

INTRODUCTION TO NODE JS

Server-side Scripting using node

- Node.js is an open-source, cross-platform runtime environment that allows developers to create server-side tools and applications using JavaScript
 - Node.js can be used to generate dynamic page content, create, open, read, write, delete, and close files on the server, collect form data, and add, delete, modify data in your database.
 - Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.
 - Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.
- Node.js = Runtime Environment + JavaScript Library

A common task for a web server can be to open a file on the server and return the content to the client.

Here is how Node.js handles a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronous programming, which is very memory efficient

What is a Node.js File?

- Node.js files contain tasks that will be executed on certain events
- A typical event is someone trying to access a port on the server
- Node.js files must be initiated on the server before having any effect
- Node.js files have extension ".js"

Before creating an actual "Hello, World!" application using Node.js, let us see the components of a Node.js application. A Node.js application consists of the following three important components –

- **Import required modules** – We use the **require** directive to load Node.js modules.
- **Create server** – A server which will listen to client's requests similar to Apache HTTP Server.
- **Read request and return response** – The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

Creating Node.js Application

Step 1 - Import Required Module

We use the **require** directive to load the http module and store the returned HTTP instance into an http variable as follows –

```
var http = require("http");
```

Step 2 - Create Server

We use the created http instance and call **http.createServer()** method to create a server instance and then we bind it at port 8081 using the **listen** method associated with the server instance. Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World".

```
http.createServer(function(request, response){  
  // Send the HTTP header  
  // HTTP Status: 200 : OK
```

```
// Content Type: text/plain
response.writeHead(200,{ 'Content-Type': 'text/plain' });

// Send the response body as "Hello World"
response.end('Hello World\n');
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

The above code is enough to create an HTTP server which listens, i.e., waits for a request over 8081 port on the local machine.

Step 3 - Testing Request & Response

Let's put step 1 and 2 together in a file called **main.js** and start our HTTP server as shown below –

```
var http = require("http");

http.createServer(function(request, response){
// Send the HTTP header
// HTTP Status: 200 : OK
// Content Type: text/plain
response.writeHead(200,{ 'Content-Type': 'text/plain' });

// Send the response body as "Hello World"
response.end('Hello World\n');
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/ ');
```

Now execute the main.js to start the server as follows –

```
$ node main.js
```

Verify the Output. Server has started.

Server running at http://127.0.0.1:8081/

Open http://127.0.0.1:8081/ in any browser and observe the following result.



Congratulations, you have your first HTTP server up and running which is responding to all the HTTP requests at port 8081.

Create a Node.js file named "myfirst.js", and add the following code:

```
myfirst.js
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('Hello World!');
}).listen(8081);
```

Save the file on your computer: C:\Users\Your Name\myfirst.js

The code tells the computer to write "Hello World!" if anyone (e.g. a web browser) tries to access your computer on port 8081.

How to run a node application:

Node.js files must be initiated in the "Command Line Interface" program of your computer.

How to open the command line interface on your computer depends on the operating system. For Windows users, press the start button and look for "Command Prompt", or simply write "cmd" in the search field.

Navigate to the folder that contains the file "myfirst.js", the command line interface window should look something like this:

```
C:\Users\Your Name>_
```

Initiate the Node.js File

To initialize a new Node.js project, you can use the **npm init** command.

This command will prompt you with a series of questions to create a [package.json](#) file for your project.

The [package.json](#) file contains metadata about your project, such as its name, version, and dependencies.

Here are the steps to initialize a new Node.js project:

1. Open your terminal and navigate to the directory where you want to create your project.
2. Type `npm init` and press enter.
3. Answer the questions that appear in the terminal. You can press enter to accept the default values or type in your own values.
4. Once you have answered all the questions, a [package.json](#) file will be created in your project directory.

That's it! You have successfully initialized a new Node.js project.

Go to the command line interface, write `node myfirst.js` and hit enter:

```
Initiate "myfirst.js":
```

```
C:\Users\Your Name>node myfirst.js
```

Now, your computer works as a server!

If anyone tries to access your computer on port 8081, they will get a "Hello World!" message in return!

Start your internet browser, and type in the address: <http://localhost:8081>

Node.js Server Architecture

Node.js uses the "Single Threaded Event Loop" architecture to handle multiple concurrent clients. Node.js Processing Model is based on the JavaScript event-based model along with the JavaScript callback mechanism.

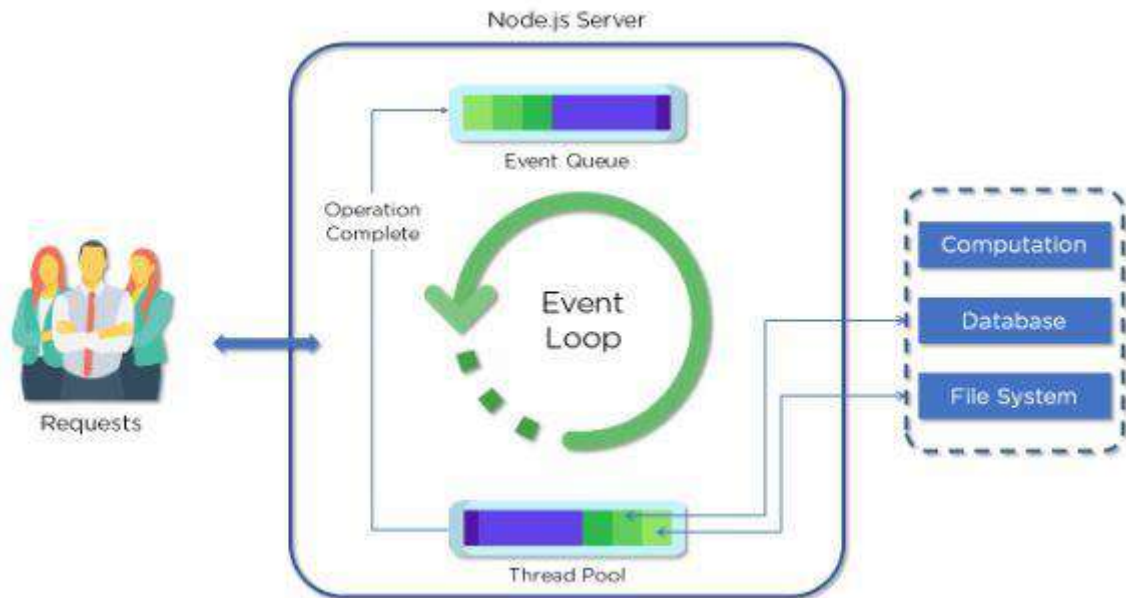


Fig: Node.js architecture

Now let's understand each part of the Node.js architecture and the workflow of a web server developed using Node.js.

Parts of the Node.js Architecture:

- **Requests**
Incoming requests can be blocking (complex) or non-blocking (simple), depending upon the tasks that a user wants to perform in a web application
- **Node.js Server**
Node.js server is a server-side platform that takes requests from users, processes those requests, and returns responses to the corresponding users
- **Event Queue**
Event Queue in a Node.js server stores incoming client requests and passes those requests one-by-one into the Event Loop
- **Thread Pool**
Thread pool consists of all the threads available for carrying out some tasks that might be required to fulfill client requests
- **Event Loop**
Event Loop indefinitely receives requests and processes them, and then returns the responses to corresponding clients
- **External Resources**
External resources are required to deal with blocking client requests, external resources are used. They can be of any type (computation, storage, etc).

The Workflow of Node.js Architecture:

A web server developed using Node.js typically has a workflow that is quite similar to the diagram illustrated below. Let's explore this flow of operations in detail.

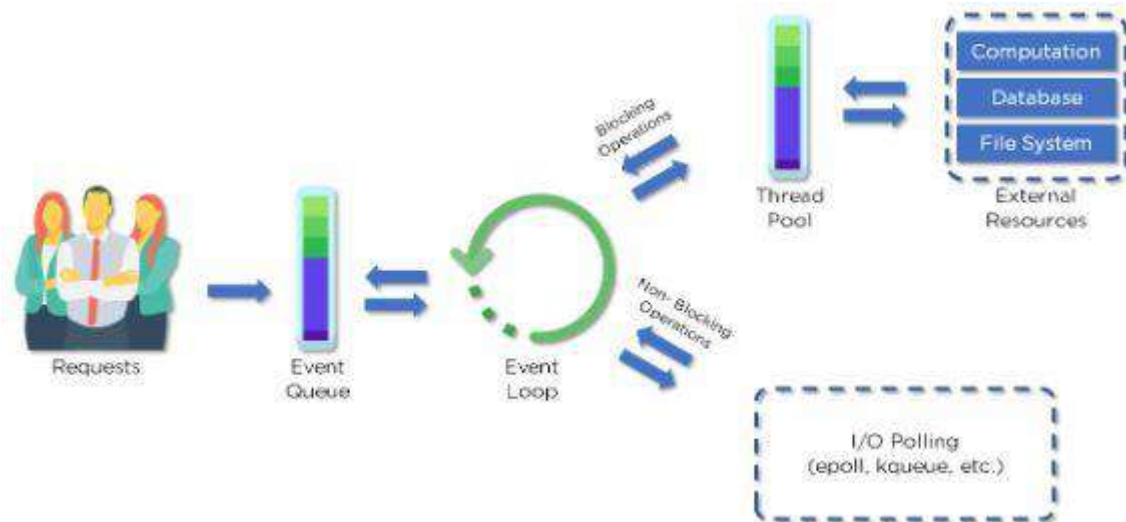


Fig: Node.js Architecture Workflow

- Clients send requests to the webserver to interact with the web application. Requests can be non-blocking or blocking:

-Querying for data

-Deleting data

-Updating the data

- Node.js retrieves the incoming requests and adds those requests to the Event Queue
- The requests are then passed one-by-one through the Event Loop. It checks if the requests are simple enough to not require any external resources
- Event Loop processes simple requests (non-blocking operations), such as I/O Polling, and returns the responses to the corresponding clients

A single thread from the Thread Pool is assigned to a single complex request. This thread is responsible for completing a particular blocking request by accessing the external resources, such as compute, database, file system, etc.

Once, the task is carried out completely, the response is sent to the Event Loop that in turn sends that response back to the Client.

Node Module System

- The Node.js module system is a key feature that enables developers to organize, encapsulate, and manage their code in a modular and efficient manner.
- It allows developers to break down their code base into smaller, reusable components called modules, which can be shared and imported as needed.
- This system plays a crucial role in promoting code maintainability, reusability, and collaboration in Node.js applications.

Node.js Built-in Modules

- Node.js provides a set of built-in modules known as core modules.
- These modules offer essential functionalities for a wide range of tasks, from working with file systems to handling networking operations.
- You can use these modules in your Node.js applications without installing any additional packages.
- To explore on the modules available do visit node js documentation at <https://nodejs.org/api>
- Here are some of the most commonly used built-in core modules:

File System(fs) Module

This module provides methods for working with the file system, allowing you to read, write, and manipulate files and directories.

Example:

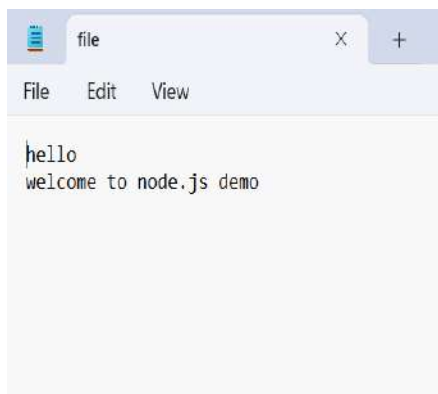
```
// Import the fs module
const fs = require('fs');

// Read a file asynchronously
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});

// Write to a file asynchronously
fs.writeFile('file.txt', 'hello \nwelcome to node.js demo', (err) => {
  if (err) throw err;
  console.log('File written successfully!');
});
```

Explanation:

- The fs module provides file I/O functionality.
- fs.readFile reads a file asynchronously and logs its content.
- fs.writeFile writes data to a file asynchronously



HTTP Module

The **http** module is used for creating HTTP servers and making HTTP requests. It's the foundation of building web applications and APIs.

Example:

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

// Create a simple HTTP server
const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, World!\n');
});

// Listen on port 3000
server.listen(3000, '127.0.0.1', () => {
  console.log('Server listening on port 3000');
});
```



Explanation:

- The http module is used to create an HTTP server.
- The createServer function takes a callback function with request (req) and response (res) objects.
- We set the response header and send a simple "Hello, World!" message.
- The server listens on port 3000.

Path Module

The **path** module provides utilities for working with file paths, making it easier to manipulate and join paths in a cross-platform manner.

Example:

```
// Import the path module
const path = require('path');
// Join two path segments
const filePath = path.join(__dirname, 'files', 'file.txt');
console.log(filePath); // Output: /path/to/your/project/files/file.txt
// Get the file extension
const ext = path.extname('example.txt');
console.log(ext); // Output: .txt
```

Explanation:

- The path module provides utility functions for working with file and directory paths.
- path.join concatenates path segments.
- path.extname returns the file extension.

Operating System Module

os module provides information about the operating system's CPU, memory, network interfaces, and other related details.

Example:

```
// Import the os module
const os = require('os');

// Get information about the operating system
console.log('Platform:', os.platform());
console.log('Architecture:', os.arch());
console.log(`Total Memory: ${os.totalmem()}`);
console.log(`Free Memory: ${os.freemem()}`);
console.log(`CPU Cores: ${os.cpus().length}`);
```

```
node osmodule.js
Platform: win32
Architecture: x64
Total Memory: 17092546560
Free Memory: 7605362688
CPU Cores: 4
```

Explanation:

- The os module provides information about the operating system.
- os.platform returns the operating system platform.
- os.arch returns the operating system architecture.
- os.totalmem returns the amount of system memory.
- os.freemem returns the amount of free system memory.
- os.cpus returns the cpus count.

URL Module

The **url** module offers utilities for parsing and formatting URLs.

Example:

```
const url = require('url');
const urlString = 'https://www.google.com/path?query=value';
const parsedUrl = url.parse(urlString, true);
console.log(parsedUrl.hostname); // Output: www.google.com
console.log(parsedUrl.query);    // Output: { query: 'value' }
```

Querystring Module

This **querystring** module provides methods for parsing and formatting query strings.

Example:

```
const querystring = require('querystring');
const queryString = 'param1=value1&param2=value2';
const parsedQuery = querystring.parse(queryString);
console.log(parsedQuery); // Output: { param1: 'value1', param2: 'value2' }
```

Creating a Module:

1. **Create a JavaScript File:** Start by creating a new **.js** file that will define your module. Let's call it **math.js**.
2. **Define Module Functionality:** In the **math.js** file, define the functionality you want to export. For example, let's create a module that provides functions for addition and subtraction:

```
// math.js
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;

module.exports = {
  add,
  subtract
};
```

Loading and Using the Module:

1. **Require the Module:** In another JavaScript file (let's call it **main.js**), you can use the **require** function to load the **math.js** module:

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

console.log(mathModule.add(5, 3)); // Output: 8
console.log(mathModule.subtract(10, 4)); // Output: 6
```

Run the Main File: Run the **main.js** file using Node.js:
node main.js

This will output the results of the addition and subtraction operations defined in the **math.js** module.

Remember that the **require** function takes the path to the module file as an argument. You can use either a relative path (starting with `./`) or an absolute path.

Exporting Different Types:

You can export various types of values from a module, including variables, functions, classes, or even an entire object. The **module.exports** object allows you to define what parts of the module are accessible when the module is imported using **require**.

For example, exporting a single function from a module:

```
// myFunction.js
const myFunction = () => {
  // Function implementation
};

module.exports = myFunction;

And importing it in another file:
// main.js
const importedFunction = require('./myFunction');

importedFunction();
```

This is the basic process of creating and loading a module in Node.js. By organizing your code into modules, you can maintain a clean and modular codebase, reuse code efficiently, and improve collaboration among team members.

Global Object:

The global object provides access to various global properties and functions that are available throughout your application without the need to import them explicitly. While this might sound similar to the global scope in a browser's JavaScript environment, it's important to note that the Node.js global object is not as extensive as the browser's global scope.

Some commonly used properties and functions available on the Node.js global object:

1. Global Variables:

- **global**: This property refers to the global object itself. It's equivalent to **this** in the global scope.

2. Console:

- **console**: Provides methods for logging and interacting with the console.
- **console.log()**, **console.error()**, **console.warn()**, etc.

3. Timers:

- **setTimeout(), setInterval(), setImmediate():** Functions to schedule asynchronous operations.

4. Buffers:

- **Buffer:** The Buffer class for working with binary data.

5. Paths and Directories:

- **__dirname:** The directory name of the current module.
- **__filename:** The full path to the current module file.
- **require():** A function to import modules.

6. Process:

- **process:** Provides information and control over the current Node.js process.
- **process.argv:** An array containing command-line arguments.
- **process.env:** An object containing environment variables.
- **process.cwd():** Returns the current working directory.
- **process.exit():** Terminates the Node.js process.

7. Modules and Exports:

- **module:** A reference to the current module object.
- **exports:** An object used to define what a module exports.

8. Error Handling:

- **Error:** The constructor function to create error objects.

9. Utilities:

- **setInterval(), setTimeout():** Functions for scheduling repeated or delayed execution of functions.

It's important to note that while these properties and functions are available globally, it's generally considered better practice to avoid polluting the global scope and to use module-level scoping instead. This means that you often declare and use variables, functions, and classes within the scope of a module, making them accessible only where needed. This approach promotes better code organization, reusability, and maintainability.

For example, instead of using the global **setTimeout()**, you might use it within a specific module `app.js`

```
setTimeout(() => {  
  console.log('Delayed message');  
}, 1000);
```

NPM (Node Package Manager)

Node Package Manager is the default package manager for Node.js and is written entirely in Javascript. Developed by Isaac Z. Schlueter, it was initially released in January 12, 2010. NPM manages all the packages and modules for Node.js and consists of command line client npm.

Node Package Manager (NPM) is a command line tool that installs, updates or uninstalls Node.js packages in your application. It is also an online repository for open-source Node.js packages. The node community around the world creates useful modules and publishes them as packages in this repository.

NPM is included with Node.js installation. After you install Node.js, verify NPM installation by writing the following command in terminal or command prompt.

```
C:\> npm -v
```

Now we are ready to create a **package.json** file and learn about how it works. This is a JSON file that holds the basic information about our project. All Node applications are going to have this package.json file. To create your package.json file, go ahead and run **npm init**.

NPM consists of two main parts:

- a CLI (command-line interface) tool for publishing and downloading packages, and
- an online repository that hosts JavaScript packages

Package.json

The package.json file is core to the Node.js ecosystem and is a fundamental part of understanding and working with Node.js, npm, and even modern JavaScript

The package.json file helps us keep track of all the installed packages in a given project. When creating a new project, it is important to start by creating this file.

After generating the package.json file, all the packages installed, along with their names and versions, will be stored in the file.

The package.json file contains the metadata information. This metadata information in package.json file can be categorized into below categories.

- Identifying metadata properties: It basically consist of the properties to identify the module/project such as the name of the project, current version of the module, license, author of the project, description about the project etc.
- Functional metadata properties: As the name suggests, it consists of the functional values/properties of the project/module such as the entry/starting point of the module, dependencies in project, scripts being used, repository links of Node project etc.

A package.json file can be created :-

Using npm init: Running this command, system expects user to fill the vital information required as discussed above. It provides users with default values which are editable by the user.

Syntax:

```
npm init or npm init -y
```

```
Press ^C at any time to quit.
package name: (npm-tutorial)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\node\npm-tutorial\package.json:

{
  "name": "npm-tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this OK? (yes) █
```

Explanation

name: The name of the application/project.

version: The version of application. The version should follow semantic versioning rules.

description: The description about the application, purpose of the application, technology used like React, MongoDB, etc.

main: This is the entry/starting point of the app. It specifies the main file of the application that triggers when the application starts. Application can be started using npm start.

scripts: The scripts which needs to be included in the application to run properly.

engines: The versions of the node and npm used. These versions are specified in case the application is deployed on cloud like heroku or google-cloud.

keywords: It specifies the array of strings that characterizes the application.

author: It consist of the information about the author like name, email and other author related information.

license: The license to which the application confirms are mentioned in this key-value pair.

dependencies: The third party package or modules installed using.

npm: are specified in this segment.

devDependencies: The dependencies that are used only in the development part of the application are specified in this segment. These dependencies do not get rolled out when the application is in production stage.

Advantages of a package.json File

- makes it possible to publish your project to the NPM registry
- makes it easy for others to manage and install your package
- helps NPM manage a module's dependencies easily
- makes your package reproducible and shareable with other developers

Installing a Node Package

There are two ways to install a package: locally or globally.

Local package installation

A locally installed package is one that you can use only in the project in which you've installed it.

To install a package locally, do the following:

Navigate to the root directory of your project from the command line.

Install your package using the NPM installation command

```
npm install package-name --save
```

Note:- that the --save command above instructs NPM to save package-name in the package.json file as one of the packages on which the project depends.

Global package installation

A globally installed package is a package that you can use anywhere on your system.

To install a package globally, run the code below on your terminal:

```
npm install package-name -g
```

Using npm packages in your projects

Once you have installed a package in `node_modules`, you can use it in your code.

Node.js module

If you are creating a Node.js module, you can use a package in your module by passing it as an argument to the `require` function.

```
var lodash = require('lodash');
var output = lodash.without([1, 2, 3], 1);
console.log(output);
In package.json, list the package under dependencies. You can optionally include
a semantic version.
{
  "dependencies": {
    "package_name": "^1.0.0"
  }
}
```

Using scoped packages in your projects

To use a scoped package, simply include the scope wherever you use the package name.

```
var projectName = require("@scope/package-name")
In package.json:
{
  "dependencies": {
    "@scope/package_name": "^1.0.0"
  }
}
```

Dependencies: Packages required by your application in production.

Dependency is an object that contains the library, which your project requires for production environments and functioning effectively. You require these packages to test and run your project on the localhost. A dependency is a library that a project needs to function effectively. If a package doesn't already exist in the `node_modules` directory, then it is automatically added. These are the libraries you need when you run your code. Included in the final code bundle.

It can add dependencies to the `package.json` file by running the below command:

```
npm install <dependencies>
```

The "dependencies" field lists all the packages a project depends on in production.

Here's an example:

```
{
  "dependencies": {
    "first-package": "^1.0.4",
```



```
"second-package": "~2.1.3"
}
}
```

So, whenever a user installs your project from the NPM registry, the dependencies property ensures package managers can automatically find and install the packages listed.

Note that you can add a package to the "dependencies" field through either of the following ways:

Manually add the name and the semantic version of each package your project depends on in production.

Run the **npm install package-name --save-prod** command on your terminal.

DevDependencies: Packages that are only needed for local development and testing.

devDependencies are those packages in the package.json file that you need only for project development purposes. Example- Babel, Webpack, etc.

These types of dependencies are required during the web application development process but not while testing or executing it

For adding devDependencies to your project, run the below command:

```
npm install <dev dependencies> --save-dev
```

The "devDependencies" field lists all the packages a project does not need in production—but requires for its local development and testing purposes.

Here's an example:

```
{
  "devDependencies": {
    "first-dev-package": "^5.8.1",
    "second-dev-package": "3.2.2—4.0.0"
  }
}
```

Note that the packages listed in the "devDependencies" field will be available in the project's development environment but not on its production server.

Keep in mind that you can add a package to the "devDependencies" field through either of the following ways:

Manually add the name and the semantic version of each package on which your project depends for its development and testing purposes.

Run **the npm install package-name --save-dev** command on your terminal.

In the package.json file of your package manager, i.e., npm, there is an element known as devDependencies. It contains all the packages with specific version numbers that you require in the development phase of your project and not in the production or testing environments. Whenever you add a library or module that is required only in the development of your project, you can find it under devDependencies.

For npm users, the below command will help to add devDependencies in the project: [Replace 'dev dependencies' with the module you're installing.]

```
npm install <dev_dependencies> --save-dev
```

DevDependencies are the packages a developer needs during development. As you install a package, npm will automatically install the dev dependencies. These dependencies may be needed at some point during the development process, but not during execution. Included in the final code bundle .

Uninstalling a package

To uninstall a package from a specific project

First, navigate to the project's root directory from the command line and run:

```
npm uninstall package-name
```

To uninstall a global package

```
npm uninstall package-name -g
```

Publishing and updating a package to npm

NPM is a free registry for public package authors.

So, it is used to publish any project (folder) from your computer that has a package.json file.

Step 1: Create package

Then in terminal write the following commands

```
npm init
```

```
npm link
```

Step 2: Log in

Login to NPMJS account from the command line like so:

```
npm login
```

Enter username and password

Step 3: Publish your package!

Go to your project's root directory and publish it like so:

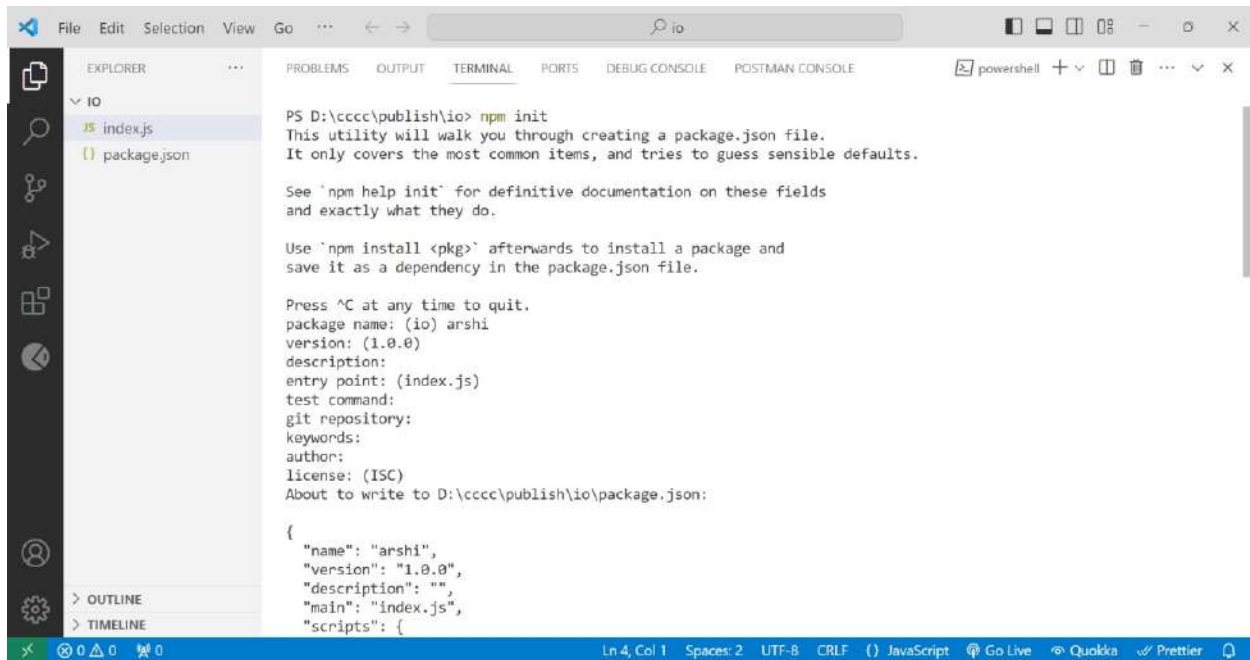
```
npm publish
```

Make sure that your package's name does not currently exist on NPM. Otherwise, you will get an error while publishing.

You can use the npm search command (or the NPM website's search bar) to search if the name you wish to use already exists on NPM.

Suppose all the suitable names for your package are already taken. In that case, NPM allows you to publish your project as a scope.

In other words, you can publish your package as a sub-section of your username. Let's see how below.



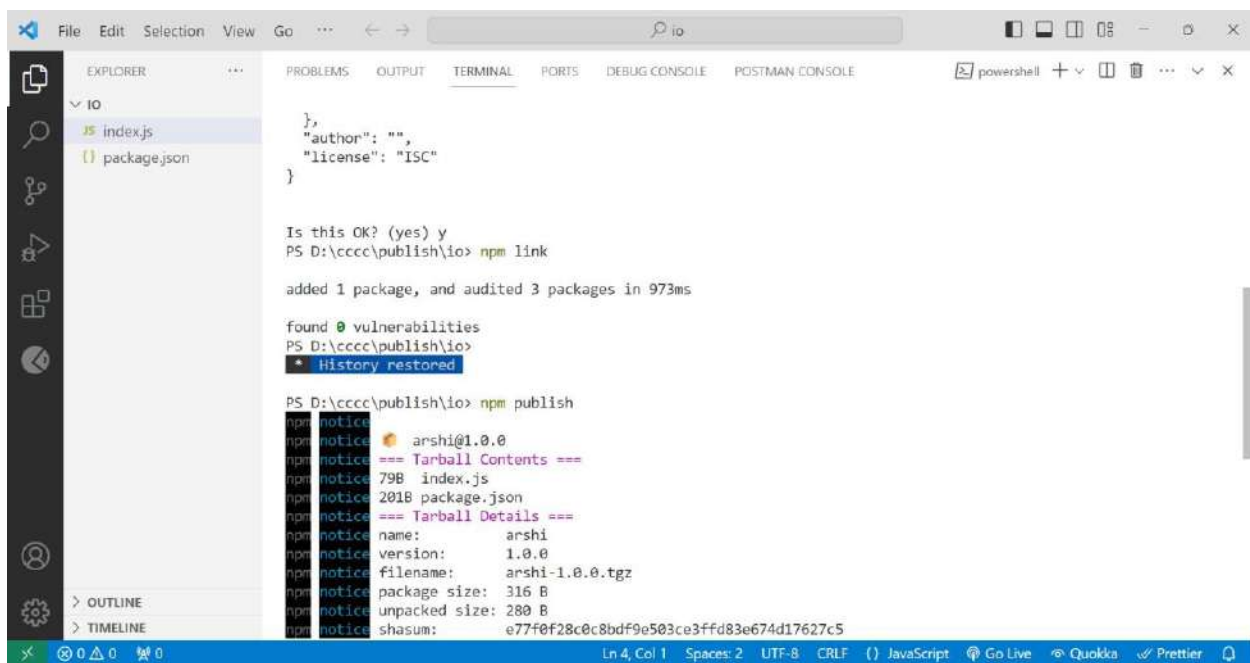
```
PS D:\cccc\publish\io> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

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: (io) arshi
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to D:\cccc\publish\io\package.json:

{
  "name": "arshi",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
```



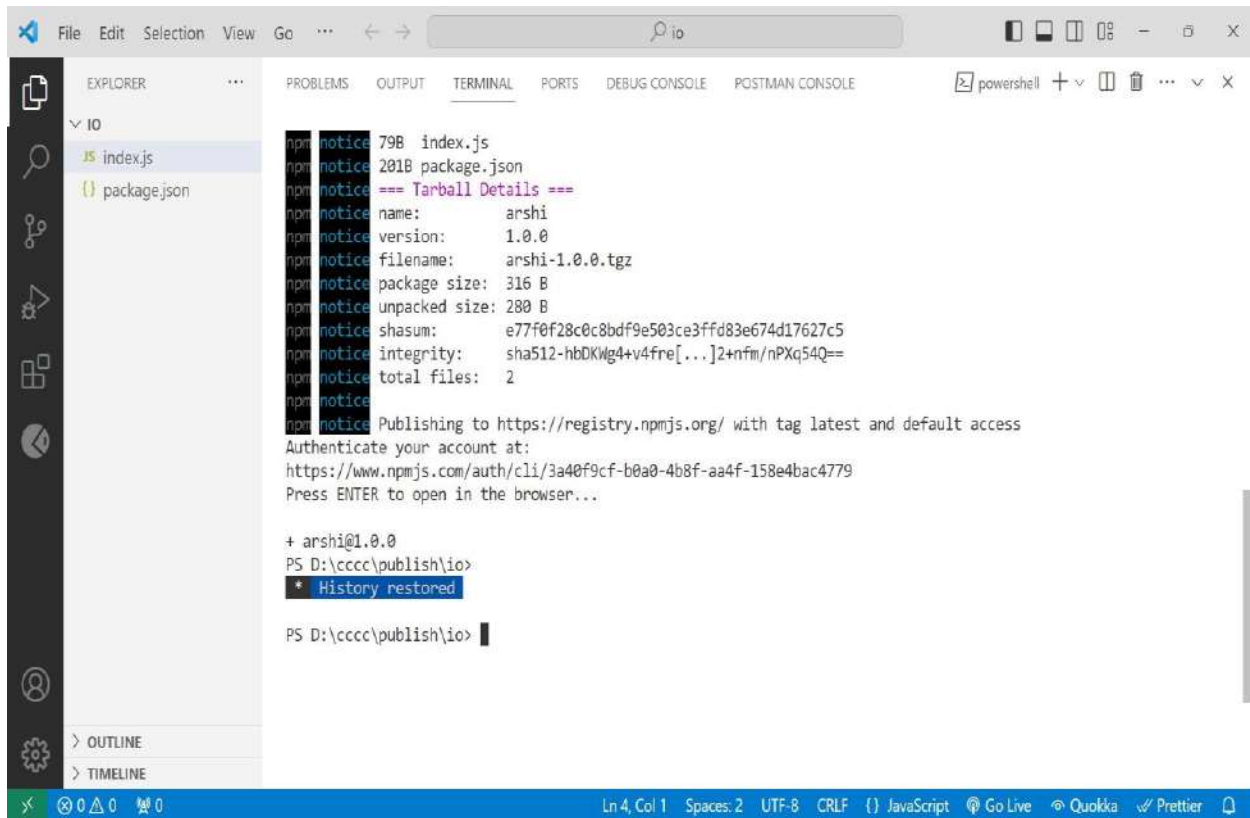
```
    },
    "author": "",
    "license": "ISC"
  }

Is this OK? (yes) y
PS D:\cccc\publish\io> npm link

added 1 package, and audited 3 packages in 973ms

found 0 vulnerabilities
PS D:\cccc\publish\io>
* History restored

PS D:\cccc\publish\io> npm publish
npm notice
npm notice 📦 arshi@1.0.0
npm notice === Tarball Contents ===
npm notice 798 index.js
npm notice 2018 package.json
npm notice === Tarball Details ===
npm notice name: arshi
npm notice version: 1.0.0
npm notice filename: arshi-1.0.0.tgz
npm notice package size: 316 B
npm notice unpacked size: 280 B
npm notice shasum: e77f0f28c0c8bdf9e503ce3ffd83e674d17627c5
```

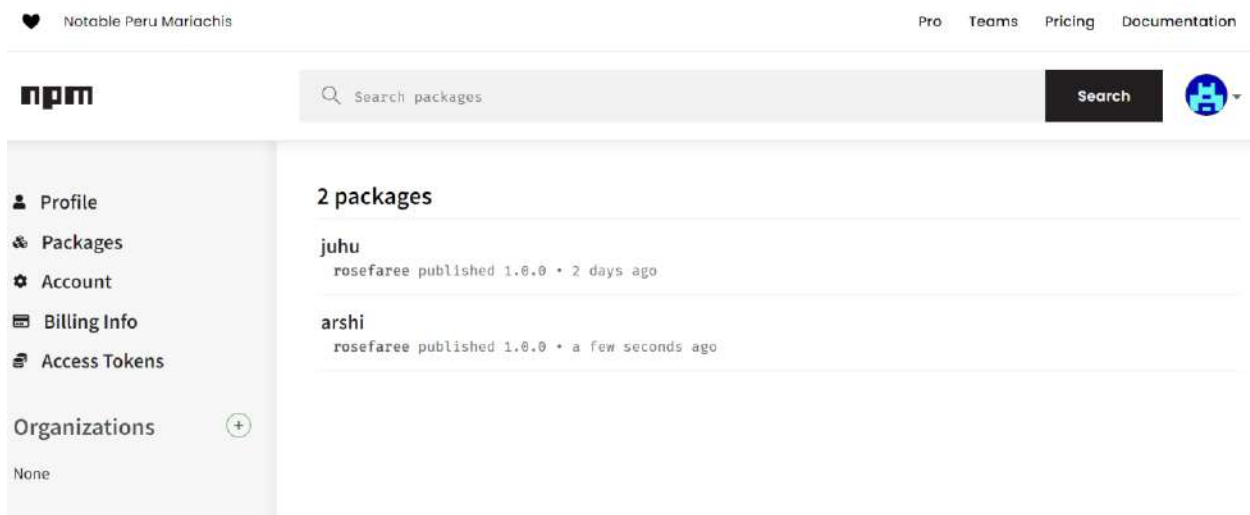
A screenshot of the Visual Studio Code interface. The Explorer sidebar on the left shows a file named 'index.js' and a 'package.json' file. The main editor area displays the output of an 'npm publish' command. The output shows details about the package 'arshi' version '1.0.0', including its filename, size, and integrity. It also shows the authentication process for publishing to the npm registry. The terminal window is titled 'powershell' and shows the command prompt 'PS D:\cccc\publish\io>' with the command 'npm publish' executed. The status bar at the bottom indicates the file is 'index.js' in the 'publish' directory, with a UTF-8 encoding and CRLF line endings.

```
npm notice 79B index.js
npm notice 201B package.json
npm notice === Tarball Details ===
npm notice name: arshi
npm notice version: 1.0.0
npm notice filename: arshi-1.0.0.tgz
npm notice package size: 316 B
npm notice unpacked size: 280 B
npm notice shasum: e77f0f28c0c8bdf9e503ce3ffd83e674d17627c5
npm notice integrity: sha512-hbDKWg4+v4fre[...]2+nfm/nPXq54Q==
npm notice total files: 2
npm notice
npm notice Publishing to https://registry.npmjs.org/ with tag latest and default access
Authenticate your account at:
https://www.npmjs.com/auth/cli/3a40f9cf-b0a0-4b8f-aa4f-158e4bac4779
Press ENTER to open in the browser...

+ arshi@1.0.0
PS D:\cccc\publish\io>
* History restored

PS D:\cccc\publish\io>
```

It is published and now to verify it go to npmjs website.



In that under my packages , the list of packages you have published will be displayed just as shown in the above picture.

To Update npm and Packages

Keeping your npm and packages up to date is the single best way to keep bugs and security flaws away from your code.

To update npm to its latest version, use the command below:

```
npm install npm@latest -g
```

This updates npm globally on your computer.

When the creators of a package introduce new features or fix bugs, they update the package on the npm registry. You then have to update your own package in order to make use of the new features.

Here is the syntax of the command you'd use to do this:

```
npm update [package name]
```

And here's a working example:

```
npm update typescript
```