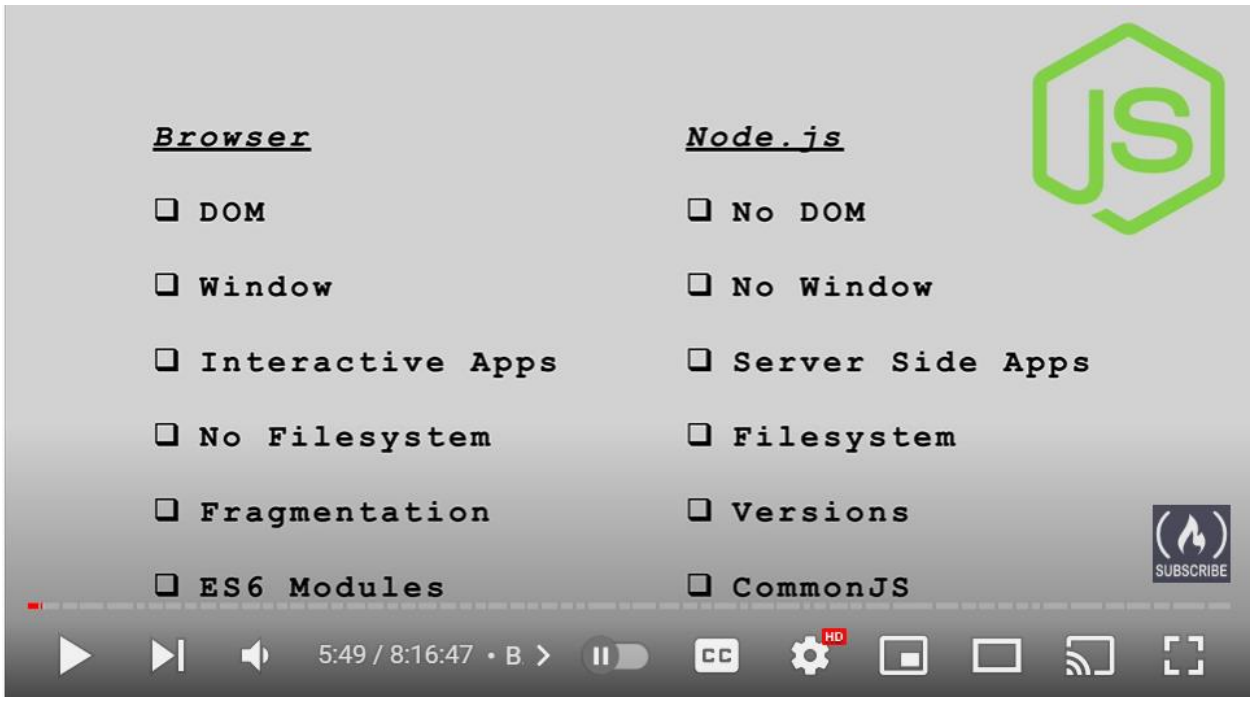
Node JS

Environment to run JS outside the browser.

Created in 2009.

Built on chrome's V8 Engine.

Every browser has an engine, a tool that compiles our code down to machine code and chrome uses it by the name V8.



Node JS uses Common JS library.

For frontend part, browser does most of the part. Our code is compiled, and browser runs our code.

How does node evaluate our code?

We have 2 options

1. REPL – Read Eval Print Loop
2. CLI Executable – running our app code in node

REPL is only for playing around but we mostly use CLI.

REPL can be accessed by opening a terminal and type node and then hit enter. Once you have entered the REPL, type const name = "Sai" and click enter.

Now, you can access the name variable which has been defined as 'Sai'.

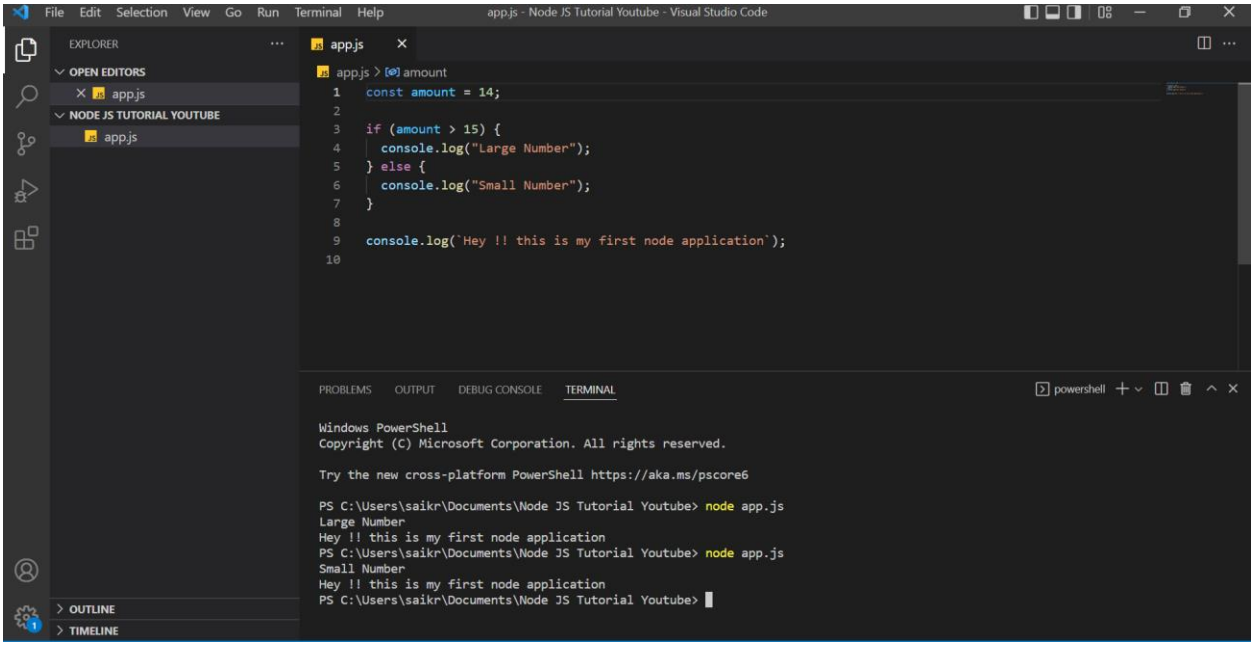
You can close the REPL by click on Ctrl + C for two times.

How to run a Node Application in a terminal

Use the command node <filename> to run the node app in terminal

Ex: node app.js

Ex: node app (we can exclude the applications file extension)

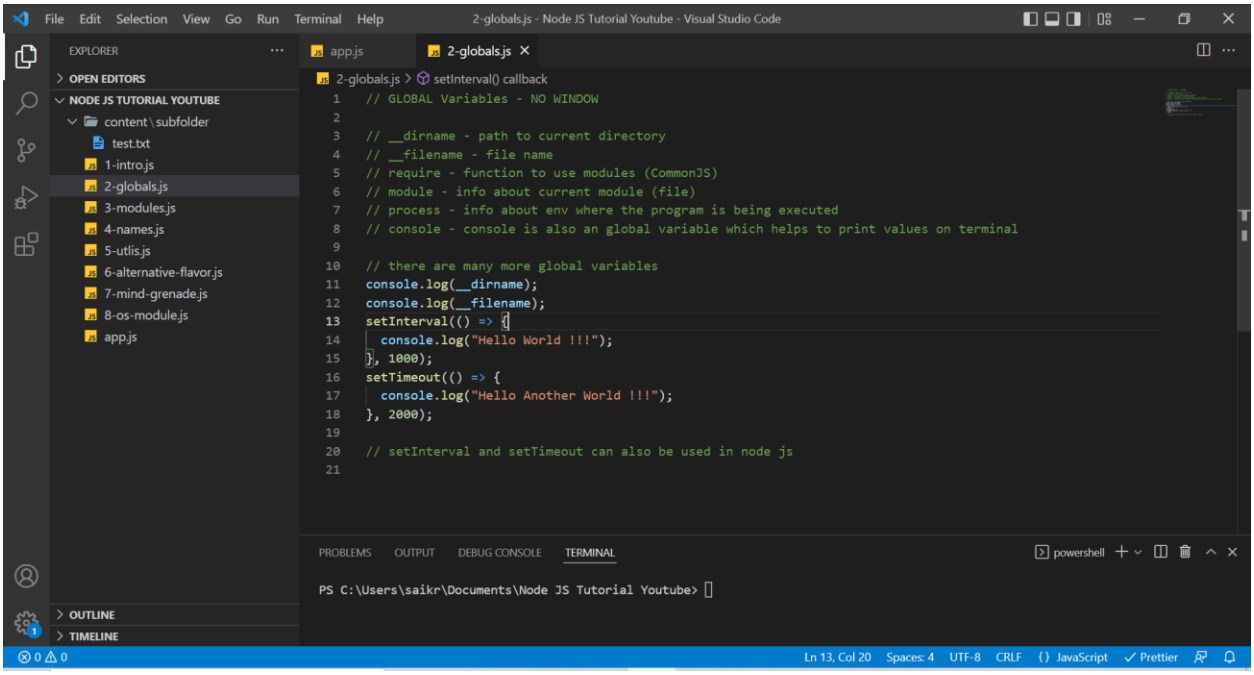


Global Variables in Node

In Vanilla JavaScript, we have access to various global variables like window. With window variable we can have access to many useful functions like querySelector().

There is no window in Node because there is no browser for Node.

Global Variables are nothing but a variable which can be accessed anywhere over the application.



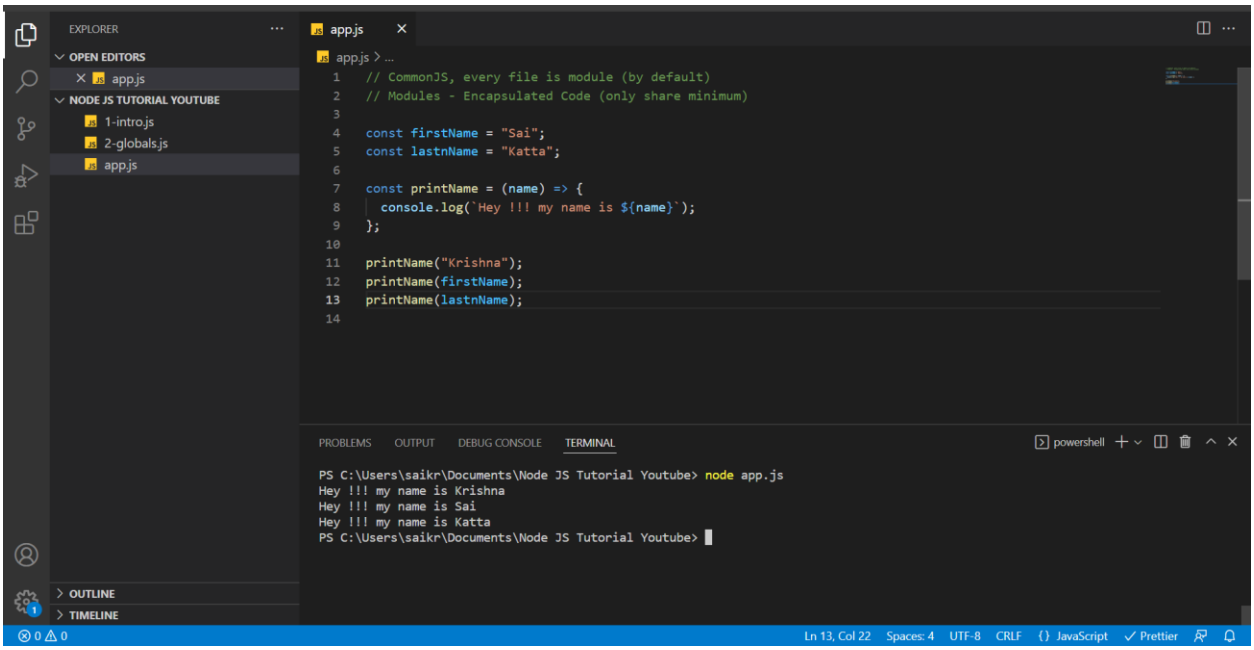
Modules in Node

Generally, our code will be split into multiple files, but all would be connected to app.js file because we run our entire application runs using one file.

Every File in node is a module.

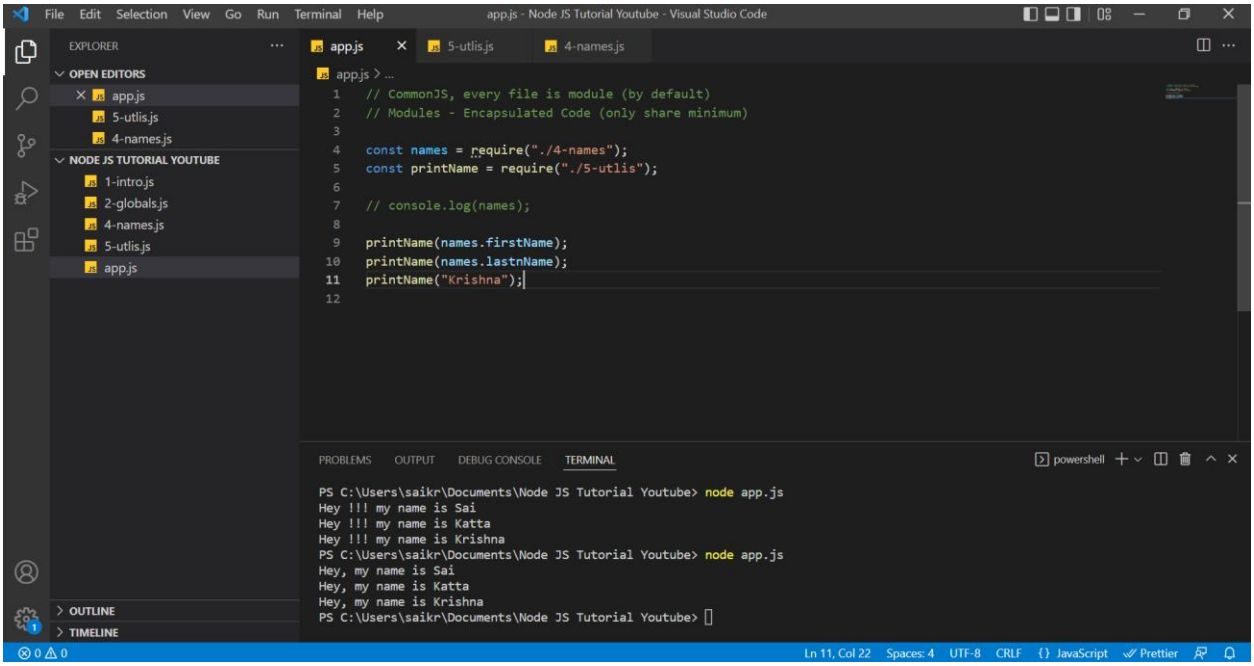
Modules are Encapsulated code (only share minimum)

Without the concept of Module

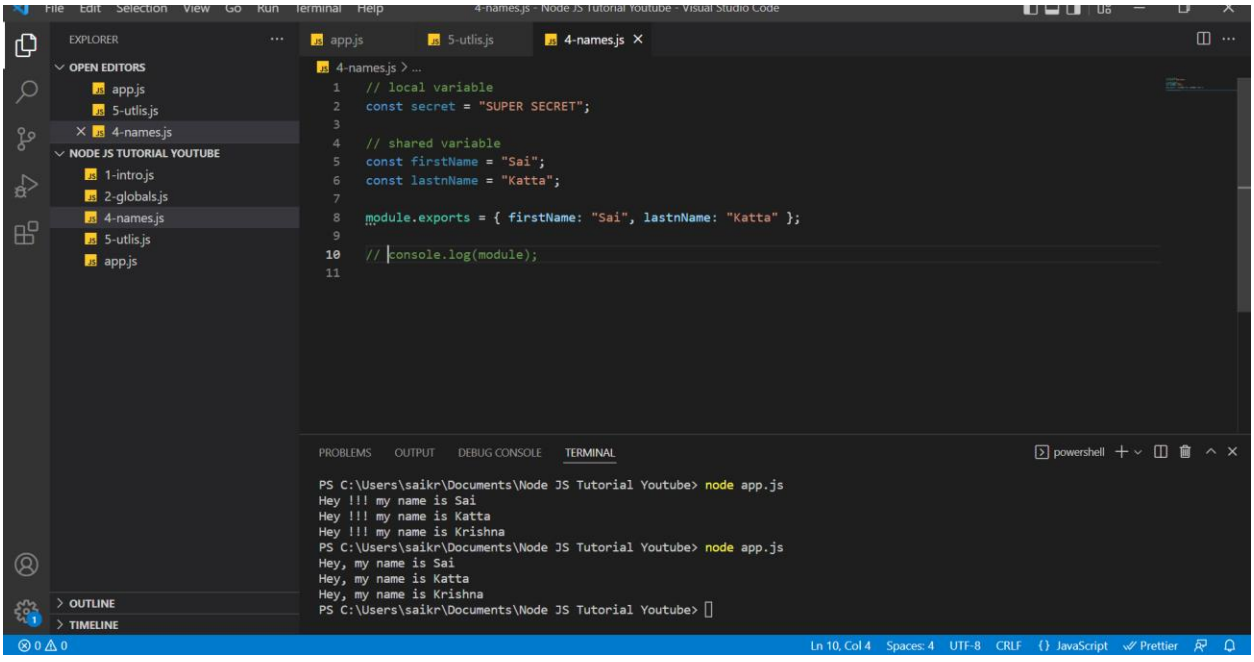


With concept of Module

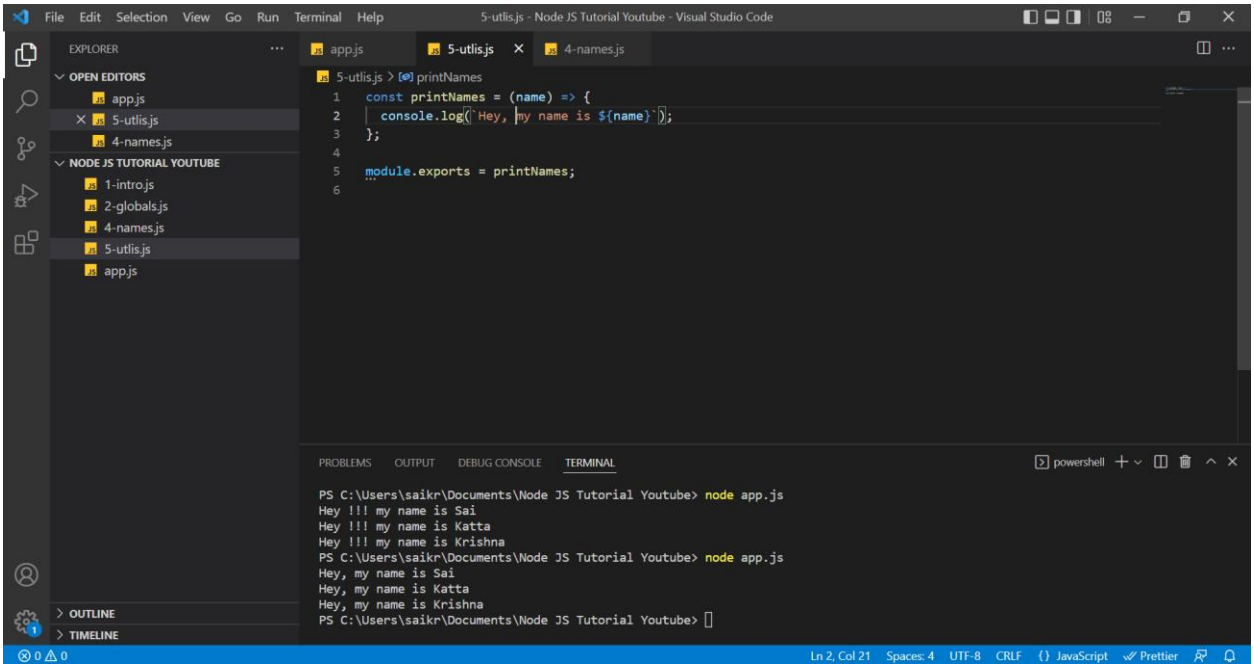
app.js



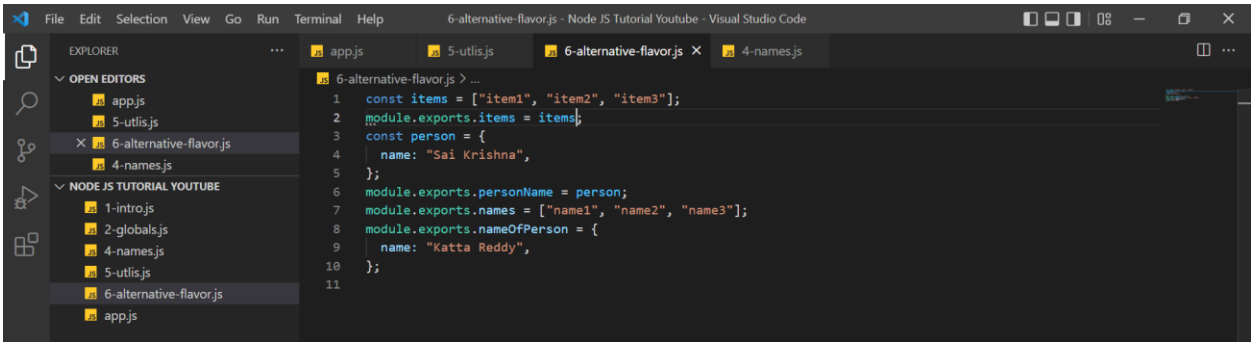
4-names.js



5-utils.js

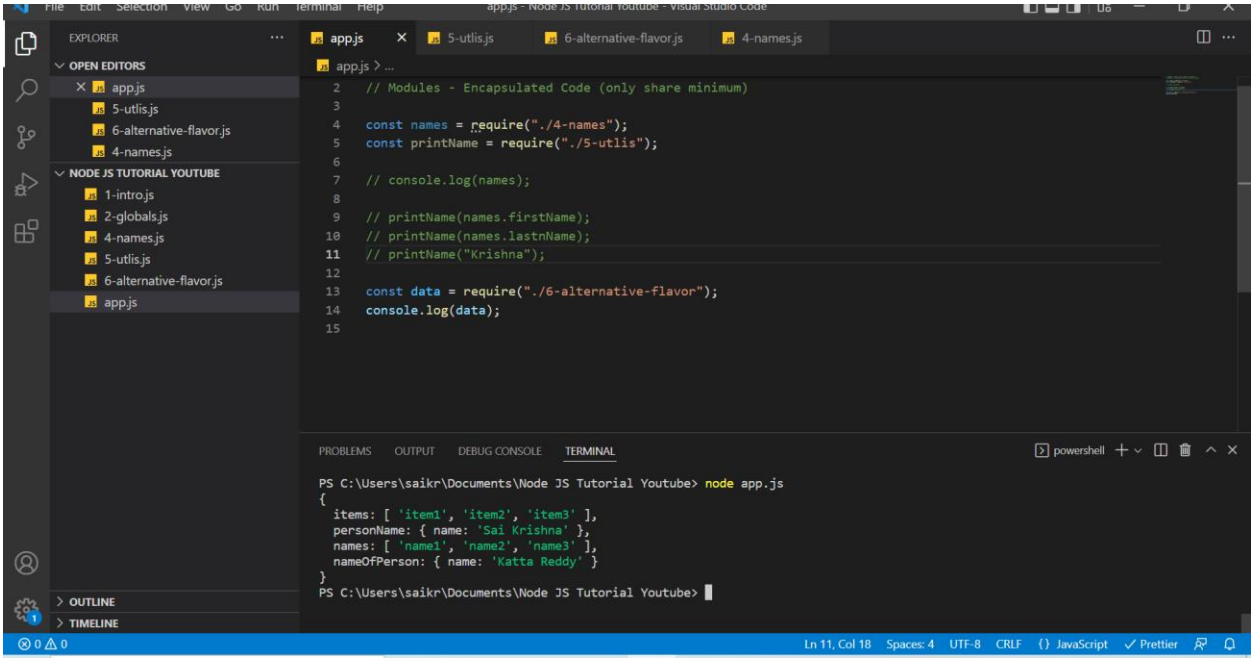


6-alternative-flavors.js



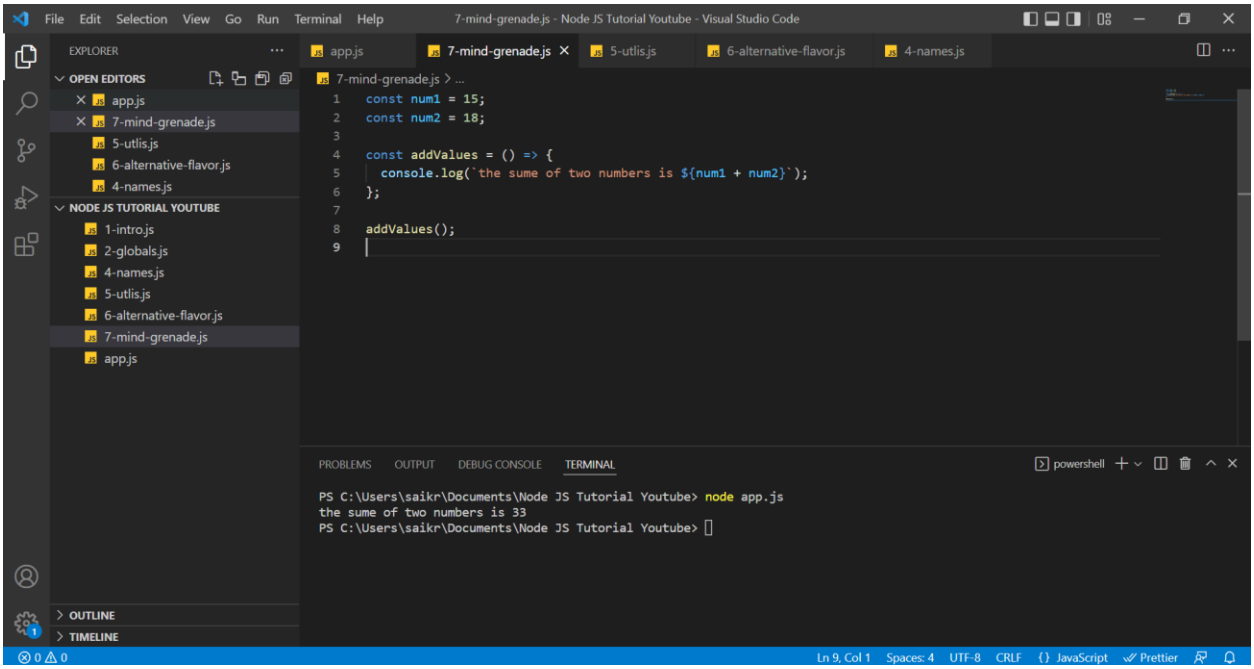
We can do multiple exports or else we can use the approach having only module export per file. It depends on the comfort of the person writing code.

In app.js, we can import it only as on single object because in modules we only use export object to export data from one module to another so that is the reason we can have multiple objects like items, personName, names, nameOfPerson from 6-alternative-js inside one object named data of app.js file.

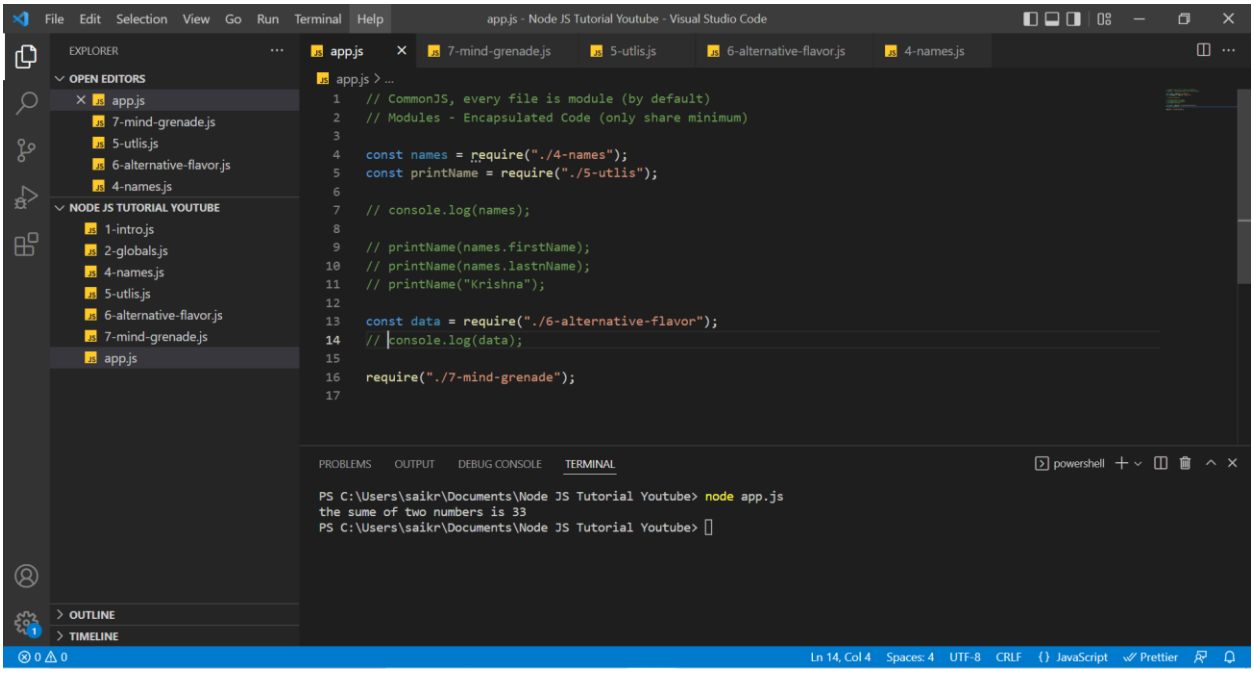


Basically, when we import a module in app.js file we invoke that module. Invoking means sometimes we may receive data through exports objects or sometimes we may invoke a function to execute some functionality in that module.

To get a good explanation about that, please have a look at the below screenshots.



In app.js, when we run node app.js, with the help of ***require*** global variable we are invoking the module 7-mind-grenade.js, which in turns executes a function named addValues() which calculates sum of two numbers.



This can be great example, since we use a bunch of third-party modules which executes functionalities in background, but we aren't able to see any of those methods on screen. We just extract few methods from the module using ***require*** global variable and use it in our code.

Built-in Modules in Node JS

Below are some of the major modules used in Node JS

OS (Operating System Module)

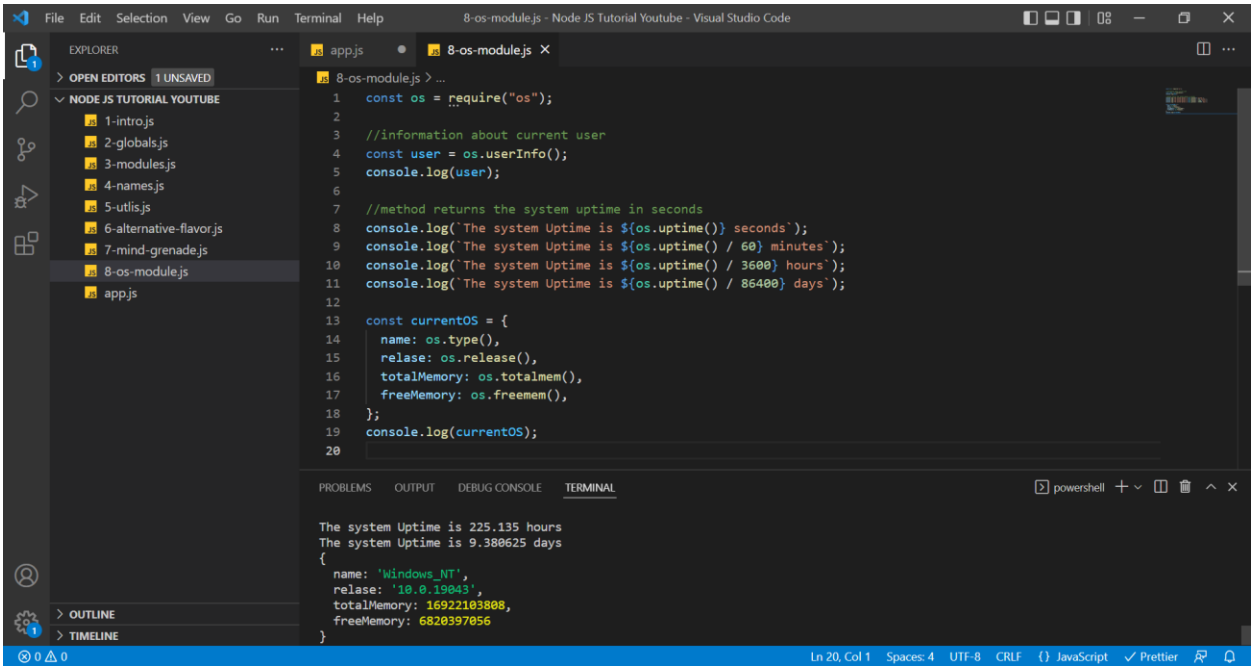
PATH Module

FS (File System Module)

HTTP Module – used to set up our HTTP server

OS (Operating System Module)

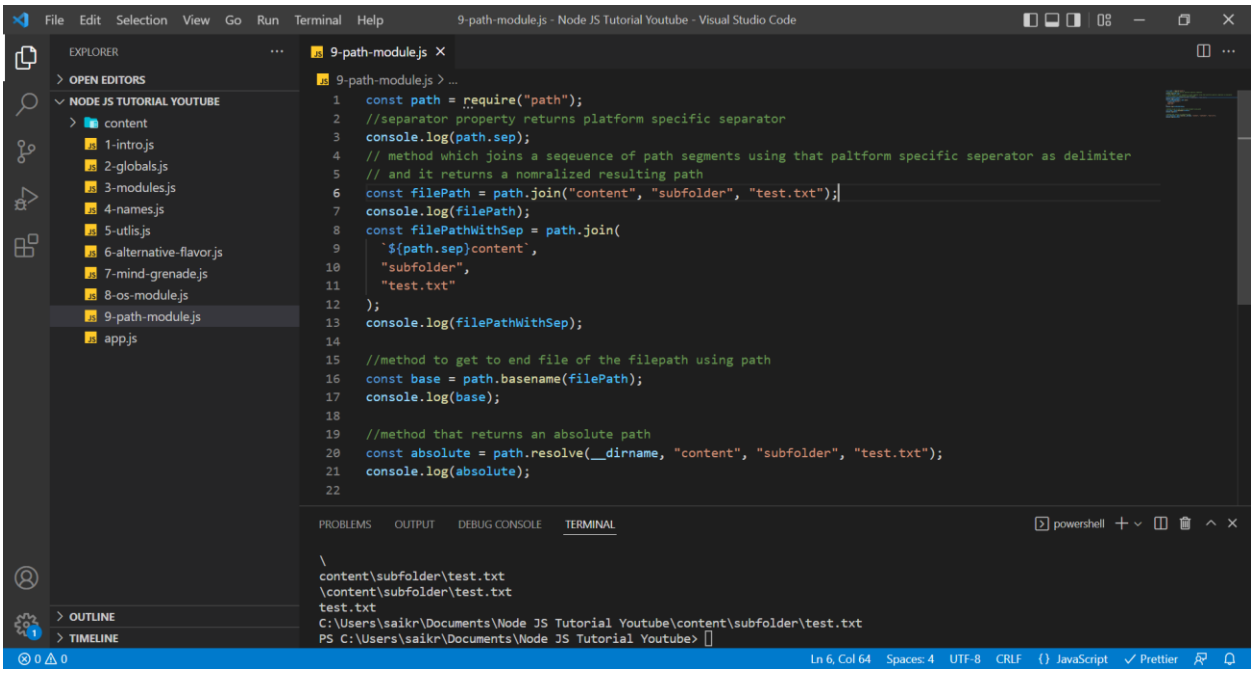
Provides methods and properties that are useful to interact with operating system and as well as server.



PATH Module

With JavaScript in browser, we cannot access File System but in node we can access the File System.

PATH Module allows us to interact with file paths easily.



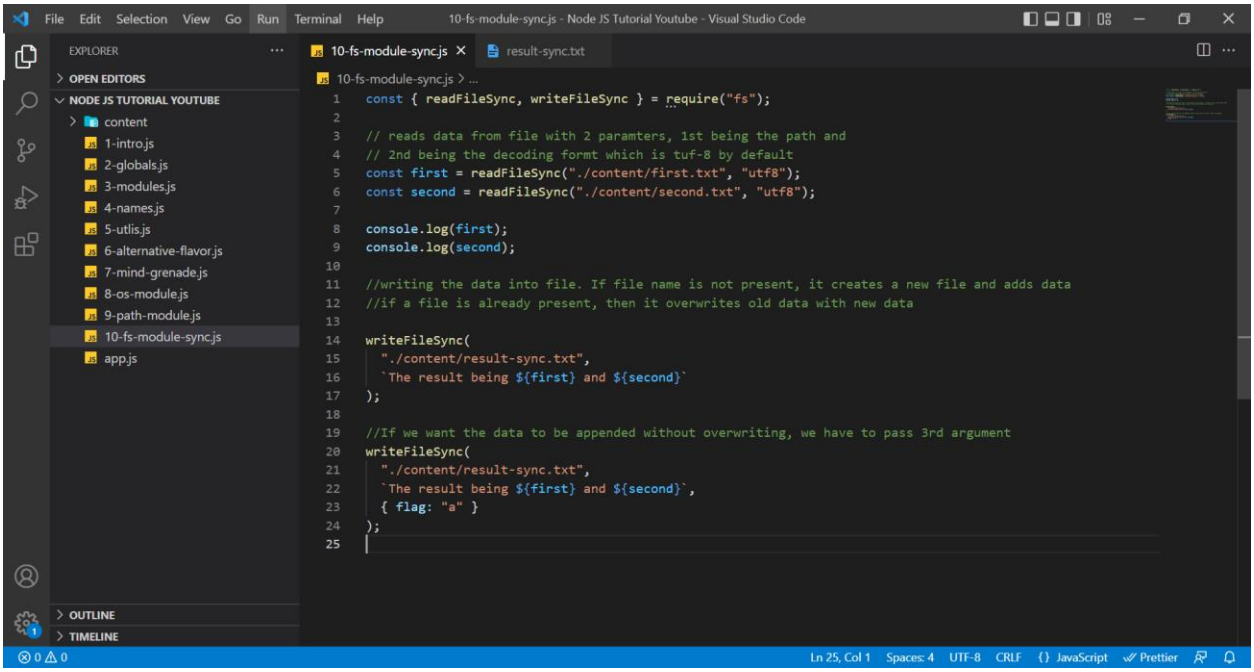
FS (File System Module)

Accessing File System Module can be done in two ways,

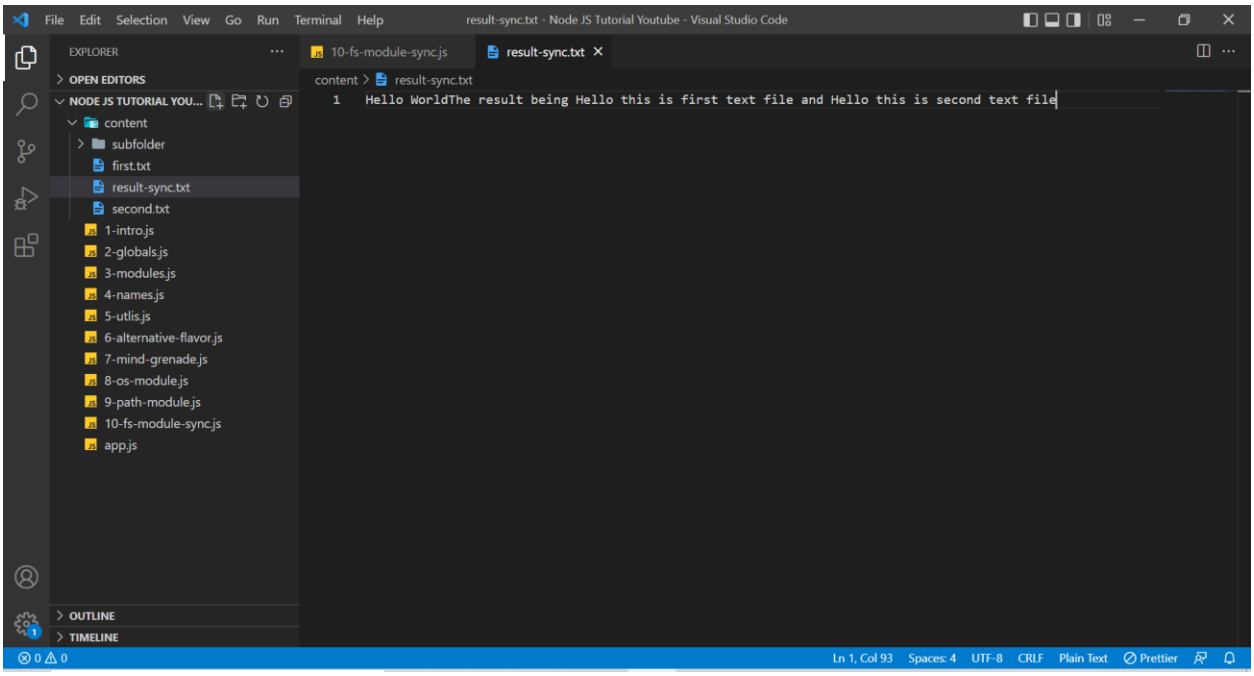
Asynchronously (Which is non-Blocking) and Synchronously (Which is Blocking)

Let us have a look at the **Synchronous** way of reading and writing data into a file.

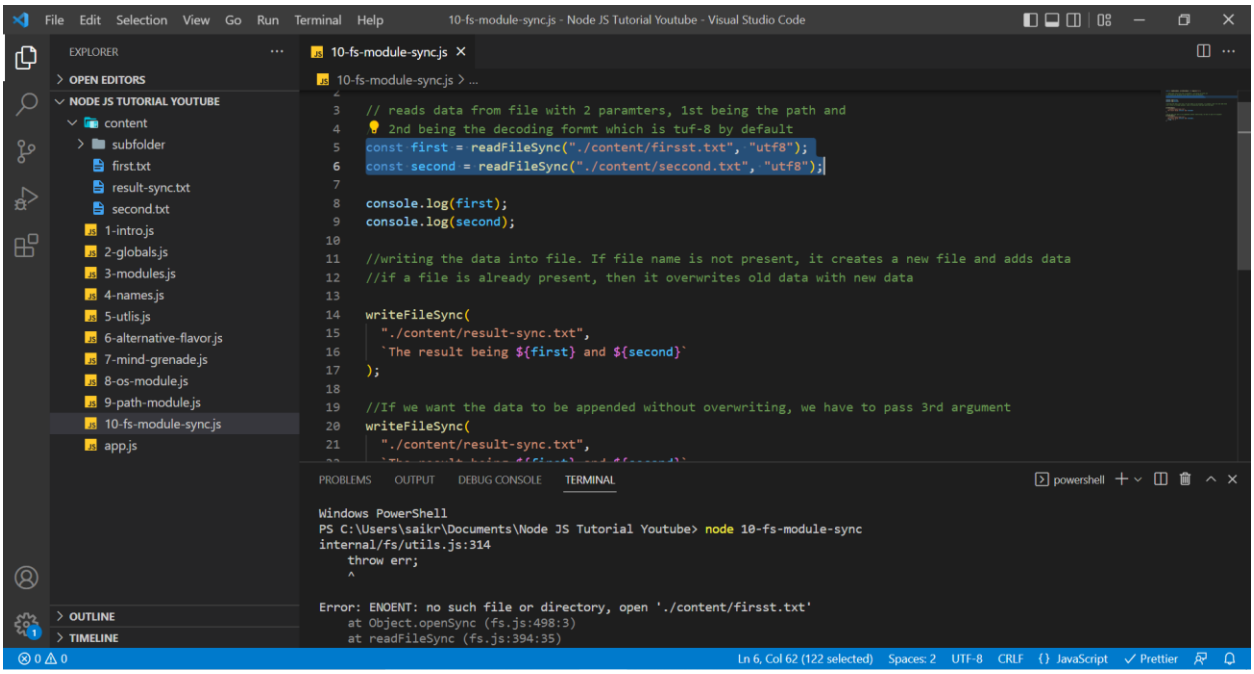
10-fs-module-sync.js



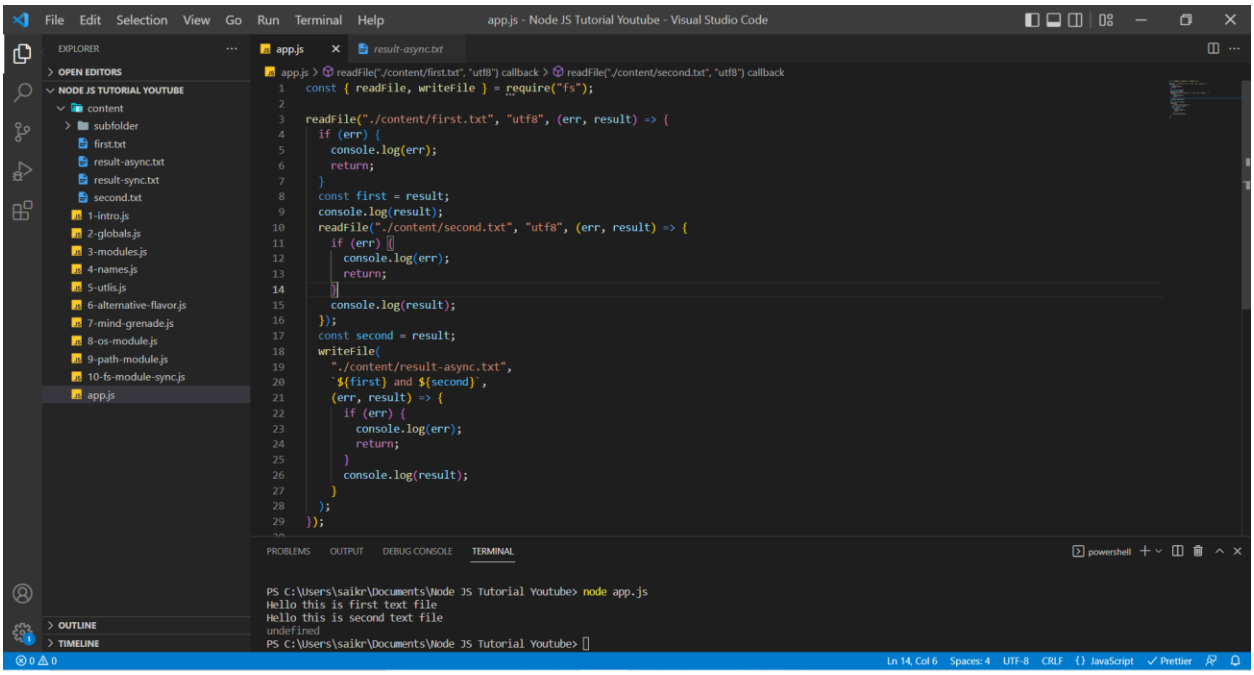
result-sync.text



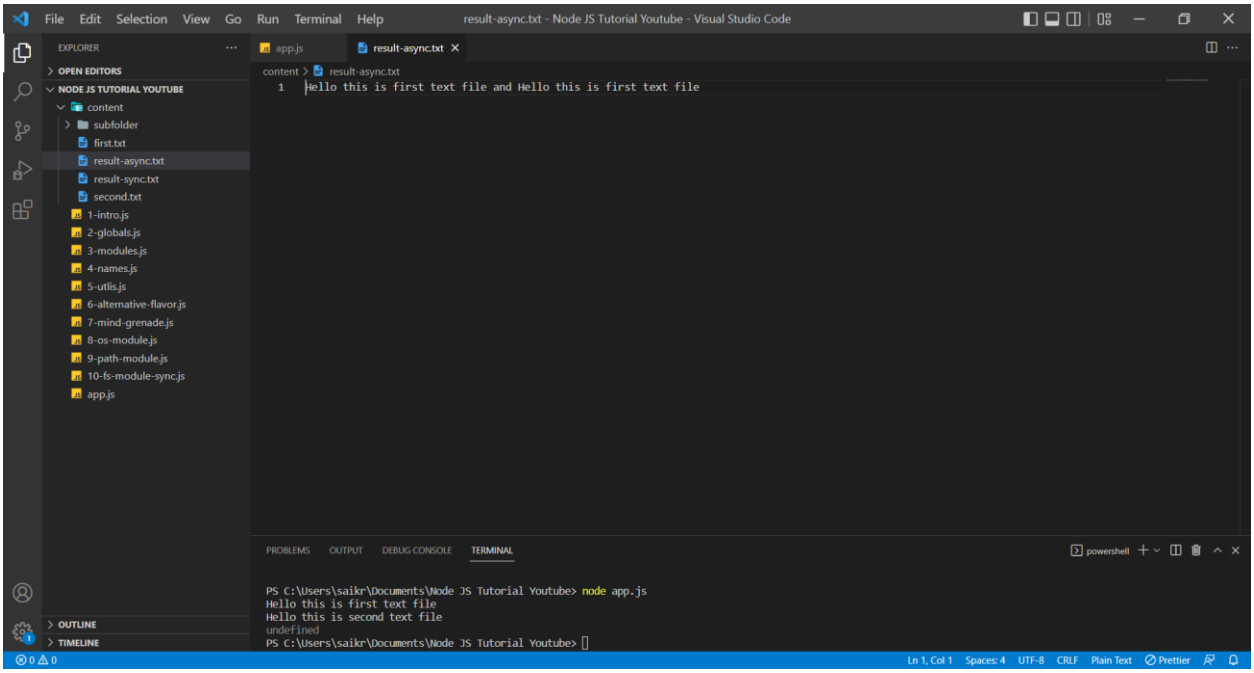
If you try to read data from a non-existed file, then node will throw an error. Here we are taking files which are not existing. Here the node is throwing an error.



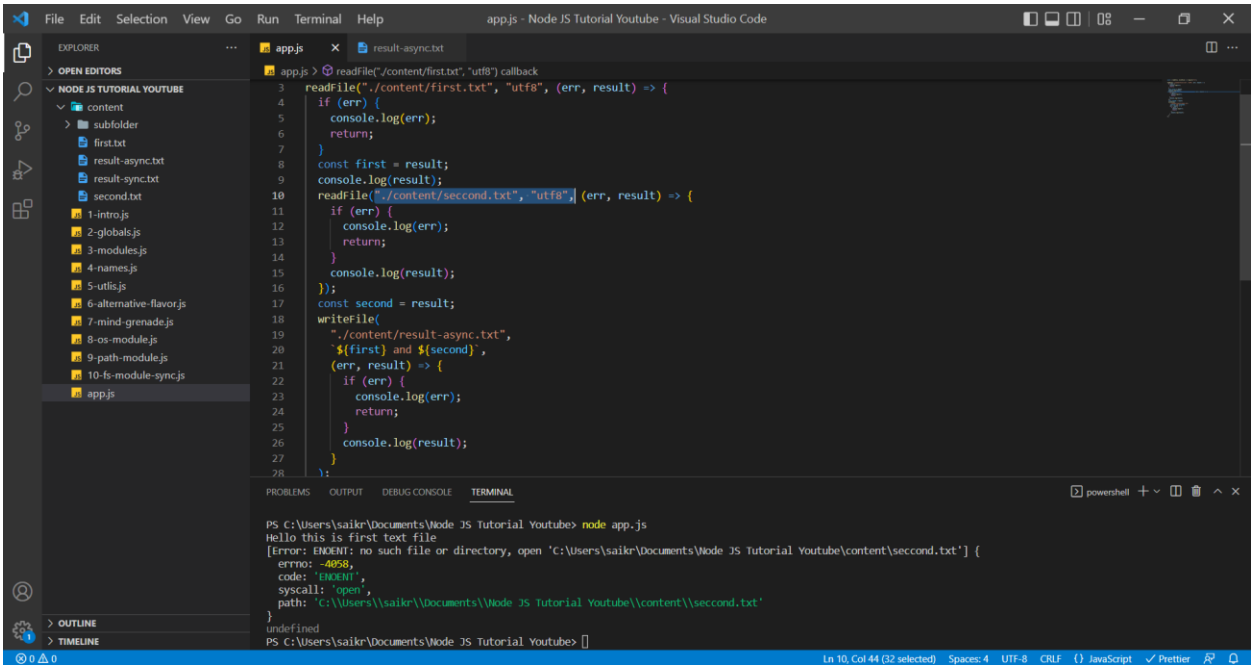
Now we are reading two files and combing their data and writing that data into a new text file result-async.txt. The result of readFile is undefined. This is also creating a callback hell.



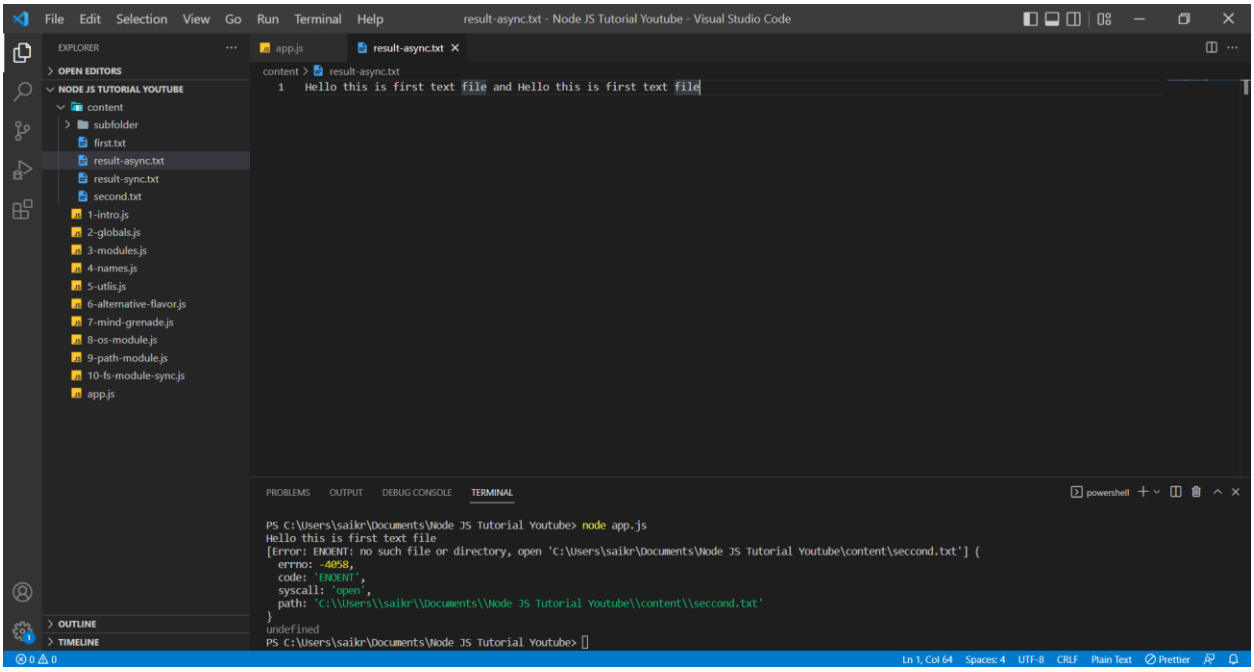
result-async.txt



When we try to read the data and couldn't find the file.

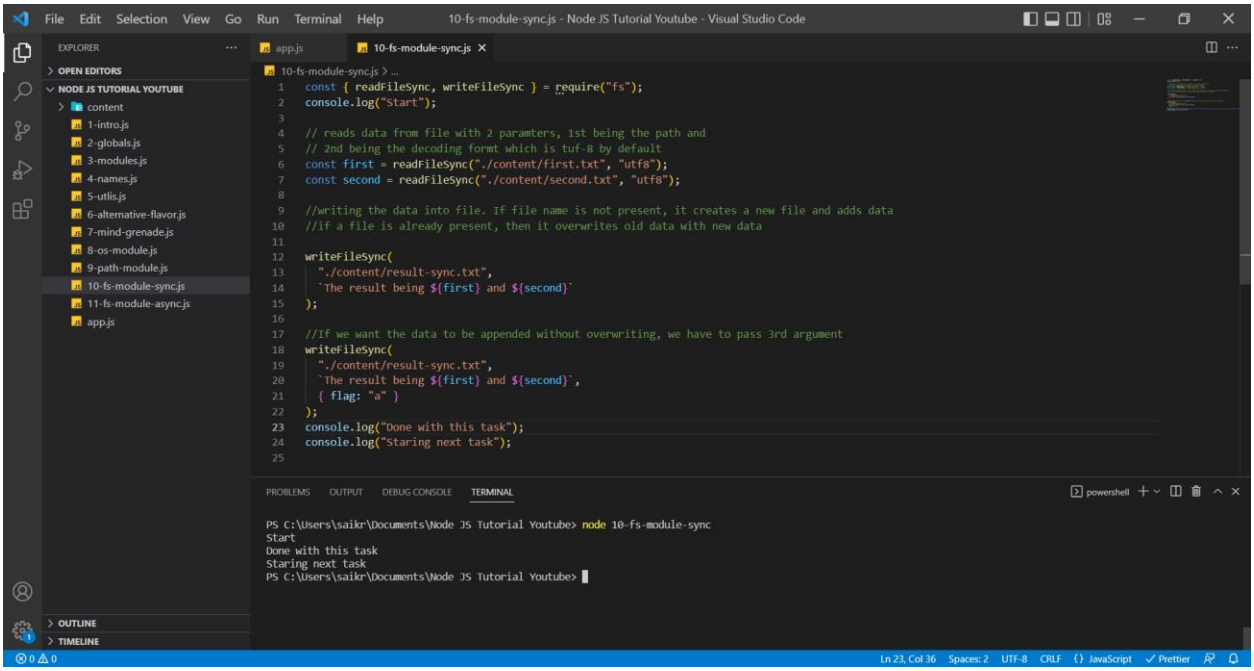


result-async.txt (result will be only from first result variable which is “Hello this is first text file”)



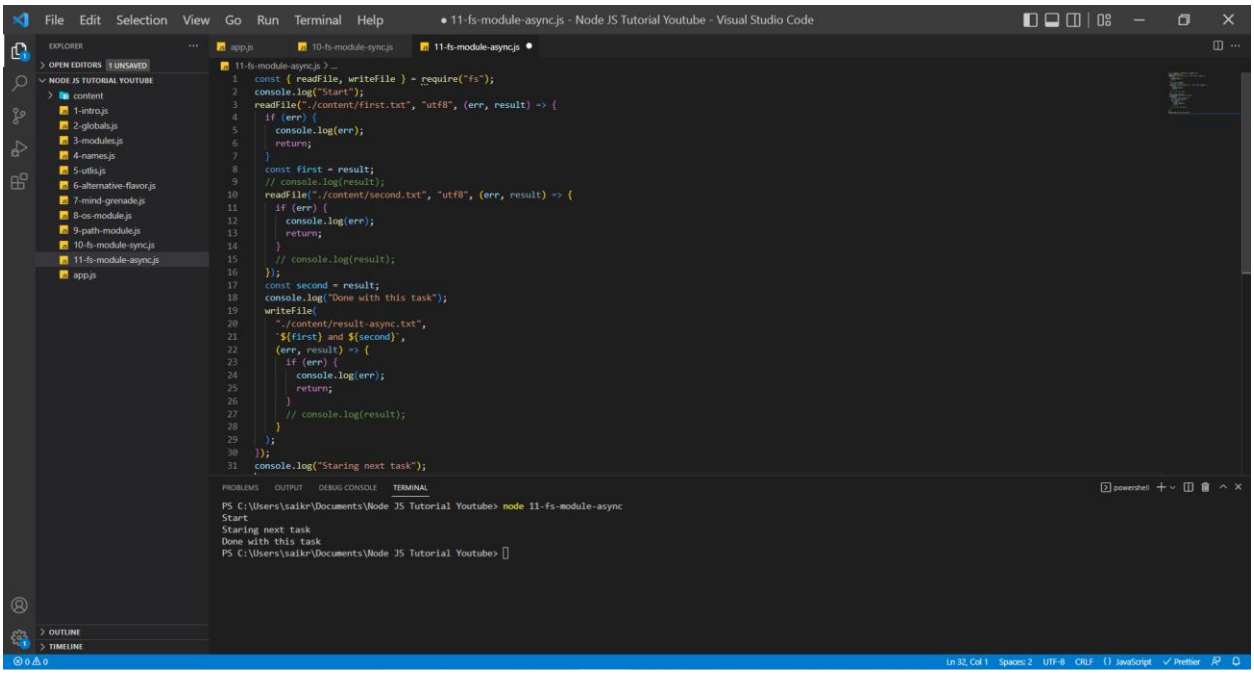
Sync vs Async

In Synchronous way, JavaScript executes code in line-by-line fashion, which means it executes line after line even if task 2 takes more time, it will wait until the task 2 execution is complete and then moves on to task 3.



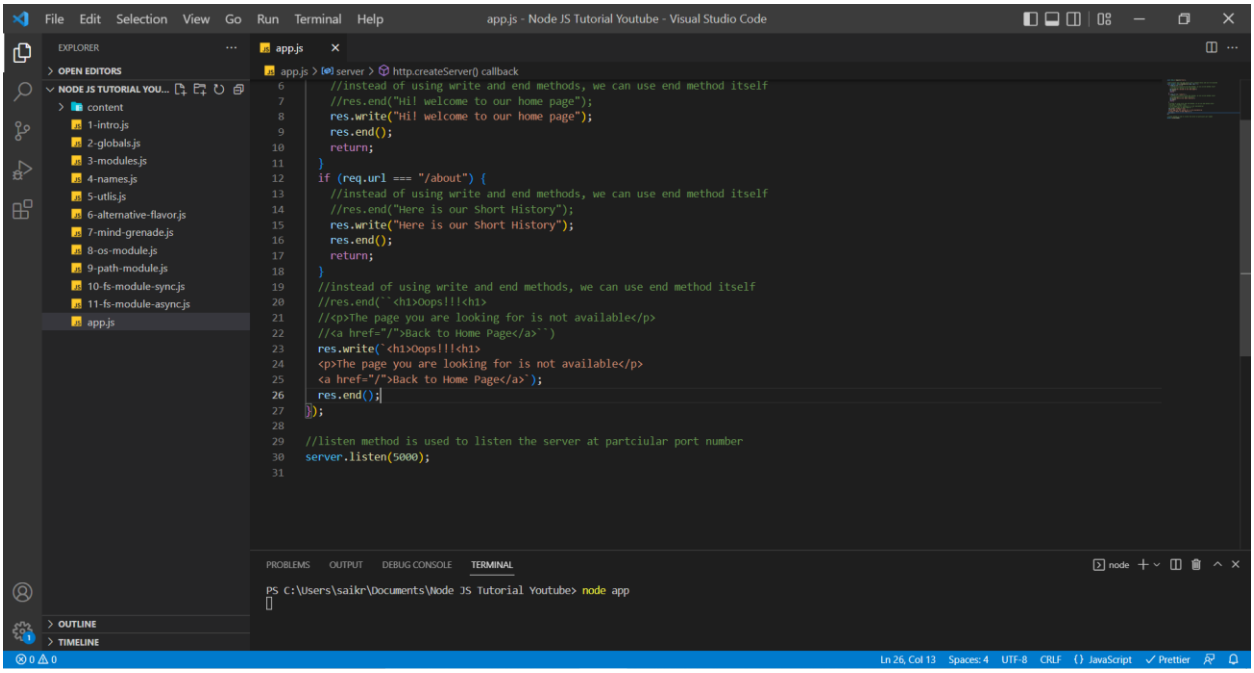
In Asynchronous way, JavaScript executes line-by-line fashion but if it sees any task (assume task 2) taking time it will offload the task2 and move on to another task but in the meanwhile the task 2 runs in the background and once it is done it will provide the results.

In Asynchronous way, we handle the tasks using the callback function but if we use too many callback functions it will create a Callback Hell and hence, we use Promises or Async and Await for asynchronous.



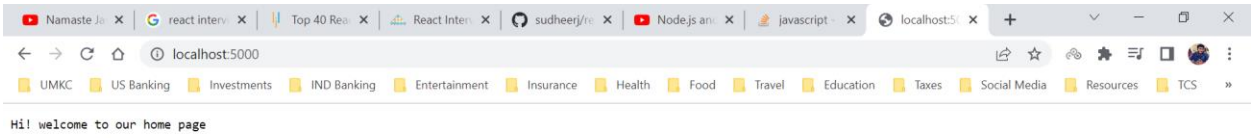
HTTP Module

HTTP Module is used to setup a web server.

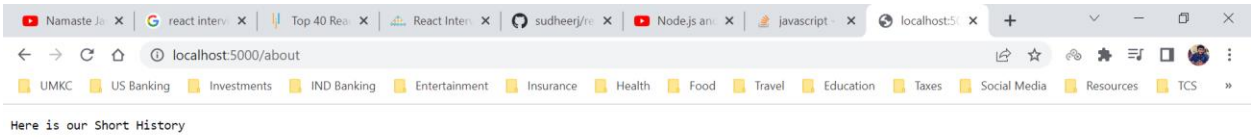


Web Server is started when we run node app.js command in the terminal.

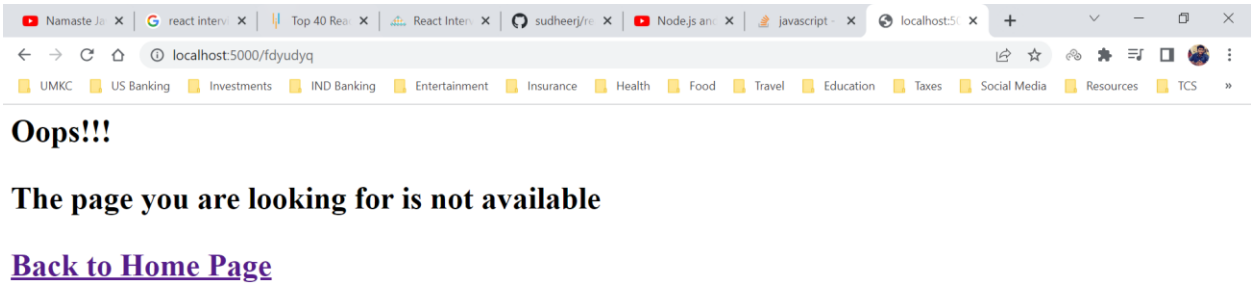
Home Page on localhost:5000



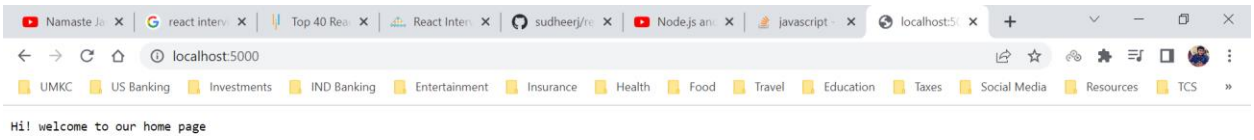
About Page on localhost:5000/about



Error Page on localhost:5000/fdyudyq



On Click of Home Page Link, we will be redirected to Home Page



NPM Information

When we install node, we automatically install NPM (Node Package Manager). NPM enables to do 3 things

1. Reuse our code in another project.
2. Use code written by other developers using a npm command.
3. Share our solutions with other developers as well.

NPM is hosted on npmjs.com, here we can find useful utility functions to full blown frameworks and libraries.

Example: React is available in npm using ***npx create-react-app <app-name>***

A typical node project will have more than few packages installed as dependencies.

NPM calls the reusable code, basically a package. A package is a folder that contains a JavaScript code. Package can be also called as dependency and module as well.

There is no quality control is npm registry. Anyone can publish anything on the registry. A good indication for good and secure package is the number of weekly downloads.

NPM in Node

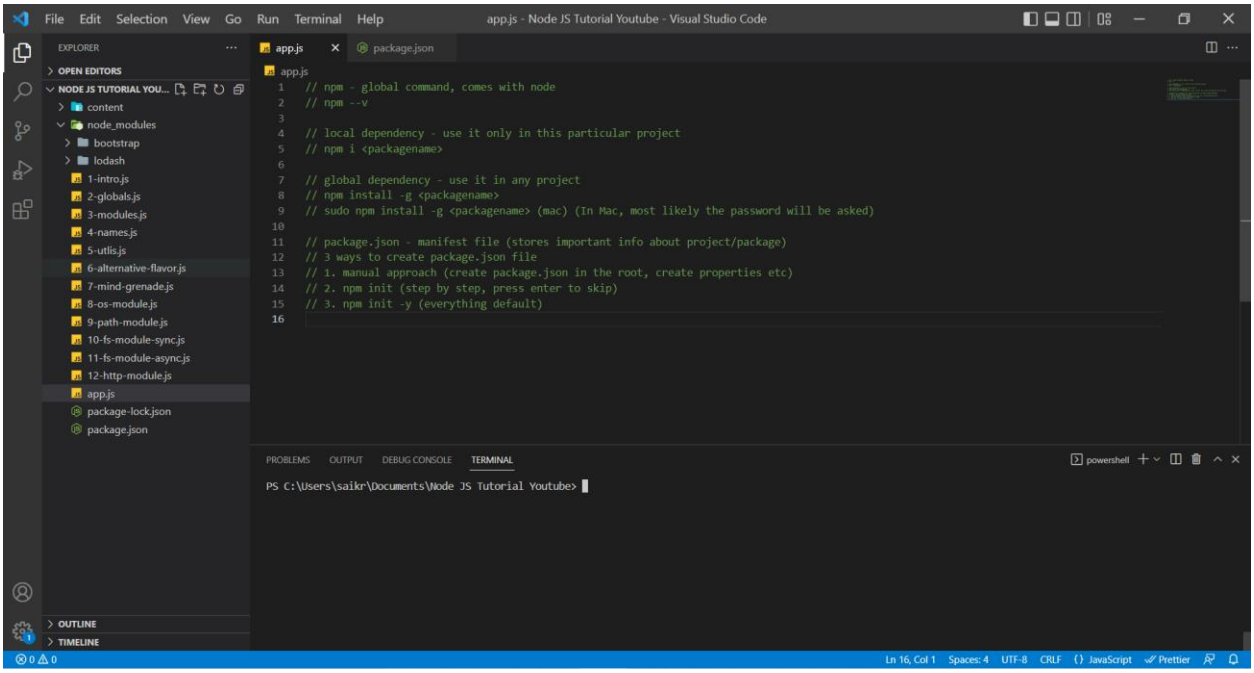
Since NPM is already installed with node, In Node JS environment we have access to NPM global variable npm.

npm - -v

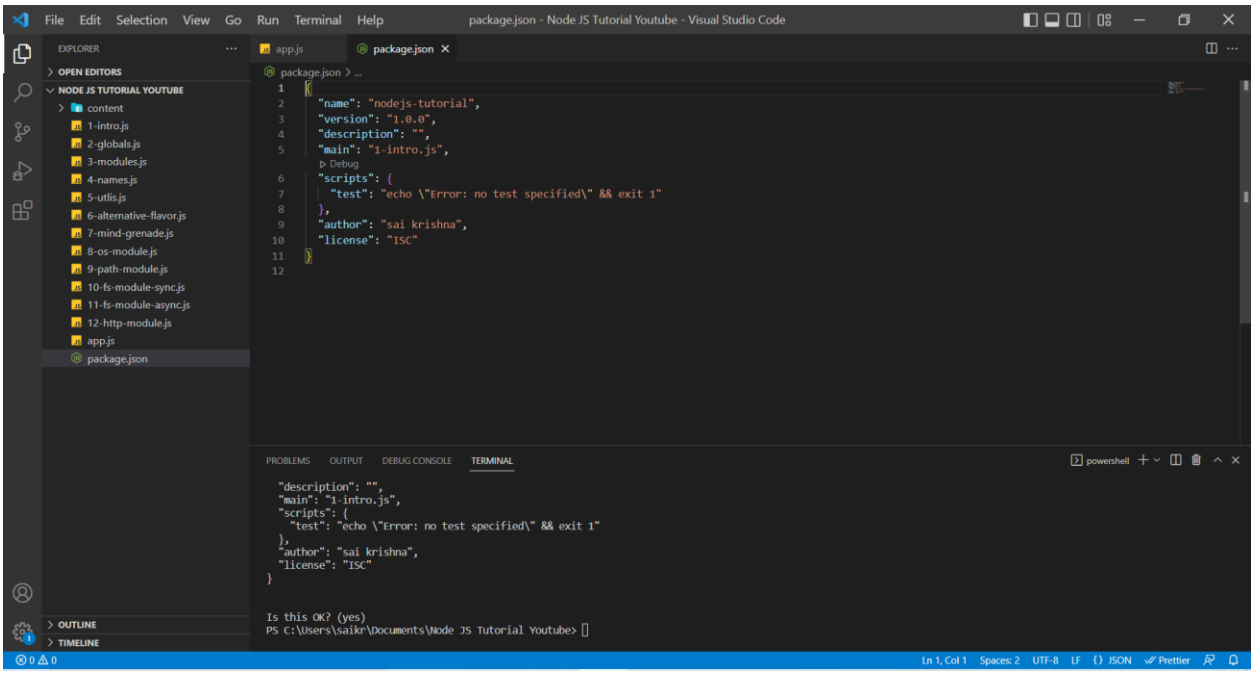
package.json - manifest file (stores important info about project/package)

3 ways to create package.json file

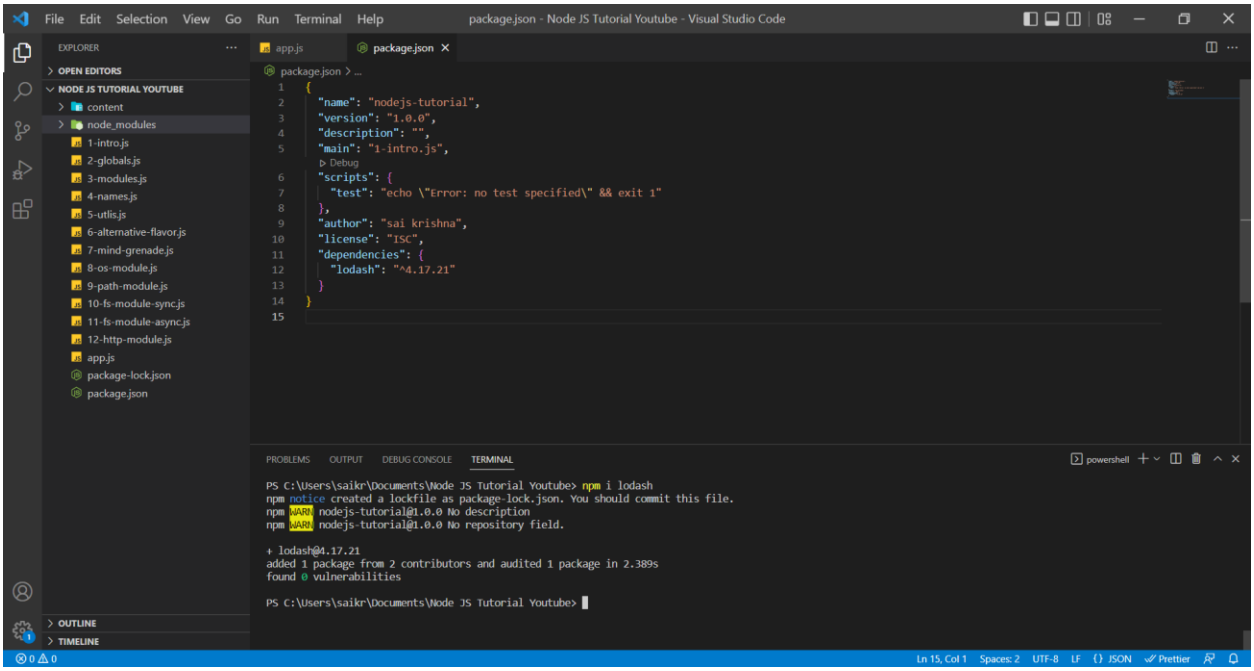
1. manual approach (create package.json in the root, create properties etc.)
2. npm init (step by step, press enter to skip)
3. npm init -y (everything default)



If you wish to publish this NPM Registry, then select a unique name for package which is not available in the NPM registry.

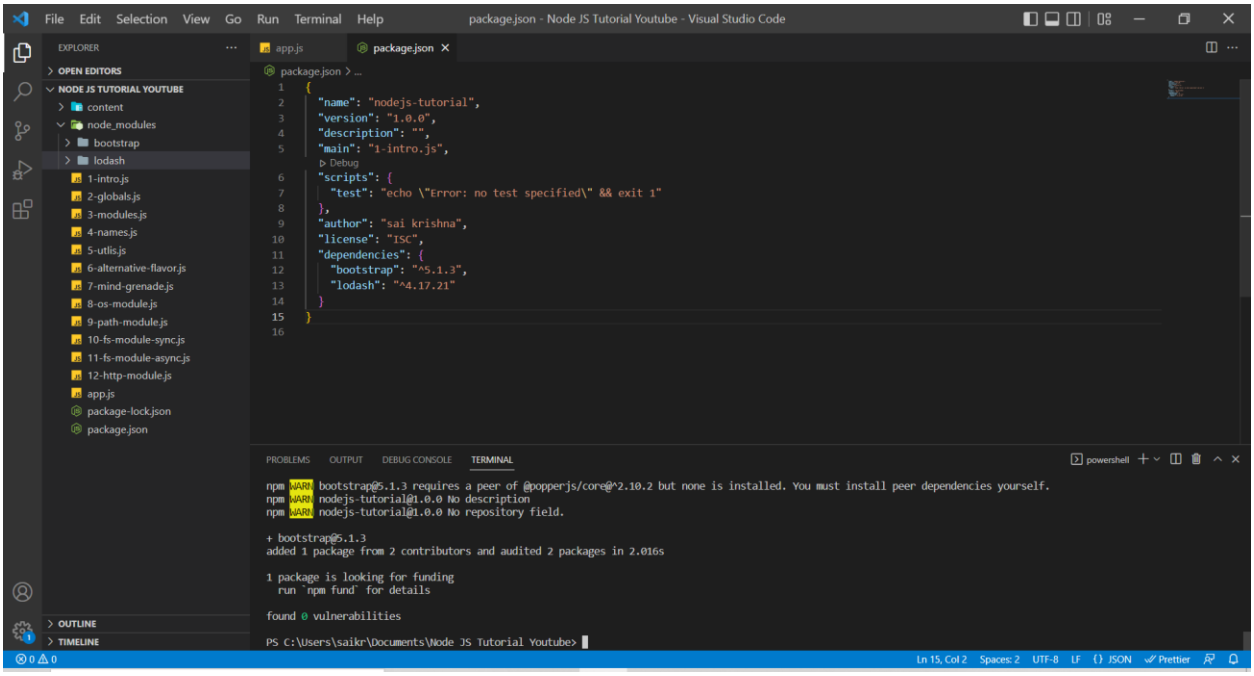


We will try to install first package into our project. The lodash is a utility library. Now we can see dependencies object inside the package.json file. We can also see node_modules folder is installed with loadash dependency.



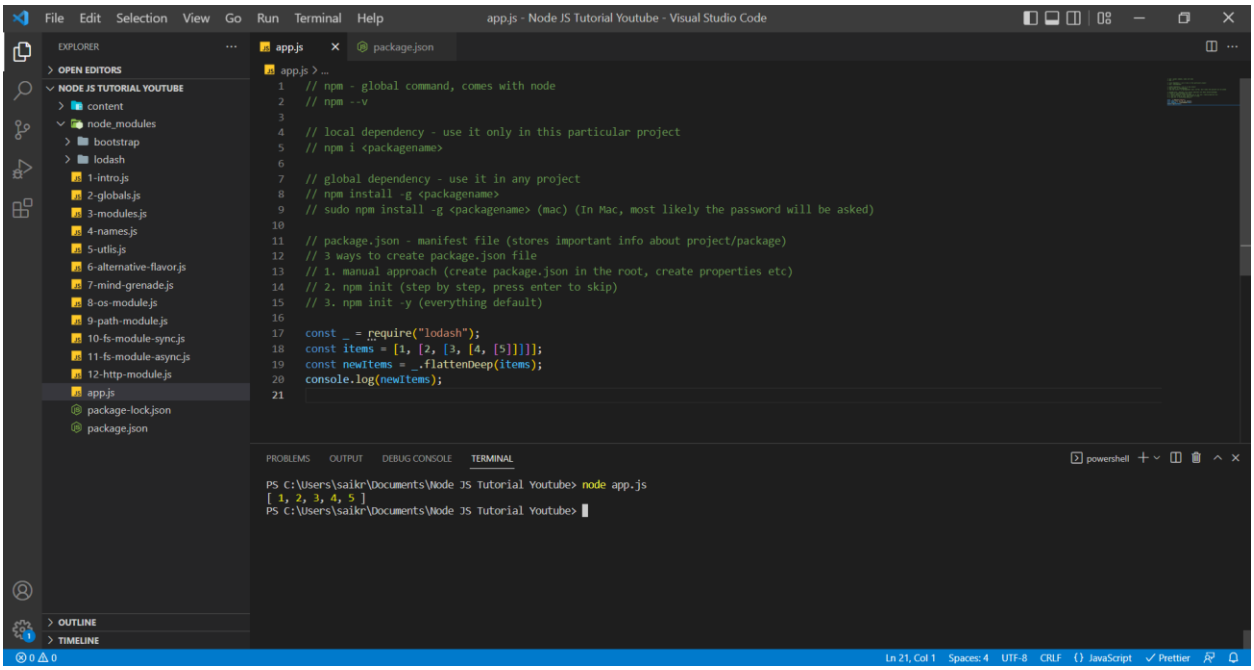
node_modules is the folder where all the dependencies are stored.

Installed bootstrap library as well.



In case of external dependencies, we need to first install them through NPM if we don't install then we won't be able to find it. In case of internal dependencies like in-built modules, we don't need to install anything.

Using loadash in our project

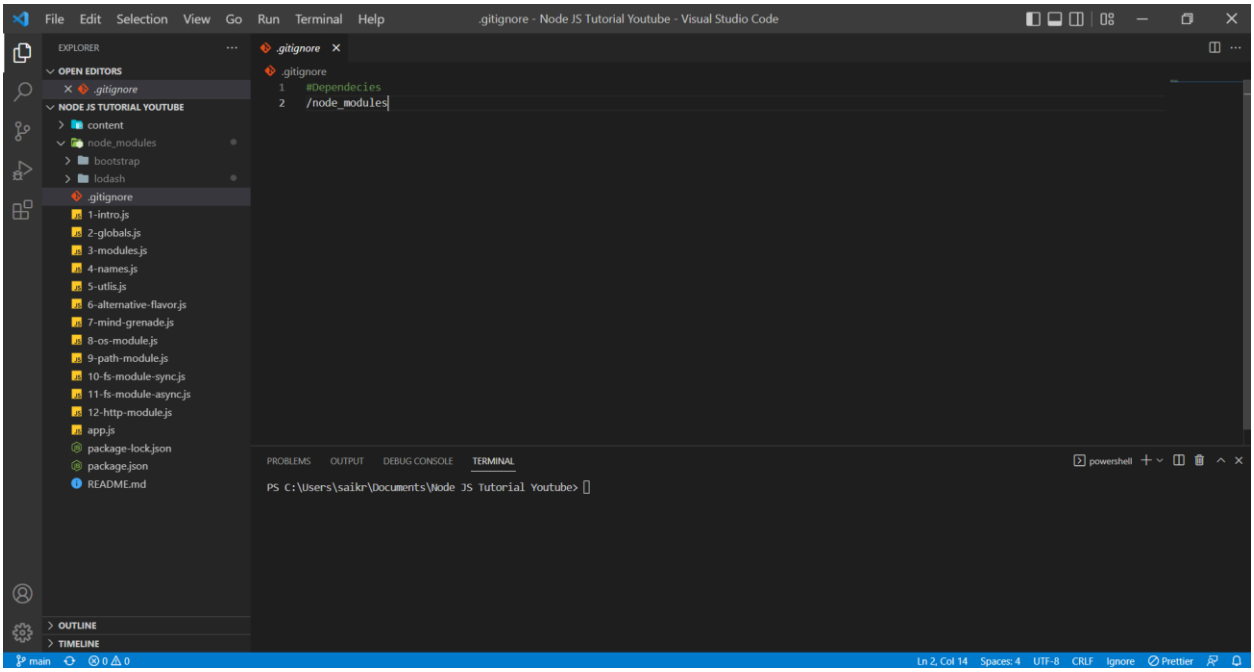


Sharing Code with Other Developers

We can share the code using the github repository. We cannot upload node_modules folder into the github repository. Hence, we create a file named .gitignore which will ignore the folders that we mention in the file.

It is very important to have package.json file because, we can take a repository from github and then use the command npm install to install all the dependencies into our project.

.gitignore file as mentioned earlier.



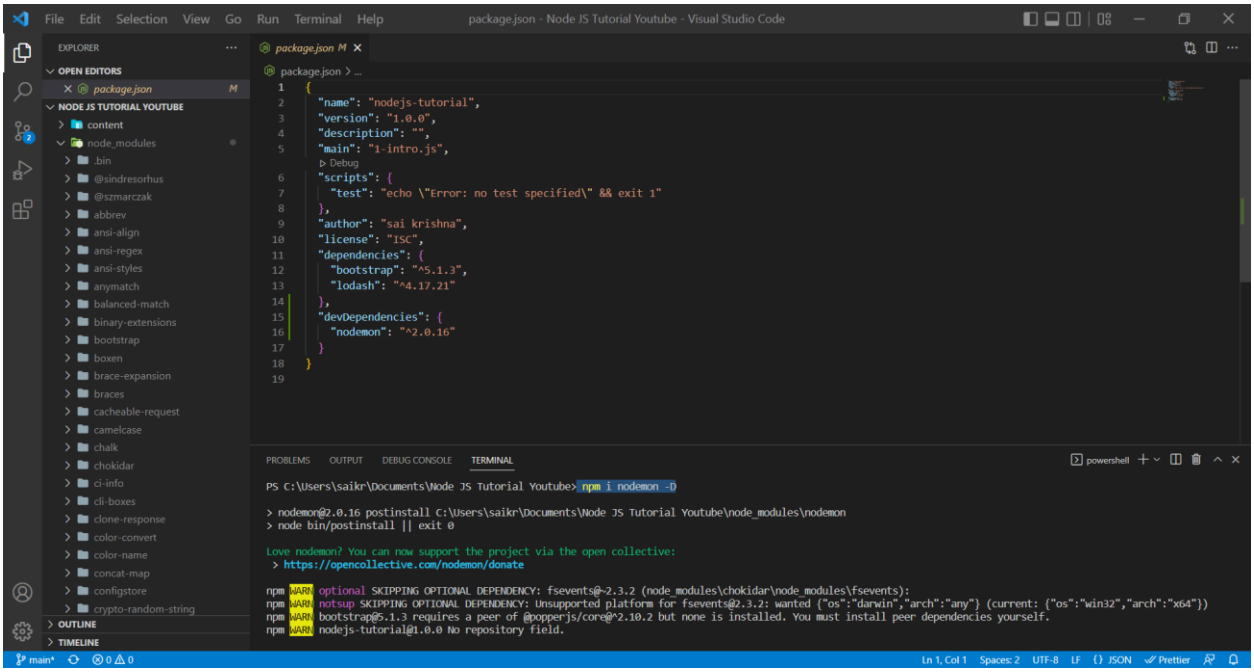
Nodemon

Nodemon is one of the packages of the npm. It is going top watch our file system and restart the application for us. In that way we don't have to type node <filename>. We can install it as dependency but here we are installing it as a dev dependency.

Command to install a package as dev dependency

npm i <package-name> -D or npm i <package-name> - -save-dev

Nodemon as dev dependency In package.json file



In production we may not use the Nodemon, but Heroku and other services will be using other serious packages but in development stage this is an option.

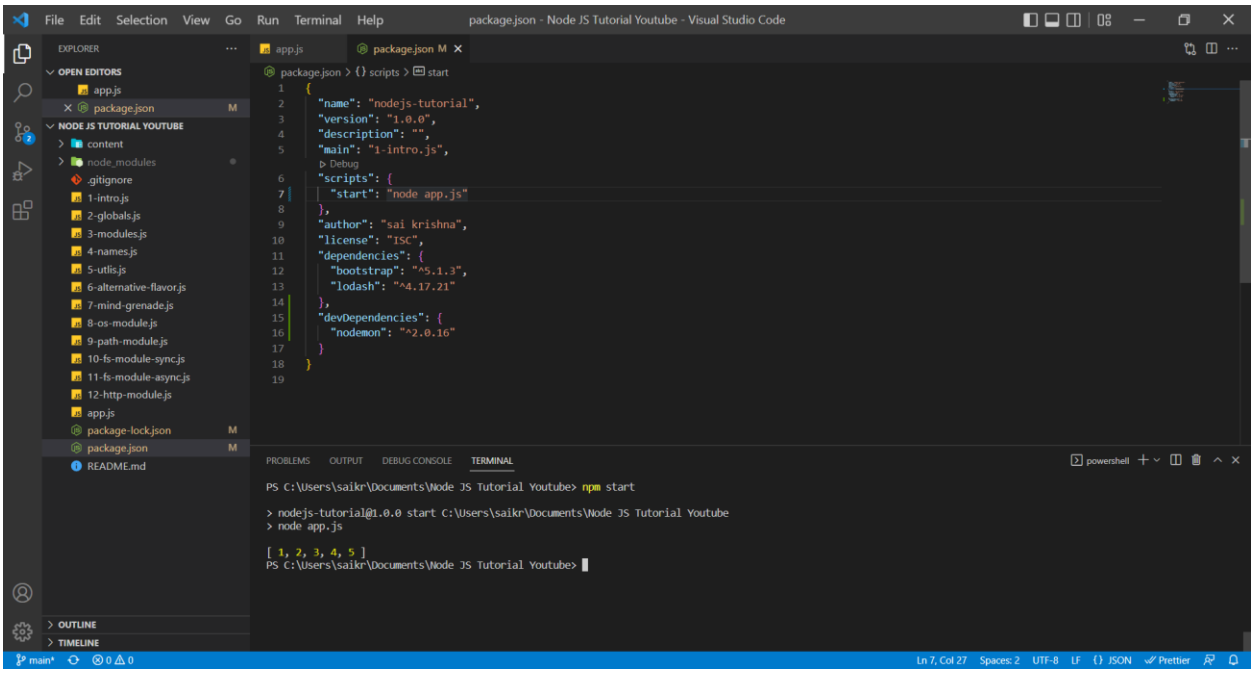
Basically, devDependencies is nothing but the dependencies that we are using for the development purposes but once we get into production, we only use the dependencies used by our application.

Inside the scripts object of the package.json file we just setup some commands which we want to run. For example, we setup start which indeed is a replacement for command ***node <file-name>.***

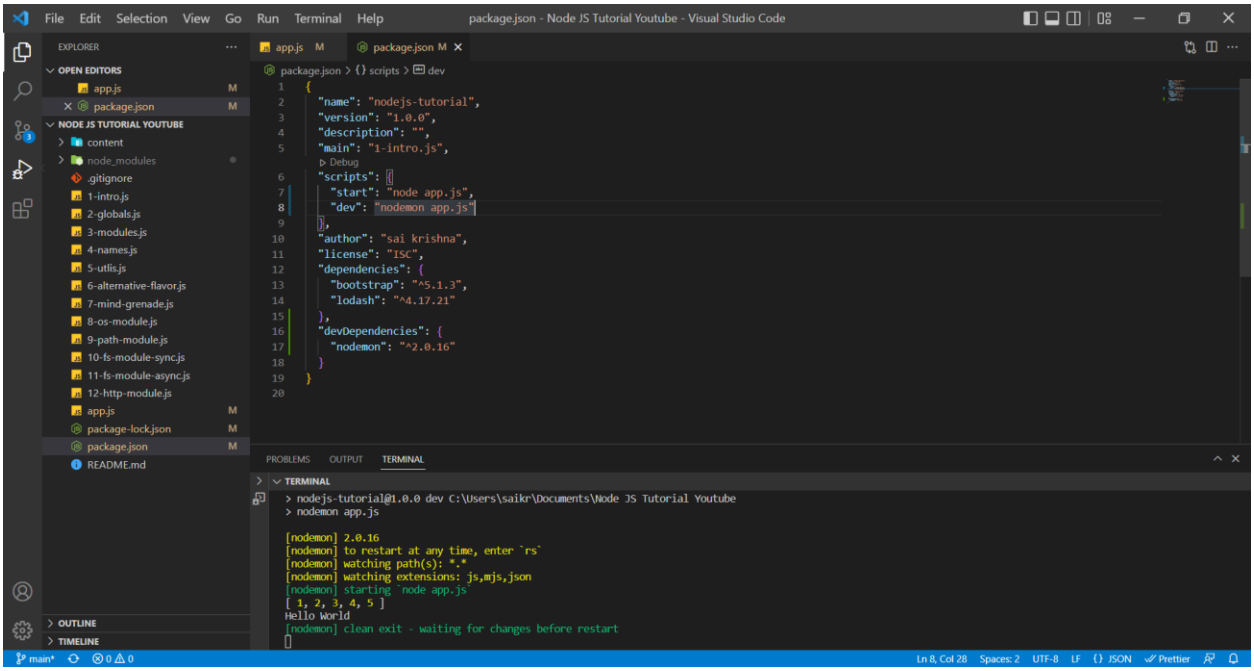
We have run command ***npm start*** which in turn is starting the node application.

For some commands, we just need to use npm <name-of-command> like ***npm start***.

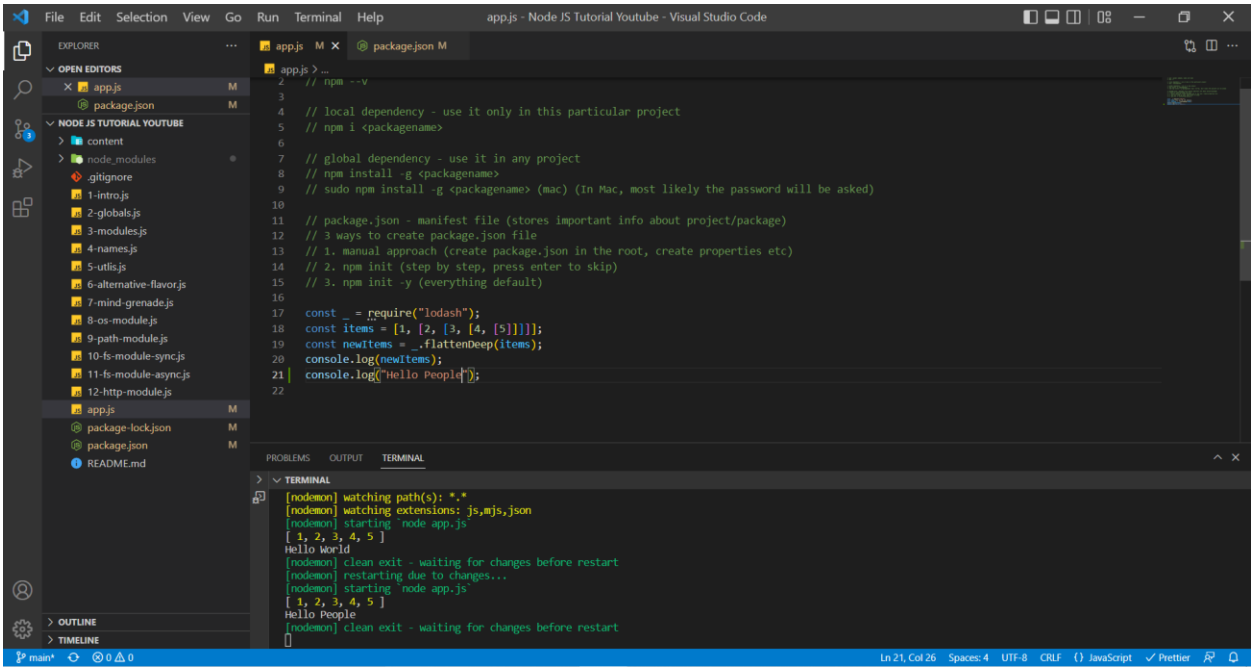
With the ***start*** command which is nothing but ***node app.js***, we are stopping the application after executing the command.



For some other commands, we need to run npm run <name-of-command> like ***npm run dev***.



With the Nodemon package, since it watches the file system and then any if change appears in our file system, it immediately restarts our application.



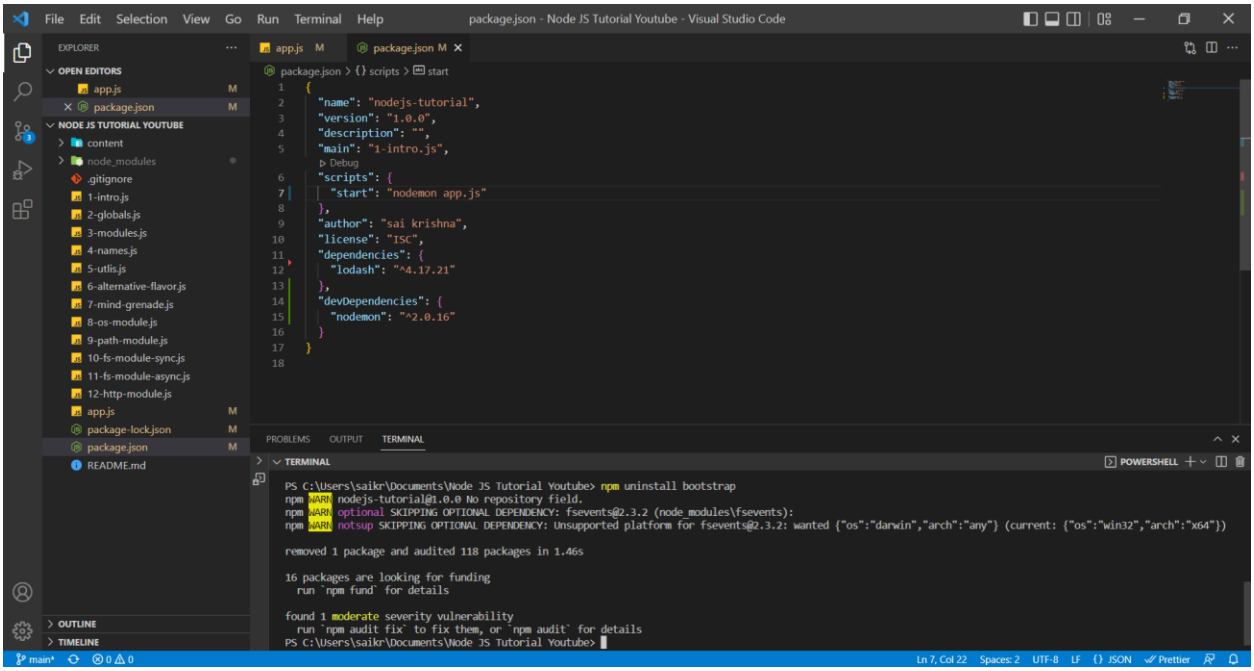
In the above example, we just changed the statement inside the console.log from “Hello World” to “Hello People” and Nodemon immediately recognized and restarted the application and executed the code.

Use Control + C command to stop the Nodemon.

Uninstall a Package in package.json file

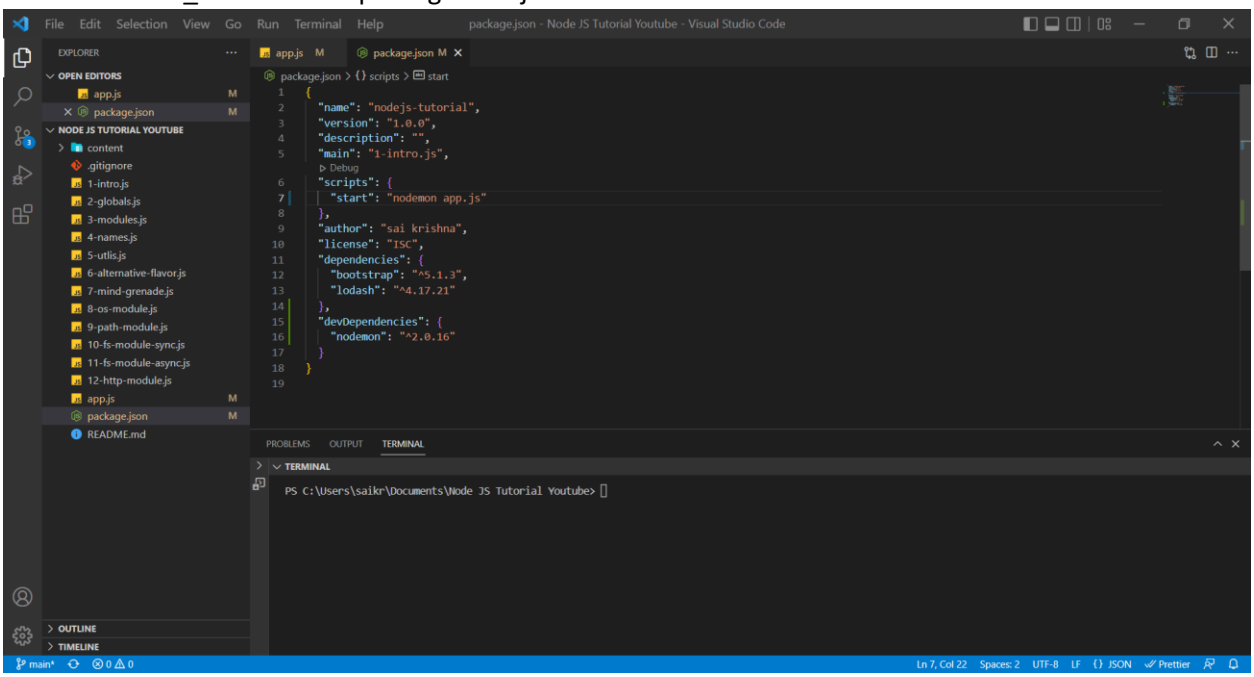
There are two ways to uninstall a package.

1. Run command npm uninstall <package-name>

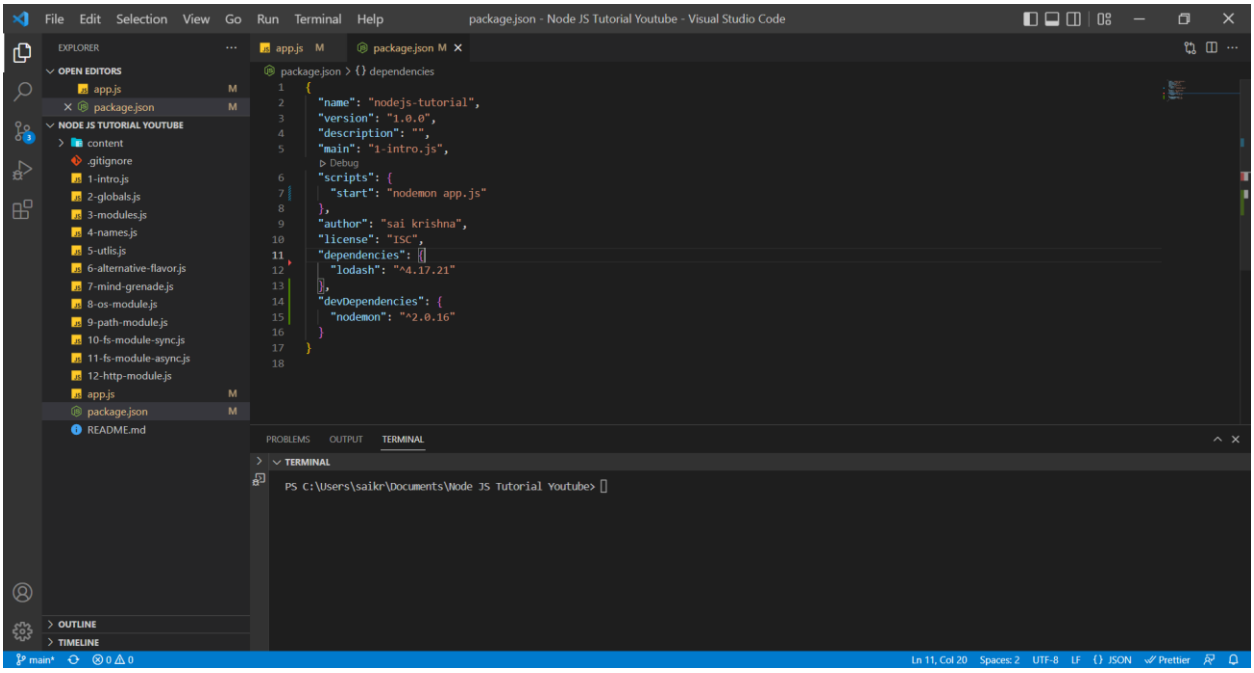


2. Nuclear Approach is deleting node_modules folder, package-lock.json file and removing the package name from package.json file and run command npm install

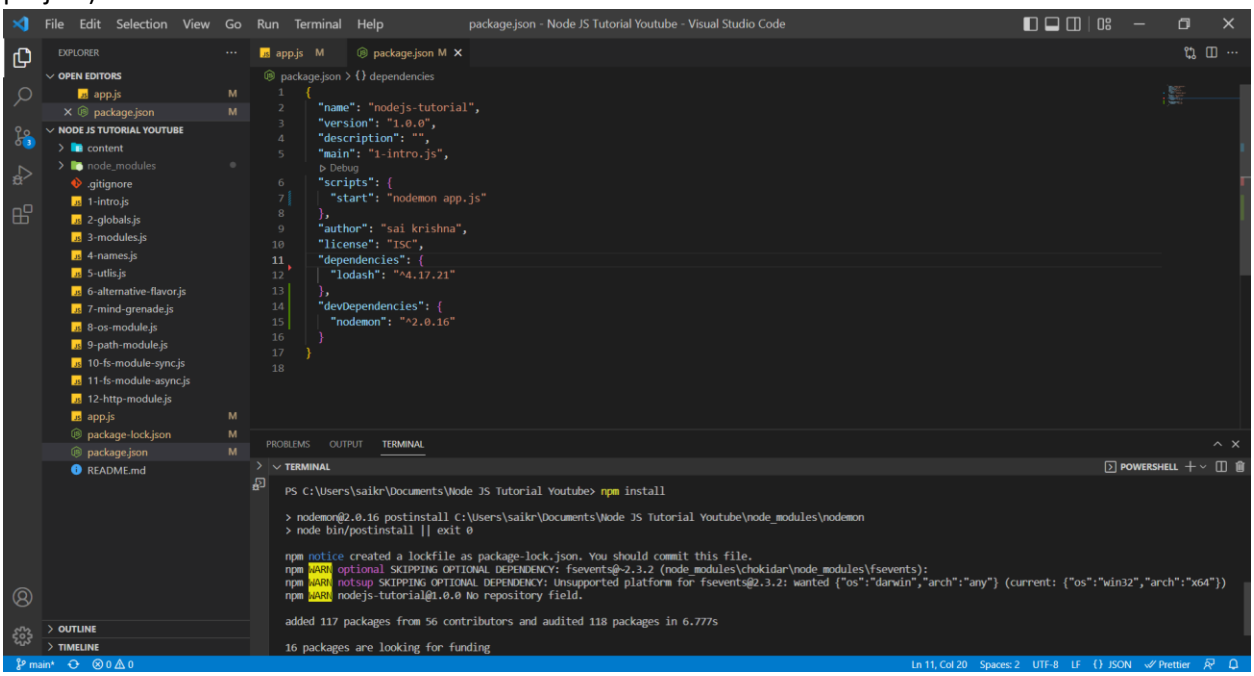
Removed node_modules and package-lock.json file



Removing the package name (bootstrap) from package.json file



Run command ***npm install*** (node_modules will installed and package-lock.json will be added to project)



Nuclear Approach is followed in Gatsby applications.

Global Install

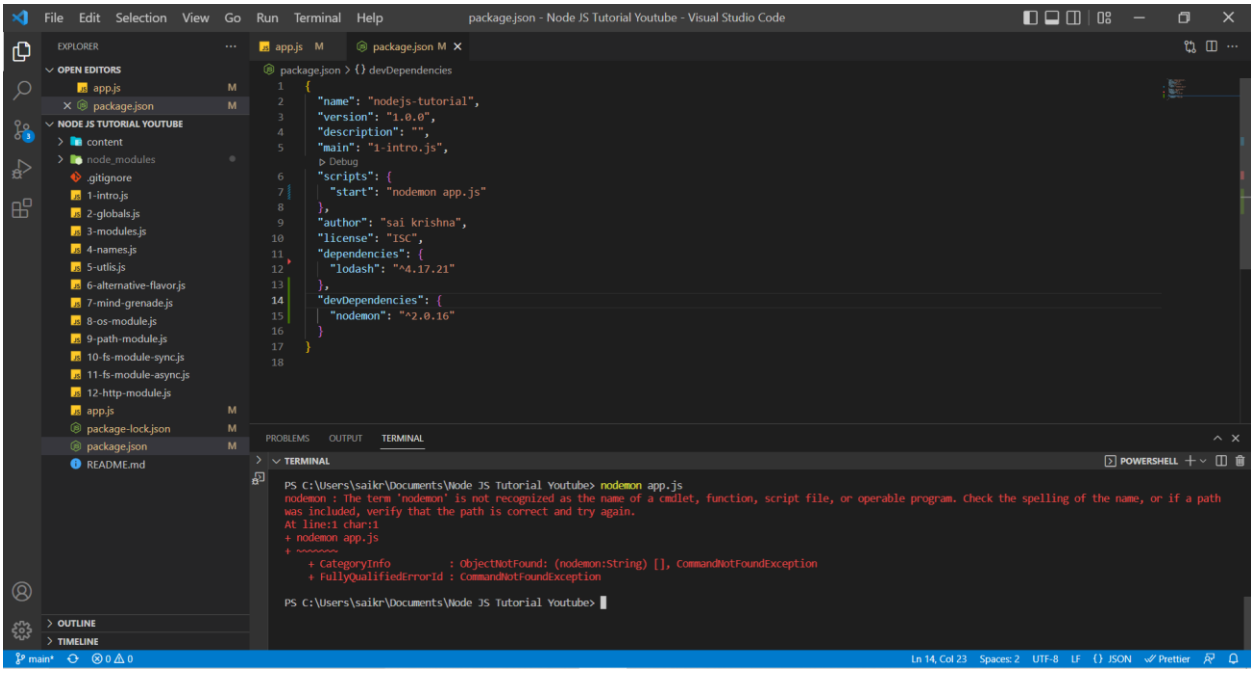
In a scenario, assume we have 20 nodal applications, and we are constantly working on node applications, so every time we can't go and change our application name in the scripts object in package.json file. To make our work easier we make nodemon package global, where we can run command ***nodemon <file-name>*** to run an application and restart the application whenever there is a change in the file system.

To Install a package globally, we run the command in the following way

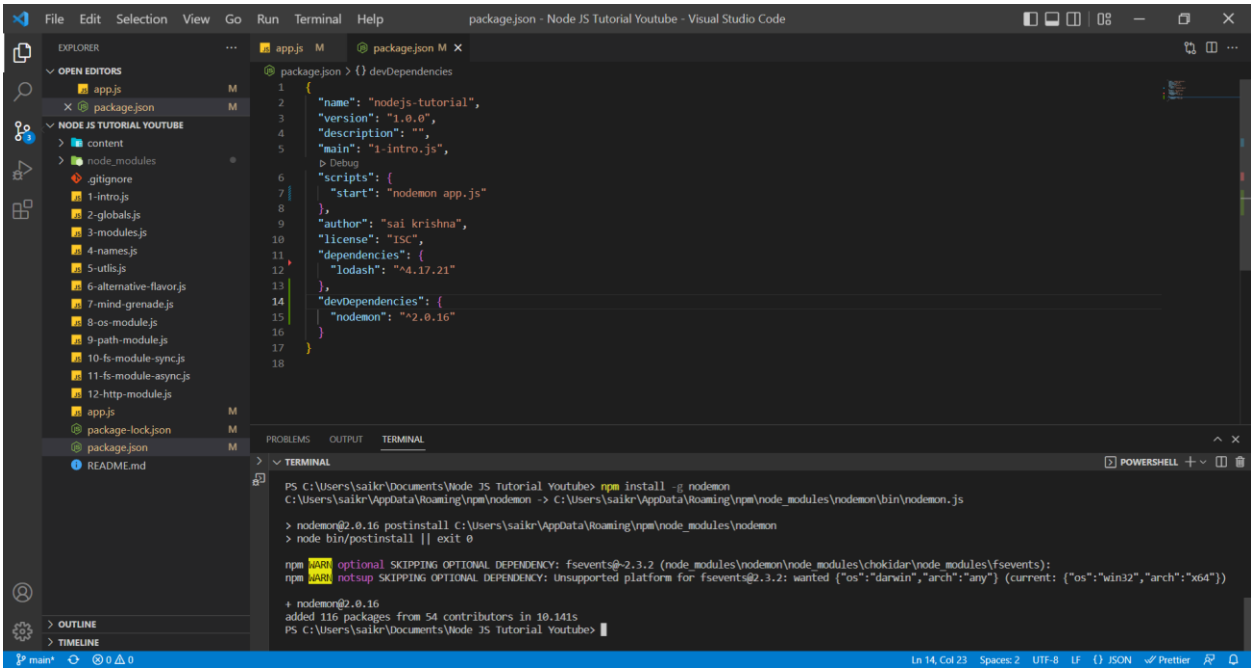
npm install -g <file-name>

Ex: ***npm install -g nodemon***

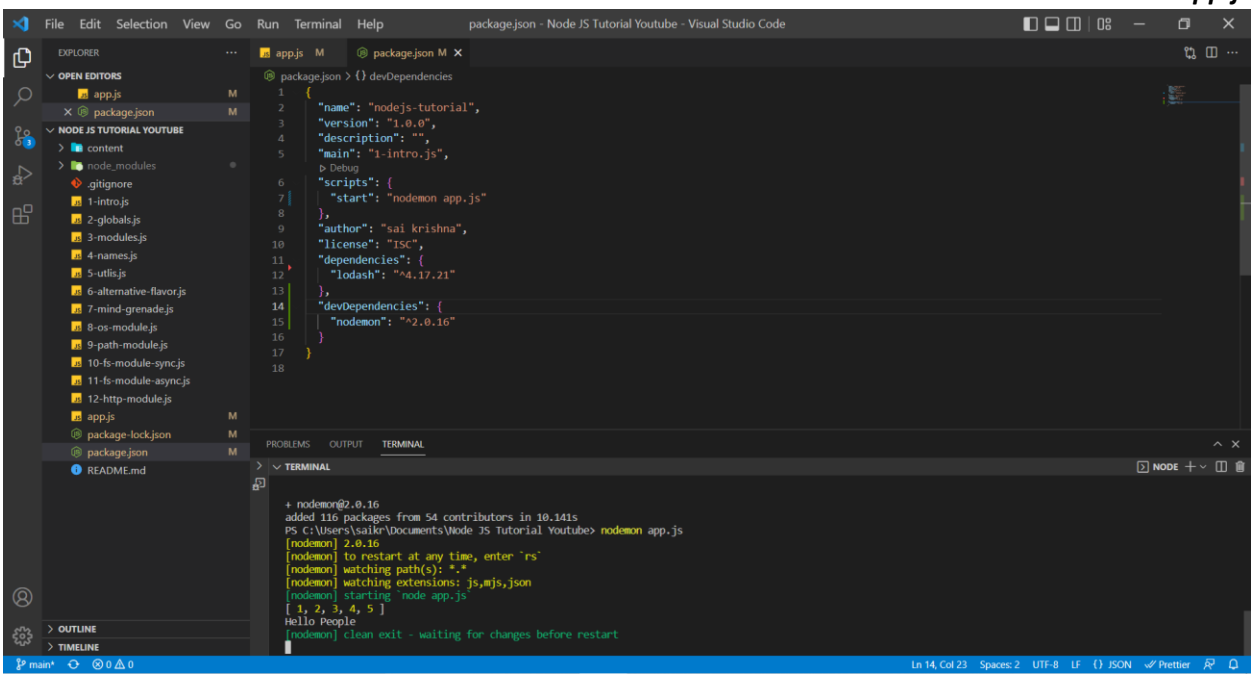
Before nodemon was installed globally, we get the error because it can't recognize it globally.



After installing globally,



Run command ***nodemon app.js***



Before installing the nodemon package globally, it won't be recognized in the terminal but when we installed it globally it can be recognized.

Earlier we used to install frontend libraries like react and Gatsby globally but when **npx** came into light, the route has shifted by installing a package globally to use **npx** command.

Ex: npx create-react-app my-app

npx stands for Node Package Execute and official name is Package Runner. It is a feature that is introduced in NPM 5.2. The main idea is to be running the CLI tool without installing the packages globally.

package-lock.json

If you remember some of the dependencies have their own dependencies like bootstrap which dependencies like jQuery and others etc. and those dependencies have versions as well.

Sometimes the dependencies of the packages may change as well by mistake when we handover code from developer to developer and it may introduce bugs into our application since there was change in package and dependencies versions.

Package-lock.json file contains all the versions of the packages and their dependencies as well.

```
"dependencies": {
   "lodash": "^4.17.21"
 },
```

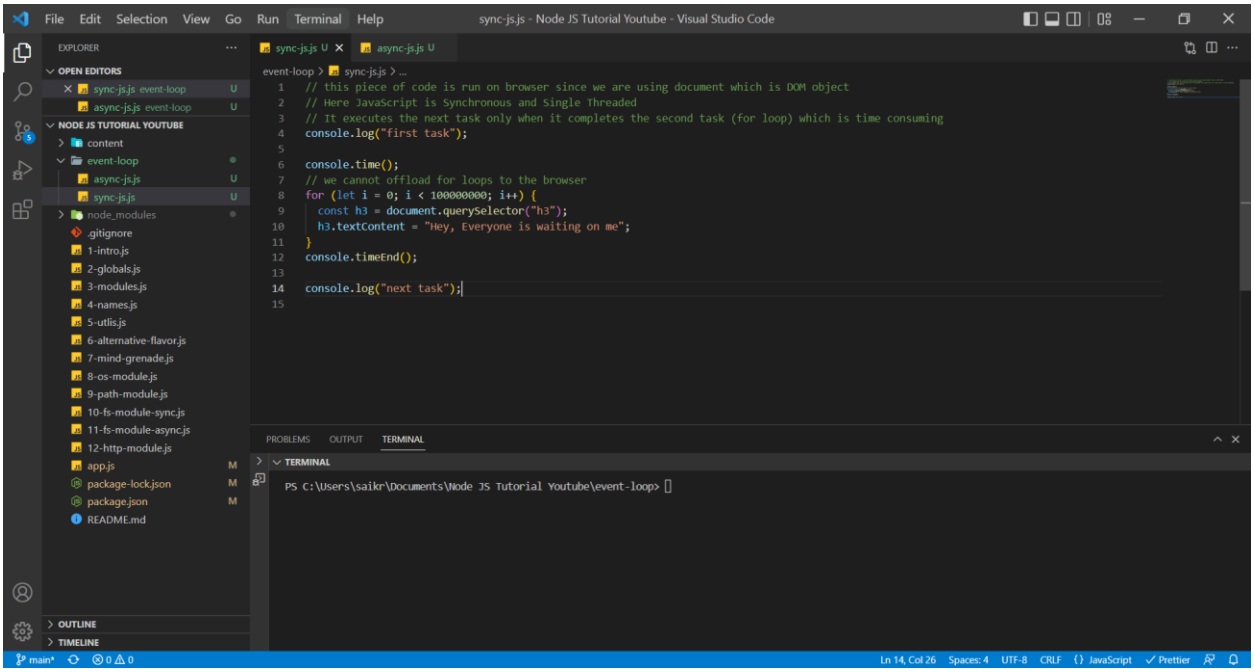
A package version has 3 values. This can be considered as contract between the person who is using the package and the person who developed.

The first number is the major change which means there are breaking changes. The second number is a minor one, which means backward compatible. If we change that to 18, we don't expect any breaking changes. The last one is patch for the bug fix. It is important to remember whenever we publish our own package.

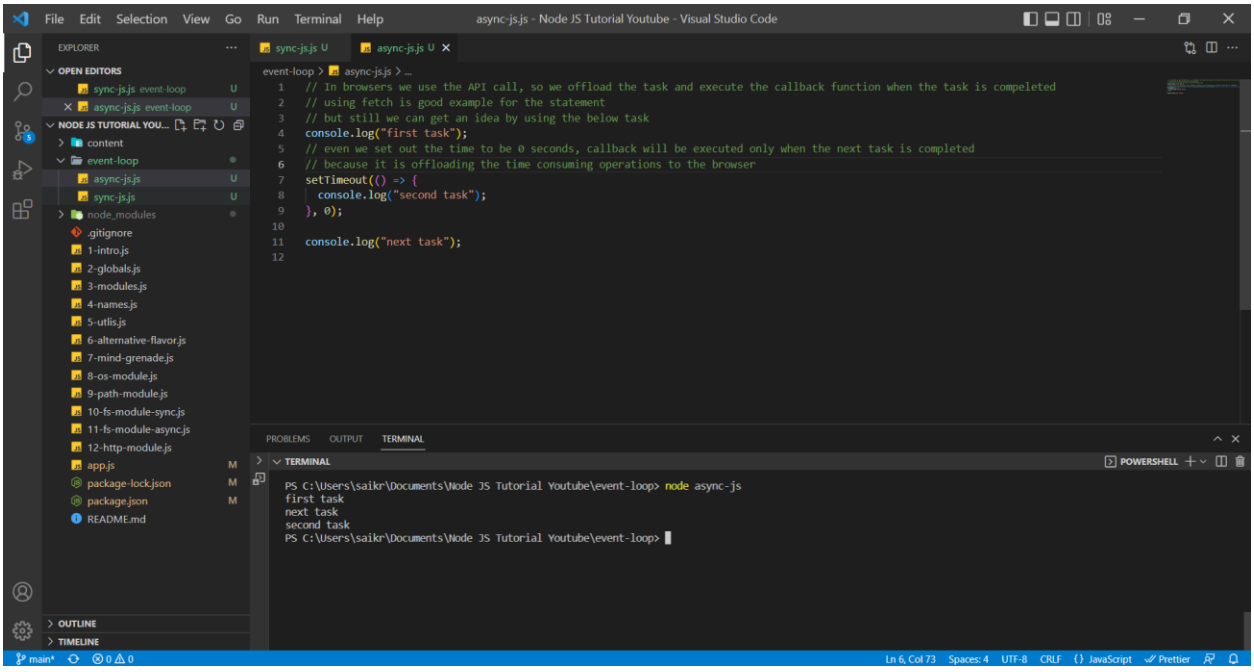
Event Loops (A complex topic)

The event loop is what allows Node JS to perform non-blocking I/O operations – even though JavaScript is single threaded – by ***offloading*** operations to the system kernel whenever possible.

sync-js.js



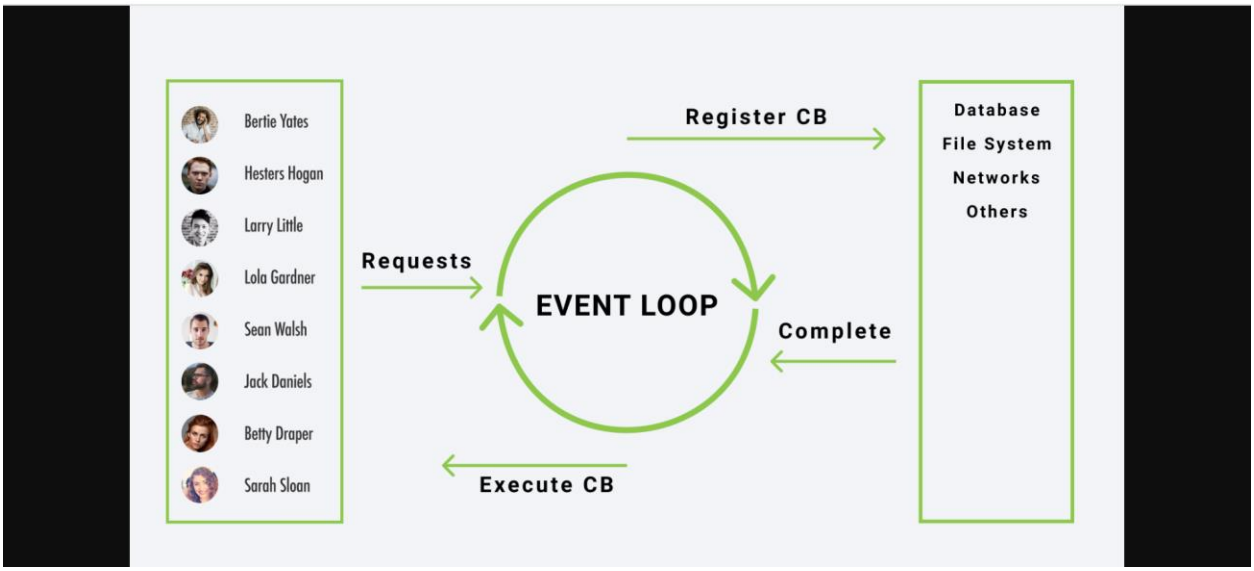
async-js.js



By saying we cannot offload for loop, we can still write blocking code in JavaScript but the browser does provide some nice API's where we can offload the time consuming tasks.

Node JS Event Loop

Assume a scenario where our application is popular and there are 8 users who are requesting information from application and as the requests are coming in the event loop is responsible for avoiding this type of scenario.



In the above scenario, assume Larry Little requesting something from our application which is time consuming, so in this case event loop registers the callback function basically it registers what needs to be done when the task is complete because if event loop wouldn't do that then we would have the above scenario where the request are coming in and because Larry is requesting something that take a long time then rest of the users have to wait.

It's just a fact that we are wasting our time on waiting for the operation to be done and then only we can serve other users but what event loop does is it registers the callback and only when the operation is complete it executes the callback function.

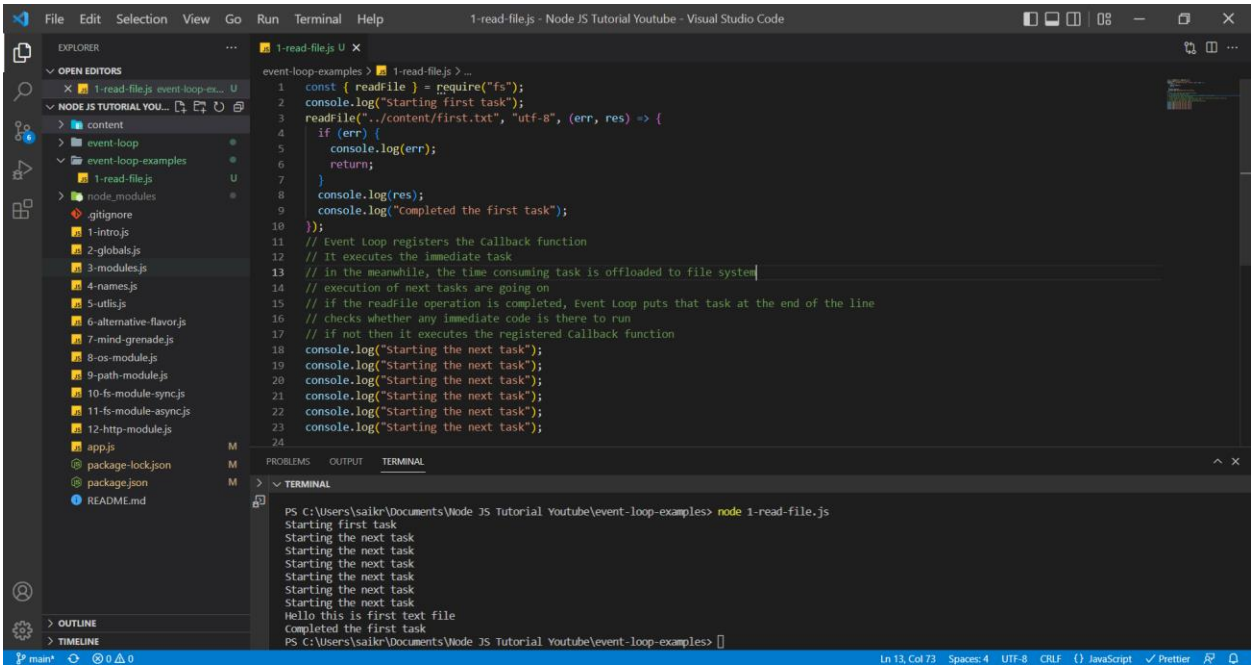
Here the requests are coming in, let's say the operation is complete we first register the callback and operation is complete and instead of executing the callback right away it effectively puts away at the end of the line and then when there is no immediate code to run then we execute the callback.

With the help of Event Loop, we can offload some time-consuming operations and effectively just keep our other requests executed instead of wasting time for the time-consuming operation to complete.

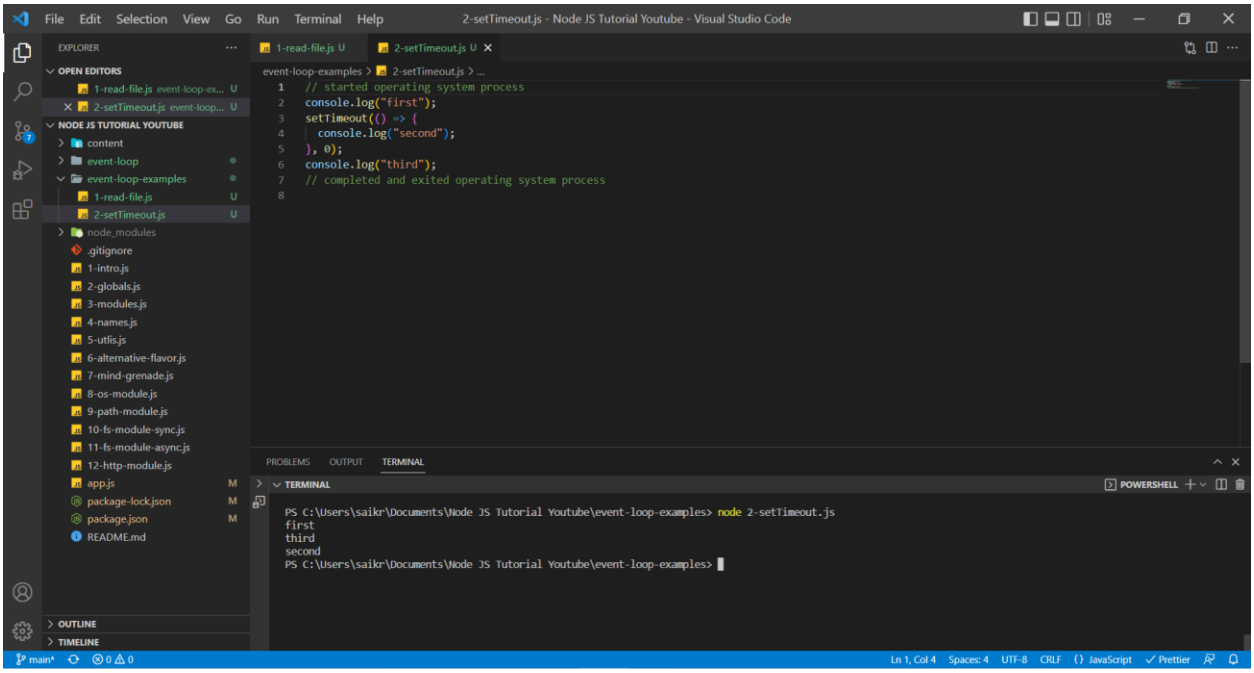
Event Loop Example Codes

Here the Event Loop registers the callback function. Event Loop will offload this task (in this case, it will offload to file system). Event Loop will execute the next tasks and in the meanwhile if the readFile operation is complete (it may provide error or data) Event Loop puts that operation at end of the process/list and executes the remaining tasks. After all the tasks are executed and when there are no immediate lines to be executed then Event Loop executes the registered callback function. That is the reason you will see ***result*** and ***completed first task*** statement at the bottom of the screen.

1-read-file.js

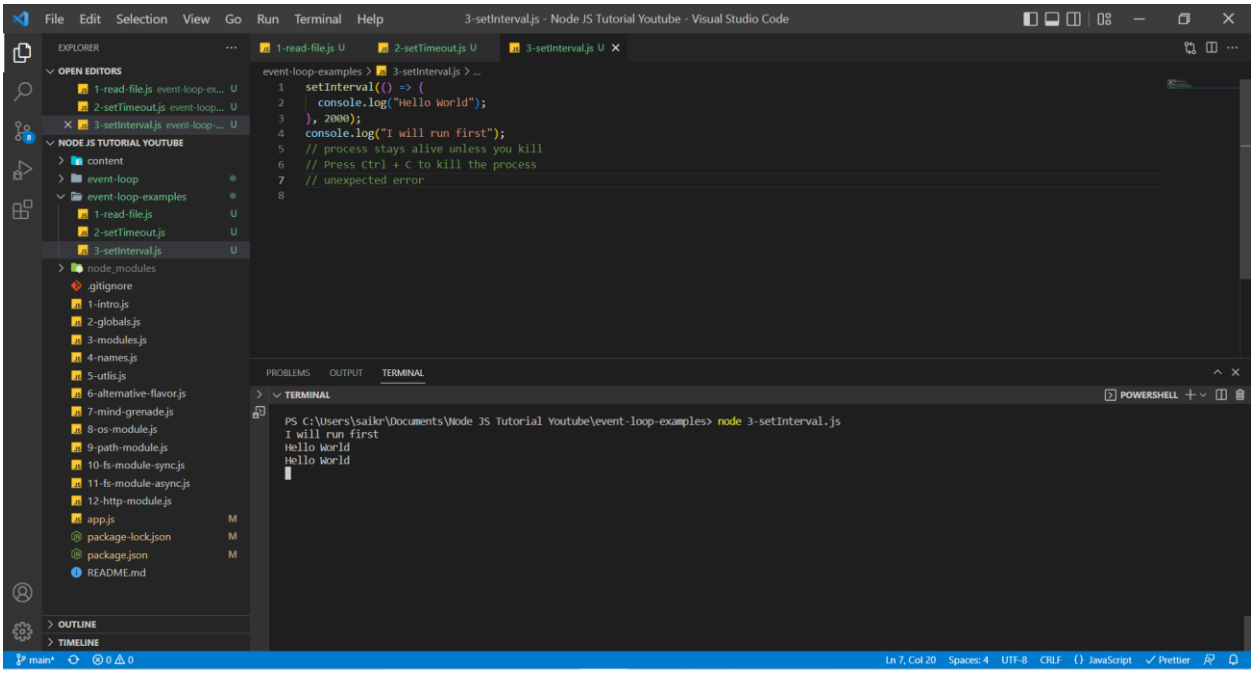


2-setTimeout.js



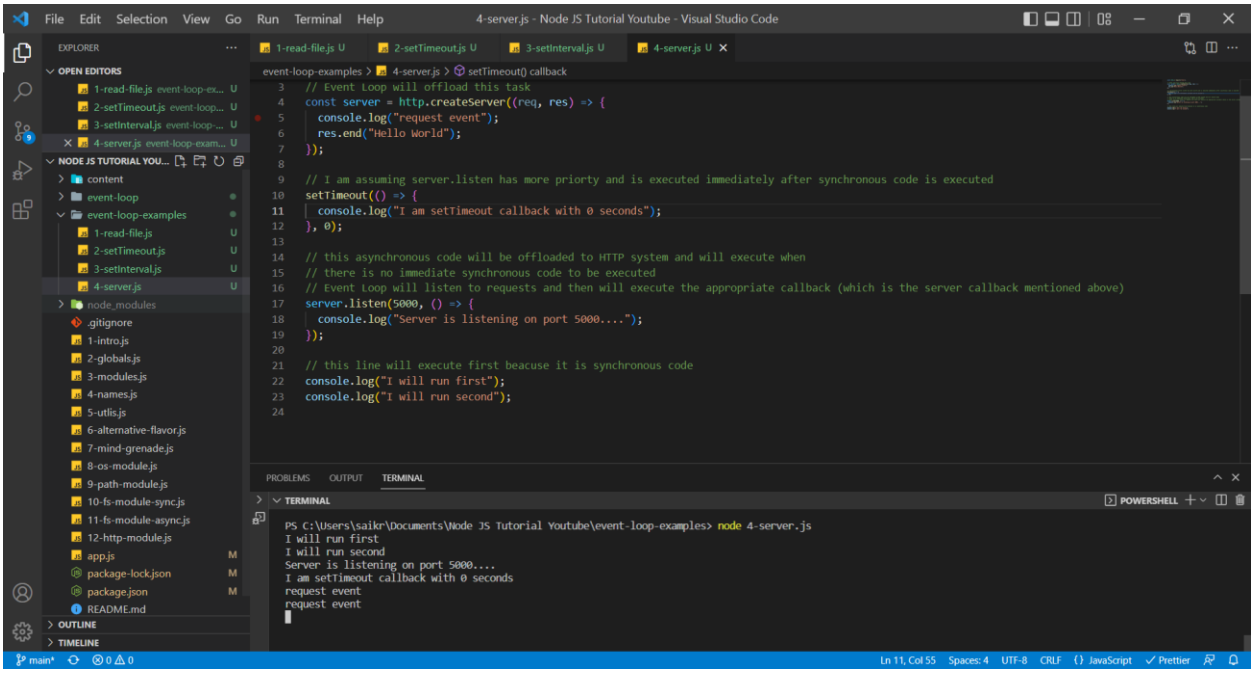
Even though setTimeout has 0 seconds (it is asynchronous code), the Event Loop will offload the task (in this case it will be offloaded to operating system) and execute the immediate tasks (synchronous code) and then there is no immediate task to execute then Event loop will execute the offloaded task (asynchronous code).

3-setInterval.js

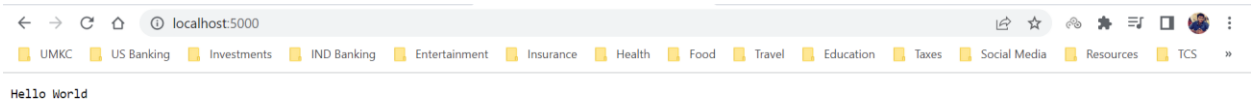


setInterval runs in intervals. So that's why we see the console statement after 2 every seconds. It will be offloaded by Event Loop and then executed when there is no immediate synchronous code.

4-server.js



Request from Webpage

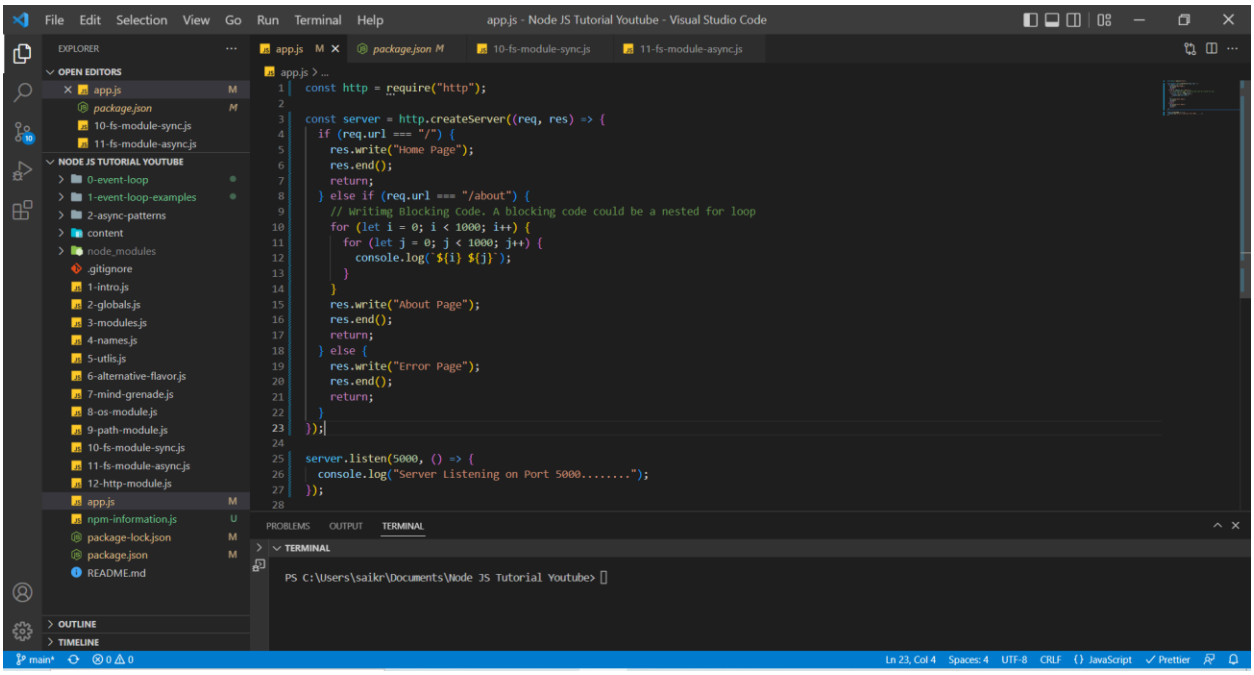


Here, we are starting the server and then listening to the server in port 5000 and whenever we listen a request from the server, we execute the appropriate callback function and return the response.

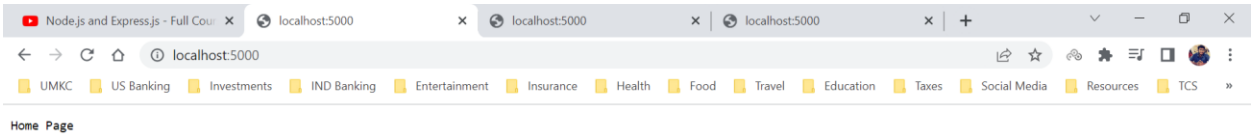
server.listen() process stays alive because listen is an asynchronous function and the moment, we set it up, Event Loop is just waiting for those requests to come in and once they come in Event Loop runs the appropriate callback function and sends the response back to the server. This process will end when we terminate or close the terminal.

Async Patterns in Node JS

As we already discussed callback hell in File System Module. Callback Hell is nothing but implementing one callback function inside another callback function and going on. Callback Hell will mess up the code and hence we use different approaches to solve the code blocking in JavaScript.

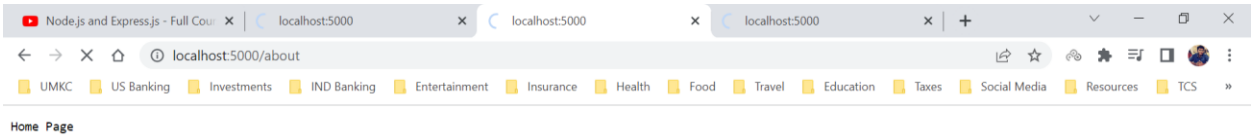


In the above code, a server is created and listening to the port at 5000 and responds to the request for three 3 URLS. We can login from multiple tabs and hit the server with a request and we will be served. This works fine until there is no blocking code, but once we have blocking code in any of the request block then we assume that only the user requesting URL but not. In real time, all the users request the server will be blocked since server is servicing the most time-consuming request which is in a synchronous code format.

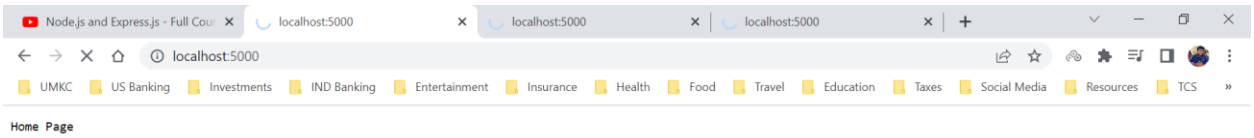


We have 3 different tabs or users requesting the server a response. Server is proving them with a response. In the code we have a blocking piece code for **localhost:5000/about** URL and when a user requests a response from that URL (localhost:5000/about) the requests from other users will also be blocked and response will be provided only that time-consuming request is completed.

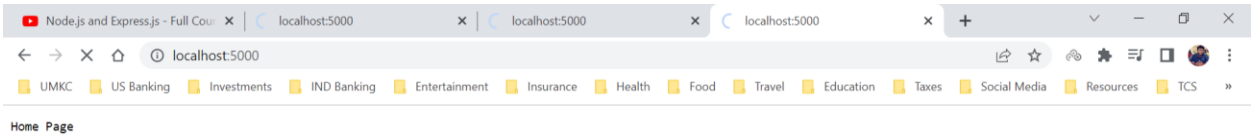
From User 2/ Tab 2 we are requesting About Page (which contains the blocking code)



After that, we requested Home Page from User 1/ Tab 1 and our page is still loading even though Home Page block doesn't contain any blocking code.



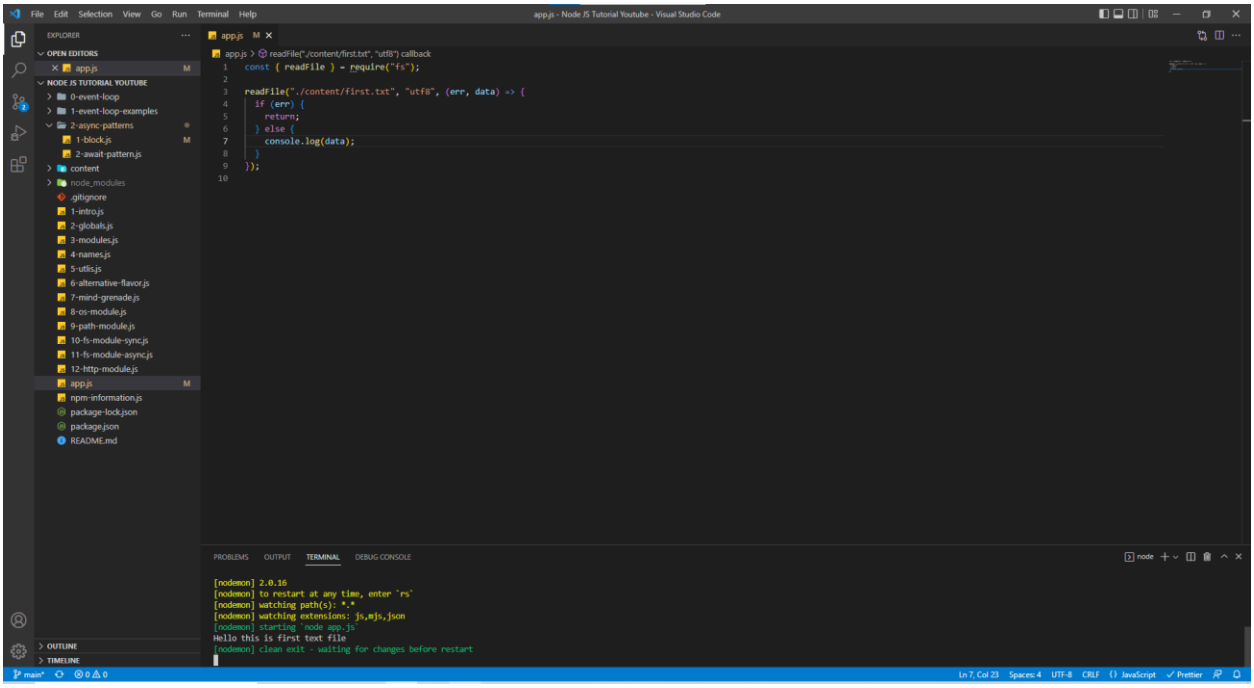
In the same way, User 3/ Tab 3 request Home Page but still page will be in loading state.



User 1 and User 3 request will be resolved only User 2 request is resolved.

This is the representation why prefer the asynchronous approach. We should always strive to set our code asynchronously.

Async Pattern – Promise Setup

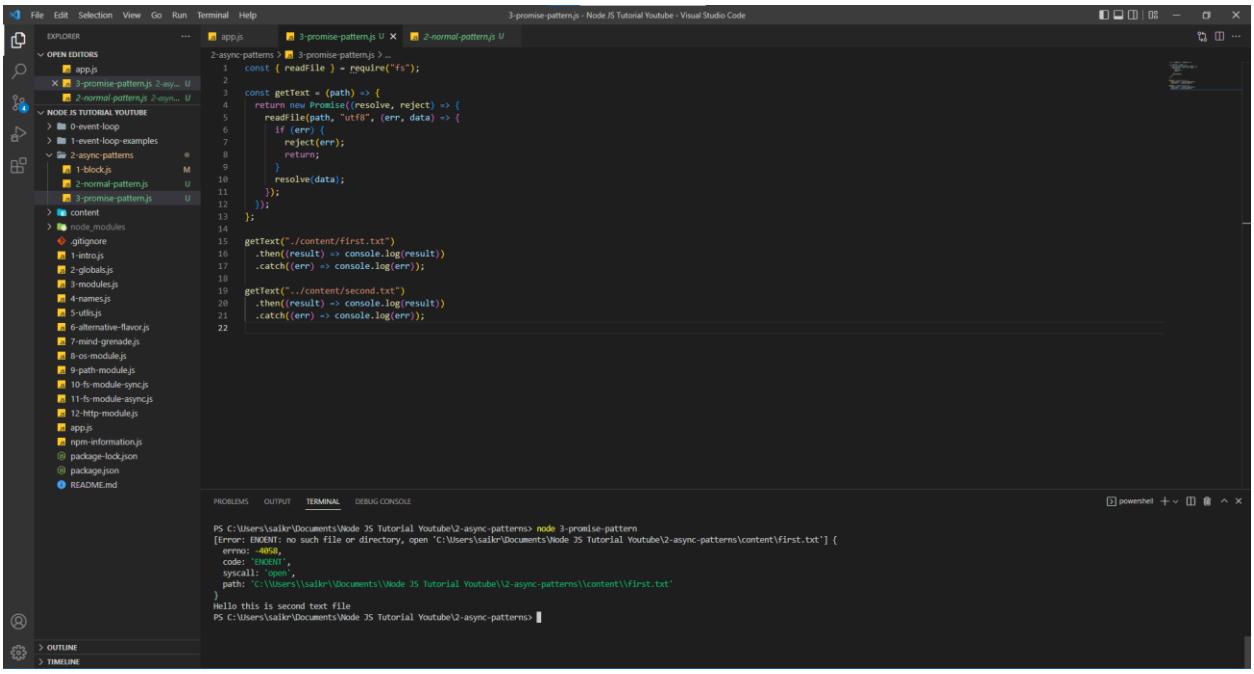


The above code works well but the real problem starts when we want to read multiple files and write data into one file.

The better solution would be turning that into a promise.

This is how an asynchronous function inside a promise looks like. Promise is divided in 3 stages which are

Pending, Reject and Resolve



If Promise is fulfilled, it is Resolved. If it isn't fulfilled or failed, then it is considered as Reject. If it is in either of those two stages, then it would be in Pending State.

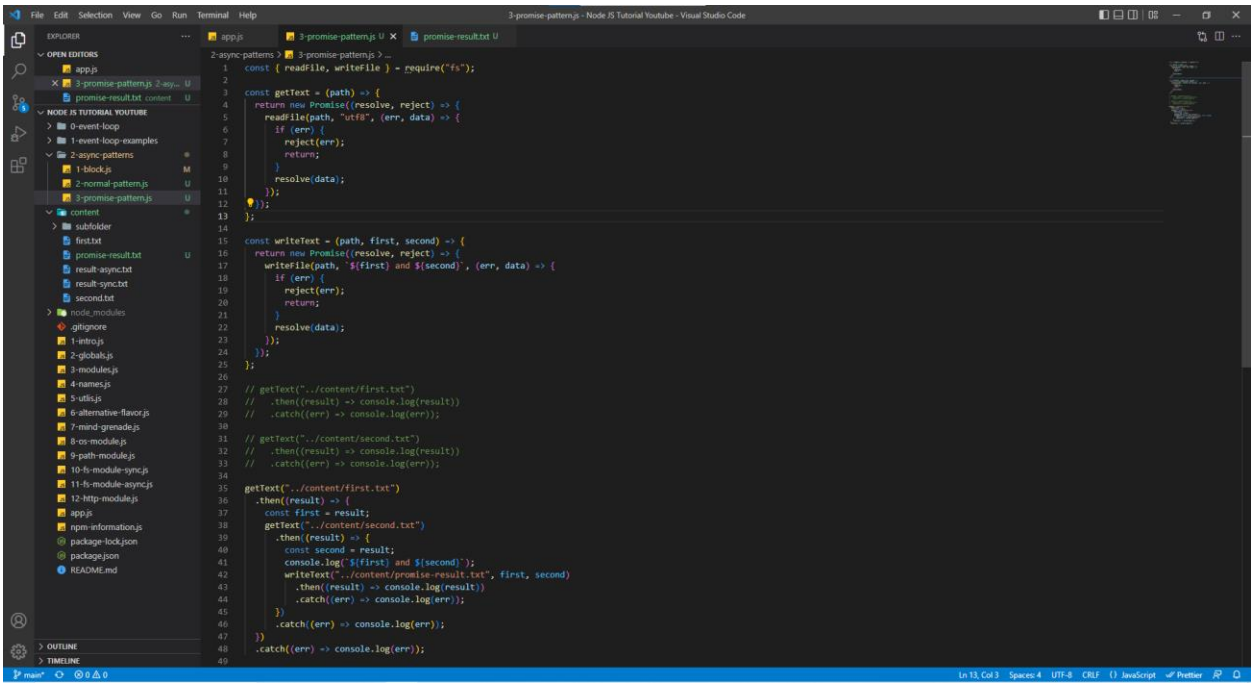
If a Promise is resolved it will enter then block, if it rejected it will enter catch block. The above screenshot depicts Resolved and Rejected states of Promise.

The above code is still not cleaner and that is why we follow different approach of setting up Async and Await.

If we want to do 2 file reads and 1 write still it would be messier because we need to chain promises one inside another, hence we use the approach Async and Await.

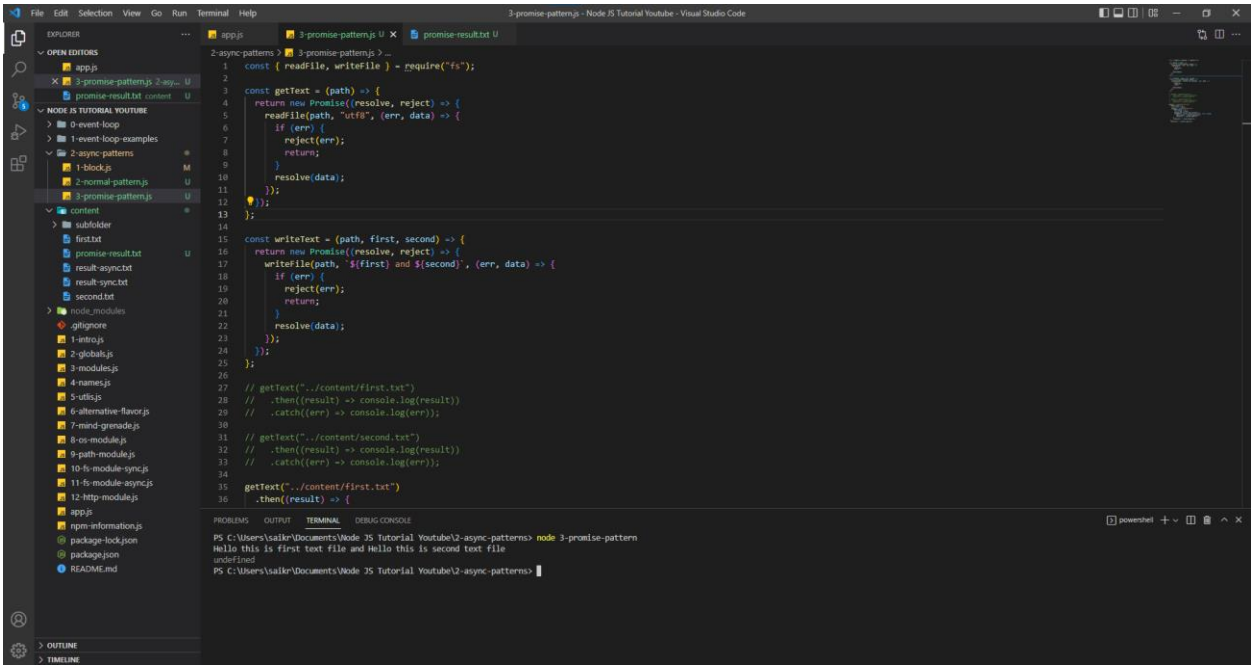
Chaining of Promises can be found below,

3-promise-pattern.js

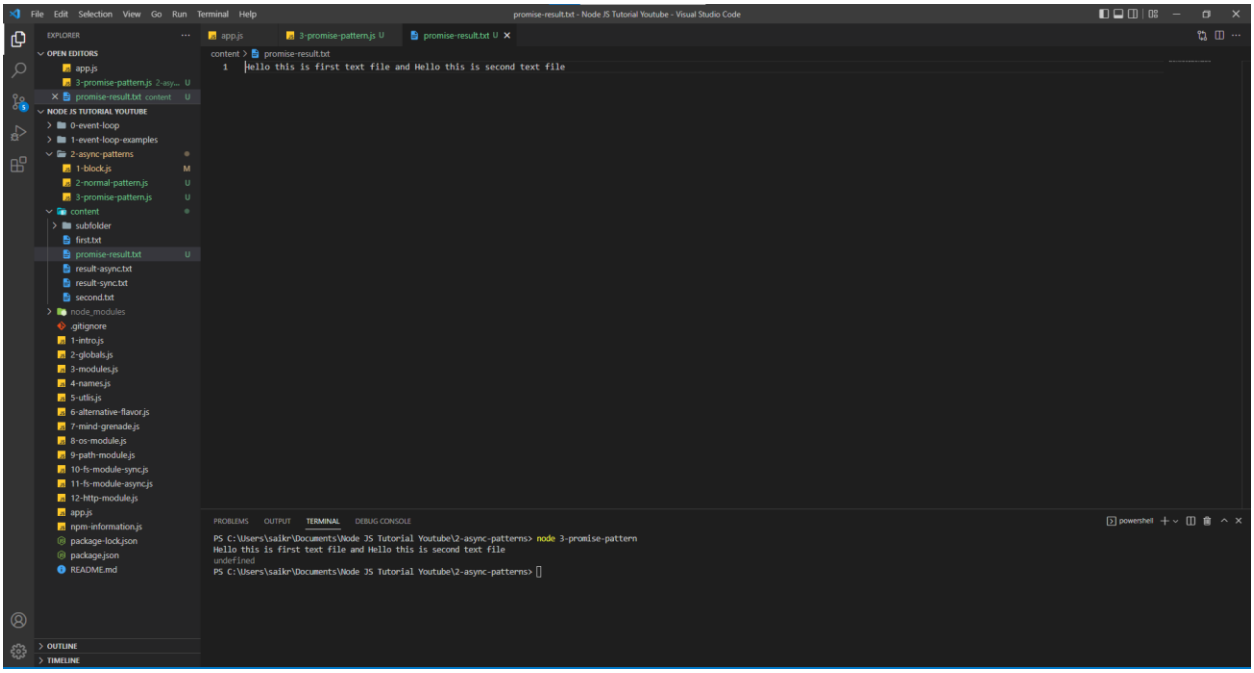


and the result can be seen in terminal and promise-result.txt file

terminal

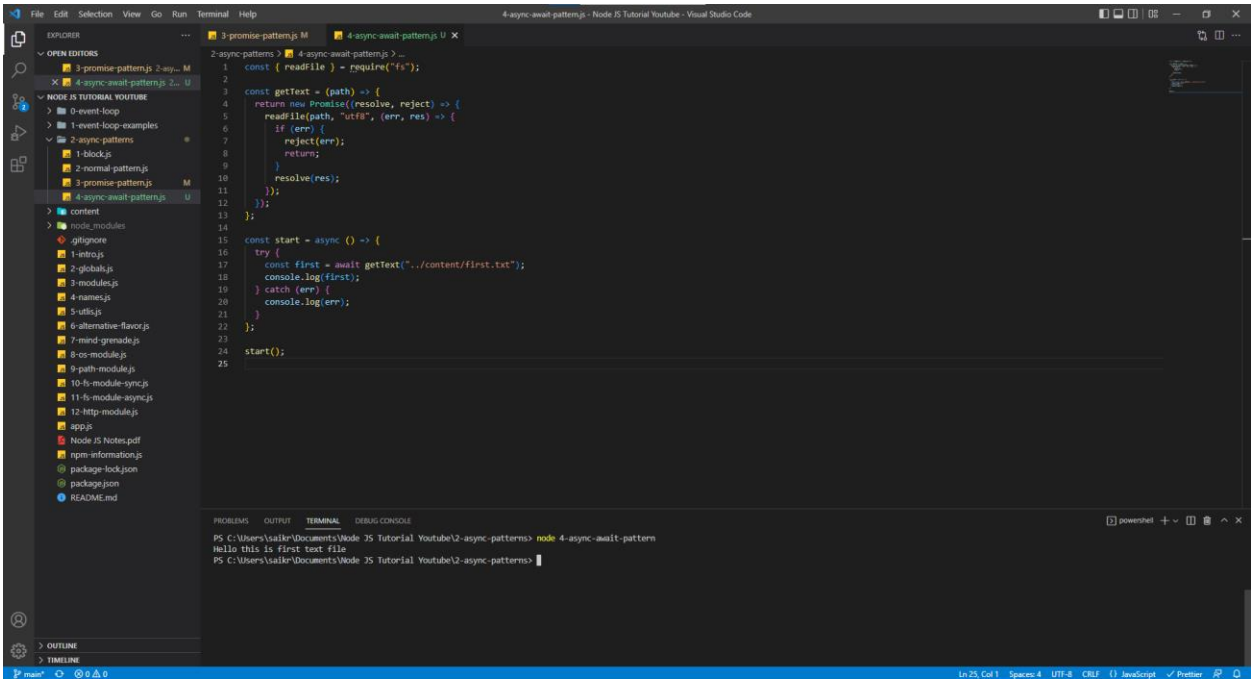


promise-result.txt



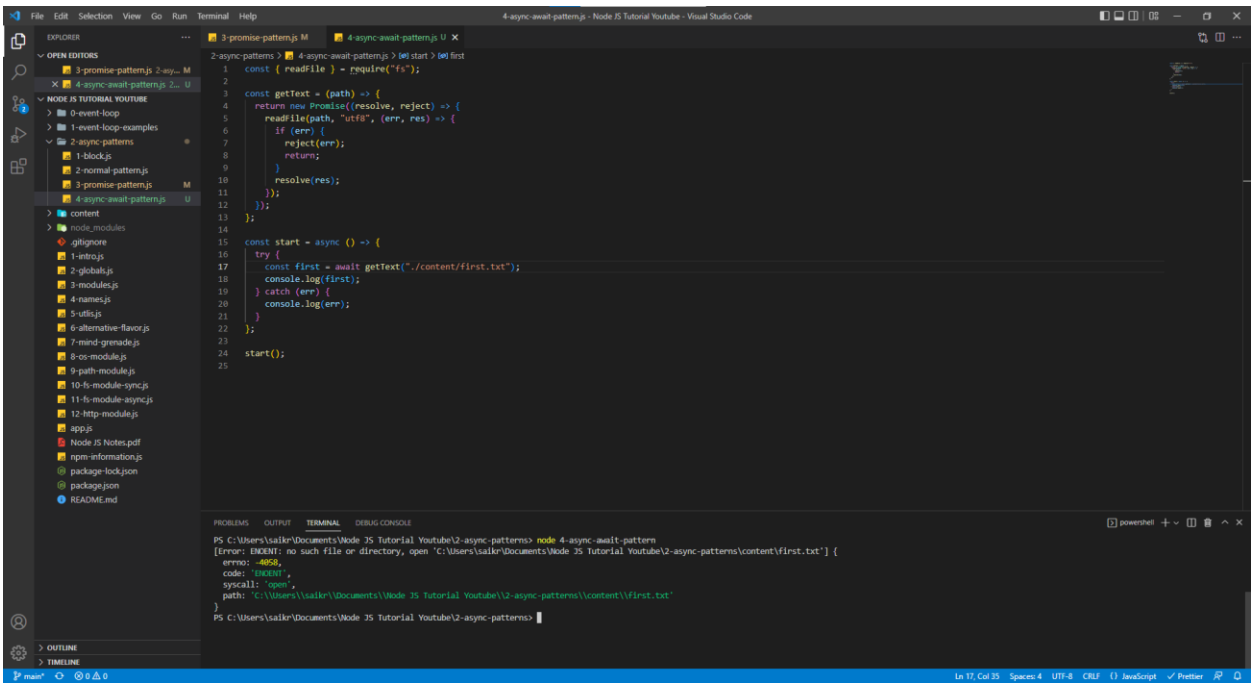
Async Patterns – Refactor to Async

Now we are refactoring the code using the async and await. With the help of async and await, we will wait till the promise is settled and then decide whatever we want to do.



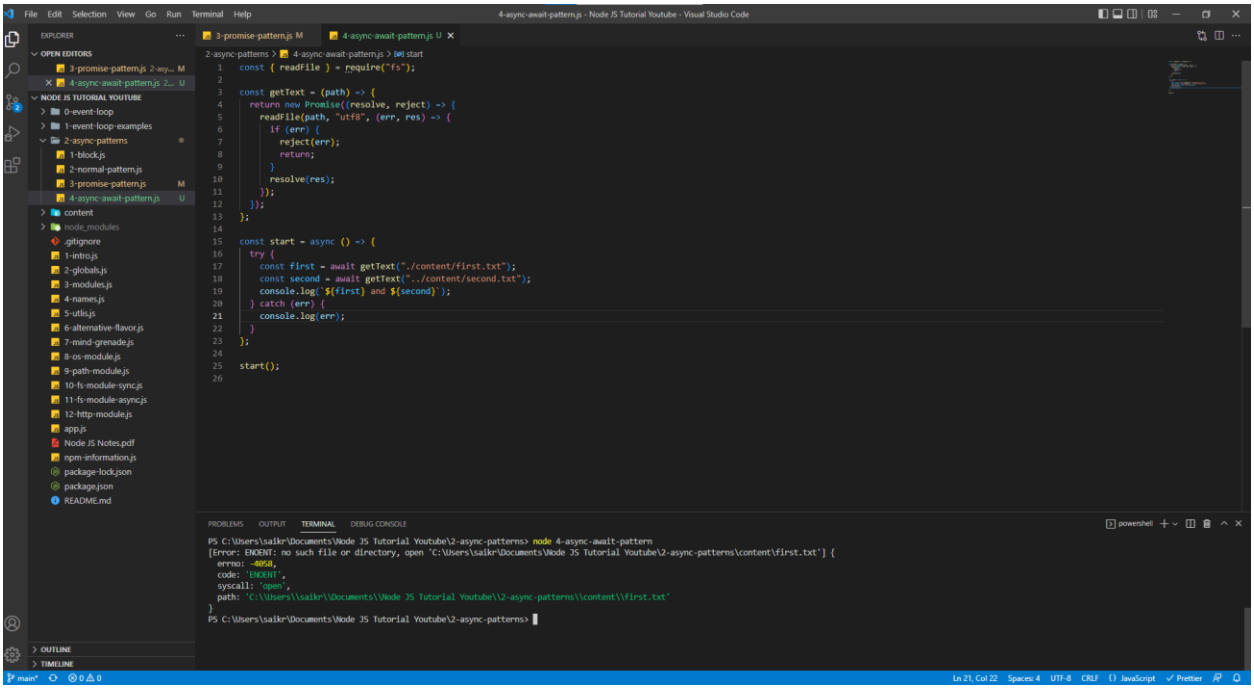
In the async await approach we wait for the promise to be settled and then follow he next steps. We always try to write code in try and catch blocks to capture any unhandled exceptions.

If the file we are trying to read is not found then we will handle the exception using the catch block.

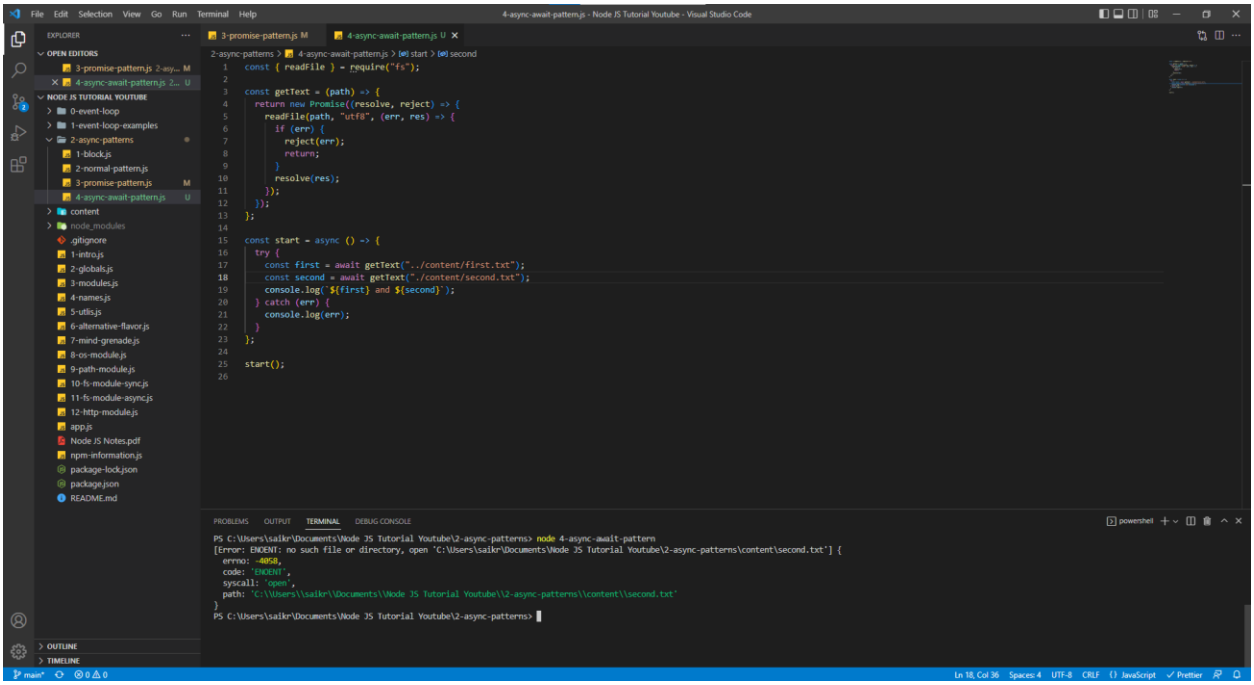


Using try and catch blocks, if any one promise is rejected then try block won't be execute and it will immediately enter the catch block to handle the error.

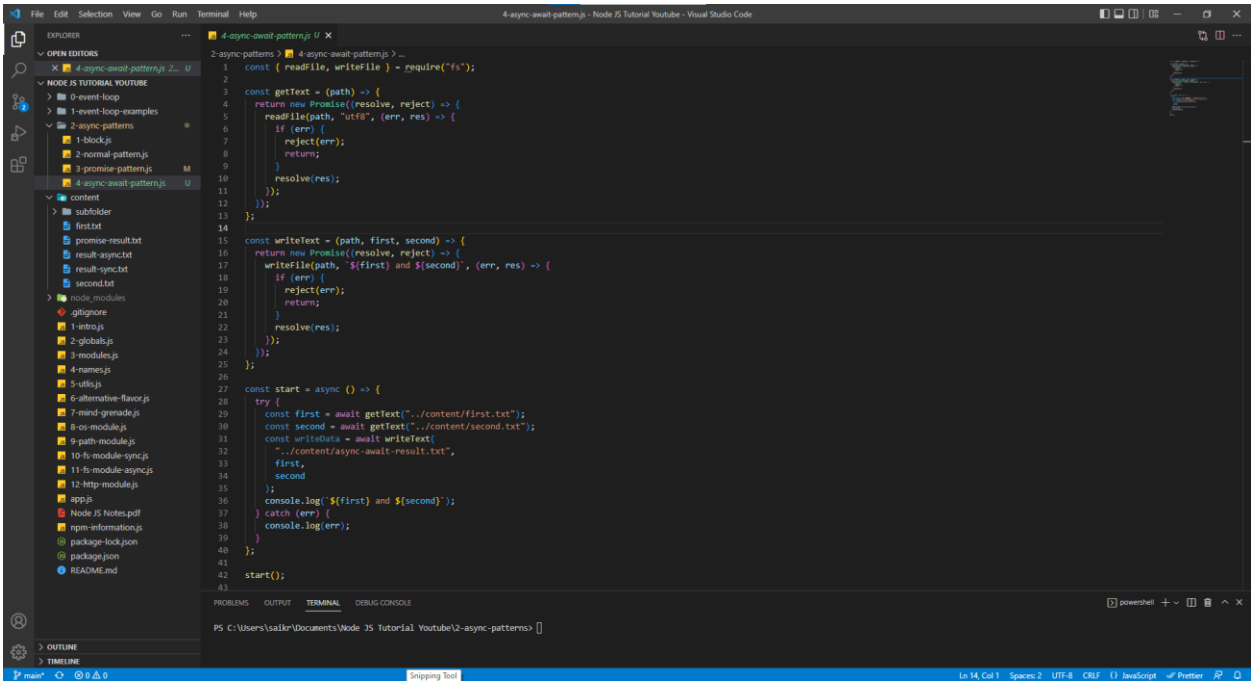
When there is an error in first file path.



When there is an error in second file path

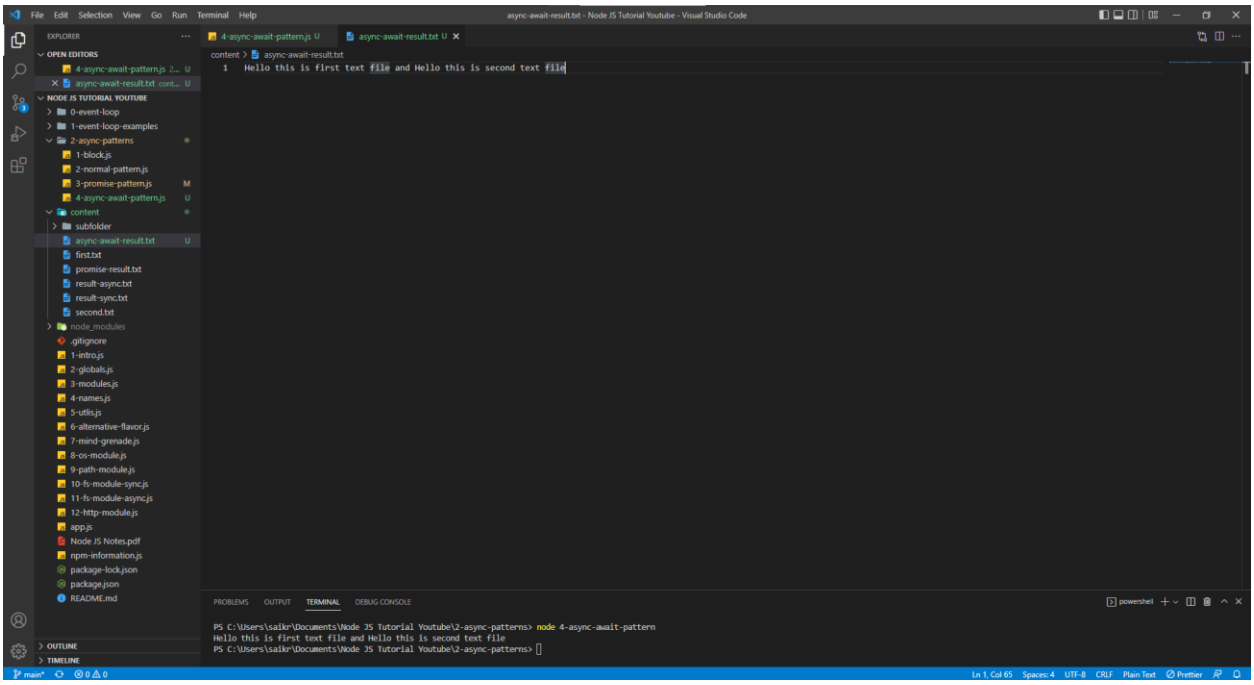


With the help of async and await pattern, this is how we implement two reads and one write



```
const { readFile, writeFile } = require("fs");

const getText = (path) => {
  return new Promise((resolve, reject) => {
    readFile(path, "utf8", (err, res) => {
      if (err) {
        reject(err);
        return;
      }
      resolve(res);
    });
  });
};

const writeText = (path, first, second) => {
  return new Promise((resolve, reject) => {
    writeFile(path, `${first} and ${second}`, (err, res) => {
      if (err) {
        reject(err);
        return;
      }
      resolve(res);
    });
  });
};

const start = async () => {
  try {
    const first = await getText("../content/first.txt");
    const second = await getText("../content/second.txt");
    const writeData = await writeText(
      "../content/async-await-result.txt",
      first,
      second
    );
    console.log(`${first} and ${second}`);
  } catch (err) {
    console.log(err);
  }
};

start();
```

async-await-result.txt



```
Hello this is first text file and Hello this is second text file
```

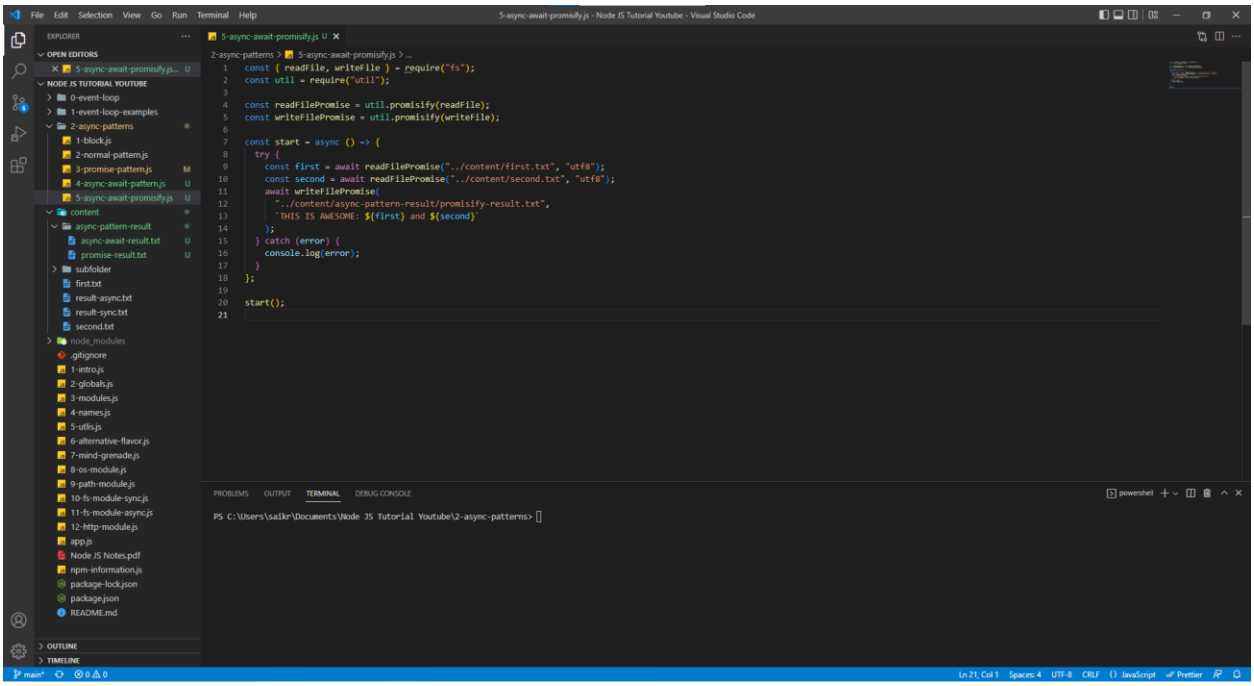
Till now we have used to wrap the function (getText and writeText) with Promises and worked with them using then() and catch() blocks (which are nothing but a promise pattern), async and await pattern. Now we will try to remove the wrapping function ad use the Node JS Native pattern using util module in Node.

Async Patterns – Node's Native Option

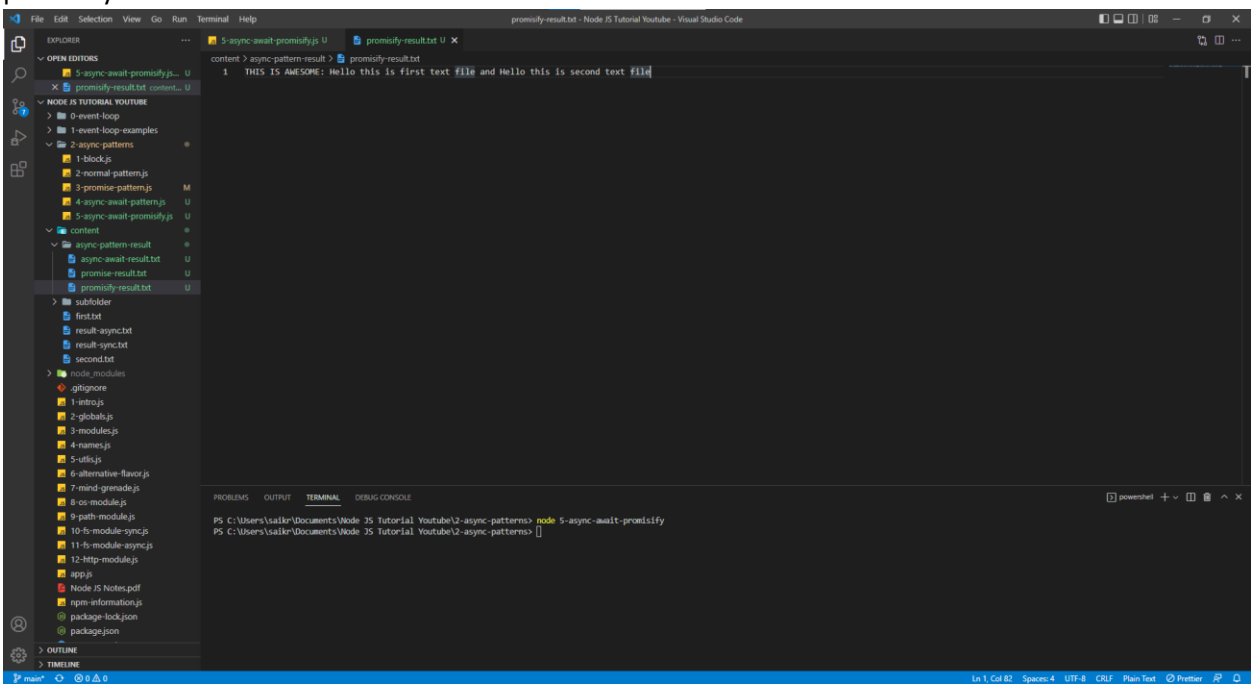
Using the node native option, we have Util module in Node which provides the promisify function which in turn provides the promise state (resolved, rejected, or pending). This approach rapidly reduces the line of code to be written and provides a cleaner code.

Using Promisify

5-async-await-promisify.js



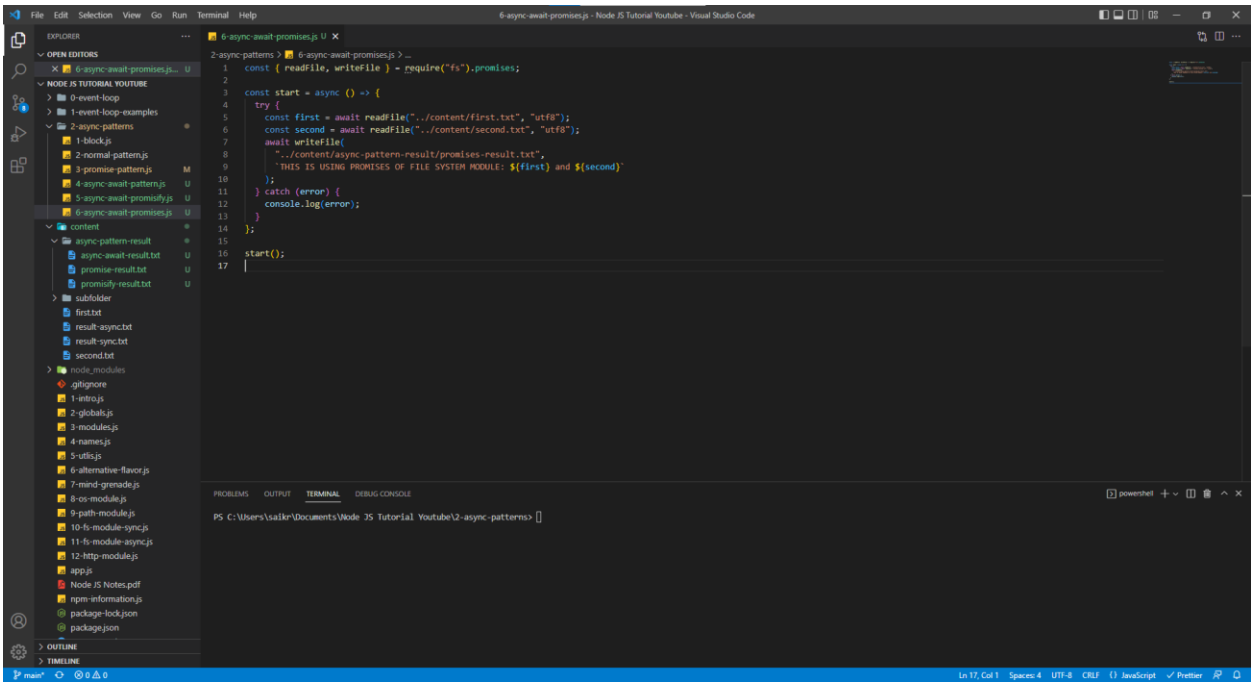
promisify-result.txt



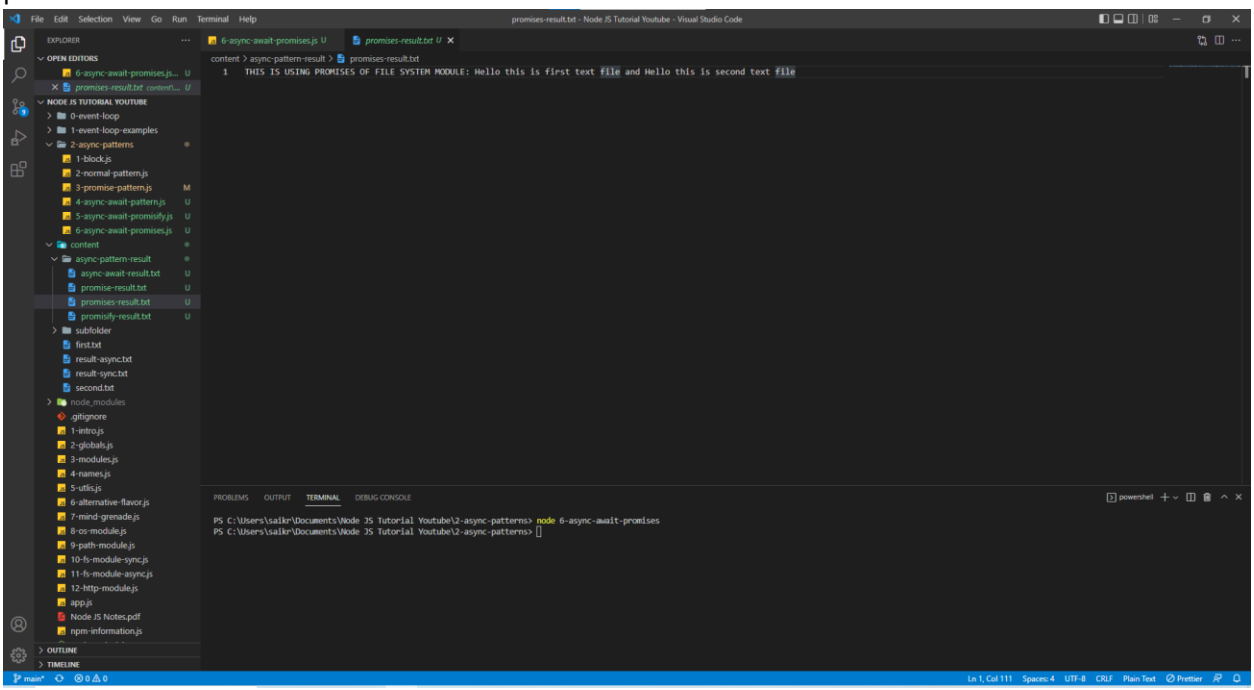
Using Promises of File System Module in Node

We can remove the promisify function of Util Module and just use the promises available in File System Module. The code looks much cleaner.

6-async-await-promises.js



promises-result.txt



Events

- Event Driven Programming
- Used Heavily in Node JS

In browser applications, events are emitted though actions performed by a user on web page like click on a button or hovering etc., Essentially our program executes at least in part it is controlled by events. In browser apps those events are mostly external. Now that style of programming is called Event Driven Programming.

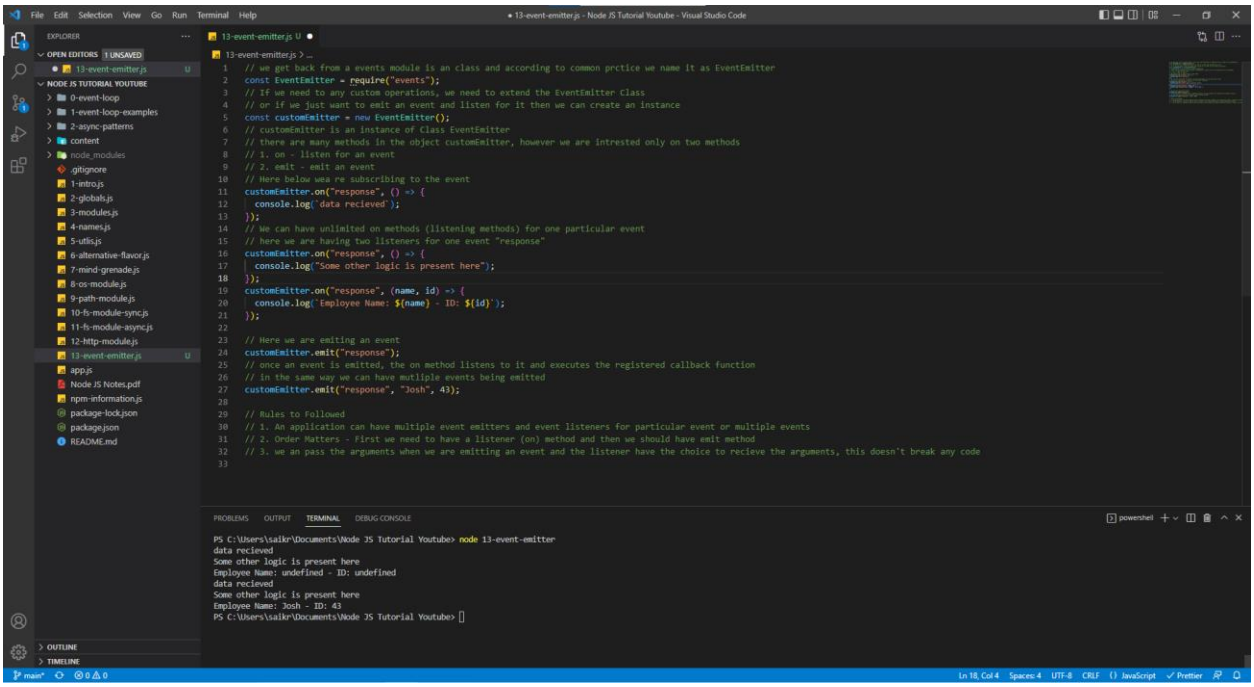
Server-side applications also use events and Event Driven Programming is heavily used in Node JS (Server Side).

Basically, we listen to specific events and register functions that will execute in response to those events. So, once an event takes place, a callback function is fired.

Many built-in modules in Node JS do use events under the hood and therefore making events an event driven programming a big part of Node JS.

Even though we don't write our own events, events are core building block of Node JS. Lot of built-in modules rely on Events.

13-event-emitter.js



In Node JS, we import the module named "events" to get access to a Class named EventEmitter. If we need to perform any operations on EventEmitter then we need to extend the EventEmitter class or if we just want to listen and emit an event, we can create an instance or object of Class EventEmitter.

There are many methods in EventEmitter class but as of we need only 2 methods namely

1. on method – listens for events
2. emit method – emits an event

Once an event is emitted, the listen method listens the event and executes the registered callback function.

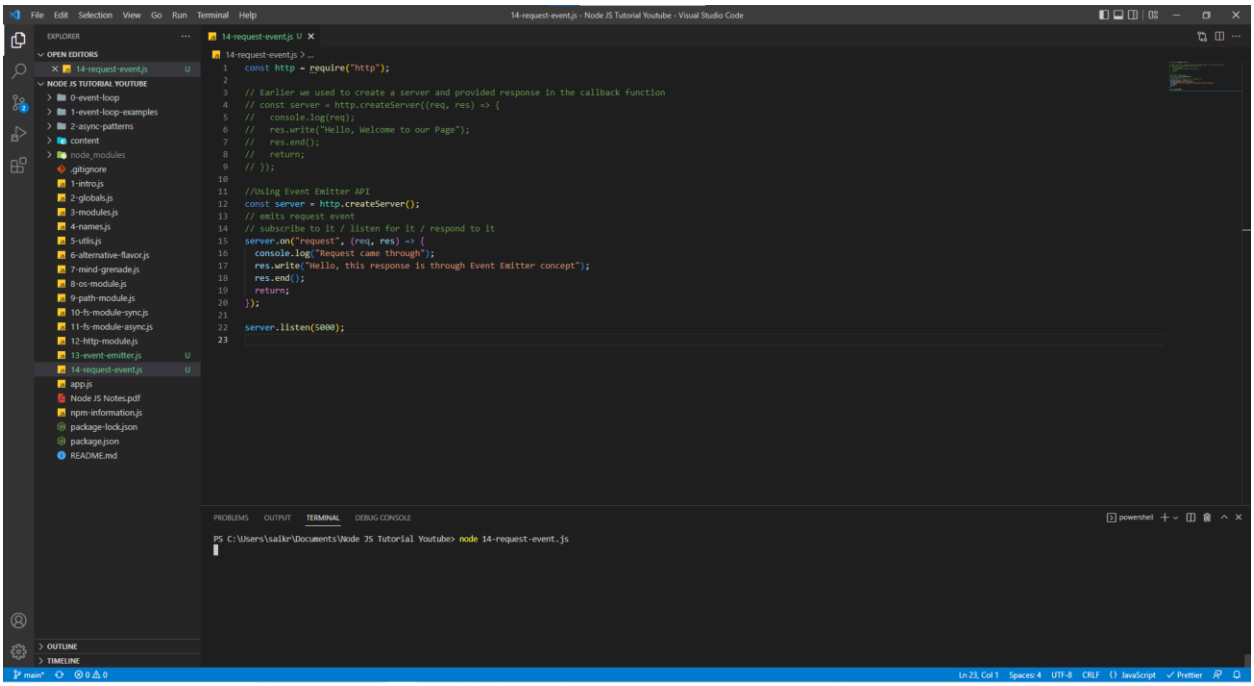
We can have multiple event listeners for one event and in the same way we can emit many events for a particular event.

Rules to be Followed in Events

1. An application can have multiple event emitters and event listeners for event or multiple events.
2. Order Matters - First we need to have a listener (on) method and then we should have emit method.
3. we can pass the arguments when we are emitting an event and the listener have the choice to receive the arguments, this doesn't break any code.

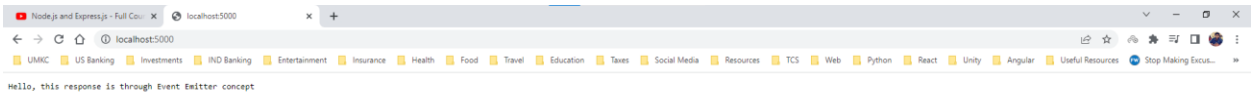
Events Emitter – Http Module Example

14-request-event.js



Earlier we used to create a server using the HTTP module and then have written a callback function inside the server but in the Event Emitter approach, Node listens to the event/ subscribe to the event/ responds to that event emitted by the server (***request event***) and then executes the registered callback.

At Port 5000, when server sends a request (emits an event), on method in the code listens to emitted event and executes the registered callback function.



Streams in Node JS

Introduction

Streams are used to read and write data sequentially. When we must handle and manipulate streaming data for ex: a continuous source or a big file, streams come handy. Thera re 4 types of streams.

1. Writeable – used to write data sequentially.
2. Readable – used to read data sequentially.
3. Duplex – used to both read and write data sequentially.
4. Transform – used to modify data when reading or writing.

Just like Events, many built-in modules implement streaming interface. Streams extend EventEmitter Class which simply means that we can use events in streams.

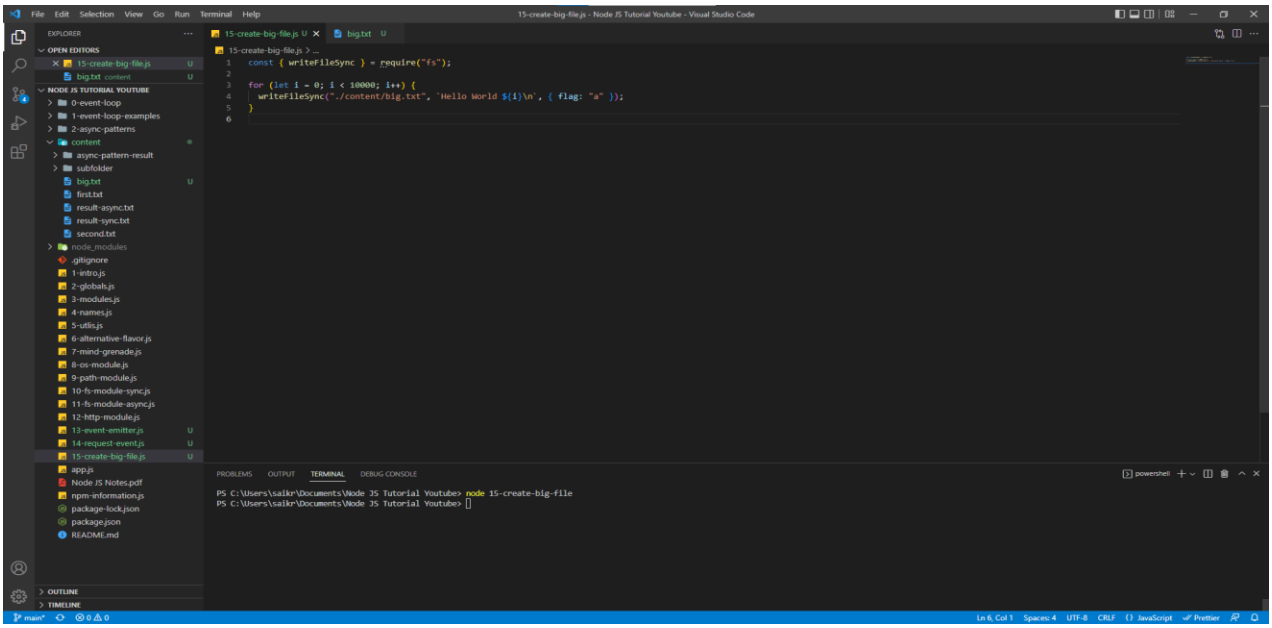
Streams – Read File

When we use readFile or readFileSync we are reading the entire file and storing it in a variable but if we read a big file then we would be eating up a lot of memory for the variable and if the file size increases the variable won't be good enough for storing the file and we would be getting an error for the size.

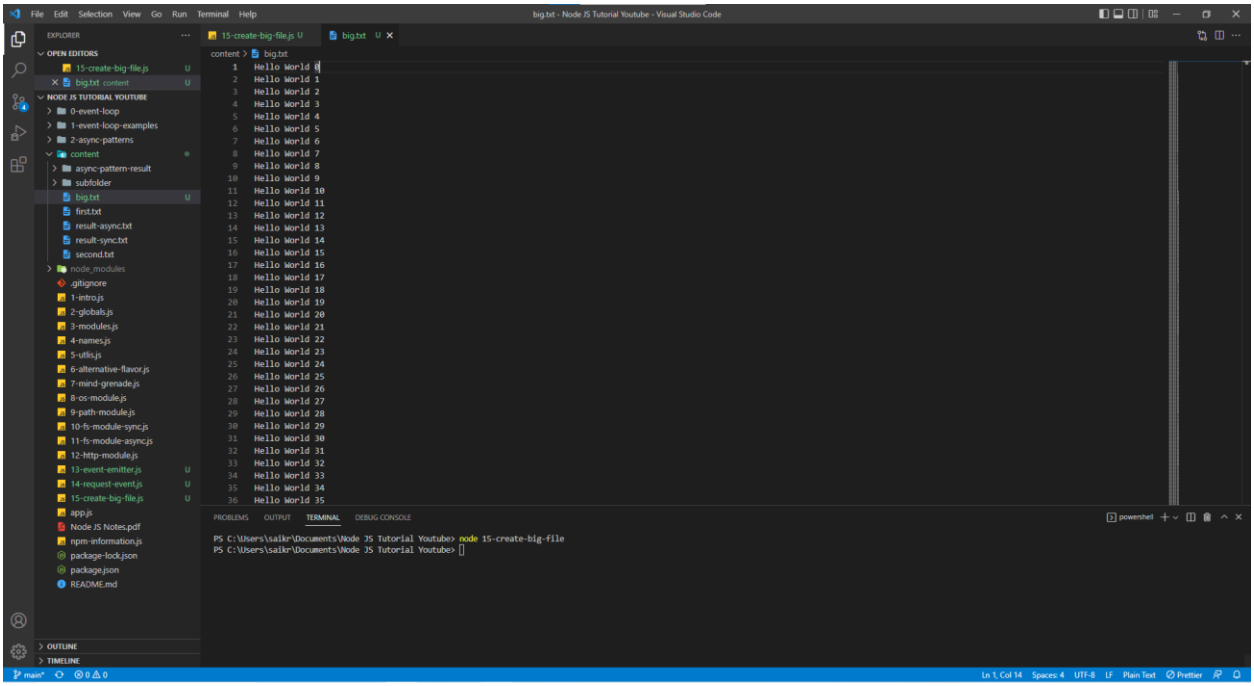
The solution for the above problem would be readStream option.

Now, let us set up a big text file which contains a lot of data.

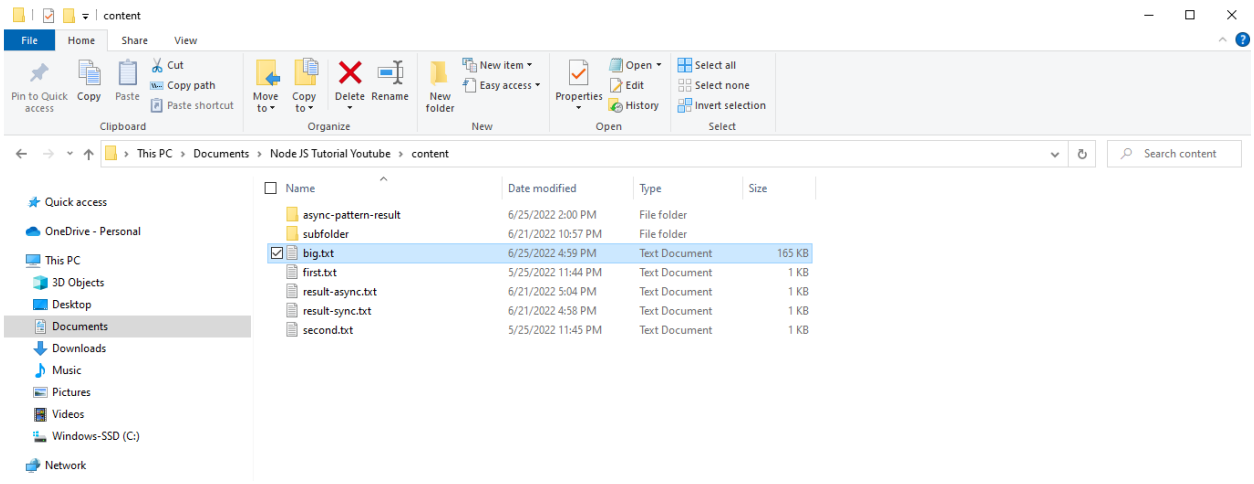
15-create-big-file.js



big.txt

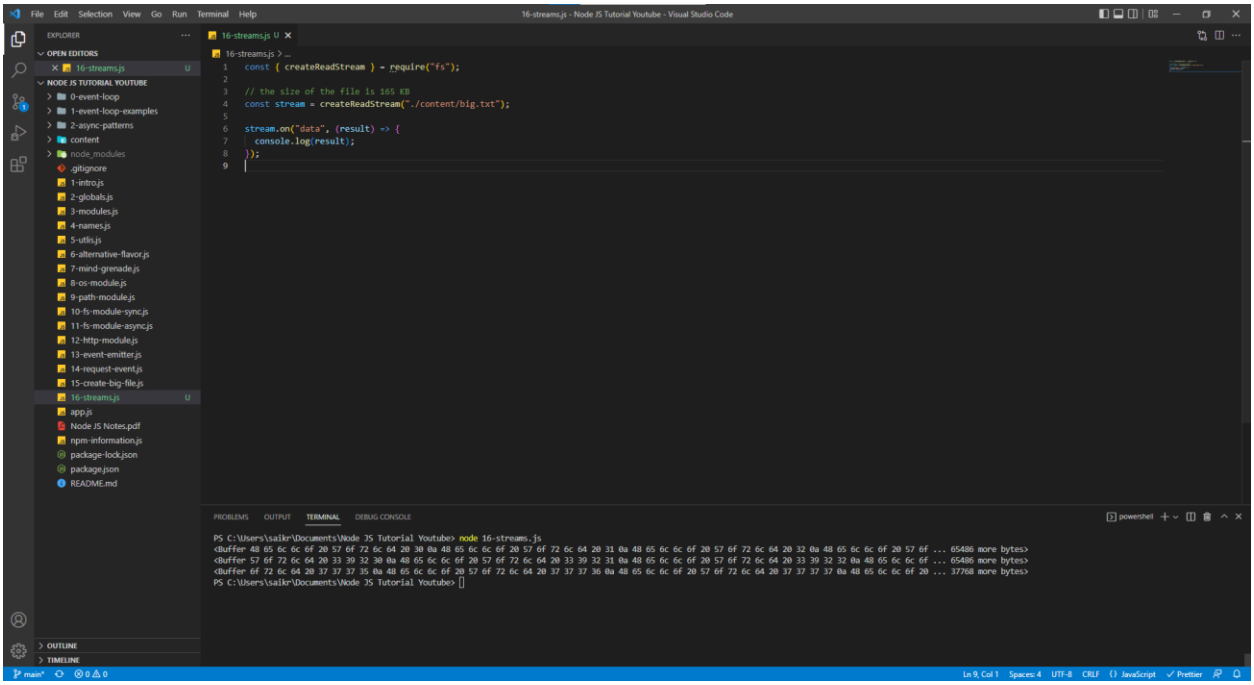


The size of big.txt file is 165 KB

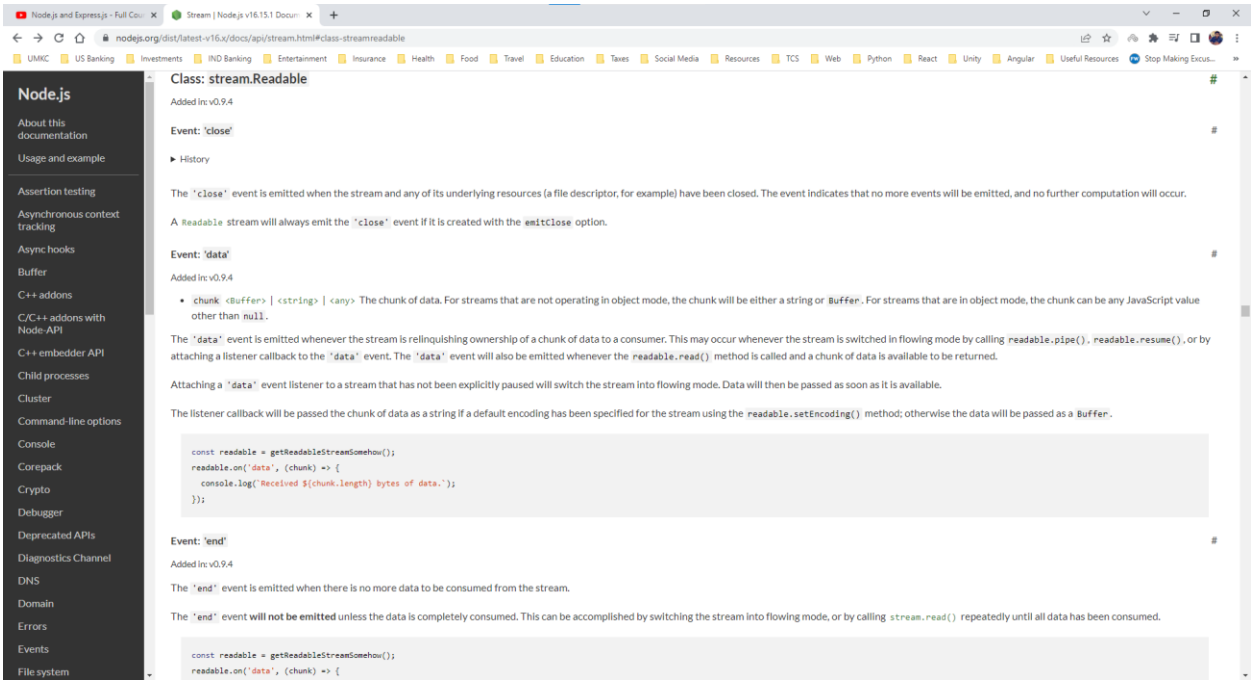


When we use the createReadStream, we are reading the data in chunks and by default each chunk of data is 64KB. We can increase size of the chunk. Instead of using the variable we are using the stream to get data in chunks. The event emitted by stream is “data”.

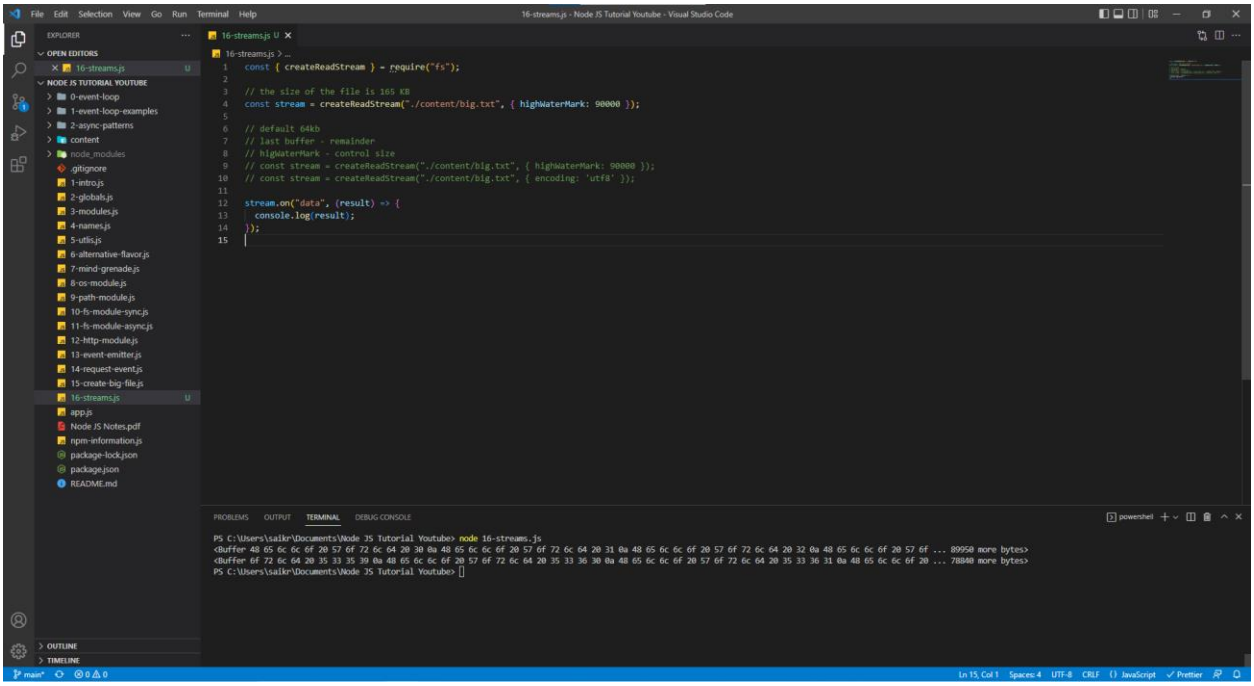
createReadStream invokes the readStream and returns readStream. readStream extends form stream.Readable . stream.Readable has the event named “data”. With options of encoding, we cosole.log the data in buffer format.



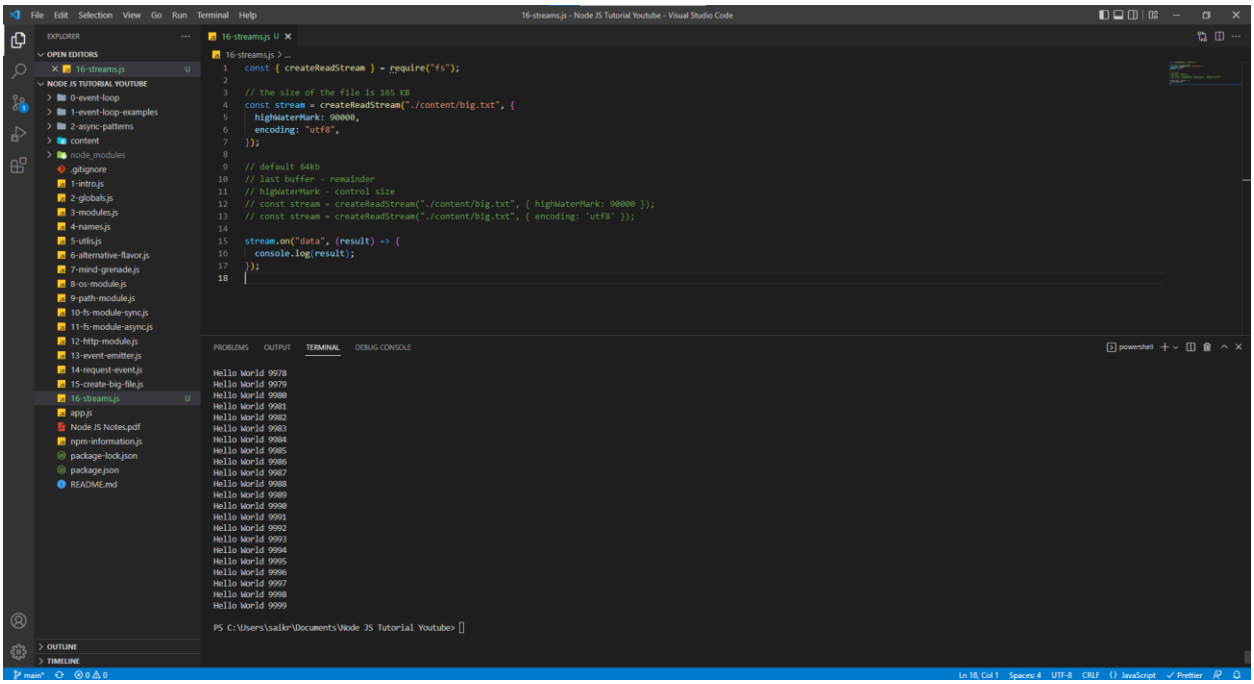
About “data” event in Node JS Docs



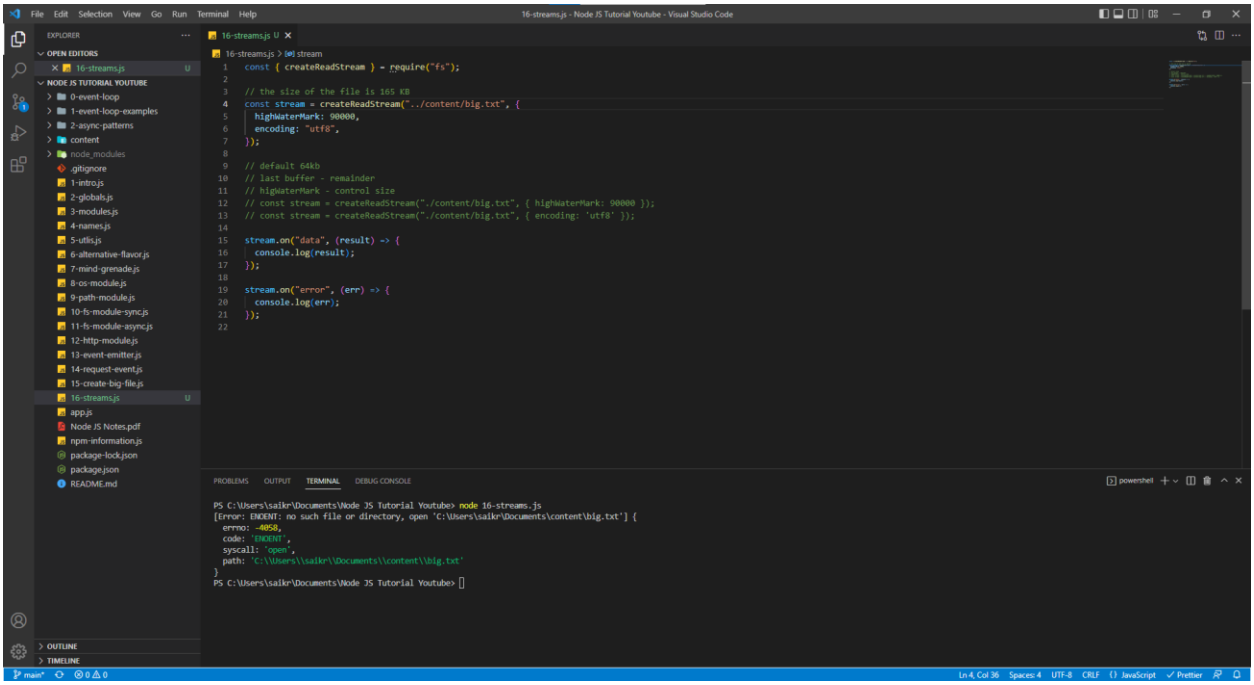
When we control the size of the chunk.



When we encode the data in utf8 format.

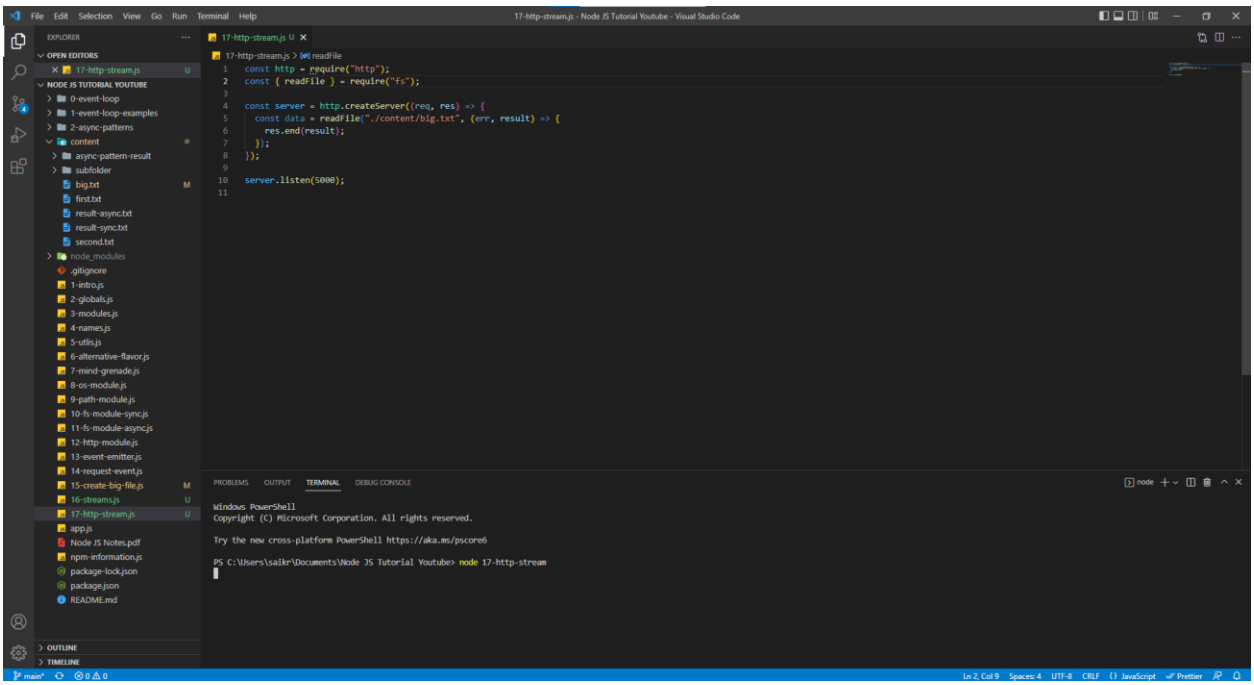


event emitter emits error event when there is an error while reading the data stream.

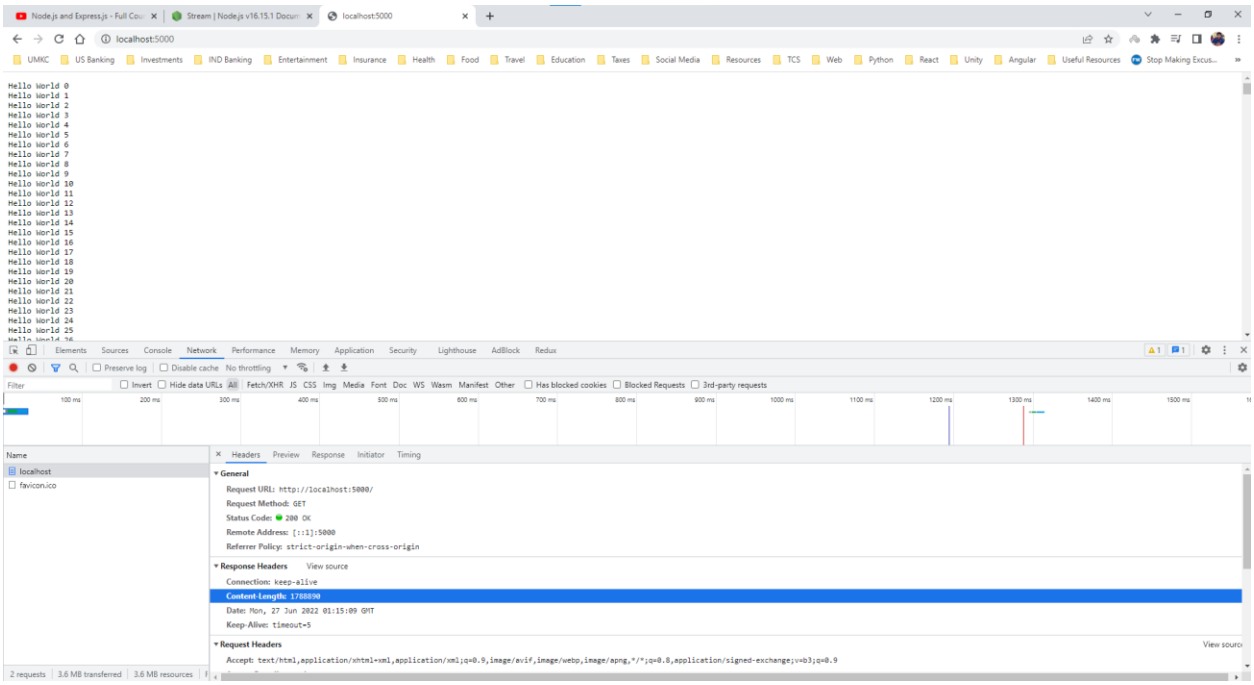


Streams – Http Example

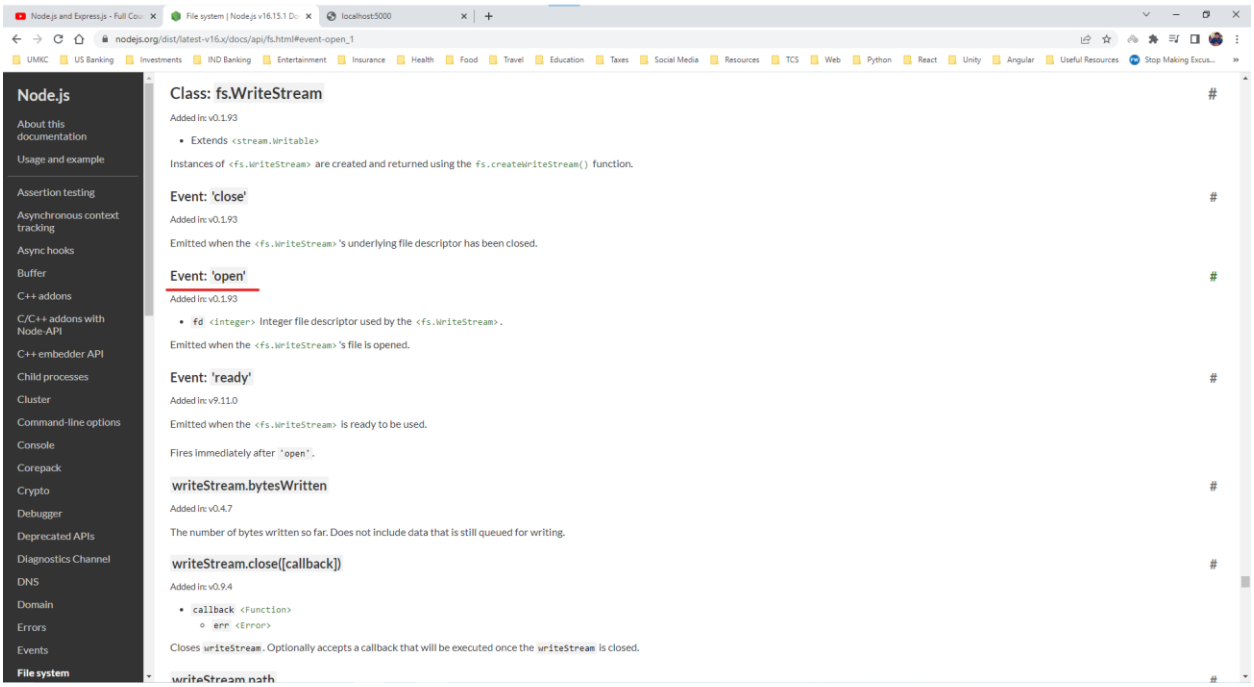
Let's see a practical example when streams come in handy. Assume a situation where are trying send the data from big.txt file we created. We have increased the size of bg.txt file to 1.8 MB and it is not ideal to transfer data of 1.8MB over the internet.



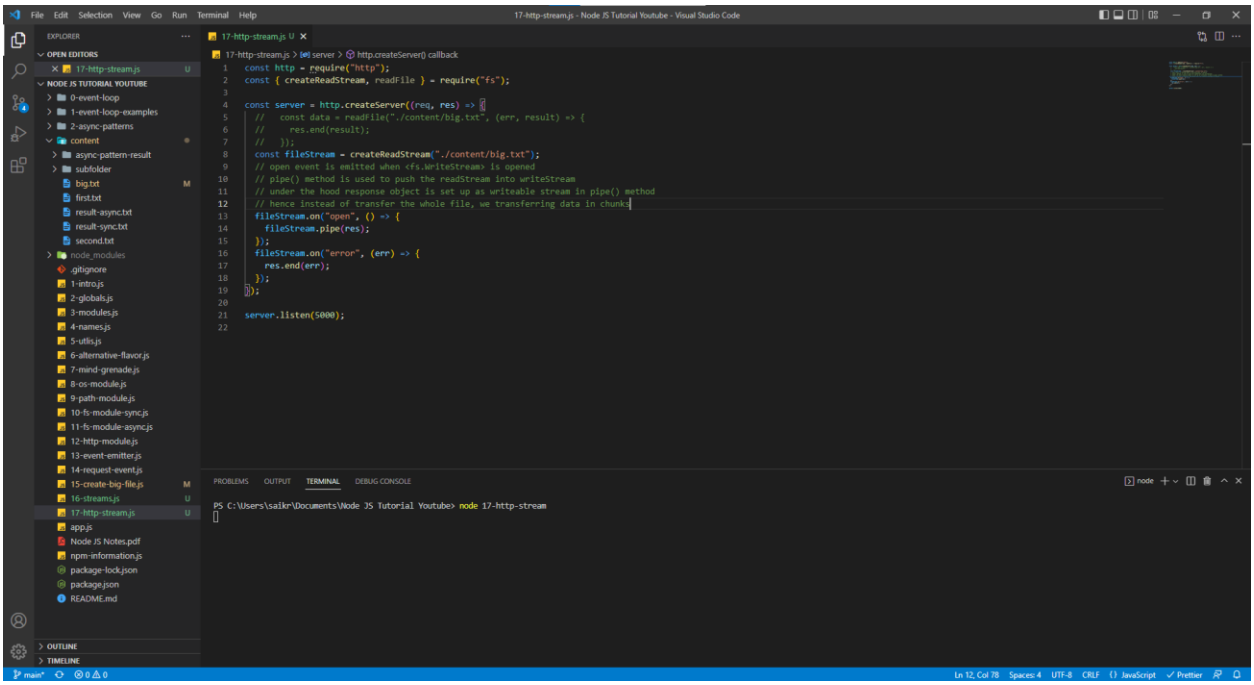
In the below screenshot we can see the size of file being approximately equal to 1.8 MB and hence we want to refactor the code and send the data in chunks.



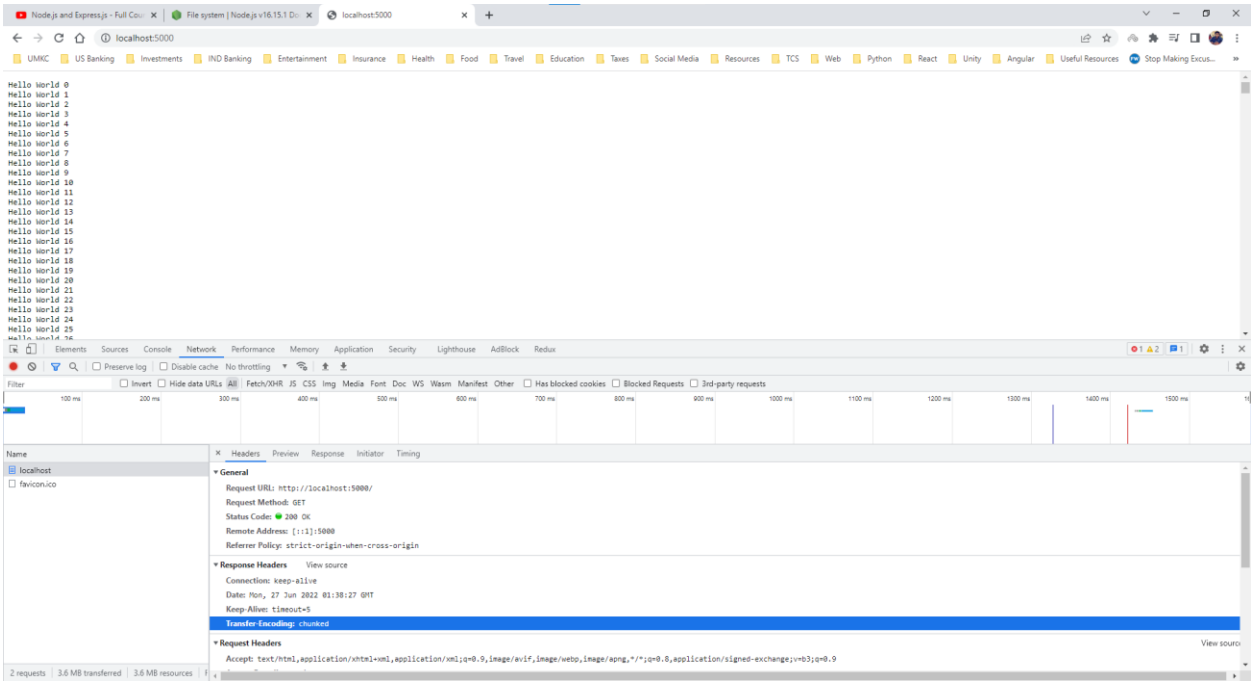
To send the data in chunks, we use createReadStream. The pipe() method is used to push the readStream into writeStream. So, we can imagine we read data in chunks then we can also write data in chunks. What happens under the hood response object is set up as writeable stream in pipe() method. ***Some information about "open event in writeStream.***



Hence in this case we are reading the data in chunks and writing the data in chunks using the pipe() method without transferring the data of large life at once.



We are receiving the data in chunks.



End of Node Tutorial Module

Until Here we are done with Node Fundamentals