

Node .JS

Node.js is a JavaScript runtime to build fast, scalable network applications. It helps us to use JavaScript in server-side coding. JavaScript is responsible for most of the client-side processing in web applications today. Express.js is the most popular framework for creating web applications in Node.js. It is lightweight and provides easy connectivity with different databases. Using Express, we can handle requests and manage routes.

Why Node .js?

Scenario 1: JavaScript has been the most used language for creating dynamic web pages. Its rising popularity has brought in a lot of changes. JavaScript has moved to the server-side too, establishing itself on the web servers as well. That entailed a capability to generate web pages on the server.

Node.js also represents the "JavaScript everywhere" paradigm, by unifying web application development around a single language, rather than using different languages for server-side and client-side.

Scenario 2: Consider a scenario where you are trying to upload videos to a social media website. You may observe that lot of time is taken for getting your request processed. This may be because the preceding requests need to be completed first by the web server of the application before executing this new request. In order to ensure the requests get served concurrently, multiple approaches have been proposed. This is where Node.js comes in. Since Node.js uses an event looping mechanism, a Node.js web server is capable of sending responses in a non-blocking way. This eventually means, a Node.js server can handle a large number of requests than the commonly used traditional web servers.

Node.js and its Benefits:

Let us have a look at the nature of Node.js and some of the benefits associated with Node.js:

1. **Node is popular:** Node.js is one of the most popular platforms for application development as it uses JavaScript. JavaScript is a powerful programming language as it provides good flexibility to developers.

2. **Full-stack JavaScript development:** Node.js has paved the way for JavaScript to be on the server-side as well. Hence applications can now have both back-end and front-end developed with the same JavaScript language. We now have two popular JavaScript software stacks for building full-stack web applications: **MEAN and MERN**.

Node is the **N** in the **MEAN** stack. In this stack, **Node** is used along with the **MongoDB** database and **Express**, which is a Node framework for back-end application development. **Angular** framework is used for front-end application development.

Node is the **N** in the **MERN** stack as well. Here, Node is used along with **MongoDB** database and **Express** for back-end application development. **React** is used for front-end application development in this stack.

3. **Very fast:** Node applications have better performance than applications developed using other technologies due to the JavaScript engine used internally in Node. The engine converts JavaScript code into machine code and provides extremely fast execution.
4. **Node is powerful:** Node uses a non-blocking I/O model and asynchronous programming paradigm, which helps in processing requests in a non-blocking way.
5. **Data Streaming:** Node has a built-in Streams API that helps in creating applications where data streaming is required.
6. **Rich Ecosystem:** Node.js provides a package manager called NPM (Node Package Manager) which is the world's largest software registry that helps open source developers to share and use packages using NPM.
7. **Modularity:** Node applications can be developed as modules which helps in dividing the application into a reusable set of code.
8. **Wide client-side and database connectivity:** Node.js has absolutely no dependencies and also goes perfectly with any possible client-side technologies like Angular, React, etc., and any database like MySQL or MongoDB.

The nature of Node.js and the benefits associated with it has helped it grab quite a lot of attention from various organizations across different domains.

Organizations using Node.js:

Node.js is used in applications developed for a wide range of domains. The below table lists out a few domains and the companies that are migrated to Node.js.

Domains	Manufacturing	Financial	E-Commerce	Media	Technology
Companies	BMW	Citigroup	Walmart	DirectTV	Salesforce.com
	Siemens	PayPal	eBay	HBO	Yahoo

All these organizations were using different technologies like Java, Rails, etc. for developing the server-side of their applications. But later, they migrated to Node.js because of the features provided by it.

The below table lists the reasons why they migrated to Node.js.

Company	Moved From	Reason
Paypal	Java	<ul style="list-style-type: none"> Created application with 33% fewer lines of code and 40% fewer files. Served double the requests per second, thereby reducing the average response time by 35%.
Walmart	Java	<ul style="list-style-type: none"> Was able to serve sophisticated features to mobile users by customizing content based on device and browser capability. Was able to use various web services to conduct business and scale up their overall growth.
eBay	Java	<ul style="list-style-type: none"> Handles a lot of I/O bound operations. Used the Node.js module to do load balancing and found a tremendous improvement in the application performance.
LinkedIn	Rails	<ul style="list-style-type: none"> Found 20x faster performance and lower memory overhead. Servers were cut down to 3 from 30, increasing 10x resource utilization.
DirectTV	Rails	<ul style="list-style-type: none"> Met the scalability needs during peak times.

Now that we have seen the importance of Node.js, let us dive deep into **Node.js**.

Introduction to Node.js:**What is Node.js?**

Node.js is an open-source JavaScript run-time environment used for building scalable network applications. It helps in developing the server-side of the application using JavaScript language. It is used for building data-intensive real-time applications.

What can we build using Node.js?

1. Complex SPA(Single Page Applications)
2. Real-time applications like Chat rooms
3. Data streaming applications
4. REST APIs
5. Server-side web applications

Features of Node.js: which makes it the most popular platform for server-side application development.

1. V8 engine

As application development in Node uses JavaScript language, the Node.js platform needs an engine for executing JavaScript. The engine used in the platform is **V8** which is an open-source high-performance engine. It is developed by Google and written in C++. V8 compiles JavaScript source code into native machine code. The same engine is used in the Google Chrome browser as well to execute JavaScript code. The performance of the Node.js application is faster due to this ultra-fast engine used in the platform.

2. Single codebase

Since coding in Node is based on JavaScript, both the client and the server-side code can be written using the same JavaScript language. It allows the front-end and back-end teams to be combined into a single unit. Also since Node.js uses JavaScript, we can quickly manipulate the JSON data retrieved from external web API sources like MongoDB, hence reducing the processing time needed per request.

3. Asynchronous and event-driven

All the APIs in Node are asynchronous i.e. non-blocking, which means Node-based server will never wait for an API to return data or to complete the request, it will move to the next request process. The notification mechanism of Node.js helps in getting the response from the previous requests after its completion.

Let us now understand the Asynchronous programming in Node.js.

Executing JavaScript code can happen in a Synchronous or Asynchronous way.

Synchronous programming

In Synchronous programming, the code execution happens synchronously. This allows only one task to execute at a time.

Consider the scenario where we need to read the content of a file and then database operation is to be executed. When the file read operation is started, the rest of the code in the program gets blocked until the file reading operation is finished. Once the file reading is done, then it continues to execute the remaining code. Though the database operation code is not dependent on the file read operation, it is getting blocked. This kind of code is considered as blocking code or synchronous code.

Asynchronous programming

Asynchronous programming is a design pattern that ensures code execution in a non-blocking way. The asynchronous code will get executed without affecting other code execution. This allows multiple tasks to happen at the same time.

Consider the same scenario of reading a file and then database operation is to be executed. On asynchronously implementing this, when the file read operation is started, it will not wait for the read operation to complete, it will just continue execution of the rest of the code. Once the file reading is done, it will be informed and the corresponding function gets called. This provides a non-blocking way of executing the code.

This improves system efficiency and throughput.

In Node.js, asynchronous programming is implemented using the callback functions.

Callback function: One approach to asynchronous programming is through callback functions. A callback is a function passed as an argument to another function and it will be executed after the task gets completed. It helps in non-blocking code execution.

```
1. setTimeout(() => {  
2.   console.log("after 20 seconds");  
3. }, 20000);
```

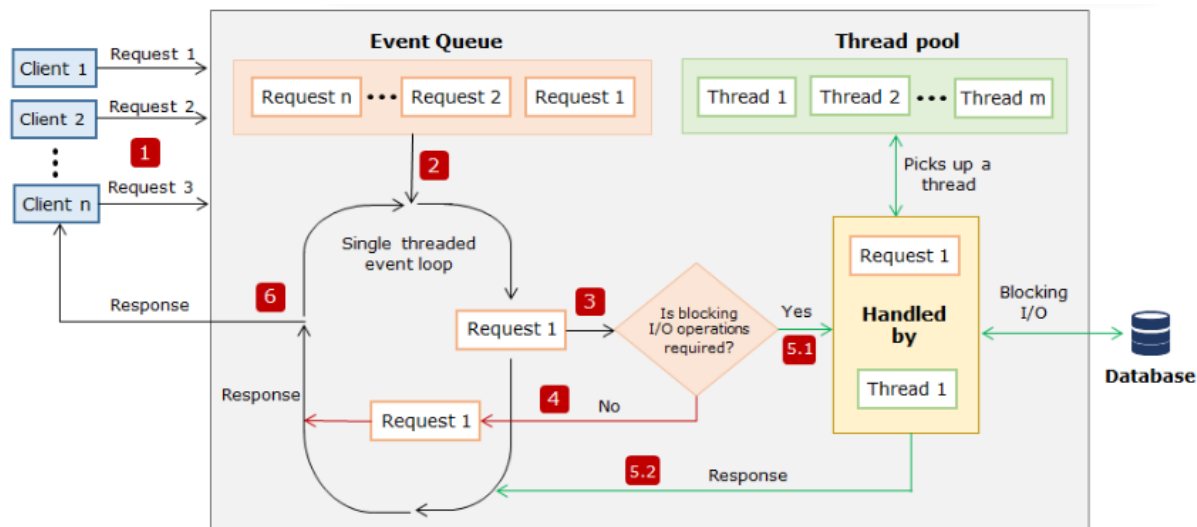
In the above example, `setTimeout()` takes two arguments. The first argument is the callback function and the second argument is the delay in milliseconds. The callback function is called after 20 seconds.

In Node.js, at the completion of each task, the respective callbacks written gets invoked. All the APIs of Node.js are written in a way to support callbacks. This makes Node.js highly scalable as it can handle a large number of requests in a non-blocking way.

4. Single-threaded event loop model

Node.js is said to be highly scalable because it handles the client request using the Single-threaded model with an event loop. Node environment follows the **Single Threaded Event Loop Model** which is based on JavaScript's callback mechanism.

Let us take a look at the Single-threaded model with the event loop of Node.js.



Single-threaded event loop model processing steps:

Step 1: Assume 'n' number of clients, send requests to the webserver to access the web application **concurrently**.

Node web server receives those requests and places them into a queue known as "**Event Queue**". The Node web server internally maintains a **limited thread pool** to provide service to the client. Let us assume that 'm' number of threads can be created and maintained.

Step 2: The Node web server internally has a component, known as "**Event Loop**". It uses the indefinite loop to receive requests and process them. But the Event loop component uses a "**Single Thread**" to process the requests.

Step 3: The event Loop component checks for any client request that is placed in the Event Queue. If no requests are present, then it waits for incoming requests. If the requests are present, then it picks up one client request from the Event Queue and starts processing that request.

Step 4: If that client request does not require any blocking I/O operations, then the request is processed till completion and the response is sent back to the client.

Step 5.1: If the client request requires some blocking I/O operations like file operations, database interactions, any external services then it checks the availability of threads from the internal thread pool.

Step 5.2: One thread is assigned from the internal pool of threads and assigned to the client request. That thread is responsible for taking that request, processing it, and performing blocking I/O operations.

Step 6: After processing the request, the response is prepared and sends back to the Event Loop. Event Loop then sends that response back to the requested client.

5. Scalability

One of the reasons for the Node.js application scalability is that it makes use of event-driven programming with the Single Threaded Event Loop Mechanism. This enables the Node application to serve a huge number of incoming requests concurrently and it scales up automatically to serve those requests efficiently.

Companies like eBay and Walmart makes use of Node.js to scale up their application to support multiple external services and handle the requests flowing in with those services.

6. I/O bound operations

Due to its asynchronous/non-blocking nature, Node.js can be used to create I/O bound applications that involve huge input-output operations, such as creating real-time applications that have real-time data flowing in. Applications like Facebook, online chat applications, and Twitter are a few examples.

An online marketing company like **eBay** makes use of Node.js to handle lots of I/O-bound operations to handle eBay-specific services that display information on the page.

7. Streaming of data

Node.js can be used to create data streaming applications that involve streaming the data quickly. Node.js has a built-in Stream API available using which we can stream the data very fast. Applications like the Twitter stream, video stream, etc. use this feature.

Media companies like **National Public Radio, Direct TV, HBO** makes use of Node.js to stream the data to their viewers.

8. Modularity

Node.js supports modular JavaScript. Instead of writing code in a single JavaScript file, the code can be written in modules which can then be accessed at multiple places in the application. This helps in easy maintenance and reusability of the code.

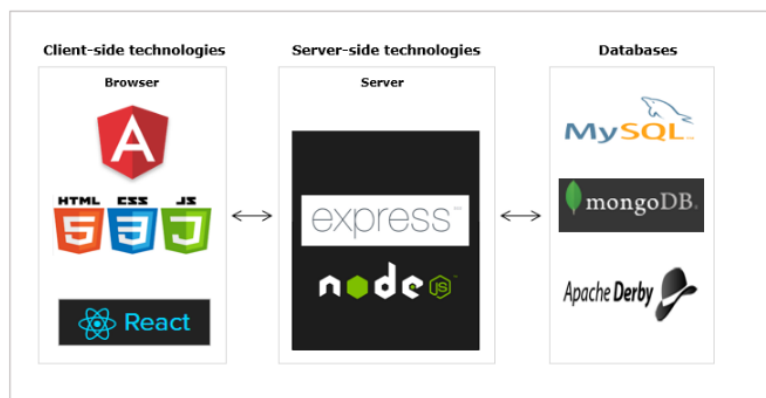
Where to use Node.js and where not to?

Node.js should not be preferred for creating applications that involve CPU intensive operations. If an application involves a lot of calculations that require CPU for processing, is not fit for Node.js.

Node.js in the web application stack :

Now let us take a look at the position of Node.js in the web application stack.

Node.js places itself in the server-side in the complete web application stack and provides a complete server-side solution for application development. Node.js works well with any client-side technology like Angular, React, etc. and any database like MongoDB, MySQL, etc. can be used for data storage.



There are different frameworks built on the Node.js platform which simplifies Node-based application development. The most popular frameworks are Express, Hapi, Loopback, etc.

How to use Node.js:

Download **Node.js** from the official site. Follow the instructions mentioned on the site for installing it on your machine. It is important to download the stable version while installing, as the Node.js team keeps upgrading the version by adding new features and enhancements.

To check whether Node.js is installed or not in your machine, open the **Node command prompt** and check the Node.js version by typing the following command.

```
1. node -v
```

The flag **-v** will display the version of Node.js installed in the machine.

```
D:\>node -v
v14.15.1
```

Node.js also provides a package manager called **NPM(Node Package Manager)** which is a collection of all open-source JavaScript libraries. It helps in installing any library or module into your machine.

Thus we have successfully installed Node.js in the machine and we can start the application development using Node.js.

How to use Node.js:

Now that we know how to install the Node.js platform in our machine, let us create our first Node.js program.

Step 1: Create a folder NodeJS in D drive and create a new JavaScript file, **first.js** inside the folder. Type the below code inside the JavaScript file.

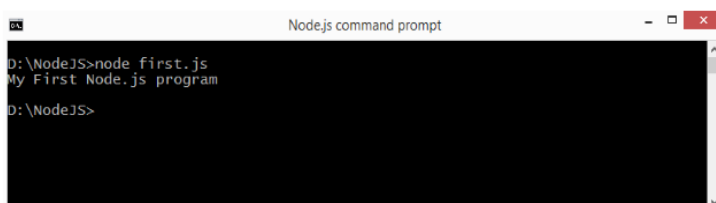
```
console.log("My First Node.js program");
```

Step 2: Navigate to the created NodeJS folder in the NodeJS command prompt and execute the JavaScript file, first.js using the **node** command.

```
node first.js
```

Step 3: After the successful interpretation of the code, we can see the output in the Node.js command prompt as shown below.

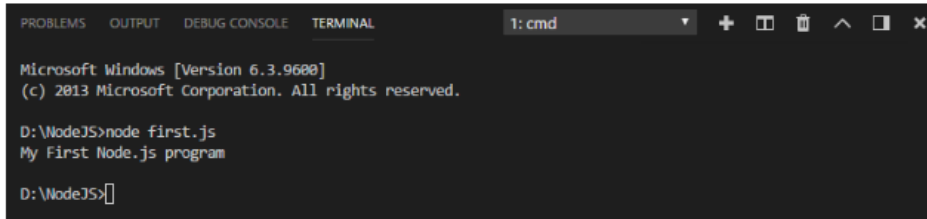
Node.js command prompt:



```
Node.js command prompt
D:\NodeJS>node first.js
My First Node.js program
D:\NodeJS>
```

Visual Studio Code IDE:

You can also open the NodeJS folder, in Visual Studio Code IDE. To execute the first.js file, open the integrated terminal and issue the command for executing the file. You will get the output as shown below:



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

D:\NodeJS>node first.js
My First Node.js program

D:\NodeJS>
```

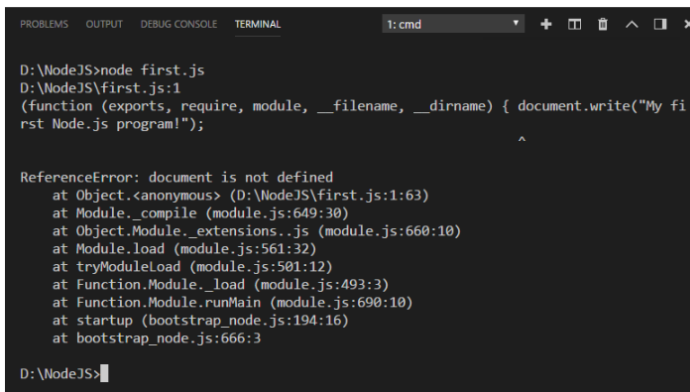
Let us now see how to execute a JavaScript file that contains code for browser interaction through Node.js.

JavaScript code executable in Node.js:

Let us make a small modification to the code written in the first.js created earlier.

```
document.write("My first Node.js program!");
```

We have replaced console.log() with document.write(). The document.write() function displays output on the browser screen. While interpreting the first.js using Node.js now, we will encounter an error as shown below:



```
D:\NodeJS>node first.js
D:\NodeJS\first.js:1
(function (exports, require, module, __filename, __dirname) { document.write("My fi
rst Node.js program!");
                                                                    ^
ReferenceError: document is not defined
    at Object.<anonymous> (D:\NodeJS\first.js:1:63)
    at Module._compile (module.js:649:30)
    at Object.Module._extensions..js (module.js:660:10)
    at Module.load (module.js:561:32)
    at tryModuleLoad (module.js:501:12)
    at Function.Module._load (module.js:493:3)
    at Function.Module.runMain (module.js:690:10)
    at startup (bootstrap_node.js:194:16)
    at bootstrap_node.js:666:3

D:\NodeJS>
```

So, we can now conclude that any JavaScript file which doesn't contain codes for browser interactions will execute successfully using the Node platform.

Create web server in Node.js:

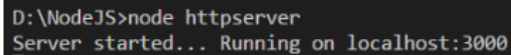
Step 1: Create a new JavaScript file **httpserver.js** and include the HTTP module.

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

Step 2: Use the createServer() method of the HTTP module to create a web server.

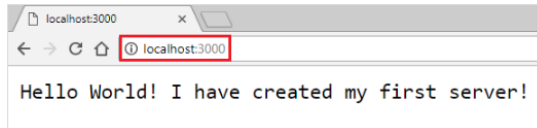
```
var server = http.createServer((req, res) => {
  res.write("Hello World! I have created my first server!");
  res.end();
});
server.listen(3000);
console.log("Server started... Running on localhost:3000");
```


Step 3: Save the file and start the server using the **node** command. When the file executes successfully, we can observe the following output in the console.



```
D:\NodeJS>node httpserver
Server started... Running on localhost:3000
```

Step 4: We will observe the following in the browser.



Thus we have created a Node server and sent the response from the server to the client.

Introduction to Node Package Manager:

We saw the usage of one of the built-in modules in Node.js: HTTP module. There are many other built-in modules as well, provided by Node.js for implementing various functionalities. It is also possible to import an external module into a Node application, using NPM (Node Package Manager).

What is NPM?

NPM stands for Node Package Manager which is a collection of all the open-source JavaScript libraries available in this world. It is the world's largest software registry maintained by the Node.js team.

The npm CLI

NPM provides a command-line interface "**npm**" to work with the NPM repository, which comes bundled with Node. This tool can be used to install, update, or uninstall any package through NPM.

How to get the packages to be installed in our application?

To install any NPM package use the below code in the command prompt:

```
npm install <package_name>[@<version>]
```

This will create a folder `node_modules` in the current directory and put all the packages related files inside it. Here `@version` is optional if you don't specify the version, the latest version of the module will be downloaded.

You can even get tools from NPM like `@angular/cli`, `typescript` (compiler), etc.

There are two modes of installation through NPM

- Global
- Local

Global installation

If we want to globally install any package or tool add `-g` to the command. On installing any package globally, that package gets added to the `PATH` so that we can run it from any location on the computer.

```
npm install -g <package_name>
```

Consider the scenario where "express" which is the most popular framework of Node to be used in our application. Then it can be installed using NPM as below.

```
npm install -g express
```

To install any package that is to be available from anywhere on the computer, then better to go for global installation.

Local installation

If we do not add **-g** to your command for installation, the modules get installed locally, within a node_modules folder under the root directory. This is the default mode, as well.

```
npm install express
```

To install any application-specific package, get it installed locally.

How to update the packages in our application?

We can also update the packages downloaded from the registry to keep the code more secure and stable. Any update for a global package can be done using the following command.

```
npm update -g <package_name>
```

To update a locally installed package, navigate to the folder the package is installed and run the below command. We can also update the packages downloaded from the registry to keep the code more secure and stable. Any update for a global package can be done using the following command.

```
npm update
```

How to uninstall the packages in our application?

We can uninstall the package or module, which we downloaded using the following command.

```
npm uninstall <package_name>[@<version>]  
npm uninstall express@2.1.0
```

NPM Alternatives

NPM is one of the oldest package managers for Node.js. There are other replacements like **YARN** that works with the NPM registry.

What is package.json file?

A Node project needs a configuration file named "package.json". It is a file that contains basic information about the project like the package name, version as well as more information like dependencies which specifies the additional packages required for the project.

To create a package.json file, open the Node command prompt and type the below command.

```
npm init
```

Enter values for package name, version, description, and so on. This will generate a package.json file similar to the one shown below:

```
{  
  "name": "mypackage",  
  "version": "1.0.0",
```

```
"description": "\"Testing publishing\"",
"main": "index.js",
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
"author": "",
"license": "ISC"
}
```

The name and version represent the module name and the version to be used while publishing. The description represents a brief description of the module.

The property main represents the entry point of the application. The test represents all the test scripts to run. The author represents the author's name and the license represents the license type of the module.

Publish custom module to NPM-Demo:

We saw that the NPM repository has a collection of modules that we can download and use in our application. It is also possible to publish the custom modules that we created to NPM so that we can make our modules to be available for others to download.

Let us understand how to publish a custom module to the NPM repository.

Steps to publish a custom module to NPM:

Step 1: Create a user account using the signup option in npmjs.com or using npm adduser command from the command prompt as below:

```
D:\>npm adduser
```

Step 2: Create a package.json file in a folder named mypackage. Specify application configurations in this file.

```
{
  "name": "mypackage",
  "version": "1.0.0",
  "description": "Testing publishing",
  "main": "webServer.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node"
  ],
  "author": "",
  "license": "ISC"
}
```

Step 3: Create a module in the same folder inside which the package.json file resides.

MyModule.js:

```
exports.myMethod = function () {
```

```
console.log('Method invoked from a module');  
};
```

Create another file where MyModule is to be imported and invoke the myMethod() function.

TestModule.js

```
const myModule = require('./MyModule');  
myModule.myMethod();
```

Step 4: Run the code using the Node command and test it.

```
D:\mypackage\>node TestModule.js
```

Step 5: Use npm publish command from the command prompt to publish the module to npm:

```
D:\mypackage\>npm publish + mypackage@1.0.1
```

The package will be published successfully. To use this module from npm, just use the "npm install mypackage" command from the command line and it will get installed.

Node Package Manager:

Step 1: Write the following statement in your terminal in order to install the **express** module using npm.

```
npm install -g express
```

Step 2: You can observe the following in your integrated terminal.

Thus **express** module will be installed in the machine. Similarly, any other module can be installed through the npm repository.

NPM audit:

To perform a quick security check know, as a moment-in-time review of your application, we can make use of npm audit which generates a report on the dependencies of your application. This report consists of security threats to your application and can help you fix vulnerabilities by providing npm commands and recommendations for further troubleshooting.

Use the command

```
npm install npm -g
```

Example:-

running npm audit against myApp:

```
cd myApp  
npm audit
```

Modularizing Node application:

Why do we need to modularize the application?

In an Enterprise application, it is not possible to have all the logic written in a single file. As the complexity of the program increases, it reduces the **readability** and **maintainability** of the application. In this kind of environment, it is easy to lose track of what a particular code does or to produce reusable code. So the application needs to be created in a **modularized** fashion.

What is modularization?

Modularization is a software design technique in which the functionality of a program is separated into independent modules, such that each module contains the desired functionality.

Advantages of modularization:

1. **Readability:** Modular code highly organizes the program based on its functionality. This allows the developers to understand what each piece of code does in the application.
2. **Easier to debug:** When debugging large programs, it is difficult to detect bugs. If a program is modular, then each module is discrete, so each module can be debugged easily by the programmer.
3. **Reusable Code:** Modular code allows programmers to easily reuse code to implement the same functionality in a different program. If the code is not organized modularly into discrete parts, then code reusability is not possible.
4. **Reliability:** Modular code will be easier to read. Hence it will be easier to debug and maintain the code which ensures smoother execution with minimum errors.

Node environment provides many built-in modules that can be used in application development. We have already seen the usage of the "HTTP" module, which is a built-in module in Node.js for creating a web server. Node.js also provides many other useful modules like fs module (used for file-related operations), os module(used for operating system-related functions), net module(used for socket programming), etc. for server-side application development.

Let us explore how to **modularize** an application in Node.

How to modularize code?

Let us explore how to modularize an application in Node.

Consider a simple Node.js application with a Javascript file calculator.js with the below code:
calculator.js

```
async function add(operator1, operator2) {  
  return 'Result: ', operator1 + operator2;  
}  
  
async function subtract(operator1, operator2) {  
  return 'Result: ', operator1 - operator2;  
}  
  
async function asyncCall() {  
  console.log('calling');  
  const result = await add(2, 3);  
  console.log(result);  
}  
asyncCall();
```

By default the functions declared in one module are visible only in that specific module. In order to export the functionalities of one module into another, Node.js has provided an object called "**exports**". This object is a built-in object of Node.js to which all the functionalities are to be attached.

We can assign the function reference to the **exports** object, which can be written as:

```
exports.add = async (operator1, operator2) => {  
  console.log("Result:", operator1 + operator2);  
};  
exports.subtract = async (operator1, operator2) => {  
  console.log("Result:", operator1 - operator2);  
};
```

Now we have exported the functions in the calculator.js file, let us see how to import it in another file.

Importing a module:

Let us now understand how to import the **calculator.js** file into another file.


Consider another file **tester.js**, where we need to import and use the functions in the **calculator.js** file. Use **require()** function and specify the module name to be imported as shown below:

```
const myCalculator = require("./Calculator");
```

The **require()** function takes the path of the file as a parameter and returns an **exports** object. Now in order to use the methods of **calculator.js** add the below code in the **tester.js** file:

```
myCalculator.add(10, 30);  
myCalculator.subtract(30, 10);
```

On executing the **tester.js** file, we will get the following output:



```
Microsoft Windows [Version 6.3.9600]  
(c) 2013 Microsoft Corporation. All rights reserved.  
  
D:\Node_Workspace>node tester.js  
Result: 40  
Result: 20  
  
D:\Node_Workspace>
```

Creating a module in Node:

Let us try out how to create and load a module in a Node application.

Step 1: Create a file DBModule.js within the NodeJS folder created earlier.

```
exports.authenticateUser = (username, password) => {  
  if (username === "admin" && password === "admin") {  
    return "Valid User";  
  } else return "Invalid User";  
};
```

Step 2: Modify the file httpserver.js file created earlier as below.

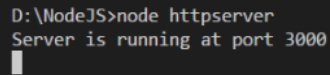
app.js

```
const http = require("http");  
var dbmodule = require("./DBModule");  
var server = http.createServer((request, response) => {  
  result = dbmodule.authenticateUser("admin", "admin");  
  response.writeHead(200, { "Content-Type": "text/html" });  
  response.end("<html><body><h1>" + result + "</h1></body></html>");  
  console.log("Request received");  
});  
server.listen(3000);
```

```
console.log("Server is running at port 3000");
```

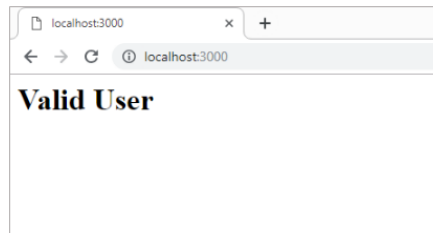
In the httpserver.js file, we are loading the module "DBModule" and then invoking the function named "authenticateUser()".

Step 3: Run the httpserver.js using the **node** command.



```
D:\NodeJS>node httpserver
Server is running at port 3000
```

Open a browser and navigate to URL "http://localhost:3000" and observe the output.



Restarting Node Application:

Whenever we are working on a Node.js application and we do any change in code after the application is started, we will be required to restart the Node process for changes to reflect. In order to restart the server and to watch for any code changes automatically, we can use the Nodemon tool.

Nodemon

Nodemon is a command-line utility that can be executed from the terminal. It provides a different way to start a Node.js application. It watches the application and whenever any change is detected, it restarts the application.

It is very easy to get started with this tool. To install it in the application, run the below command.

```
npm install nodemon -g
```

Once the 'nodemon' is installed in the machine, the Node.js server code can be executed by replacing the command "node" with "nodemon".

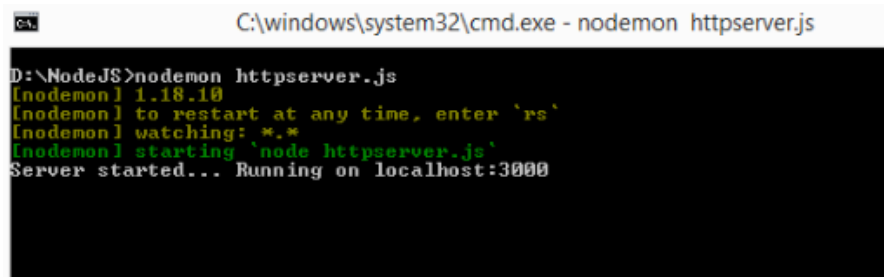
```
nodemon app.js
```

Thus the 'nodemon' starts the application in watch mode and restart the application when any change is detected.

Consider a simple Node.js code for creating a web server.

```
const http = require("http");
var server = http.createServer((req, res) => {
  res.write("Hello World! I have created my first server!");
  res.end();
});
server.listen(3000);
console.log("Server started... Running on localhost:3000");
```

Observe the console on starting the nodemon in a command prompt window.



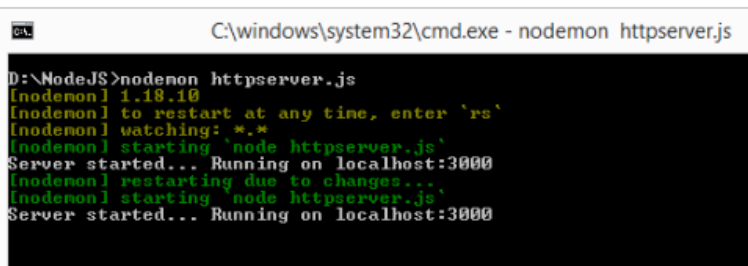
```
C:\windows\system32\cmd.exe - nodemon httpserver.js

D:\NodeJS>nodemon httpserver.js
[nodemon] 1.18.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting 'node httpserver.js'
Server started... Running on localhost:3000
```

Now open the application code and do changes in the code as below.

```
const http = require("http");
var server = http.createServer((req, res) => {
  console.log("Request URL is " + req.url);
  res.write("Hello World! I have created my first server!");
  res.end();
});
server.listen(3000);
console.log("Server started... Running on localhost:3000");
```

Observe the console message in the command prompt. Nodemon automatically restarted the server on observing changes in the code.



```
C:\windows\system32\cmd.exe - nodemon httpserver.js

D:\NodeJS>nodemon httpserver.js
[nodemon] 1.18.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting 'node httpserver.js'
Server started... Running on localhost:3000
[nodemon] restarting due to changes...
[nodemon] starting 'node httpserver.js'
Server started... Running on localhost:3000
```

To avoid over and over restarting of server for small changes to reflect. It is important to have an automatic restart of the server of your application. Use nodemon for this purpose.

Need for File system module:

Let's consider a scenario, the user wants to store the information about the requested URLs to the application/errors that occurred in the application. To store this information, the user can choose any of the following methodologies:

- File
- Database

Storing log details in the Database is not an optimal solution because it may increase the number of API calls for DB interaction. This will impact the performance of the application.

The best solution to use the file system, here. The information will be stored locally in the application without impacting the performance of the application.

One of the important operations in server-side programming is to be able to read or write the content of a file. Node.js comes with the file system (fs) module fs to perform file operations. It provides many useful functionalities to interact with the file system.

What is fs module?

The fs module provides a wrapper that contains the standard file operations for manipulating files and dealing with the computing platform's file system.

How to use fs module?

To include the File System module, use the require() method:

Syntax:

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

Once imported, we can use different methods provided by this module for doing any file manipulations.

fs module – Operations

Some of the file operations that we will be discussing are:

- Writing data to a file
- Reading data from a file
- Updating content in a file

We will now begin our file system operations by understanding how to write data to a file.

Writing data to a file:

The File System module has the following methods for creating a new file and writing data to that file:

- writeFile()
- appendFile()

The fs.writeFile() method will overwrite the content if the content already exists.

If the file does not exist, then a new file will be created with the specified name and content.

Syntax:

```
fs.writeFile(file, data, callback);
```

- file: Placeholder to give the file name in which you are going to write the data.
- data: The data/content must be written to the file.
- callback: The callback method, that will be executed, when 'writeFile()' function is executed. This callback will be executed in both success as well as failure scenarios.

Writing data to a file – Async/Await

Method 1:

Before Node.js v8, If we want to avoid callbacks, we have to manually promisify the fs.writeFile function. Let's manually promisify and wrap it in a function:

```
//Method 1
//promisifying writeFile method
const fs = require('fs');
const writeFilePromise = (file, data) => {
  return new Promise((resolve, reject) => {
    fs.writeFile(file, data, (err) => {
      if (err) reject('Could not write file');
```

```
    resolve('success');
  });
});
};
//Invoking the promise which we have created. Self-invocation function
(async () => {
  try {
    await writeFilePromise('myData.txt', `Hey @ ${new Date()}`);
    console.log('File created successfully with promisify and async/await!');
  } catch (err) {
    console.log(err);
  }
})();
```

Method 2:

In the latest versions of Node.js, 'util.promisify()' will allow us to convert I/O functions that return callbacks into I/O functions that return promises.

```
// Method 2
const fs = require('fs');
const { promisify } = require('util');
const writeFile = promisify(fs.writeFile);
(async () => {
  try {
    await writeFile('myData.txt', `Hey @ ${new Date()}`);
    console.log('File created successfully with promisify and async/await!');
  } catch (err) {
    console.log(err);
  }
})();
```

Appending data to a file :

The appendFile() method first checks if the file exists or not. If the file does not exist, then it creates a new file with the content, else it appends the given content to the existing file.

Syntax:

```
fs.appendFile(path, data, callback)
```

- **path:** Placeholder to give the file name in which you are going to append the data.
- **data:** The data/content which must be appended to the file.
- **callback:** The callback method, that will be executed, when 'appendFile()' function is executed. This callback will be executed in both success as well as failure scenarios.

Look at the following code and try it out. You might have an existing file from the previous demo, paste the below code in that file.

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

```
const { promisify } = require('util');
const appendFile = promisify(fs.appendFile);
(async () => {
  try {
    await appendFile('myData.txt', `\nHey @ ${new Date()}`);
    console.log(
      'File content appended successfully with promisify and async/await!'
    );
  } catch (err) {
    console.log(err);
  }
})();
```

Save the file and run the code using the node command. Open the myData.txt file, we can observe that the new log information has been appended to the file.

Try out some more:

Delete the myData.txt file and re-run the code multiple times with various different inputs and observe.

Reading data from a file:

The fs.readFile() method is used to read the content from a given file.

Syntax:

```
fs.readFile(path, encoding, callback);
```

- **path:** Path where the file with data/content resides, with respect to the root folder.
- **encoding:** an optional parameter that specifies the type of encoding to read the file. Possible encodings are 'ascii', 'utf8', and 'base64'. If no encoding is provided, the default is utf8.
- **callback:** The callback method, that will be executed, when readFile() function is executed.

Consider the below sample code for reading data from a file "myData.txt".

```
const fs = require('fs');
const { promisify } = require('util');
const readFile = promisify(fs.readFile);
(async () => {
  try {
    const fileData = await readFile('myData.txt', 'utf8');
    console.log(fileData);
  } catch (err) {
    console.log(err);
  }
})();
```

Why Express?

Organizations from different domains are using Node.js as a preferred runtime environment for their web applications. But some common **web-development tasks** mentioned below needs a lot of coding in Node.js.

- Implementing Routing for different paths and implementation of route handlers for different HTTP verbs POST, Get.
- Serving HTML, CSS static files.
- Sending dynamic response using templates etc.,

Express is a **layer** built on Node.js which helps us to **manage our web server** and **routes**. Since Express is built on top of Node.js, it is a perfect framework for performing **high-speed input and output operations**.

Advantages of using Express:

- Has Robust API's which will help us to easily configure routes for sending and receiving the data between browser and database.
- Takes less time and fewer lines of code thereby simplifying the application development and making it simple to write safe and modular applications.
- Have been used in organizations in different domains like business, finance, news, social, etc.

What is Express?

Express is the standard server-side framework for Node.js. It simplifies the code development for Node.js developers by introducing a set of APIs which extend Node.js and are easy to implement.

The Express framework provides the following functionalities using which we can:

- Implement **request handlers** for different URL paths in a straightforward manner.
- Use **template engines** like pug, stylus to generate views quickly for a particular response.
- Add **cross-cutting concerns** like logging, authentication, etc. easily, by using middleware concepts,

Express has the following features:-

- **Minimal:** Web framework, only with the essential features.
- **Lightweight:** Robust enough to support simple, complex, or hybrid applications.
- **Flexible:** For implementing different functionalities, middleware which is pluggable JavaScript components, can be used.
- **Powerful:** It gives complete access to the core Node.js APIs.
- **Unopinionated:** It gives the full flexibility to the developer for doing server-side operations. There is no restriction regarding the number of files or the directory structure from the framework.

Express.js is a popular web framework for Node.js that is widely used for building server-side applications. It offers a range of features that make it a popular choice among developers, including:

Easy routing: Express makes it easy to define routes for your application and handle HTTP requests and responses.

Middleware support: Express provides a flexible middleware architecture that allows you to easily add functionality to your application, such as authentication, logging, and error handling.

Template engines: Express supports a variety of template engines, including popular options like Handlebars and Pug, allowing you to easily generate dynamic HTML pages.

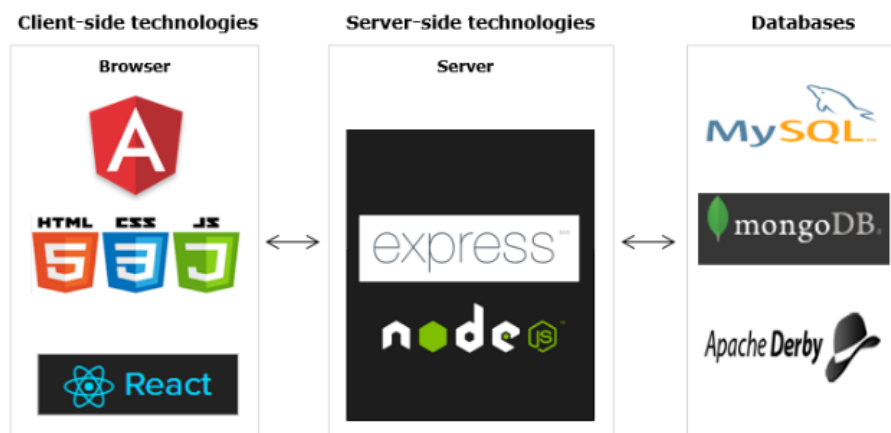
Scalability: Express is designed to be lightweight and flexible, making it a great choice for building scalable applications that can handle a large number of users and requests.

Large ecosystem: Express has a large and active community, with a wide range of plugins and modules available that can be used to add additional functionality to your application.

Overall, Express.js provides developers with a simple, efficient, and flexible way to build web applications with Node.js, making it a popular choice for building both small and large-scale web applications.

Express in web application stack:

Let us see where Express fits in the entire web application stack.



Express places itself on the server-side in the complete application stack and provides a complete server-side solution for application development. Express works well with any client-side technology like Angular, React, etc. and any database like MongoDB, MySQL can be used.

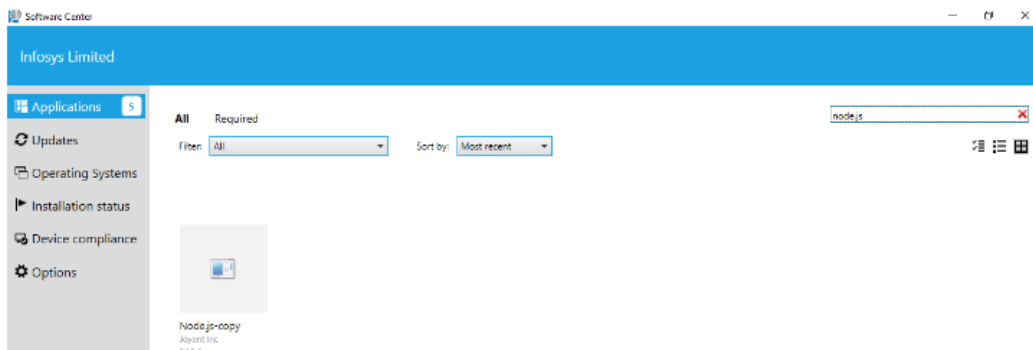
Express Development Environment – Internal

To develop an Express application on a local machine, a development environment with the installation of the below packages are to be set up.

- Node.js
- Express
- Express application generator (optional)

Step 1:

Install Node.js from the software house/software center as shown below or take help from CCD to get it installed.



Follow the instruction to check the Node version installed in the machine.

Open Node command prompt and give the following command.

```
node -v
```

This will display the Node version which is installed.

Step 2:

Installation of Express is straightforward if you have Node.js installed on your machine. Express can be installed using the node package manager (npm).

Note: We must set the proxy in the Infosys network for npm to work. Please visit <http://wiki/Npm> for more details.

```
npm config set registry http://infynexus/nexus/repository/npm-all/
```

Now install Express using the following command:

```
npm install express -g
```

Step 3:

Express provides a scaffolding tool called express-generator which helps to quickly generate an Express application with typical support for routes. The express-generator tool helps in generating the application skeleton which can be later populated with site-specific routes, templates, and database calls.

For installing the express-generator tool globally, use the following command.

```
npm install express-generator -g
```

Once the generator is installed, it is extremely simple to create the application using the 'express' command. To generate a new Express application, use the "express" command, and specify a new application name as shown below.

```
express <<application_name>>
```

The express application generator allows us to configure any template engine and any CSS stylesheet engine while creating a new Express application. To specify any template engine use --view and a CSS generation engine can be specified using --css.

The below command sets 'Pug' as the template engine and 'Stylus' as the CSS generation engine.

```
express --view=pug --css=stylus <<application_name>>
```


It will automatically generate a folder with the supplied application name. Inside the folder, there will be a set of folders and files which are created by the generator tool.

Development Environment Demo

1. Install Express.

```
D:\Demos>npm install express -g
```

2. Install Express-generator

```
D:\Demos>npm install express-generator -g
```

3. Open a command prompt and execute the below command to create a new Express application named 'myApp'.

```
D:\Demos>express myApp
```

```
D:\Demos>express myApp

warning: the default view engine will not be jade in future releases
warning: use '--view=jade' or '--help' for additional options

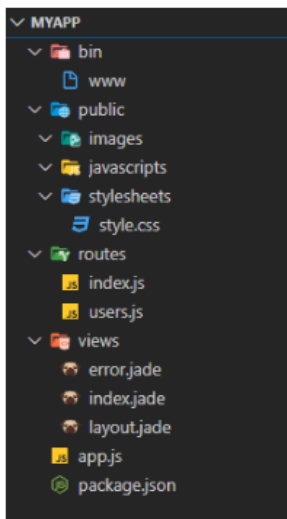
create : myApp\
create : myApp\public\
create : myApp\public\javascripts\
create : myApp\public\images\
create : myApp\public\stylesheets\
create : myApp\public\stylesheets\style.css
create : myApp\routes\
create : myApp\routes\index.js
create : myApp\routes\users.js
create : myApp\views\
create : myApp\views\error.jade
create : myApp\views\index.jade
create : myApp\views\layout.jade
create : myApp\app.js
create : myApp\package.json
create : myApp\bin\
create : myApp\bin\www

change directory:
> cd myApp

install dependencies:
> npm install

run the app:
> SET DEBUG=myapp:* & npm start
```

4. Observe the new project structure generated.



Folder description:

- **bin:** Contains the configuration file for the environment.
- **public:** Contains the static files which we can use in the application.
- **routes:** Contains all the routes created in the application.
- **views:** Contains the view templates, default jade template files.
- **app.js:** Contains application-level configurations and the starting point of the application.
- **package.json:** The package.json file is usually present in the root directory of a Node.js project. This file helps npm to identify the project and handle its dependencies. It consists of other metadata, vital to end-users, such as the description of the project, its version, license information, other configuration data, etc.

Best Practices Tips: Add scripts to your package.json

Simply add a scripts property and object to your package.json with a start key. Its value should be the command to launch your app. For example,

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

Code linting:

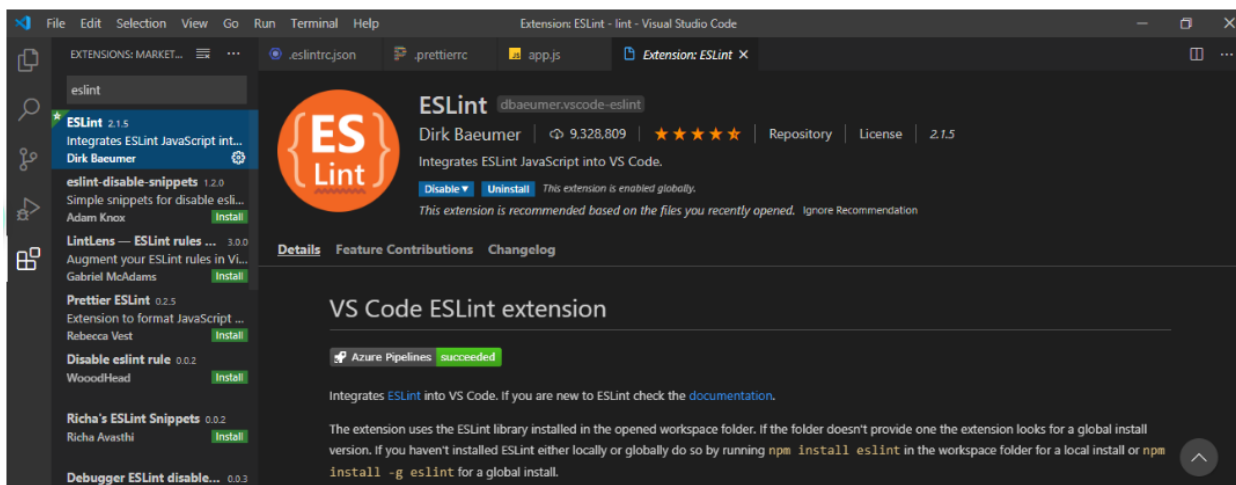
We will be writing the logic in the application based on the requirements provided, but as a developer, we must take care of the standards of the code. It can be taken care of by the developer by knowing all the standards or we can make use of some packages and extensions which are available.

We will explore ESLint and learn how to ensure the coding standards are followed in the code.

Install eslint module global to the machine.

```
npm install eslint -g
```

After installing the node modules enable ESLint from the extensions of VSCode.



Now let us learn how to configure the projects.

Basic Configuration:

VLITS, Vadlamudi.

To create a basic eslint config file follow the steps given. [If you want to set up with prettier and other advanced options proceed to the advanced configuration details on the next page]

1. Create an eslint config file for this folder using the below command:

```
eslint --init
```

2. The following 3 options will be available for the user to choose from.

```
D:\demo>eslint --init
? How would you like to use ESLint? (Use arrow keys)
  To check syntax only
> To check syntax and find problems
  To check syntax, find problems, and enforce code style
```

Based on the requirement we can choose any of these.

3. Choose the type of module which will be used in the project.

```
D:\demo>eslint --init
? How would you like to use ESLint? To check syntax and find problems
? What type of modules does your project use? (Use arrow keys)
> JavaScript modules (import/export)
  CommonJS (require/exports)
  None of these
```

4. Choose the framework based on the requirement.

```
D:\demo>eslint --init
? How would you like to use ESLint? To check syntax and find problems
? What type of modules does your project use? JavaScript modules (import/export)
? Which framework does your project use?
  React
  Vue.js
> None of these
```

5. Select whether Typescript is used in the application.

```
D:\demo>eslint --init
? How would you like to use ESLint? To check syntax and find problems
? What type of modules does your project use? JavaScript modules (import/export)
? Which framework does your project use? None of these
? Does your project use TypeScript? (y/N) N
```

6. Choose where the application will primarily run.

```
D:\demo>eslint --init
? How would you like to use ESLint? To check syntax and find problems
? What type of modules does your project use? JavaScript modules (import/export)
? Which framework does your project use? None of these
? Does your project use TypeScript? No
? Where does your code run? (Press <space> to select, <a> to toggle all, <i> to invert selection)
(*) Browser
> ( ) Node
```

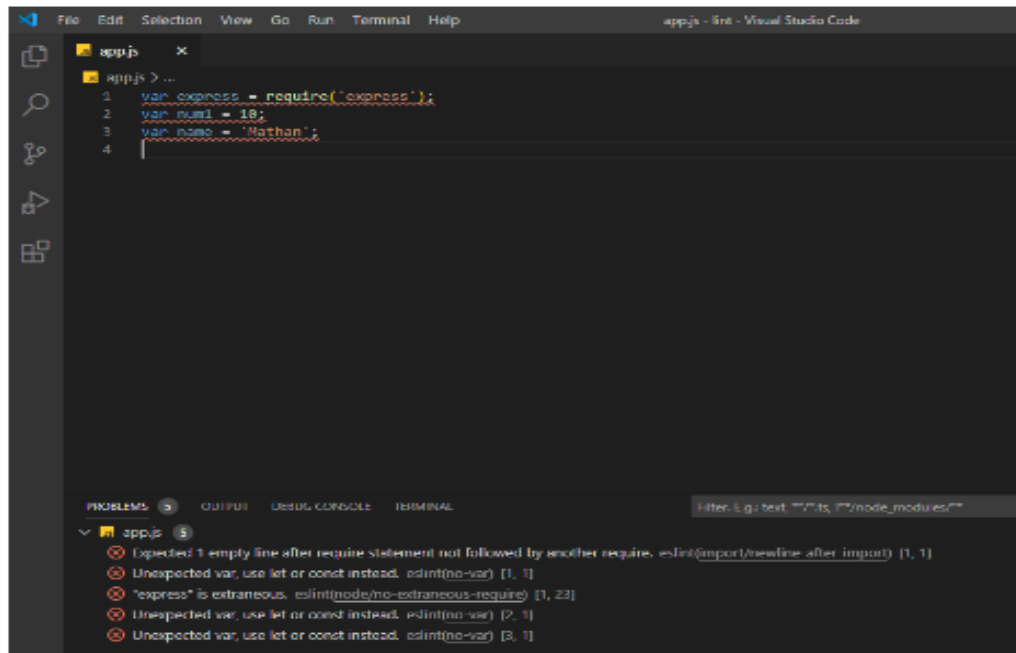
7. Select the format in which you need to create the config file.

```
D:\demo>eslint --init
? How would you like to use ESLint? To check syntax and find problems
? What type of modules does your project use? JavaScript modules (import/export)
? Which framework does your project use? None of these
? Does your project use TypeScript? No
? Where does your code run? Browser
? What format do you want your config file to be in? (Use arrow keys)
> JavaScript
  YAML
  JSON
```

8. On successful installation of all required modules, the config file **".eslintrc.json"** will be created in the project.

```
.eslintrc.json X
.eslintrc.json > ...
1 {
2   "env": {
3     "browser": true,
4     "es2020": true
5   },
6   "extends": "eslint:recommended",
7   "parserOptions": {
8     "ecmaVersion": 11,
9     "sourceType": "module"
10  },
11  "rules": {
12  }
13 }
14
```

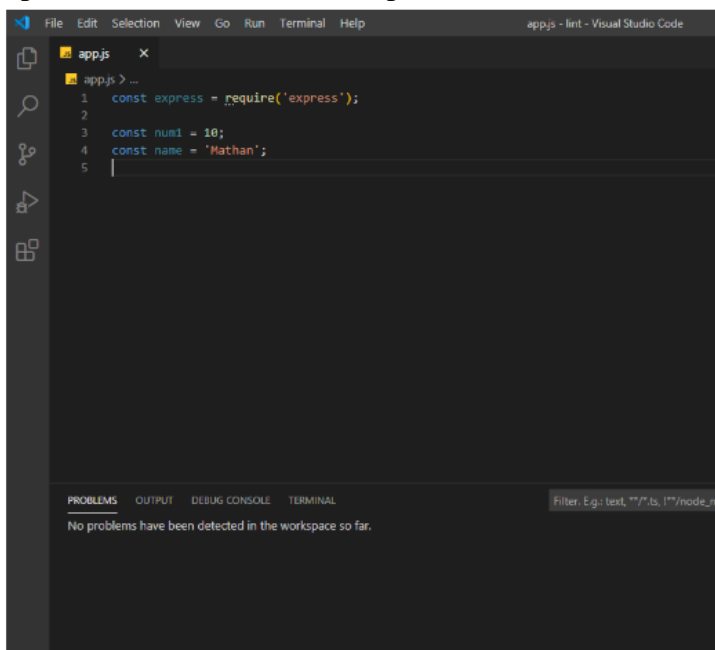
Code with ESLint problems:



The screenshot shows the Visual Studio Code editor with a file named `app.js`. The code contains four lines of JavaScript code using `var` for variable declarations. The bottom panel shows the `PROBLEMS` view with five ESLint errors:

- Expected 1 empty line after require statement not followed by another require. `eslint(import/newline after import)` [1, 1]
- Unexpected var, use let or const instead. `eslint(no-var)` [1, 1]
- "express" is extraneous. `eslint(node/no-extraneous-require)` [1, 23]
- Unexpected var, use let or const instead. `eslint(no-var)` [2, 1]
- Unexpected var, use let or const instead. `eslint(no-var)` [3, 1]

Updated code without ESLint problems:



The screenshot shows the Visual Studio Code editor with the same file `app.js`, but the code has been updated to use `const` instead of `var`. The bottom panel shows the `PROBLEMS` view with the message: "No problems have been detected in the workspace so far."

Why Routing?

Routing is a fundamental concept in web development that refers to the process of directing incoming requests to the appropriate handler functions in the server. In Express.js, routing is used to define the endpoints or URLs of the application and map them to specific handler functions that perform certain tasks or return responses.

Routing in Express.js provides several benefits:

Separation of Concerns: By defining routes in a separate module, the application logic is separated into different components, making it easier to manage and maintain.

Reusability: Routes can be reused across different parts of the application, reducing code duplication and making it easier to add new features.

URL Parameters: Express.js allows developers to define URL parameters that can be passed to the route handlers. This enables dynamic URL generation and more flexible request handling.

Middleware: Express.js middleware functions can be used to perform pre-processing or post-processing of requests or responses. Middleware can be used to handle authentication, logging, error handling, and more.

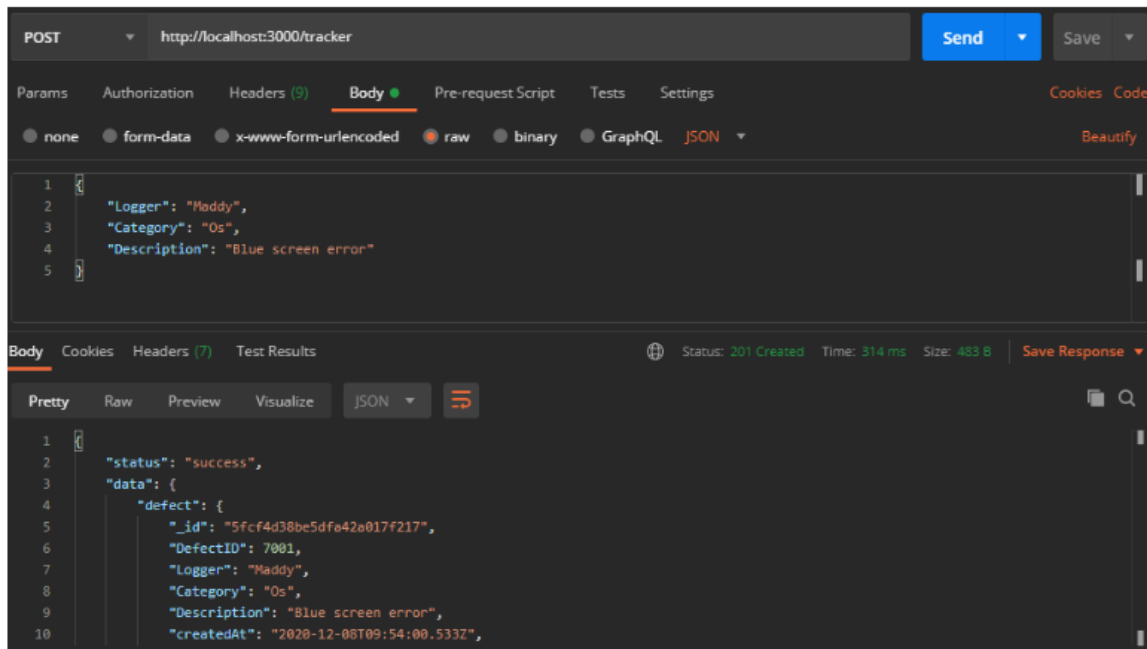
Overall, routing in Express.js provides a flexible and powerful mechanism for defining endpoints and handling requests in web applications.

routing using applic

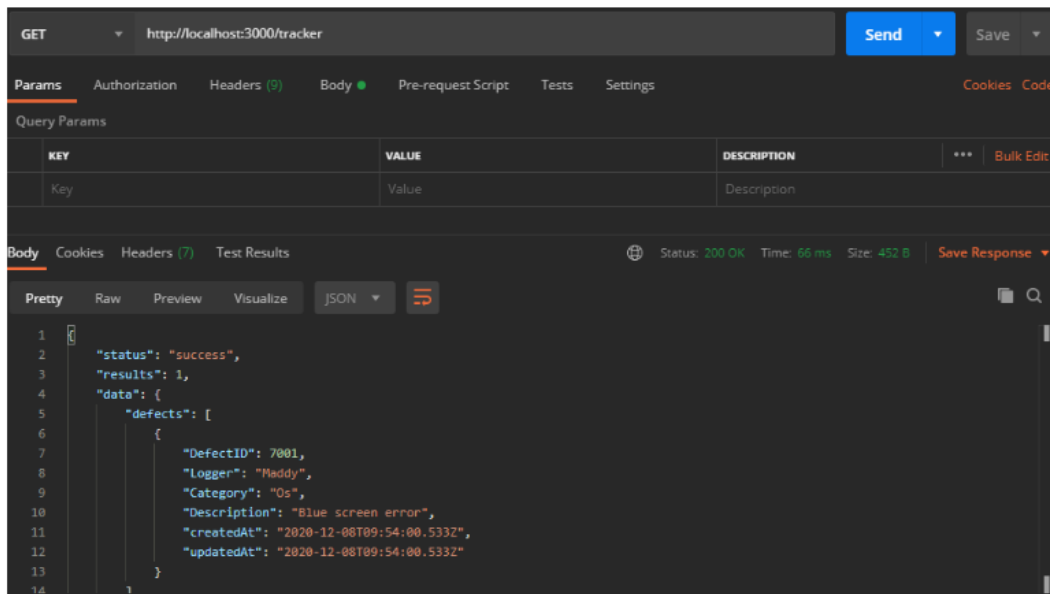
To explore the various features available in any web application, we need to request the correct URL for the feature specific feature we are currently interested in.

For a web application, all the URLs will reach the same server, but paths will be different. It is the server's responsibility to route the URLs to the appropriate features depending on the path.

In the techDesk application, whenever the user sends a POST request to "http://localhost:3000/tracker", the server receives the POST request to log a new defect. Here, "http://localhost:3000/" has the address and port number of the machine where the server is hosted and "/" is the path.



When a GET request is sent to the server with the URL <http://localhost:3000/tracker>, the GET request is fired which fetches all defects.



In the above two examples, we see that all the requests are received by the same server running on port 3000. However, depending on the path and HTTP method, different responses are coming from the server. The server needs to decide which response is to be sent for a particular path in the URL. Express provides routing API to help to manage these requirements whenever a request is received by the server.

What is Routing?

Defining the endpoints of an application and the way the application responds to the incoming client requests are referred to as routing.

A route is nothing but a mixture of a URI, HTTP request method, and some handlers for that particular path.

Routing can be defined using two ways

- Application Instance
- Router class of Express

Method 1

routing using application instance in express js

In Express.js, routing is implemented using the application instance returned by calling `express()`. The application instance is an object that represents the entire Express.js application, and it is used to define routes, middleware, and other application-level settings.

To define a route using the application instance, you can use the HTTP methods like `get()`, `post()`, `put()`, `delete()`, etc. These methods take two arguments: the route path and the route handler function.

Here is an example of defining a route using the application instance in Express.js:

In this example, we define three routes: one for the root URL, one for the `/users` URL, and one for the `/users/:id` URL, which includes a parameter for the user ID. The routes are defined using the `get()` method of the application instance, and each route has a corresponding route handler function that sends a response to the client.

Finally, we start the server by calling the `listen()` method of the application instance and passing the port number to listen on.

We can do routing using the application object. Look at the below code snippet which does the routing using the application object.

```
const express = require('express');
const app = express();
app.get('/', myController.myMethod);
app.get('/about', myController.aboutMethod);
```

Method 2

Let's use the **express.Router** class. Router class helps in grouping the route handlers together for a site and allows them to access them using a common route-prefix.

The below code explains how to define routes in an Express application.

route.js file:

```
const express = require('express');
const router = express.Router();
router.get('/', myController.myMethod);
router.get('/about', myController.aboutMethod);
module.exports = router;
```

Now to use the route module created above, we need to import it and then associate it with the application object. In **app.js** file include the below code:


```
const express = require('express');
const router = require('./routes/route');
const app = express();
app.use('/', router);
```

This application will now be able to route the paths defined in the route module.

Defining a route:

A route can be defined as shown below:

```
router.method(path,handler)
```

router: express instance or router instance

method: one of the HTTP verbs

path: is the route where request runs

handler: is the callback function that gets triggered whenever a request comes to a particular path for a matching request type

Route Method:

The application object has different methods corresponding to each of the HTTP verbs (GET, POST, PUT, DELETE). These methods are used to receive HTTP requests.

Below are the commonly used route methods and their description:

Method	Description
get()	Use this method for getting data from the server.
post()	Use this method for posting data to a server.
put()	Use this method for updating data in the server.
delete()	Use this method to delete data from the server.
all()	This method acts like any of the above methods. This can be used to handle all requests.

Examples:

```
routing.get("/notes", notesController.getNotes);
routing.post("/notes", notesController.newNotes);
routing.all("*", notesController.invalid);
```

Here, notesController is the custom js file created to pass the navigation to this controller file. Inside the controller, we can create methods like getNotes, newNotes, updateNotes, etc. to perform various database related queries like reading from the database, inserting into the database, etc. More on this will be explained later module of the course.

Route Paths

Route path is the part of the URL which defines the endpoint of the request.

- For the URL "**http://localhost:3000/**" - **'/'** is the endpoint
- For the URL "**http://localhost:3000/about**" - **'/about'** is the endpoint

We can handle the endpoints as follows, in an Express application.

```
router.get('/',myController.myDefaultMethod);
router.get('/about',myController.aboutMethod);
```

In the above example, we have strings like '/' and '/about' as part of the route path. The path can also be string patterns or regular expressions.

String-based routes: Pass a string pattern as the first parameter of the routing method.

```
router.get('/ab*cd', myController.myDefaultMethod);
```

The above route definition responds to all the paths starting with "ab" and ending with "cd". For eg: "/abxyzcd", "/abbcd"). The request will be handled by the route handler, myDefaultMethod written in myController.

Regular expression routes: Pass a regular expression object as the first parameter to the routing method.

```
router.get('/x/',myController.myMethod);
```

The above route definition responds to any path with an 'x' in the route name.

In a general web application, string-based routes are used mostly and regular expression-based routes are used only if required absolutely.

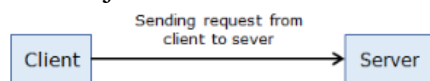
Handling Routes:

Route handler can be defined as functions that get executed every time the server receives a request for a particular URL path and HTTP method.

The route handler function needs at least two parameters: **request object** and **response object**.

Request object

The HTTP request object is created when a client makes a request to the server. The variable named **req** is used to represent this object.



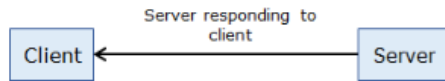
The request object in Express provides many properties and methods to access the request related information sent by the client to the server.

Below are some commonly used methods of the request object.

Property	Description
req.app	Provides the Express application instance's reference.
req.body	It contains the data sent as part of the body in HTTP post request as key-value pairs.
req.cookies	contains the information about the cookie sent with the request.
req.hostname	The host of the request HTTP header.
req.ip	The remote IP address of the request.
req.params	It contains the parameter send along with the request URL.
req.path	Provides the path from request URL.

Response object

The HTTP response object has information about the response sent from the server to the client. The response object is created along with the request object and is commonly represented by a variable named **res**.



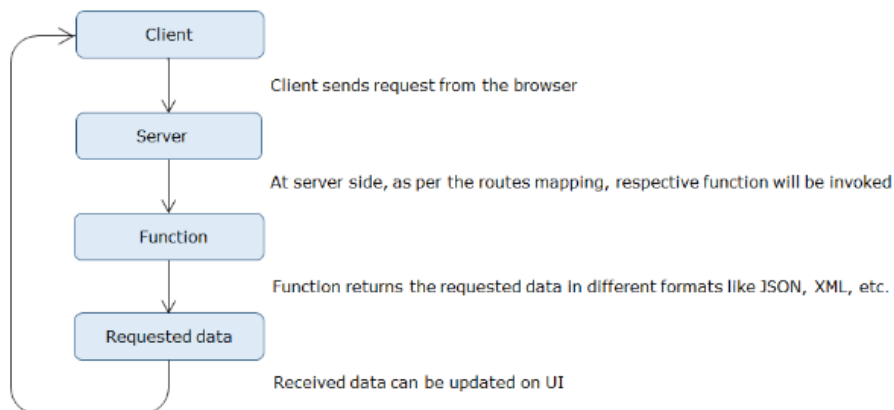
The route handler function sends a response using `res.send()` method whenever a GET request is received.

```
res.send('About us page');
```

The response object in Express also supports many methods for sending different types of responses. Have a look at some commonly used methods of the response object.

Property	Description
<code>res.download()</code>	To download a file.
<code>res.end()</code>	To end the process of response.
<code>res.json()</code>	To send response in JSON format.
<code>res.jsonp()</code>	To send response in JSON format with JSONP support.
<code>res.redirect()</code>	To redirect a request to different URL.
<code>res.render()</code>	To render response through a view template.
<code>res.send()</code>	To send any type of response.
<code>res.sendFile</code>	To send a file as response.
<code>res.sendStatus()</code>	To set the status code of response

Have a look at the basic architecture of Express routing.



The above diagram shows the flow of data while request handling.

Whenever a client sends a request to the server, that request is forwarded to the corresponding route handler function. The function processes the request and then generates the response that is to be sent back to the client. The client will be able to see the response in the browser.

Handling Routes-an example

Let us look at the below router code to understand how routes can be handled.

```
routing.get("/notes", notesController.getNotes);
```

To the express routing object, we can first configure the HTTP method and then pass a route handler function written in a custom controller file like the notesController.js file.

The getNotes model method written inside notesController.js can be used to send back a response message in JSON format as shown below:

For example:

```
exports.getNotes = async (req, res) => {  
  try {  
    res.status(200).json({  
      message: 'You can now get the requested notes for your request ',  
    });  
  } catch (err) {  
    res.status(404).json({  
      status: 'fail',  
      message: err,  
    });  
  }  
};
```

Here we are returning a JSON response with a message field:

```
try {  
  res.status(200).json({  
    message: 'You can now get the requested notes for your request ',  
  });  
}
```

Let us take one more example with the HTTP post method.

```
routing.post("/notes", notesController.newNotes);
```

The handler can be written as,

```
exports.newNotes = async (req, res) => {  
  try {  
    res.status(201).json({  
      data: 'New notes added for the POST request',  
    });  
  } catch (err) {  
    res.status(404).json({  
      status: 'fail',  
      message: err.errmsg,  
    });  
  }  
};
```

Further, in the course, we will learn how to connect to the database for the above handlers to make the operations work with the database.

Route parameters

Route parameters are useful for capturing the values indicated at certain positions in the URL. These are defined as named URL segments.

For example, in the URL, *'http://localhost:3000/user/Smith'*, if we want to capture the portion of URL which contains "Smith", then

we can configure the route parameter as below:

```
router.get('/user/:username',myController.getMethod);
```

Here inside the `getMethod` of Controller, `username` is the route parameter. We access it using the below syntax.

```
req.params < parameter_name >
exports.getMethod = async (req, res) => {
  const name = req.params.username;
  res.send(`Welcome ${name}`);
};
```

If more than one parameter is passed as part of the request URL, then the required information can be extracted as shown below.

```
router.get('/user/:username/:id',myController.getMethod)
exports.getMethod = async (req, res) => {
  const username = req.params.username;
  res.send(`Welcome ${username} with id ${req.params.id}`);
};
```

Query Parameters:

Query strings are the data appended as part of the request URL. Query strings start with a question mark and the name-value pair in it is separated by an `&`(ampersand).

For example, in the below URL,

'http://localhost:3000/login?username=john&email=john%40i.com&login=Login',

the querystring is

`"username=john&email=john%40i.com&login=Login"`

Consider the below example HTML form which submits the data to the login path using HTTP get method.

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

```
<body>
  <form name="loginform" action="http://localhost:3000/login" method="get">
    Enter your name
    <input type="text" name="username" value="">
    <br> Enter the email
    <input type="email" name="email" value="">
    <br>
    <input type="submit" name="submit1" value="Register">
  </form>
</body>
</html>
```

In **Routing.js**, we can handle this request as shown below:

```
const express = require('express');
const routing = express.Router();
const notesController = require('../Controller/myNotes');
routing.get('/login', notesController.loginMethod);
```

Since the form submits data using HTTP get method, the data, i.e., the username and email will be appended as part of the query string. While handling the route, we can extract these in the Controller, as shown below:

```
exports.loginMethod = async (req, res) => {
  const name = req.query.username;
  res.send(` You have registered in with username ${name} `);
};
```

The server grabs the query string using request object's **query** property followed by name of the query string name-value pair.

```
request.query.<querystring name-value pair>
```

Why Middleware?

Consider the scenario where the following tasks are to be designed in our application.

- Authenticate or authorize requests
- Log the requests
- Parse the request body
- End a request-response cycle

The above jobs are not the core concerns of an application. But they are the cross-cutting concerns that are applicable to the entire application.

In the Express framework, these cross-cutting concerns can be implemented using **middleware**.

The middleware concept helps to keep the router clean by moving all the logic into corresponding external modules. The middleware is a great concept for organizing the code.

What is a Middleware?

A middleware can be defined as a function for implementing different cross-cutting concerns such as authentication, logging, etc.

The main arguments of a middleware function are the **request** object, **response** object, and also the **next** middleware function defined in the application.

A function defined as a middleware can execute any task mentioned below:

- Any code execution.
- Modification of objects - request and response.
- Call the next middleware function.
- End the cycle of request and response.

Middleware functions are basically used to perform tasks such as parsing of requests body, cookie parsing for cookie handling, or building JavaScript modules.

How Middleware works?

In order to understand how middleware works, let us take a scenario where we want to log the request method and request URL before the handler executes.

Below is the route definition for which we want to add the middleware.

```
app.get('/login', myController.myMethod);
exports.myMethod = async (req, res, next) => {
  res.send('/login');
};
```

The middleware for the above requirement could be written as below.

```
const mylogger = async (req, res, next) => {
  console.log(new Date(), req.method, req.url);
  next();
};
```

The arguments of this function are:

- **req**: an object containing all the information about the request.
- **res**: an object containing all the information about the response sent from server to client.
- **next**: tells Express when the middleware is done with the execution.

Inside the function, we have logic to log the request method and request URL along with the date. The **next()** method ensures that after the execution of middleware logic, the handler is executed.

Now we can modify the route definition to add the middleware using **app.use()** method.

```
app.use(mylogger);
app.get('/login', myController.myMethod);
```

Now, whenever any request is coming to the path '/login', **mylogger** middleware function will get executed, and then the corresponding handler function will be invoked.

Any middleware can be loaded in an Express application using the **app.use()** method.

The **app.use()** method accepts one string parameter path, which is optional, and one function parameter callback which is the mandatory middleware function.


```
app.use(PATH, CALLBACK)
```

Whenever any request comes to a particular "path", the middleware callback function will get triggered.

To associate a middleware directly with the handler function use **app.use()** method.

```
app.use('/login', (req, res, next) => {  
  console.log(new Date(), req.method, req.url);  
  next();  
});
```

Chaining of Middleware

- We can create a chain of middlewares before the request reaches the handler.
- Consider a middleware which logs the request time and another middleware which logs the request URL as shown.

```
const logtime = async (req, res, next) =>{  
  console.log('Request received at ' + Date.now());  
  next();  
};  
const logURL = async (req, res, next) =>{  
  console.log('Request URL is ' + req.url);  
  next();  
};  
  
app.use(logtime);  
app.use(logURL);
```

- Both middlewares are associated with the application object using **app.use()** method.
- Now whenever an HTTP request arrives at the application, it goes through these two middlewares. Both middlewares in the chain can modify the request and response object based on the requirement.

Middleware Ordering

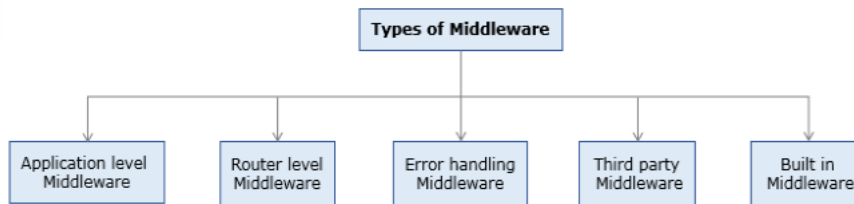
- The order of execution of middleware depends on the order in which the route handler functions and other middleware functions are declared in the application.
- Consider the below code where middleware is configured, which logs the request URL. The middleware is added between two route definitions.

```
app.get('/', myController.getMethod);  
app.use(myLogger);  
app.post('/', myController.postMethod)
```

In the above code, the middleware will never run for the route handler for the GET request, as it is declared after the route definition. The middleware will get executed only for the POST handler as it is declared before the route definition for the POST handler.

Types of Middlewares

The express framework provides the following five different types of middlewares.



Let us explore each of these middlewares in detail.

Application Level Middleware

- Application-level middlewares are functions that are associated with the application object. These middleware function will get executed each time an application receives a request.
- The middleware that we have discussed earlier was also an application-level middleware.

`app.use('/', route);`

Demo steps:

1. Modify the app.js file in the application by adding middleware to it.

```
const express = require('express');
const router = require('./Routes/routing');
const app = express();
// Custome logger middleware
const mylogger = function (req, res, next) {
  console.log(`Req method is ${req.method}`);
  console.log(`Req url is ${req.url}`);
  next();
};

// using app object make use of logger middleware function
app.use(mylogger);
app.use('/', router);
app.listen(3000);
console.log('Server listening in port 3000');
```

2. The routing.js file content.

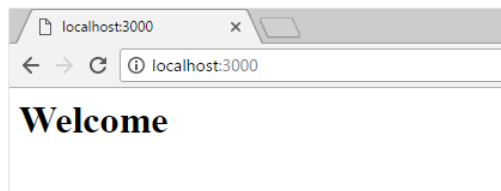
```
const express = require('express');
const router = express.Router();
```

```
const myController = require('../Controller/myController');
router.get('/', myController.myMethod);
router.get('/about', myController.aboutMethod);
module.exports = router;
```

3. In Controller, add the below-mentioned code.

```
exports.myMethod = async (req, res, next) => {
  res.send('<h1>Welcome</h1>');
};
exports.aboutMethod = async (req, res, next) => {
  res.send('<h1>About Us Page</h1>');
};
```

4. Run the application using 'node app' and observe the output.



On changing the URL to 'http://localhost:3000/about', we get the below message:



The middleware output can also be seen in the console as shown below:

```
Server listening in port 3000
Req method is GET
Req url is /
Req method is GET
Req url is /about
```

Router Level Middleware

Router-level middlewares are functions that are associated with a route. These functions are linked to `express.Router()` class instance.

Demo steps:

1. Modify routing.js file by adding a middleware in it.

```
const express = require('express');
const router = express.Router();
const myController = require('../Controller/myController');
router.use((req, res, next) => {
  console.log(`Req method is ${req.method}`);
  console.log(`Req url is ${req.url}`);
  next();
});
```

```
});  
router.get('/', myController.myMethod)  
router.get('/about', myController.myaboutMethod)  
module.exports = router;
```

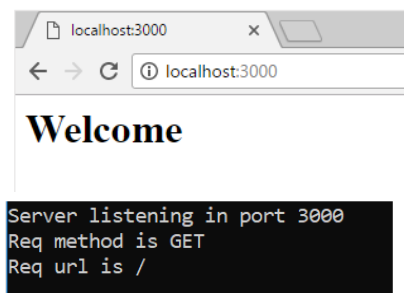
2. Add the below code to the Controller:

```
exports.myMethod = async (req, res) => {  
  res.send('Welcome');  
};  
exports.myaboutMethod = async (req, res) => {  
  res.send('About us');  
};
```

3. The app.js file in TestApp.

```
const express = require('express');  
const router = require('./Routes/routing');  
const app = express();  
app.use('/', router);  
app.listen(3000);  
console.log('Server listening in port 3000');
```

4. Open a command prompt and start the server. Observe the output



Demo 2: Router Level Middleware

If we want to use middleware for only a specific route, we can use the below code:

Demo steps:

1. Modify routing.js file by adding a middleware in it.

```
const express = require('express');  
const router = express.Router();  
const myController = require('../Controller/myController');  
router.use('/about', (req, res, next) => {  
  console.log(`Req method is ${req.method}`);  
  console.log(req.originalUrl);  
  next();  
});  
router.get('/', myController.myMethod)
```

```
router.get('/about', myController.myaboutMethod)
module.exports = router;
```

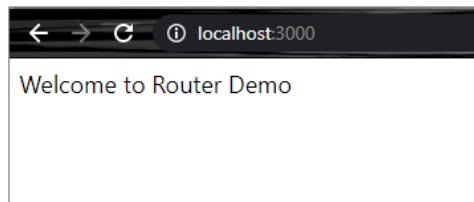
2. In app.js add the below-mentioned code:

```
const express = require('express');
const router = require('./Routes/routing');
const app = express();
app.use('/', router);
app.listen(3000);
console.log('Server listening in port 3000');
```

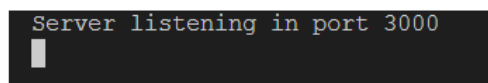
3. In Controller, we have the below code.

```
exports.myMethod = async (req, res) => {
  res.send('Welcome to Router Demo');
};
exports.myaboutMethod = async (req, res) => {
  res.send('About us');
};
```

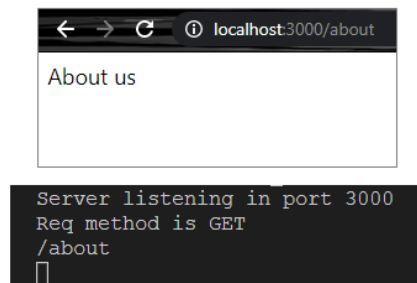
4. Run the application and observe the below output.



In the console we can observe the middleware is not logged:



On changing the URL to http://localhost:3000/about:



Error Handling Middleware

Handling errors is an important aspect of any application. Error handling middleware provides the error-handling capability in an Express application. Whenever any run-time error occurs in the application, Express will call the error-handling middleware. The middleware can handle any type of run-time error and also format the error into a user-friendly error message.

The best way to handle the errors is by creating an error object with the error status code and error message and by throwing the error object. This can be passed to the errorlogger[custom method] which can handle it appropriately.

app.js

```
const express = require('express');
const errorLogger = require('./errorlogger')
const app = express();

app.all('*', (req, res, next) => {
  let err = new Error();
  err.message = 'Invalid Route';
  err.status = 404;
  next(err);
});

app.use(errorLogger)
app.listen(3000);
console.log('Server listening in port 3000');
module.exports = app;
```

errorlogger.js

This code will log the details to the ErrorLogger.txt and will send the response to the client.

```
const fs = require('fs');
let errorLogger = (err, req, res, next) => {
  if (err) {
    fs.appendFile('ErrorLogger.txt', `${new Date().toDateString()} - ${err.message}\n`, (error) => {
      if (error) {
        console.log("logging failed");
      }
    });
    if(err.status){
      res.status(err.status);
    }
    else{
      res.status(500)
    }
    res.json({
      status: 'error',
      message: err.message
    })
  }
}
module.exports = errorLogger;
```

Third Party Middleware

Third-party middleware provides the capability of adding more functions to an Express application using a third party module.

For using any third-party middleware in the Express application, the corresponding module for that particular functionality is to be installed, and then it is to be loaded at the application level or router level based on the requirement.

Some of the commonly used middlewares are:

- body-parser
- cookie-parser
- helmet
- session

Body Parser

Let us explore one of the commonly used middleware called body-parser.

Body-parser:

The users are always prompted to enter data in a form, which is a powerful tool in any web application. On submitting the forms using the POST method, the data is sent as part of the request's body. The form data is hence not exposed in the URL. POST submission handling is therefore different from GET submissions and it needs more processing.

The Express framework provides a middleware 'body-parser' for extracting the data coming as part of the request body, whenever the client sends any request. To access the POST parameters, it provides a body property for the request object.

This middleware can be installed using the below command, for use in an Express application:

```
npm install body-parser
```

Consider the code snippet to use this middleware in the application.

```
const bodyParser = require('body-parser');  
app.use(bodyParser.json());
```

The body-parser middleware is imported into the application and then associated with the application object using app.use() method. The above code snippet uses bodyParser.json() method to parse the content.

Depending on the specific middleware that is used, the body-parser parses the body using one of the below four strategies:

- **bodyParser.raw():** Reads the buffered up contents
- **bodyParser.text():** Reads the buffer as plain text

- **bodyParser.urlencoded():** Parses the text as URL encoded data
- **bodyParser.json():** Parses the text as JSON

Once the middleware is configured, the data submitted through the POST method can be accessed using **req.body.variable_name**.

Build-In Middleware

- Built-in middleware is a set of functions that are built-in with Express.
- Let us see an example for a built-in middleware in Express which is `express.static`. It has the responsibility of serving static assets like images, CSS files, and JavaScript files within the application.
- Consider the configuration of `express.static` middleware, which is associated with an application object.

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

- The static files stored in the "public" directory is accessible from the browser directly after configuring this middleware.
- If there are multiple static asset directories, modify the code as below by calling the `express.static()` function multiple times.

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

```
app.use(express.static('./downloads'));
```

```
app.use(express.static('./files'));
```

Express looks up the files in the order in which we set the static directories with this middleware function.

Connecting to MongoDB with Mongoose:

Interacting with databases

Organizations use various databases to store data and to perform various operations on the data based on the requirements.

Some of the most popular databases are:

- Cassandra
- MySQL
- MongoDB
- Oracle
- Redis
- SQL Server

A user can interact with a database in the following ways.

- Using query language of the respective databases' - eg: SQL
- Using an ODM(Object Data Model) or ORM(Object-Relational Model)

In this module, we will be dealing with the usage of the Object Data Model (ODM) to interact with the database.

Need for ODM

Let us assume, we wish to maintain the details of the employees in the below format in the MongoDB database.

empName	empId	Location
Alex	123456	Europe
Sam	432567	California

If we try to insert the below employee record into the collection, it will be inserted.

```
{empName : "John", employeeId : 324123, Location : "France"}
```

But if we carefully observe, the below record did not exactly match the format as we discussed. After inserting the above record, the collection will be looking as shown below:

empName	empId	Location	employeeId
Alex	123456	Europe	
Sam	432567	California	
John		France	324123

MongoDB stores the data in the form of a document in the collection. Since there is no predefined structure and constraints on values for fields, a document of any structure having any value for fields can be inserted into the collection. It becomes the responsibility of developers to strictly maintain the structure of the document and apply constraints on the values for fields, which can be easily violated. So, we need a library that can maintain or model the structure of documents and impose restrictions on the values for fields.

You can observe that the structure of the database has been violated in the above example. In order to ensure that the structure of the database is maintained, we need an **Object Data Modeler**, which ensures the maintenance of the collection.

ODM

- An ODM is used to map the data as JavaScript objects to the underlying database format. They provide an option to perform validation and checking data to ensure data integrity.
- There are many ODM based libraries available on NPM. One such library is **Mongoose**.

Mongoose

Mongoose is an Object Data Modeling (ODM) tool, which is created to work on an asynchronous MongoDB environment. It imposes the structure and constraints by mapping JavaScript objects to documents in the database. It also provides the functionality for validating field values and methods for selecting, adding, modifying, and deleting documents in the collection.

If we insert the same employee record with the help of Mongoose, then the collection will be looking as shown below:

empName	empId	Location
Alex	123456	Europe
Sam	432567	California
John		France
empName	empId	Location
Alex	123456	Europe
Sam	432567	California
John		France

Here, we can observe that the field which did not match the name in the collection will not be inserted. If we had set a required constraint for the empId field, then the document will not be inserted at all.

Let us now try to insert the below employee record into the collection. Notice here that the below record did not match the value as expected, for the field empId.

```
{ empName : "John", empId : "324123", location : "France" }
```

Without using Mongoose, if you try to insert the above record, the record will be inserted successfully and the collection will be looking as shown below:

empName	empId	Location
Alex	123456	Europe
Sam	432567	California
John	"324123"	France

If we try to insert the above record with the help of Mongoose, then the collection will be looking as shown below:

empName	empId	Location
Alex	123456	Europe
Sam	432567	California
John	324123	France

We can observe that the value for the empId field got type-cast to Number type. This is because Mongoose did a type-cast. If Mongoose is not able to convert the value to the specified type, then the document will not be inserted.

While working with Mongoose we will come across some common terminologies like Schema and Models. Let us begin our understanding of Mongoose ODM by visiting these terms.

Schema and Models

Schema:

The structure of the document to be inserted in the database collection is defined through 'schema' in mongoose. Using schema, we can perform validations, provide default values to the properties, etc.

Example:

```
const empSchema = [
  {
    empId: Number,
    empName: String,
    address: {
      doorNo: String,
      lane: String,
      pincode: Number,
    },
  },
];
```

Models:

Through schema, we can define the structure of the document. Now, we need to interact with the database collection to store or retrieve data. In order to perform database interaction, mongoose provides a 'Model' method that takes the predefined schema and creates an instance of a document that is like the records of the relational database. It provides an interface to the database for creating, querying, updating, deleting records, etc. A Mongoose model is a wrapper on the Mongoose schema.

Creating a Mongoose model comprises primarily of three parts:

- Referencing Mongoose
- Defining the Schema
- Creating a Model

Let us look at each of these parts in detail, but first, let us begin by learning to install the mongoose library.

Referencing Mongoose

1. Installing Mongoose:

For installing the Mongoose library, issue the below command in the Node command prompt.

```
npm install --save mongoose
```

Executing the above command downloads the Mongoose library and add a dependency entry in your 'package.json' file.

2. Connecting to MongoDB using Mongoose:

To establish a connection to the MongoDB database, we need to create a connection object using Mongoose. This is done through the below code:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/Database_Name');
```

In Line 1, we are importing the Mongoose library into our application.

In Line 2, the connect() method takes MongoDB protocol followed by the host and the database name as a parameter.

Let us now establish a connection to the MongoDB database IntellectNotes for storing details using the below code:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/IntellectNotes', {
  useNewUrlParser: true,
  useCreateIndex: true,
  useFindAndModify: false,
  useUnifiedTopology: true,
})
```

To create a collection using mongoose, we must start by creating a Mongoose schema. Let's understand about Mongoose schema, next.

Introduction to Mongoose Schema

A schema defines document properties through an object, where the key name corresponds to the property name in the collection. The schema allows you to define the fields stored in each document along with their validation requirements and default values.

To create a schema, we need to use the following lines of code:

```
const mongoose = require('mongoose');
const schema = new mongoose.Schema({ property_1: Number, property_2: String });
```

In a schema, we will define the data types of the properties in the document.

The most commonly used types in the schema are:

- String

To store string values. For e.g: employee name

- Number

To store numerical values. For e.g: employee Id

- Date

To store dates. The date will be stored in the ISO date format. For e.g: 2018-11-15T15:22:00.

- Boolean

It will take the values true or false and is generally used for performing validations. We will discuss validations in the next resource.

- ObjectId

Usually, ObjectId is assigned by MongoDB itself. Every document inserted in MongoDB will have a unique id which is created automatically by MongoDB and this is of type ObjectId.

- Array

To store an array of data, or even a sub-document array. For e.g: ["Cricket","Football"]

Creating Schema for myNotes collection – Demo

- **myNotes** wants to store notes id, name, and data for multiple users.
- The code to create the required schema is given below:

```
const myNotesSchema = new mongoose.Schema(  
  {  
    notesID: {  
      type: Number,  
    },  
    name: {  
      type: String,  
    },  
    data: {  
      type: String,  
    },  
  },  
  {  
    timestamps: {  
      createdAt: true,  
      updatedAt: true,  
    },  
  }  
);
```

- Line 1: Using the Schema class provided by the Mongoose library, we can create the schema with all the necessary fields.
- **Note:** It is a standard practice to have timestamps - (created at & updated at) field for each document inserted into the collection. This can be done by adding a timestamp option to the Schema.
- To ensure data entered in the collection is as per the requirement, we can configure validations to the schema. Let us now look at adding validations for our schema.

Validation types and Defaults:

Validation through mongoose validator

Mongoose provides a way to validate data before you save that data to a database. Data validation is important to make sure that "**invalid**" data does not get persisted in your application. This ensures data integrity. A benefit of using Mongoose, when inserting data into MongoDB is its built-in support for data types, and the automatic validation of data when it is persisted.

Mongoose's validators are easy to configure. When defining the schema, specific validations need to be configured.

The following rules should be kept in mind while adding validation:

- Validations are defined in the Schema
- Validation occurs when a document attempts to be saved
- Validation will not be applied for default values
- Validators will not be triggered on undefined values. The only exception to this is the required validator
- Customized validators can be configured
- Mongoose has several built-in validators
- The **required** validator can be added to all SchemaTypes
- Number schema type has **min** and **max** validators
- Strings have **enum** and **match** validators

Let us see how to add a validator to a Schema.

Validation types and default values

Required

If you want any field to be mandatory, use the required property.

```
const schema = mongoose.Schema({
  name: {
    required: true,
  },
});
```

Types

You can declare a schema type using the type directly, or an object with a type property.

Below code, snippet shows how to declare schema type using an object with a type property.

```
const schema = mongoose.Schema({
  property_1: {
    type: String,
  },
  property_2: {
    type: Number,
  },
});
```

Default

- Your schema can define default values for certain properties. If you create a new document without that property set, the default value provided to that property will be assigned.
- The below code snippet shows how to add a default to schema type:

```
const schema = mongoose.Schema({
  property_1: {
    default: "Client"
  },
  property_2: {
    default: 1
  },
});
```

Custom validations

In Mongoose we can specify custom validators that are tailored specifically to fields if the built-in validators are not enough. They are provided as values of validate property.

The below code snippet shows how to add a custom validator to the schema type.

```
const schema = mongoose.Schema({
  property_1: {
    validate: (value) => {
      /*Validation code*/
    }
  },
});
```

For more information on validators, please visit, [mongoose official site](#).

Demo: Adding Validation, types and default to myNotes Schema

Let us now add validation and specify types for myNotes Schema.

```
const myNotesSchema = new mongoose.Schema(
{
  notesID: {
    type: Number,
    unique: true,
    required: [true, 'Required field'],
  },
  name: {
    type: String,
    required: [true, 'Required field'],
  },
  data: {
    type: String,
  },
},
{
  {
```

```
timestamps: {  
  createdAt: true,  
  updatedAt: true,  
},  
}  
);
```

Now that our myNotesSchema is ready, let us create a model for the schema.

Creating a model

- To use our schema definition, we need to wrap the **Schema** into a **Model** object we can work with.
- The model provides an object which provides access to query documents in a named collection. Schemas are compiled into models using the **model()** method.

```
const Model = mongoose.model(name , schema)
```

- The first argument is the singular name of the collection for which you are creating a Model.
- The model() function makes a copy of the schema. Make sure that you have added everything you want to schema before calling the model().
- Let us now create a model for our **myNotes** collection using the below code.

```
const NotesModel = mongoose.model("mynotes", myNotesSchema);
```

- Here **NotesModel** is a collection object that will be used to perform CRUD operations.
- Now we have created the model for our database collection, let us insert some data into the collection.

CRUD Operations

Create:

Insert Document – Demo

Mongoose library offers several functions to perform various CRUD (Create-Read-Update-Delete) operations.

Inserting a document into the collection

To insert a single document to MongoDB, use the create() method. It will take the document instance as a parameter. Insertion happens asynchronously and any operations dependent on the inserted document must happen by unwrapping the promise response.

We can insert a new document into our myNotes Collection using the below code:

```
exports.newNotes = async (req, res) => {
```



```
try {
  const noteObj = {
    notesID: 7558,
    name: 'Mathan',
    data: 'Mongo Atlas is very easy to configure and use.',
  };
  const newNotes = await NotesModel.create(noteObj);
  console.log(newNotes);
} catch (err) {
  console.log(err.errmsg);
}
};
```

- In line 1, a new async function newNotes is created with request and response objects as parameters.
- In line 3, a new object is created with the name noteObj with all the necessary keys that need to be inserted.
- In line 8, we are referring to the NotesModel which we had created previously, and then making use of the create() method to insert the document into the collection. This will insert the documents into the collection based on the schema and will return the documents which got inserted to the const newNotes.
- Try catch block will ensure all the exceptions are handled in the code.
- In line 9, after successful insertion, you will get the following output in the console:

```
{
  _id: 5edf6fc802e4ce1fd2726360,
  notesID: 7558,
  name: 'Mathan',
  data: 'Mongo Atlas is very easy to configure and use.',
  createdAt: 2020-06-09T11:17:28.666Z,
  updatedAt: 2020-06-09T11:17:28.666Z,
  __v: 0
}
```

Insert multiple documents – Demo

In order to insert multiple documents into a collection, we can use any one of the following two methods: insert() or insertMany(). The syntax for inserting multiple documents into a collection is:

```
Model.insert([document1, document2]);
or
Model.insertMany([document1, document2]);
```

Let us now insert more documents into our myNotes collection using the below code:

```
exports.newNotes = async (req, res) => {
  try {
    const noteObj = [
      {
```

```
notesID: 7555,
name: 'Mathan',
data:
  'This includes macOS package notarization and a change in configuration... ',
},
{
  notesID: 7556,
  name: 'Sam',
  data: 'AMD processor support in android studio.',
},
{
  notesID: 7557,
  name: 'Mathan',
  data: 'MERN',
},
];
const newNotes = await NotesModel.create(noteObj);
console.log(newNotes);
} catch (err) {
  console.log(err.errmsg);
}
};
```

- In line 1, a new async function newNotes is created with request and response objects as parameters.
- In line 3, a new array of objects is created with the name noteObj with all the necessary keys that need to be inserted.
- In line 16, we are referring to the same NotesModel which we had created earlier. We can now make use of the create() method to insert the documents into the collection. It will insert the documents into the collection based on the schema and will return the documents which got inserted to the const newNotes.
- Try catch block will ensure all the exceptions are handled in the code.
- In line 17, after successful insertion, you will get the following output in the console:

```
[
  {
    _id: 5edf706627f169388e6cc8ae,
    notesID: 7555,
    name: 'Mathan',
    data: 'This includes macOS package notarization and a change in configuration...',
    createdAt: 2020-06-09T11:20:06.871Z,
    updatedAt: 2020-06-09T11:20:06.871Z,
    __v: 0
  },
  {
    _id: 5edf706627f169388e6cc8af,
    notesID: 7556,
    name: 'Sam',
    data: 'AMD processor support in android studio.',
    createdAt: 2020-06-09T11:20:06.872Z,
    updatedAt: 2020-06-09T11:20:06.872Z,
    __v: 0
  },
  {
    _id: 5edf706627f169388e6cc8b0,
    notesID: 7557,
    name: 'Mathan',
    data: 'MERN',
    createdAt: 2020-06-09T11:20:06.872Z,
    updatedAt: 2020-06-09T11:20:06.872Z,
    __v: 0
  }
]
```

```
[
  {
    _id: 5edf706627f169388e6cc8ae,
    notesID: 7555,
    name: 'Mathan',
    data: 'This includes macOS package notarization and a change in configuration...',
    createdAt: 2020-06-09T11:20:06.871Z,
    updatedAt: 2020-06-09T11:20:06.871Z,
    __v: 0
  },
  {
    _id: 5edf706627f169388e6cc8af,
    notesID: 7556,
    name: 'Sam',
    data: 'AMD processor support in android studio.',
    createdAt: 2020-06-09T11:20:06.872Z,
    updatedAt: 2020-06-09T11:20:06.872Z,
    __v: 0
  },
  {
    _id: 5edf706627f169388e6cc8b0,
    notesID: 7557,
    name: 'Mathan',
    data: 'MERN',
    createdAt: 2020-06-09T11:20:06.872Z,
    updatedAt: 2020-06-09T11:20:06.872Z,
    __v: 0
  }
]
```

Read document(s)

Documents can be retrieved through find, findOne, and findById methods.

Let us now retrieve all the documents that we have inserted into our myNotes Collection.

```
exports.getNotes = async (req, res) => {
  try {
    const notes = await NotesModel.find({}, { _id: 0, __v: 0 });
    if (notes.length > 0) {
      console.log(notes);
    }
  } catch (err) {
    console.log(err.errmsg);
  }
}
```

```
};
```

- In line 1, a new async function getNotes is created with request and response objects as parameters.
- In line 3, we are referring to the same NotesModel which we had created earlier and made use of the find() method to fetch the document(s) from the collection. The retrieved documents will be available in the const notes.
- In line 4, we are ensuring the data by checking the length and displaying the returned data in the console.
- Try catch block will ensure all the exceptions are handled in the code.

On successful retrieval of the documents, you will get the following output in the console:

```
[
  {
    notesID: 7558,
    name: 'Mathan',
    data: 'Mongo Atlas is very easy to configure and use.',
    createdAt: 2020-06-09T11:17:28.666Z,
    updatedAt: 2020-06-09T11:17:28.666Z
  },
  {
    notesID: 7555,
    name: 'Mathan',
    data: 'This includes macOS package notarization and a change in configuration...',
    createdAt: 2020-06-09T11:20:06.871Z,
    updatedAt: 2020-06-09T11:20:06.871Z
  },
  {
    notesID: 7556,
    name: 'Sam',
    data: 'AMD processor support in android studio.',
    createdAt: 2020-06-09T11:20:06.872Z,
    updatedAt: 2020-06-09T11:20:06.872Z
  },
  {
    notesID: 7557,
    name: 'Mathan',
    data: 'MERN',
    createdAt: 2020-06-09T11:20:06.872Z,
    updatedAt: 2020-06-09T11:20:06.872Z
  }
]
```

Retrieving data based on the condition

Let us try to retrieve notes details based on notesID from the myNotes Collection using the below code:

```
exports.getNotes = async (req, res) => {
  try {
    const notes = await NotesModel.find({ notesID: 7555 }, { _id: 0, __v: 0 });
    if (notes.length > 0) {
      console.log(notes);
    }
  } catch (err) {
    console.log(err.errmsg);
  }
};
```

- In line 1, a new async function getNotes is created with request and response objects as parameters.
- In line 3, we are referring to the same NotesModel which we had created earlier and made use of the find() method to fetch the document(s) from the collection by providing the condition(s) based on which documents should be retrieved. The retrieved documents will be available in the const notes.
- In line 4, we are ensuring the data by checking the length and displaying the returned data in the console.
- Try catch block will ensure all the exceptions are handled in the code.

On successful retrieving of the document, you will get the following output in the console:

```
[
  {
    notesID: 7555,
    name: 'Mathan',
    data: 'This includes macOS package notarization and a change in configuration...',
    createdAt: 2020-06-09T11:20:06.871Z,
    updatedAt: 2020-06-09T11:20:06.871Z
  }
]
```

Update document(s)

We will update the details of the myNotes collection using the below code:

```
exports.updateNotes = async (req, res) => {
  try {
    const noteObj = {
      name: 'Mathan',
      data: 'Updated notes',
    };

    const notes = await NotesModel.findOneAndUpdate(
      { notesID: 7555 },
      noteObj,
      {
        new: true, //to return new doc back
        runValidators: true, //to run the validators which specified in the model
      }
    );
    if (notes !== null) {
      console.log(notes);
    }
  } catch (err) {
    console.log(err.errmsg);
  }
};
```

- In line 1, a new async function updateNotes is created with request and response objects as parameters.
- In line 3, a new array of objects is created with the name noteObj with all the necessary keys that need to be updated.
- In line 8, we are referring to the same NotesModel which we had created earlier and made use of the **findOneAndUpdate()** method to find and update one document in the collection. It will update the document in the collection based on the schema and will return the document which got updated to the const notes.

The parameters of findOneAndUpdate() method are:

- The condition based on which the document should be updated.
- The new values to be updated.
- Additional options to return the updated document and to validate the values based on the schema definition.

Try catch block will ensure all the exceptions are handled in the code.

In line 18. On successful updating of the document, you will get the following output in the console.

```
{
  _id: 5edf706627f169388e6cc8ae,
  notesID: 7555,
  name: 'Mathan',
  data: 'Updated notes',
  createdAt: 2020-06-09T11:20:06.871Z,
  updatedAt: 2020-06-09T11:26:34.289Z,
  __v: 0
}
```

Delete document(s)

The below code shows how to delete the contents of myNotes collection:

```
exports.deleteNotes = async (req, res, err) => {
  const delDet = await NotesModel.deleteOne({ notesID: 7555 });
  console.log(delDet);
};
```

- In line 1, a new async function deleteNotes is created with the request, response, and error objects as parameters.
- In line 2, we are referring to the same NotesModel which we had created earlier and made use of **deleteOne()** method to delete the document from the collection based on the condition. It will delete the document from the collection based on the condition and will return the details about the delete operation performed, which is initialized to the const delDet.
- In line 3, on the successful deletion of the document, you will get the following output in the console.

```
{ n: 1, ok: 1, deletedCount: 1 }
```

API Development: Introduction

We are going to learn how to create a set of APIs with all CRUD operations.

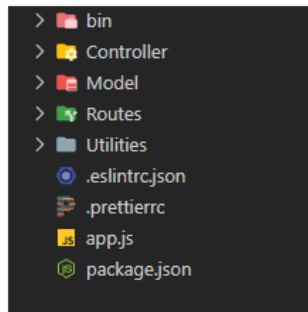
This project covers the following:

- Create a web server

- Routing
- Middleware
- Database connectivity to MongoDB using mongoose

Let's begin with creating a project in the express.

A typical folder structure with all the necessary folders will look like the below image:



The following describes the folder structure of the project:

- Controller – Business logic and Database operations of the application.
- Model – MongoDB schema and model.
- Routes - All the routes that are created for the application.
- Utilities – Helper files for the application. E.g. Custom validators, loggers, etc.
- app.js – The starting point of the application.

app.js file

Application execution will start from the app.js file. All the necessary imports and middlewares need to be configured in this file.

In the app.js, include the following code:

```
const express = require('express');
const bodyparser = require('body-parser');
const myReqLogger = require('./Utilities/requestLogger');
const route = require('./Routes/routing');
const app = express();
app.use(bodyparser.json());
app.use(myReqLogger);
app.use('/', route);
const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}...`);
});
```

API Routes

- In the Routes folder, create a file with the name "**routing.js**".
- This file contains route definitions for all the APIs of the application.

```
const express = require('express');
const routing = express.Router();
const notesController = require('../Controller/myNotes');

routing.get('/notes', notesController.getNotes);

routing.post('/notes', notesController.newNotes);

routing.put('/notes/:id', notesController.updateNotes);

routing.delete('/notes/:id', notesController.deleteNotes);

routing.all('*', notesController.invalid);

module.exports = routing;
```

Model

- In the Model folder, create a file with the name "**myNotesSchema.js**".
- Logic to connect to MongoDB using mongoose, schema, and the model creation logic will be included in this file.

```
const mongoose = require('mongoose');
mongoose
  .connect('mongodb://localhost:27017/IntellectNotes', {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
    useUnifiedTopology: true,
  })
  .then(() => console.log('DB connection successful!'));

//Schema
const myNotesSchema = new mongoose.Schema(
  {
    notesID: {
      type: Number,
      unique: true,
      required: [true, 'Required field'],
    },
    name: {
      type: String,
      required: [true, 'Required field'],
    },
  },
```



```
data: {
  type: String,
},
},
{
  timestamps: {
    createdAt: true,
    updatedAt: true,
  },
}
);

//Model
const NotesModel = mongoose.model('mynotes', myNotesSchema);

module.exports = NotesModel;
```

Controllers

- In the Controller folder, create a file with the name "**myNotes.js**".
- This file contains business logic for all the APIs of the application.
- Custom validators will be imported and accessed based on the requirement.
- All the controllers will send the response back to the client based on the request received. Error handling also is taken care of accordingly.

```
const NotesModel = require('../Model/myNotesSchema');
const validators = require('../Utilities/validator');

exports.getNotes = async (req, res) => {
  try {
    const notes = await NotesModel.find({}, { _id: 0, __v: 0 });
    if (notes.length > 0) {
      res.status(200).json({
        status: 'success',
        results: notes.length,
        data: {
          notes,
        },
      });
    } else {
      res.status(400).json({
        status: 'success',
        data: {
          message: 'No notes available in the repo',
        },
      });
    }
  } catch (error) {
    res.status(500).json({
      status: 'error',
      message: error.message,
    });
  }
};
```

```
    },
  });
}
} catch (err) {
  res.status(404).json({
    status: 'fail',
    message: err,
  });
}
};

exports.newNotes = async (req, res) => {
  try {
    if (validators.ValidateName(req.body.name)) {
      const newNotes = await NotesModel.create(req.body);
      res.status(201).json({
        status: 'success',
        data: {
          newNotes,
        },
      });
    } else {
      res.status(400).json({
        status: 'error',
        results: 'Enter valid name',
      });
    }
  } catch (err) {
    res.status(404).json({
      status: 'fail',
      message: err.errmsg,
    });
  }
};

exports.updateNotes = async (req, res) => {
  try {
    const notes = await NotesModel.findOneAndUpdate(
      { notesID: req.params.id },
      req.body,
      {
        new: true, //to return new doc back
      }
    );
  } catch (err) {
    res.status(404).json({
      status: 'fail',
      message: err.errmsg,
    });
  }
};
```

```
    runValidators: true, //to run the validators which specified in the model
  }
);
if (notes !== null) {
  res.status(200).json({
    status: 'success',
    data: {
      notes,
    },
  });
} else {
  res.status(400).json({
    status: 'success',
    data: {
      message: `No notes available with ID ${req.params.id} `,
    },
  });
}
} catch (err) {
  res.status(404).json({
    status: 'fail',
    message: err,
  });
}
};

exports.deleteNotes = async (req, res) => {
  const delDet = await NotesModel.deleteOne({ notesID: req.params.id });
  if (delDet.deletedCount === 0) {
    res.status(404).json({
      status: 'fail',
      message: 'No notes available for this ID',
    });
  } else {
    res.status(200).json({
      status: 'success',
      message: `Notes with ${req.params.id} ID deleted`,
    });
  }
};

exports.invalid = async (req, res) => {
  res.status(404).json({
```

```

status: 'fail',
message: 'Invalid path',
});
};

```

Utilities

- In the application, we will need specific helper files. For instance, we need to have a logger that should keep track of all the requests that comes into the application or some custom validators which we can make use in the controller.
- Inside the Utilities folder create the following files with the content.

requestLogger.js

```

const fs = require('fs');
const { promisify } = require('util');
const appendFile = promisify(fs.appendFile);
async function requestLogger(req, res, next) {
  try {
    const logMessage = `${new Date()} - ${req.method} - ${req.url} \n`;
    await appendFile('RequestLogger.log', logMessage);
    next();
  } catch (err) {
    next(err);
  }
}
module.exports = requestLogger;

```

validator.js

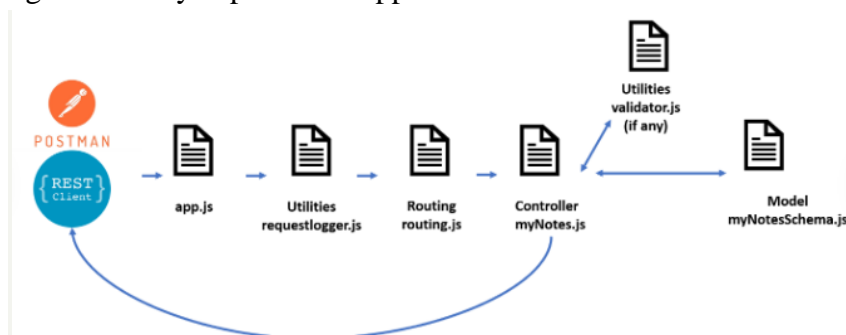
```

exports.ValidateName = function (name) {
  if (name.trim().length > 0) {
    return true;
  }
  return false;
};

```

Project flow

The following figure visually explains the application data flow.



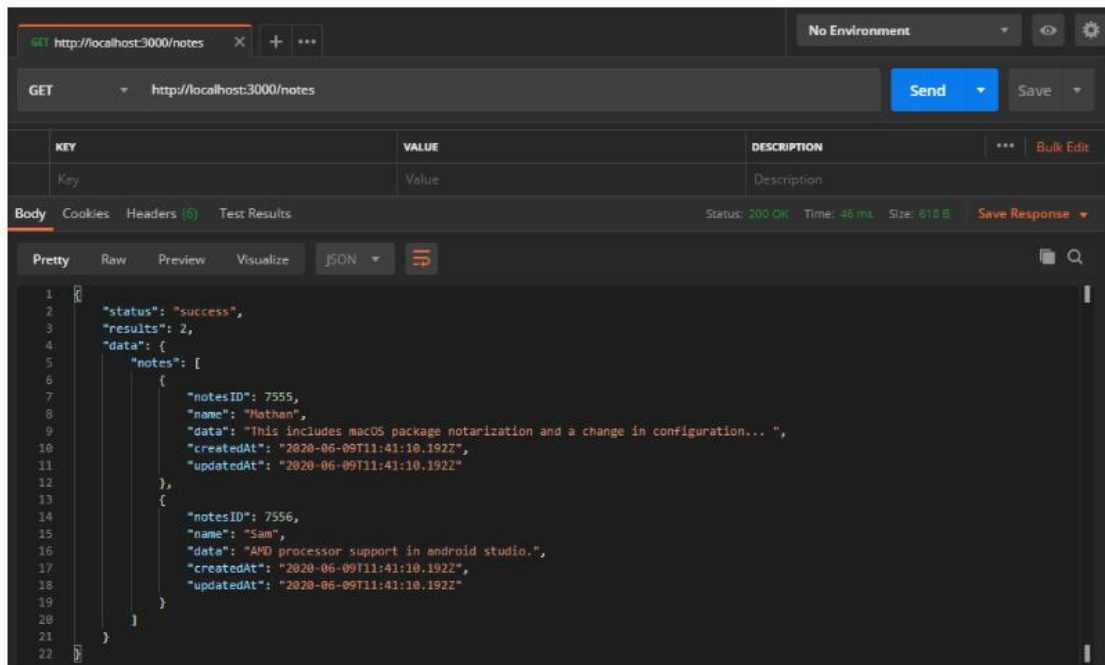
Project Execution

API – 1

URL – <http://localhost:3000/notes>

Method - GET

Description – API to fetch all the notes.

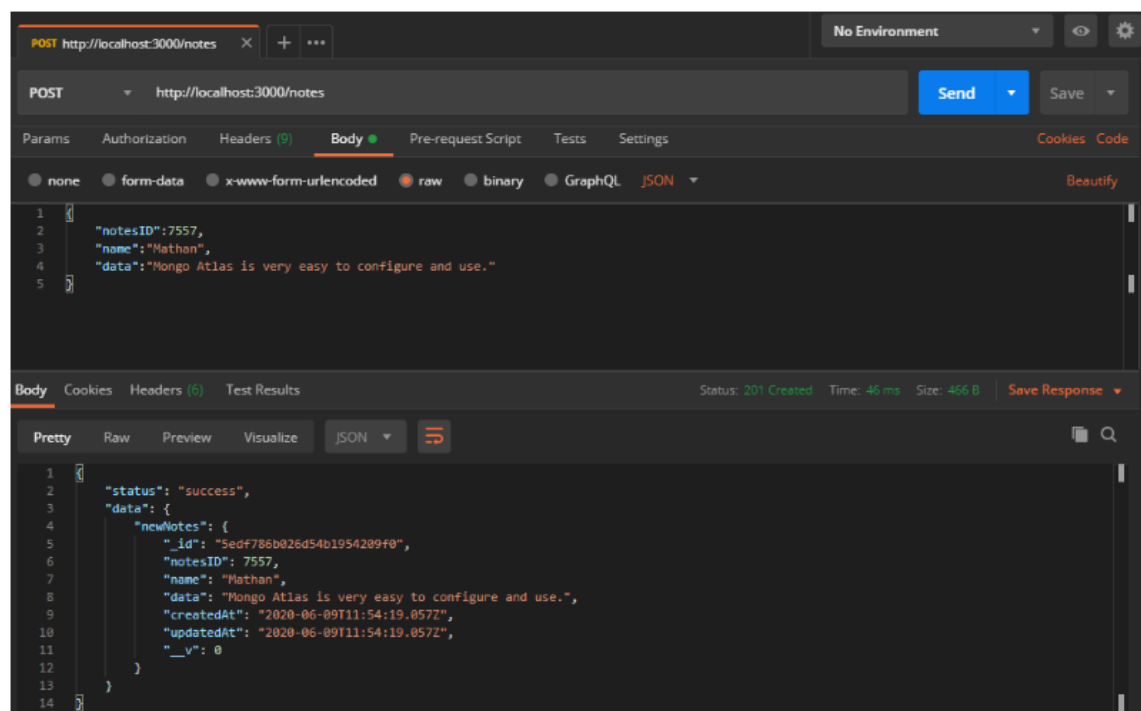


API – 2

URL – <http://localhost:3000/notes>

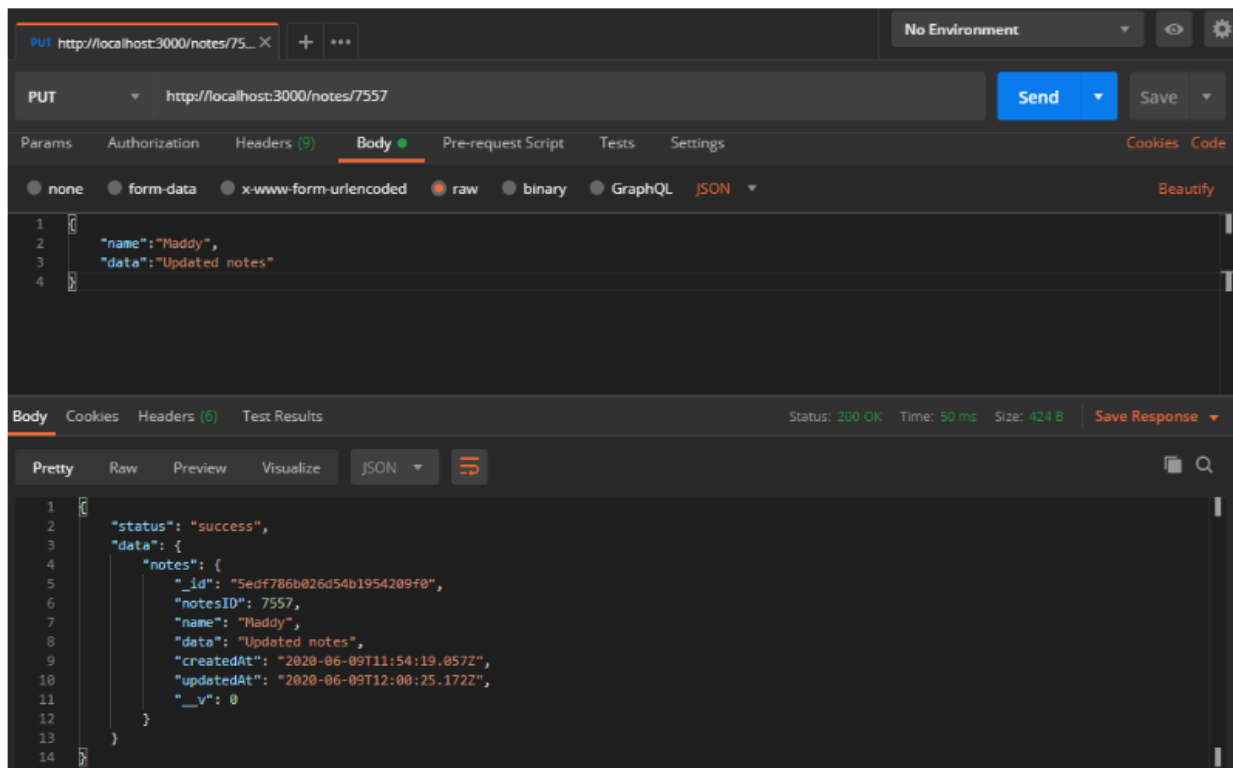
Method – POST

Description – API to add new notes.



API – 3

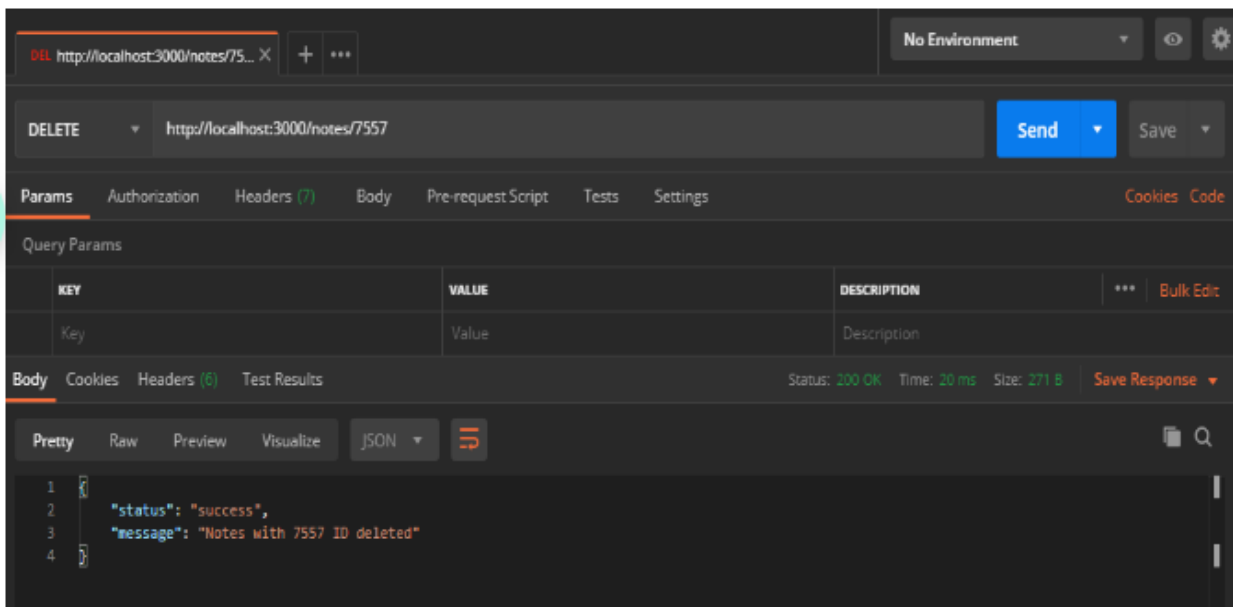
URL – <http://localhost:3000/notes/7558> **Method** - PUT **Description** – API to update existing notes.

**API – 4**

URL – <http://localhost:3000/notes/7558>

Method - DELETE

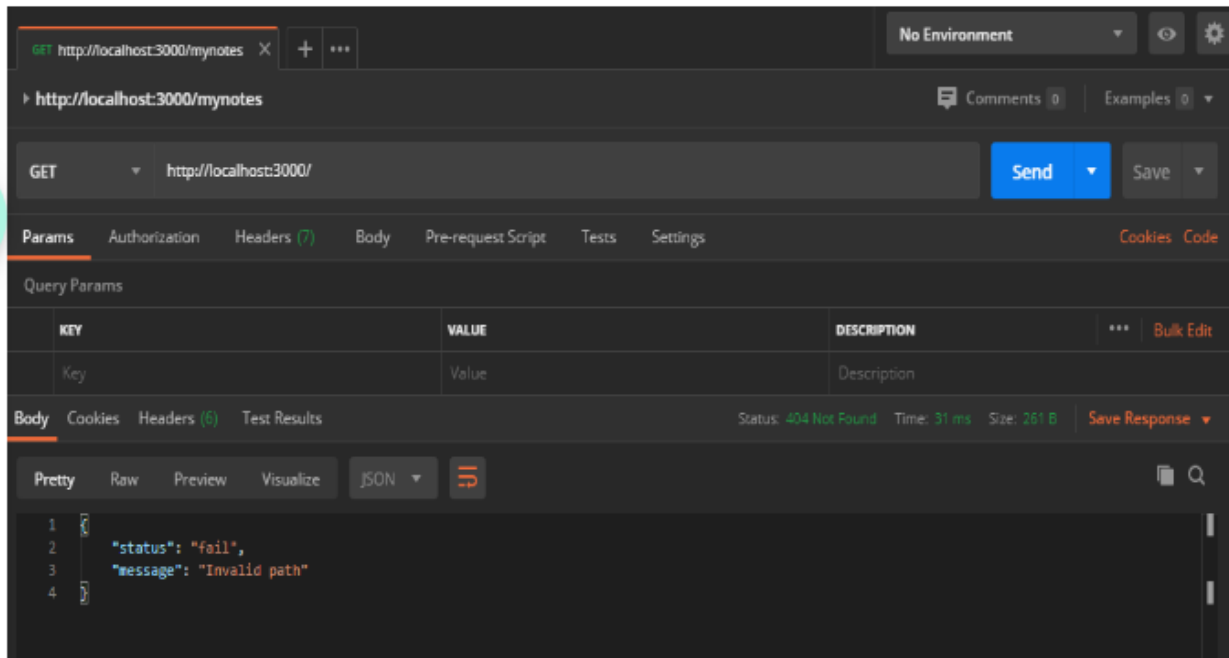
Description – API to delete existing notes.

**API – 5**

URL – <http://localhost:3000/mynotes>

Method - ALL

Description – default API to handle invalid routes.



Why Session management?

Every user interaction with an application is an individual request and response. The need to persist information between requests is important for maintaining the ultimate experience for the user for any web application.

Session management is useful in the scenarios mentioned below:

- We need to maintain that a user has already been authenticated with the application.
- To retain various personalized user information that is associated with a session.

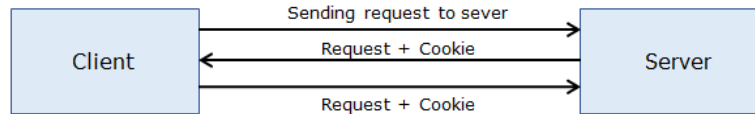
Let us now understand how we can set up sessions securely in our application to lessen risks like session hijacking.

Session Management is a technique used by the webserver to store a particular user's session information. The Express framework provides a consistent interface to work with the session-related data. The framework provides two ways of implementing sessions:

- By using cookies
- By using the session store

Introduction to cookies

Cookies are a piece of information sent from a website server and stored in the user's web browser when the user browses that website. Every time the user loads that website back, the browser sends that stored data back to the website server, to recognize the user.



Configuration

- The Express framework provides a **cookie API** using **cookieParser** middleware. The middleware **cookieParser** parses the cookies which are attached to the request object.
- To install the cookieParser middleware, issue the below command in the Node command prompt.

```
npm install cookie-parser
```

The above middleware can be configured in Express application as shown below:

```
const express = require('express');
const cookieParser = require('cookie-parser');
const app = express();
app.use(cookieParser());
```

The cookie-parser middleware is imported and then associated with the application object using the `app.use()` method.

Setting cookies:

The below code shows how to set a cookie:

```
routing.get('/user/:name', notesController.user1);
exports.user1 = async (req, res) => {
  res.cookie('name', req.params.name);
  res.send('<p>Cookie set:<a href="/user"> View here </a>');
};
```

When the user navigates to URL `"/user/<username>"`, the cookie is set with the name **username** and the value is retrieved using the **params** property of the request object. Thus for setting cookies, the general syntax is as shown below:

```
res.cookie('cookieName', value, { expires: new Date(), maxAge: 99999 });
```

The arguments of the above method are

- the name of the cookie
- cookie value
- the options to be used while configuring cookie, an optional object.

Some of the options that can be specified while configuring a cookie are shown below:

Options	Description
path	The path for the cookie.
expires	The expiry date of the cookie (GMT).
maxAge	Expiry time of the cookie relative to the current time (milliseconds).

Reading cookies:

To access a particular cookie, use the **cookies** property of the **request** object.

```
exports.user2 = async (req, res) => {  
  res.send(req.cookies.name);  
};
```

Whenever the user visits the `"/user"` route, the value of the cookie is fetched using **request.cookies** property and displayed to the user.

Updating cookies:

A cookie can be updated by re-creating it with new properties. For example, if we need to update a cookie named **username**:

```
res.cookie('username', new_value)
```

Deleting cookies:

It is possible to remove a cookie with the response object's **clearCookie()** method. This method accepts the name of the cookie which we want to delete. We can also delete the cookies from the browser developer's tools.

```
app.get('/user', myController.myMethod);  
  
exports.myMethod = async (req, res) => {  
  res.clearCookie('username');  
  res.send(req.cookies.username);  
};
```

Types of cookies:**1. Session cookies**

- The cookies which exist for a browser session are defined as Session cookies and they are rejected whenever the browser is closed.
- While configuring a cookie, if the 'expires' option is not specified with any value, then a session cookie is created by default.

```
res.cookie('name', 'John')
```

A session cookie is also created on setting value as `"0"` for the `"expires"` option.

```
res.cookie('name', 'John', { expires: 0 })
```

2. Signed cookies

- The cookies which have a signature attached to its value are considered as Signed cookies. To generate this signature, any secret string value can be used while adding the cookie middleware into the application object.
- A signed cookie can be created as shown below, by passing a string value while instantiating it:

```
app.use(cookieParser('S3CRE7'))
```

To access the signed cookies, use the property called **signedCookies** of the **request** object. For example, if we need to access the value of a signed cookie:

```
request.signedCookies.username
```

Session cookies are deleted on closing the browser. The other cookies are deleted on reaching the expiry date.

Demo of Using Cookies:

1. Modify the app.js file in TestApp as shown below:

```
const express = require('express');
const cookieParser = require('cookie-parser');
const router = require('./Routes/routing');

const app = express();
app.use(cookieParser());
app.use('/', router);
app.listen(3000);
console.log('Server listening in port 3000');
```

2. Modify the routing.js file in TestApp as shown below:

```
const express = require('express');
const myController = require('./Controller/myController');
const router = express.Router();

router.get('/user/:name', myController.user1);
router.get('/user', myController.user2);

module.exports = router;
```

3. In Controller, add the below code:

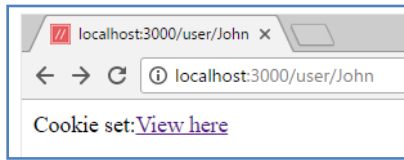
```
exports.user1 = async (req, res) => {
  res.cookie('name', req.params.name);
  res.send('<p>Cookie set:<a href="/user"> View here </a>');
};

exports.user2 = async (req, res) => {
  res.send(req.cookies.name);
};
```

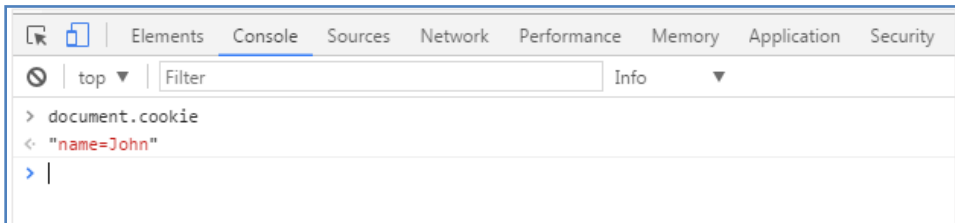
4. Save the files. Start the server.

```
D:\Demos\TestApp>node app
Server listening in port 3000
```

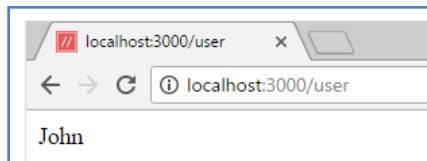
4. Access the URL "http://localhost:3000/user/<username>" and observe the output.



5. Open the developer tools in the browser window and click on the Console tab and type **document.cookie** and verify the value displayed.



6. On click of the "View here" link, the cookie is retrieved and displayed as below:



7. Open the developer tools in the browser window and click on the Console tab and type **document.cookie** and verify the value displayed.

Sessions:

Session store for session management

A session store can be used to store the session data in the back-end. Using this approach, a large amount of data can be stored, and also the data will be hidden from the user, unlike a cookie.

Configuration

Express provides a middleware called **express-session** for storing session data on the server-side.

To install **express-session** middleware, use npm.

```
npm install express-session
```

To use this module in the Express application, first, import it as shown below:

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

Then the session is to be initialized as follows:

```
app.use(session());
```

The session middleware also accepts an object '**options**' for defining different configuration options.

```
app.use(session({
  secret: 'keyboard',
  resave: false,
  saveUninitialized: true,
  cookie: { secure: true }
}));
```

- **secret** - used for signing
- **resave** - a boolean value which makes the session to be saved when not modified as well
- **saveUninitialized** - a boolean value which makes the session which is uninitialized to be saved to the store
- **cookie** - configurations for a session cookie

The built-in session store called MemoryStore is used in Express by default. But it is not suggested for use in a production environment due to memory leakage.

Express also supports the usage of any third-party module like RedisStore which stores session data in Redis, which is an in-memory data store.

Session Variables

- Session variables are the variables that are associated with a user's session.
- Consider the example code below which uses a session variable called **page_views**.

```
exports.myMethod = async (request, response) => {  
  if (request.session.views) {  
    request.session.views++;  
    response.send(`You visited this page ${request.session.views} times`);  
  } else {  
    request.session.views = 1;  
    response.send('Welcome to this page for the first time!');  
  }  
};
```

- According to the above code snippet, a session variable called **views** is created when the user visits the website. The session variable is updated accordingly, whenever the user comes next time.
- The session variables are set independently for each user and it can be accessed using the **session** property of the request object as shown below:

```
request.session
```

Setting session variables

- A session variable can be set by attaching it to the request object either using the dot notation or the substring syntax.

```
request.session.username = 'John';  
request.session['username'] = 'John';
```

Reading session variables

- To read a session variable, use the request object as shown below:

```
const username = request.session.username  
const username = request.session['username']
```

Updating session variables

- A session variable can be updated using the req.session object or use a new value to overwrite the existing one.

```
request.session.username.push('value');  
request.session.username = 'John';
```

Demo- Using Session store

1. Open the TestApp folder in the command prompt and run the command "npm install" for installing the dependencies mentioned in the package.json file.
2. In the file server-session.js, use the below code:

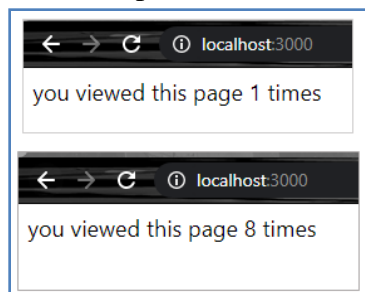
```
const express = require('express');  
  
const app = express();  
const session = require('express-session');  
  
app.use(  
  session({  
    name: 'my-session',  
    cookie: {  
      //Current Server time + maxAge = expire datetime  
      maxAge: 1000 * 60,  
  
      //Allows for the cookie to be read only by a server (no JS)  
      httpOnly: true,  
  
      //Set for the root of the domain.  
      path: '/',  
    },  
    secret: 'mypasswordinfy',  
  })  
);  
  
app.use((req, res, next) => {  
  let { visit } = req.session;  
  
  if (!visit) {  
    visit = req.session.visit = {  
      count: 1,  
    };  
  } else {  
    visit.count++;  
  }  
});
```

```
next();
});
app.get('/', (req, res) => {
  res.send(`you viewed this page ${req.session.visit.count} times`);
});
app.listen(3000);
console.log('Server started...running with port 3000');
```

2. Start the Server.

```
D:\demos\TestApp>node server-session.js
Server started...running with port 3000
```

3. Observe the below output:



Securing Express applications

Why Security?

- Attackers may attempt to exploit security vulnerabilities present in web applications in various ways. They may plan attacks like clickjacking, cross-site scripting, placing insecure requests and identifying web application frameworks etc.
- It is important to follow secure coding techniques for developing secure, robust and hack-resilient applications.

What is Security?

Security is basically about protecting the application assets like the web page, the database, and so on.

Security depends on the following elements:

- **Authentication:** the process of exclusively identifying the clients of your applications and services.
- **Authorization:** a process that manages the resources and operations that the authenticated user is permitted to access.
- **Auditing:** process to assure that a user cannot deny performing an operation or initiating a transaction.
- **Confidentiality:** the process of making sure that data remains private and confidential.
- **Integrity:** the promise that data is protected from malicious modification.
- **Availability:** means that systems remain available for legitimate users.

Use of Helmet Middleware

- Express applications can be secured by using a middleware called **helmet**.
- The helmet middleware is a set of 14 small middleware functions that help in setting up security-related HTTP headers with default values and also removes unnecessary headers which expose the application related information.
- Below are the set of middleware functions that helmet provides to protect an Express application:

Middleware	Description
Csp	Adds the Content-Security-Policy header which prevents attacks like cross-site scripting.
Crossdomain	Sets X-Permitted-Cross-Domain-Policies header. It prevents Adobe Flash and Adobe Acrobat from loading content on site.
dnsPrefetchControl	Sets X-DNS-Prefetch-Control header.
expectCt	Sets Expect-CT header to handle certificate Transparency.
featurePolicy	Sets Feature-Policy header. It can restrict certain browser features like fullscreen, notifications, etc.
hidePoweredBy	Removes the “X-Powered-By” header which exposes the information that application is run with Express server.
Hpkp	Sets Public Key Pinning headers which prevents attacks like man in the middle (MIM).
Hsts	Sets Strict-Transport-Security header which enforces usage of HTTPS for connecting to the server.
ieNoOpen	Sets the X-Download-Options header which prevents Internet Explorer browser users from downloads.
noCache	Enables Cache-Control, Pragma, Expires, and Surrogate-Control headers which prevents client caching site resources, which are old.
noSniff	Sets X-Content-Type-Options.
Frameguard	Sets the “X-Frame-Options” header which blocks attacks like clickjacking which prevents the web page to be accessed on other sites.
xssFilter	Sets the Cross-site scripting (XSS) filter latest web browsers.
referrerPolicy	Sets Referrer-Policy header.

If you're writing a web application, there are a lot of common best practices that you should follow to secure your application as mentioned in the above list.

Helmet configuration

To install helmet in any Express application, use node package manager and run the below command.

```
npm install helmet
```

Once helmet is installed, it is to be associated with the Express application object.

```
var helmet = require('helmet');
app.use(helmet());
```

Express also provides the option of enabling pieces of helmet middleware individually as shown below.

```
app.use(helmet.noCache());  
app.use(helmet.frameguard());
```

It also provides the functionality of disabling a middleware that's normally enabled by default based on the application requirement.

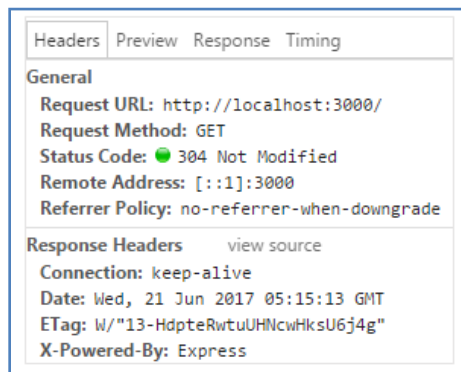
For example, **frameguard** middleware mitigates clickjacking attacks by setting a header 'X-Frame-Options'. For disabling this middleware in the helmet stack, modify the code as shown below.

```
app.use(helmet({  
  frameguard: false  
}));
```

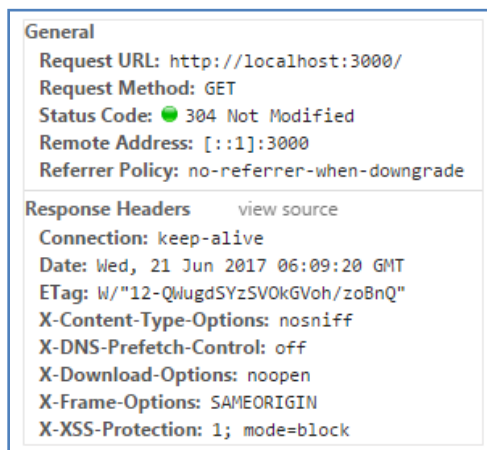
Benefit of using Helmet Middleware

Consider an Express application server code that does not have helmet middleware added.

Once the server is started and when the client sends a request to the server, have a look at the headers generated as part of the response.



- When any request is sent to the server from a client, the server will process the request and sends the response back to the client without any protections that the helmet middleware could have possibly provided, if it was used.
- In the above response headers, we can see a header "X-Powered-By" with the value as "Express". From this, we can understand that the application is running on an Express server. We can also see that there are no clickjacking protections available and no cache control headers are set for the application.
- Now if the helmet middleware was used, here's what the response headers would look like:



We can see that some extra headers get set.

- The **X-Content-Type-Options** header is set to value **nosniff**. This does not allow the browser to guess the MIME type of a file.
- The **X-Download-Options** header is set to value **noopen**. This disables the option to open a file directly on download.
- The **X-Frame-Options** header is set to value **SAMEORIGIN**. This is used to indicate whether a browser should be allowed to render a page within a <frame> or <iframe>, but only from the originating host.
- The **X-XSS-Protection** header is set to value **1; mode=block**. This instructs Internet Explorer to enable the anti-cross-site scripting filter.
- The **X-Powered-By** header is removed so that the attackers cannot see which server is used for running the application.

Stylus CSS Preprocessor

Stylus is a CSS preprocessor that allows you to write CSS in a more concise and flexible way by using indentation-based syntax instead of brackets and semicolons. It offers features like variables, mixins, functions, and nested selectors, making CSS code more maintainable and readable.

To use Stylus in your project, follow these steps:

1. **Installation:** Install Stylus globally using npm:
npm install -g stylus
2. **Create Stylus Files:** Create **.styl** files for your stylesheets.
3. **Write Stylus Code:** Write your styles using Stylus syntax.

```
// styles.styl
$primary-color = #007bff
body
  font-family Arial, sans-serif
  color $primary-color
.container
  width 100%
  max-width 1200px
  margin 0 auto
button
  padding 10px 20px
  background-color $primary-color
  color #fff
  border none
  border-radius 5px

  &:hover
    background-color darken($primary-color, 10%)
```

4. **Compile Stylus Files:** Compile your Stylus files into regular CSS using the **stylus** command:
stylus styles.styl -o dist

5. **Link Compiled CSS:** Link the compiled CSS file (styles.css) in your HTML file:

`<link rel="stylesheet" href="dist/styles.css">`

Now your Stylus styles are compiled into regular CSS and ready to use in your project.

You can also integrate Stylus into your build process using task runners like Gulp or Grunt, or bundlers like Webpack. This allows you to automatically compile Stylus files when changes are made, among other things. Additionally, there are plugins available for various text editors and IDEs that provide syntax highlighting and other helpful features for working with Stylus.

Stylus Watcher:

To set up a watcher with Stylus, you can use the **-w** or **--watch** flag along with the stylus command. This flag tells Stylus to watch for changes in the specified Stylus file(s) and recompile them automatically whenever changes are detected.

1. Set up your project directory:

Create a directory for your Stylus files and CSS output. For example:

```
project/
├── src/
│   └── styles.styl
└── dist/
```

2. Watch and compile Stylus files:

Open your terminal and navigate to the root of your project. Then run the following command:

stylus src/styles.styl -o dist -w

-o dist specifies the output directory for the compiled CSS files.

-w or --watch flag watches for changes in the Stylus files and recompiles them automatically.

Stylus Compress:

Compressing your CSS files will render them difficult to read for humans but reduces the sizes of them. This is useful for when you are working on a website that you plan to make public for others to use. The smaller the files, the less data that needs to be transmitted.

- **-c** or **--compress** flag compresses the output CSS files.
- To compile this Stylus file and compress the output CSS, you can run the following command:

stylus styles.styl -o dist -c

- By compressing it, your styles.css would look like this:

```
body{ font-family:Arial,sans-serif;color:#007bff}.container{ width:100%;max-width:1200px;margin:0 auto}button{padding:10px 20px;background-color:#007bff;color:#fff;border:none;border-radius:5px}button:hover{ background-color:#0056b3}
```

Using template engines with Express

Template engine middleware in Express.js is used to render dynamic content to HTML files by injecting data into predefined templates. There are several popular template engines available for Express.js, such as Handlebars, EJS (Embedded JavaScript), Pug (formerly known as Jade), and Mustache.

To use a template engine middleware in Express.js, you first need to install the desired template engine package. Let's take EJS as an example:

Install EJS:

```
npm install ejs
```

Set Up Express with EJS:

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

// Set EJS as the view engine
app.set('view engine', 'ejs');

// Define a route to render an EJS template
app.get('/', (req, res) => {
  // Render the 'index.ejs' template
  res.render('index', { title: 'Express with EJS' });
});

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Create EJS Templates:

Create EJS templates in a directory named views. For example, create a file named index.ejs in the views directory:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title><%= title %></title>
</head>
<body>
  <h1>Welcome to <%= title %></h1>
</body>
</html>
```

Start the Server: Run your Express.js server:

```
node app.js
```

Now, when you navigate to **http://localhost:3000/**, you should see the rendered HTML page with the dynamic content injected from the Express route.

You can replace EJS with any other template engine by installing the respective package and configuring it similarly. The key is to set the template engine using **app.set('view engine', 'engine-name')** and use **res.render()** to render templates with dynamic data.

Example2: using ‘pug’ engine:**Install pug:**

```
npm install pug --save
```

Set Up Express with pug:

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

// Set EJS as the view engine
app.set('view engine', 'pug');

// Define a route to render an EJS template
app.get('/', (req, res) => {
  res.render('index', { title: 'Hey', message: 'Hello there!' })
})

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Create pug Templates:

Create pug templates in a directory named views. For example, create a file named **index.pug** in the views directory:

```
html
  head
    title= title
  body
    h1= message
```

Start the Server: Run your Express.js server:

```
node app.js
```