find thousands of open-source packages on the npm registry.
2. **Express.js**: Express is a popular web application framework for Node.js. It simplifies routing, middleware handling, and request/response management. Many developers choose Express for building APIs, web servers, and single-page applications.
3. **Socket.io**: For real-time communication, Socket.io is a go-to library. It enables bidirectional communication between the server and clients using WebSockets or fallback mechanisms.
4. **Mongoose**: If you're working with MongoDB (a NoSQL database), Mongoose provides an elegant way to model your data and interact with the database. It offers schema validation, middleware, and query building.

2.3 NODE PACKAGE MANAGER

NPM (Node Package Manager) is a package manager for Node.js modules. It helps developers manage project dependencies, scripts, and third-party libraries. By installing Node.js on your system, NPM is automatically installed, and ready to use.

- It is primarily used to manage packages or modules—these are pre-built pieces of code that extend the functionality of your Node.js application.
- The NPM registry hosts millions of free packages that you can download and use in your project.
- NPM is installed automatically when you install Node.js, so you don't need to set it up manually.

How to Use NPM with Node.js?

To start using NPM in your project, follow these simple steps

Step 1: Install Node.js and NPM

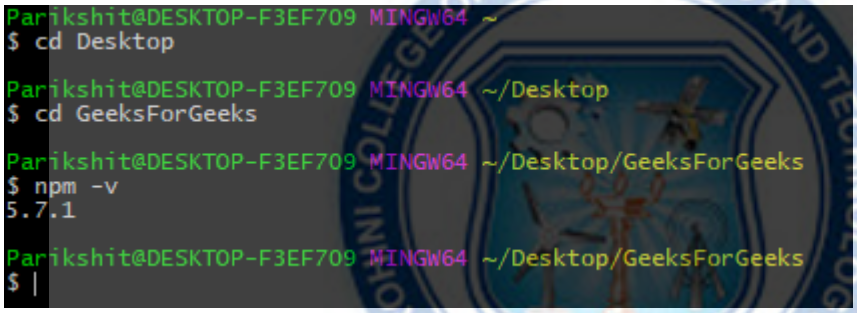
First, you need to install Node.js. NPM is bundled with the Node.js installation. You can follow our article to Install the Node and NPM- How to install Node on your system

Step 2: Verify the Installation

After installation, verify Node.js and NPM are installed by running the following commands in your terminal:

node -v
npm -v

These commands will show the installed versions of Node.js and NPM.



NodeJS NPM Version

Step 3: Initialize a New Node.js Project

In the terminal, navigate to your project directory and run:

npm init -y

This will create a package.json file, which stores metadata about your project, including dependencies and scripts.

Step 4: Install Packages with NPM

To install a package, use the following command

npm install <package-name>

For example, to install the Express.js framework

npm install express

This will add express to the node_modules folder and automatically update the package.json file with the installed package information.

Step 5: Install Packages Globally

To install packages that you want to use across multiple projects, use the -g flag:

npm install -g <package-name>

Step 6: Run Scripts

You can also define custom scripts in the package.json file under the "scripts" section. For example:

```
{
  "scripts": {
    "start": "node app.js"
  }
}
```

Then, run the script with

npm start

Using NPM Package in the project

Create a file named **app.js** in the project directory to use the package

```
//app.js
const express = require('express');//import the required package
const app = express();
app.get('/', (req, res) => {
  res.send('Hello, World!');
});
app.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

- **express()** creates an instance of the Express app.
- **app.get()** defines a route handler for HTTP GET requests to the root (/) URL.
- **res.send()** sends the response "Hello, World!" to the client.
- **app.listen(3000)** starts the server on port 3000, and console.log() outputs the server URL.

Now run the application with

node app.js

Visit **http://localhost:3000** in your browser, and you should see the message: **Hello, World!**

Managing Project Dependencies

1. Installing All Dependencies

In a Node.js project, dependencies are stored in a package.json file. To install all dependencies listed in the file, run:

npm install

This will download all required packages and place them in the node_modules folder.

2. Installing a Specific Package

To install a specific package, use:

npm install <package-name>

You can also install a package as a development dependency using:

npm install <package-name> --save-dev

Development dependencies are packages needed only during development, such as testing libraries.

To install a package and simultaneously save it in package.json file (in case using Node.js), add **–save** flag. The **–save** flag is default in npm install command so it is equal to **npm install package_name** command.

Example:

npm install express --save

Usage of Flags:

- **–save:** flag one can control where the packages are to be installed.
- **–save-prod :** Using this packages will appear in Dependencies which is also by default.
- **–save-dev :** Using this packages will get appear in devDependencies and will only be used in the development mode.

Note: If there is a package.json file with all the packages mentioned as dependencies already, just type npm install in terminal

3. Updating Packages

You can easily update packages in your project using the following command

npm update

This will update all packages to their latest compatible versions based on the version constraints in the package.json file.

To update a specific package, run

npm update <package-name>

4. Uninstalling Packages

To uninstall packages using npm, follow the below syntax:

npm uninstall <package-name>

For uninstall Global Packages

npm uninstall package_name -g

Popular NPM Packages

NPM has a massive library of packages. Here are a few popular packages that can enhance your Node.js applications:

- **Express**: A fast, minimal web framework for building APIs and web applications.
- **Mongoose**: A MongoDB object modeling tool for Node.js.
- **Lodash**: A utility library delivering consistency, customization, and performance.
- **Axios**: A promise-based HTTP client for making HTTP requests.
- **React**: A popular front-end library used to build user interfaces

2.4 SERVING FILES WITH HTTP MODULES

Node.js HTTP module allows you to create and manage web servers and handle HTTP requests and responses. Unlike other frameworks like Express.js, the HTTP module is built directly into Node.js, meaning you don't need to install any additional packages. It provides low-level control over HTTP requests and responses.

What is HTTP Module?

The HTTP module in Node.js provides functionality to create and manage HTTP servers and clients. It includes methods to handle incoming requests, send responses, and interact with HTTP headers and status codes. Since it's a built-in module, you can use it without installing any third-party libraries.

The HTTP module is fundamental to Node.js and can be used as the backbone for web servers, though many developers prefer to use higher-level frameworks like Express.js to simplify the process.

Syntax:

const http = require('http');

Creating Servers:

The HTTP module allows you to create a server using the http.createServer() method, which listens for incoming requests and handles them using a callback function.

Handling Requests:

You can handle HTTP requests and responses by accessing the request and response objects within the callback function of createServer(). The request object contains data from the client, while the response object is used to send data back.

Making Requests:

The HTTP module also provides methods for making HTTP requests from a Node.js application. You can use http.request() or http.get() to send requests to other servers and

Features

- Easily create and configure HTTP servers with custom logic.
- Handle HTTP methods, headers, and data streams.
- No need for external libraries to handle HTTP in Node.js.

// Filename: max.js

const http = require('http');

```
// Create a server
http.createServer((request, response) => {

    // Sends a chunk of the response body
    response.write('Hello World!');

    // Signals the server that all of
    // the response headers and body
    // have been sent
    response.end();
}).listen(3000); // Server listening on port 3000

console.log("Server started on port 3000");
```

Step to run this program: Run this **max.js** file using the below command:

node max.js

Output:

```
Hello World!
```

Node.js HTTP Module

To make requests via the HTTP module **http.request()** method is used.

Syntax:

http.request(options[, callback])

2.5 INTRODUCTION TO EXPRESS

Express is a small framework that sits on top of Node.js's web server functionality to simplify its APIs and add helpful new features.It makes it easier to organize your application's functionality with middle ware and routing; it adds helpful utilities to Node.js's HTTP objects;it facilitates the rendering of dynamic HTTP objects.

Express is a part of MEAN stack, a full stack JavaScript solution used in building fast, robust, and maintainable production web applications.

MongoDB(Database)

ExpressJS(Web Framework)

AngularJS(Front-end Framework)

NodeJS(Application Server)

Installing Express on Windows (WINDOWS 10)

Assuming that you have installed node.js on your system, the following steps should be followed to install express on your Windows:

STEP-1: Creating a directory for our project and make that our working directory.

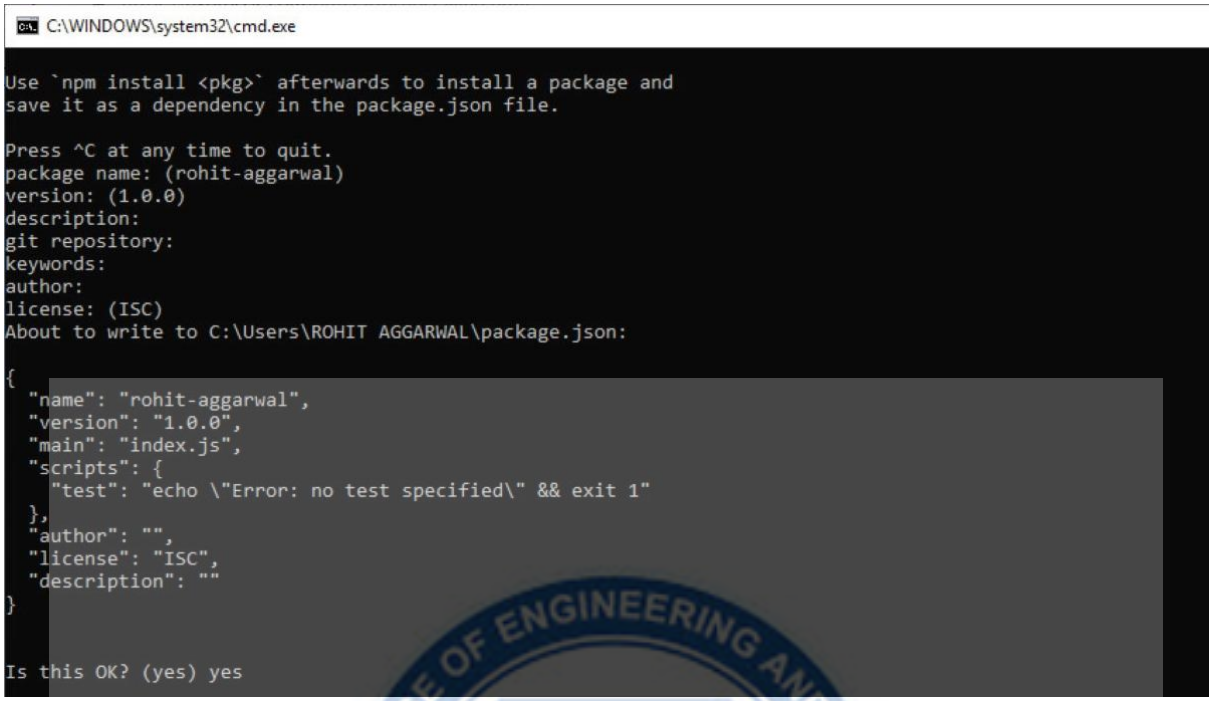
$ mkdir gfg

$ cd gfg

STEP-2: Using npm init command to create a package.json file for our project.

$ npm init

This command describes all the dependencies of our project. The file will be updated when adding further dependencies during the development process, for example when you set up your build system.

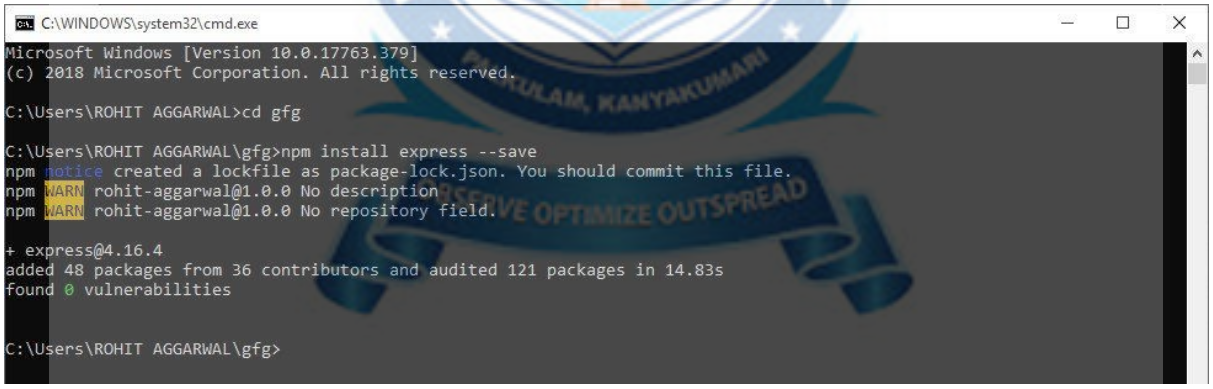


Keep pressing enter and enter "yes/no" accordingly at the terminus line.

STEP-3: Installing Express

Now in your *gfg(name of your folder)* folder type the following command line:

$ npm install express --save

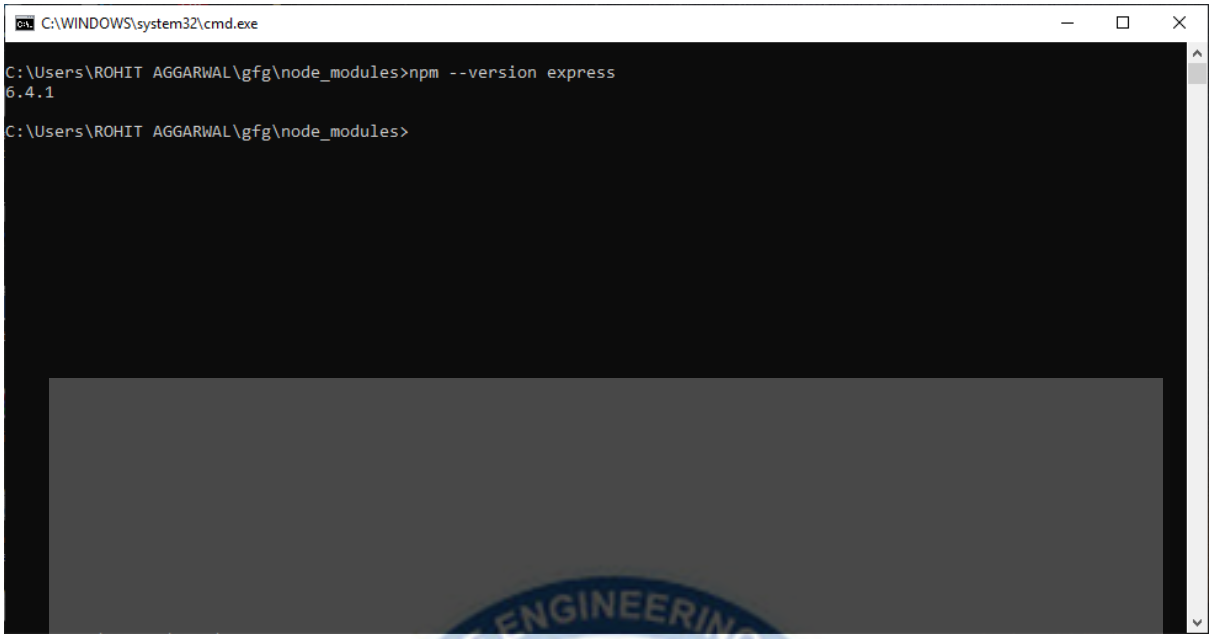


NOTE- Here *"WARN"* indicates the fields that must be entered in STEP-2.

STEP-4: Verify that Express.js was installed on your Windows:

To check that express.js was installed on your system or not, you can run the following command line on cmd:

C:\Users\Admin\gfg\node_modules>npm --version express



The version of express.js will be displayed on successful installation.

2.6 SERVER SIDE RENDERING WITH TEMPLATING ENGINES

Server-side rendering involves generating HTML on the server and sending it to the client, as opposed to generating it on the client side using JavaScript. This improves initial load time, and SEO, and enables dynamic content generation. Express is a popular web application framework for NodeJS, and EJS is a simple templating language that lets you generate HTML with plain JavaScript

Basic Approach:

- Using Express with EJS templating engine to render HTML on the server side.
- In this approach, we set up an Express server with EJS as the templating engine. We define routes and render EJS templates directly. Data can be passed to the templates as variables.

Advanced Approach (with Data Fetching):

- Utilizing asynchronous data fetching within Express routes to dynamically render content with EJS.
- This approach extends the basic one by incorporating asynchronous data fetching within Express routes. This could involve fetching data from databases, external APIs, or other sources before rendering the EJS templates.

Steps to Create Application (And Installing Required Modules):

Step 1: Create a new directory for your project:

mkdir express-ssr

Step 2: Navigate into the project directory:

cd express-ssr

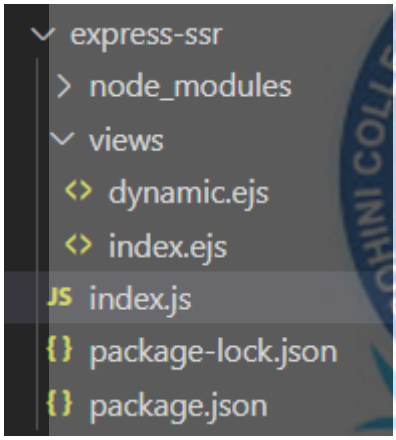
Step 3: Initialize npm (Node Package Manager) to create a package.json file:

npm init -y

Step 4: Install required modules (Express and EJS) using npm:

npm install express ejs

Project Structure:



The updated dependencies in **package.json** file will look like**:**

```
"dependencies": {
   "ejs": "^3.1.9",
   "express": "^4.18.3"
}
```

Example: Below is an example of ServerSide Rendering With Express and EJS Templates.

//views/index.ejs

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Express SSR with EJS</title>
</head>
<body>
    <h1>Hello, <%= name %>!</h1>
</body>
</html>
```

Start your server using the following command.

node index.js

Hello, World!

2.7 STATIC FILES

Introduction

A Node.js framework, Express facilitates data in a server and includes rendering your static files on the client-side such as images, HTML, CSS, and JavaScript.

Step 1 — Setting up Express

To begin, run the following in your terminal:

Create a new directory for your project named express-static-file-tutorial:

```
mkdir express-static-file-tutorial
```

Change into your new directory:

```
cd express-static-file-tutorial
```

Initialize a new Node project with defaults. This will set a package.json file to access your dependencies:

```
npm init -y
```

Create your entry file, index.js. This is where you will store your Express server:

```
touch index.js
```

Install Express as a dependency:

```
npm install express --save
```

Within your package.json, update your start script to include node and your index.js file.

package.json

```
{
  "name": "express-static-file-tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "keywords": [],
  "author": "Paul Halliday",
  "license": "MIT"
}
```

This will allow you to use the npm start command in your terminal to launch your Express server.

Step 2 — Structuring Your Files

To store your files on the client-side, create a public directory and include an index.html file along with an image. Your file structure will look like this:

```
express-static-file-tutorial
  |- index.js
  |- public
    |- shark.png
    |- index.html
```

Now that your files are set up let's begin your Express server.

Step 3 — Creating Your Express Server

In your index.js file, require in an Express instance and implement a GET request:

index.js

```
const express = require('express');
const app = express();
const PORT = 3000;


app.get('/', (req, res) => {
   res.send('Hello World!');
});


app.listen(PORT, () => console.log(`Server listening on port: ${PORT}`));
```

Now let's tell Express to handle your static files.

Step 4 — Serving Your Static Files

Express provides a built-in method to serve your static files:

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

When you call app.use(), you're telling Express to use a piece of middleware. *Middleware* is a function that Express passes requests through before sending them to your routing functions, such as your app.get('/') route. express.static() finds and returns the static files requested. The argument you pass into

express.static() is the name of the directory you want Express to serve files. Here, the public directory.

In index.js, serve your static files below your PORT variable. Pass in your public directory as the argument:

index.js

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

const app = express();

const PORT = 3000;


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


app.get('/', (req, res) => {
   res.send('Hello World!');
});


app.listen(PORT, () => console.log(`Server listening on port: ${PORT}`));
```

With your Express server set, let's focus on the client-side.

Step 5 — Building Your Web Page

Navigate to your index.html file in the public directory. Populate the file with body and image elements:

[label index.html]

```
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
    <img src="shark.png" alt="shark">
```

```
</body>
</html>
```

Notice the image element source to shark.png. Since you've served the public directory through Express, you can add the file name as your image source's value.

Step 6 — Running Your Project

In your terminal, launch your Express project:

```
npm start
```

Server listening on port: 3000

Open your web browser, and navigate to http://localhost:3000. You will see your project:

Conclusion

Express offers a built-in middleware to serve your static files and modularizes content within a client-side directory in one line of code.

2.8 ASYNC AND AWAIT

Async and Await in JavaScript is used to simplify handling asynchronous operations using promises. By enabling asynchronous code to appear synchronous, they enhance code readability and make it easier to manage complex asynchronous flows.

```
async function fetchData()
{
  const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
  const data = await response.json();
  console.log(data);
}
fetchData();
```

Output:

```
{
  userId: 1,
  id: 1,
  title: '....',
  body: ....}
```

Syntax:

```
async function functionName() {
  try {
    const result = await someAsyncFunction();
    console.log(result);
  } catch (error) {
    console.error("Error:", error.message);
  }
}
```

Async Function

The async function allows us to write promise-based code as if it were synchronous. This ensures that the execution thread is not blocked. Async functions always return a promise. If a value is returned that is not a promise, JavaScript automatically wraps it in a resolved promise.

Syntax:

```
async function myFunction()                                    {
  return "Hello";
}
const getData = async () => {
    let data = "Hello World";
    return data;
}
getData().then(data => console.log(data));
```

Output

Hello World

Await Keyword

The await keyword is used to wait for a promise to resolve. It can only be used within an async block. Await makes the code wait until the promise returns a result, allowing for cleaner and more manageable asynchronous code.

```
const getData = async () => {
    let y = await "Hello World";
    console.log(y);
}
console.log(1);
getData();
console.log(2);
```

Output

Hello World

- The **async** keyword transforms a regular JavaScript function into an asynchronous function, causing it to return a Promise.
- The **await** keyword is used inside an async function to pause its execution and wait for a Promise to resolve before continuing.

Error Handling in Async/Await

JavaScript provides predefined arguments for handling promises: resolve and reject.

- **resolve:** Used when an asynchronous task is completed successfully.
- **reject:** Used when an asynchronous task fails, providing the reason for failure.

```
async function fetchData() {
  try {
    let response = await fetch('https://api.example.com/data');
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}
```

Advantages of Async and Await

- **Improved Readability:** Async and Await allow asynchronous code to be written in a synchronous style, making it easier to read and understand.
- **Error Handling:** Using try/catch blocks with async/await simplifies error handling.
- **Avoids Callback Hell:** Async and Await prevent nested callbacks and complex promise chains, making the code more linear and readable.
- **Better Debugging:** Debugging async/await code is more intuitive since it behaves similarly to synchronous code.

2.9 FETCHING JSON FROM EXPRESS

The **express.json()** function is a built-in middleware in Express that is used for parsing incoming requests with JSON payload. The **express.json middleware** is important for parsing incoming JSON payloads and making that data available in the **req.body** or further processing within the routes. Without using express.json, Express will not automatically parse the JSON data in the request body.

By using the **express.json middleware**, you can handle POST, PUT, or PATCH requests that send JSON data from the client to the server.

Syntax:

express.json([options])

Parameters: The options parameter has various properties like inflate, limit, type, etc.

Return Value: It returns an Object.

How express.json() Works?

The primary function of express.json() is to parse requests with a Content-Type header of application/json. Once parsed, the resulting data is stored in the req.body, allowing easy access to the JSON content sent from the client.

Steps To Use express.json() Function

Step 1: Create a Node.js application using the following command.

```
mkdir nodejs
cd nodejs
npm init -y
```

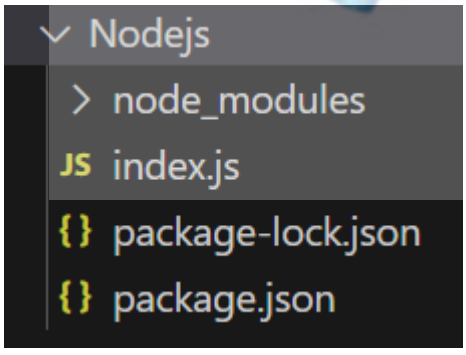
Step 2: Install the required dependencies.

```
npm install express
```

Step 3: Create the required files and start the server.

```
node index.js
```

Project Structure



Project Structure

Example 1: Below is the code example of the **express.json().**

// Filename - index.js

```
const express = require('express');
const app = express();
const PORT = 3000;

app.use(express.json());

app.post('/', function (req, res) {
    console.log(req.body.name)
    res.end();
})

app.listen(PORT, function (err) {
    if (err) console.log(err);
    console.log("Server listening on PORT", PORT);
});
```

Steps to run the program: Run the index.js file using the below command:

node index.js

Output: Now make a **POST** request to **http://localhost:3000/** with header set to **'content-type: application/json'** and body **{"name":"GeeksforGeeks"}**, then you will see the following output on your console:

```
Server listening on PORT 3000
GeeksforGeeks
```

Example 2: Below is the code example of the **express.json().**

```
// Filename - index.js
```

```
const express = require('express');
const app = express();
const PORT = 3000;

// Without this middleware
// app.use(express.json());
app.post('/', function (req, res) {
    console.log(req.body.name)
    res.end();
})

app.listen(PORT, function (err) {
    if (err) console.log(err);
    console.log("Server listening on PORT", PORT);
});
```

Steps to run the program: Run the index.js file using the below command:

```
node index.js
```

Output: Now make a **POST** request to **http://localhost:3000/** with header set to **'content-type: application/json'** and body **{"name":"GeeksforGeeks"}**, then you will see the following output on your console:

```
Server listening on PORT 3000
TypeError: Cannot read property 'name' of undefined
```

Benefits of Using express.json()

- **JSON Parsing Made Simple:** The middleware automatically parses incoming JSON data, making it available in req.body. This eliminates the need to manually parse the data, reducing complexity in your application logic.

- **Seamless Integration:** It integrates easily into any Express application and handles all JSON-related requests without requiring additional configuration or packages.
- **Handling JSON Payloads:** It enables easy handling of JSON payloads sent in POST, PUT, or PATCH requests, which is critical for interacting with modern front-end applications or APIs.