Computer Setup

To start developing with the Node.js, you need to have three things on your

computer:

1. A web browser

2. A code editor

3. The Node.js program

Installing Chrome Browser

Any web browser can be used to browse the Internet, but for development you need to have a browser with sufficient development tools.

The Chrome browser developed by Google is a great browser for web development, and if you don't have the browser installed, you can download it here:

https://www.google.com/chrome/

The browser is available for all major operating systems. Once the download is complete, follow the installation steps presented by the installer to have the browser on your computer.

Next, we need to install a code editor. There are several free code editors available on the Internet, such as Sublime Text, Visual Studio Code, and Notepad++.

Out of these editors, my favorite is Visual Studio Code because it's fast and easy to use.

Installing Visual Studio Code

Visual Studio Code or VSCode for short is a code editor application created for the purpose of writing code. Aside from being free, VSCode is fast and available on all major operating systems.

You can download Visual Studio Code here:

https://code.visualstudio.com/

When you open the link above, there should be a button showing the version compatible with your operating system as shown below:

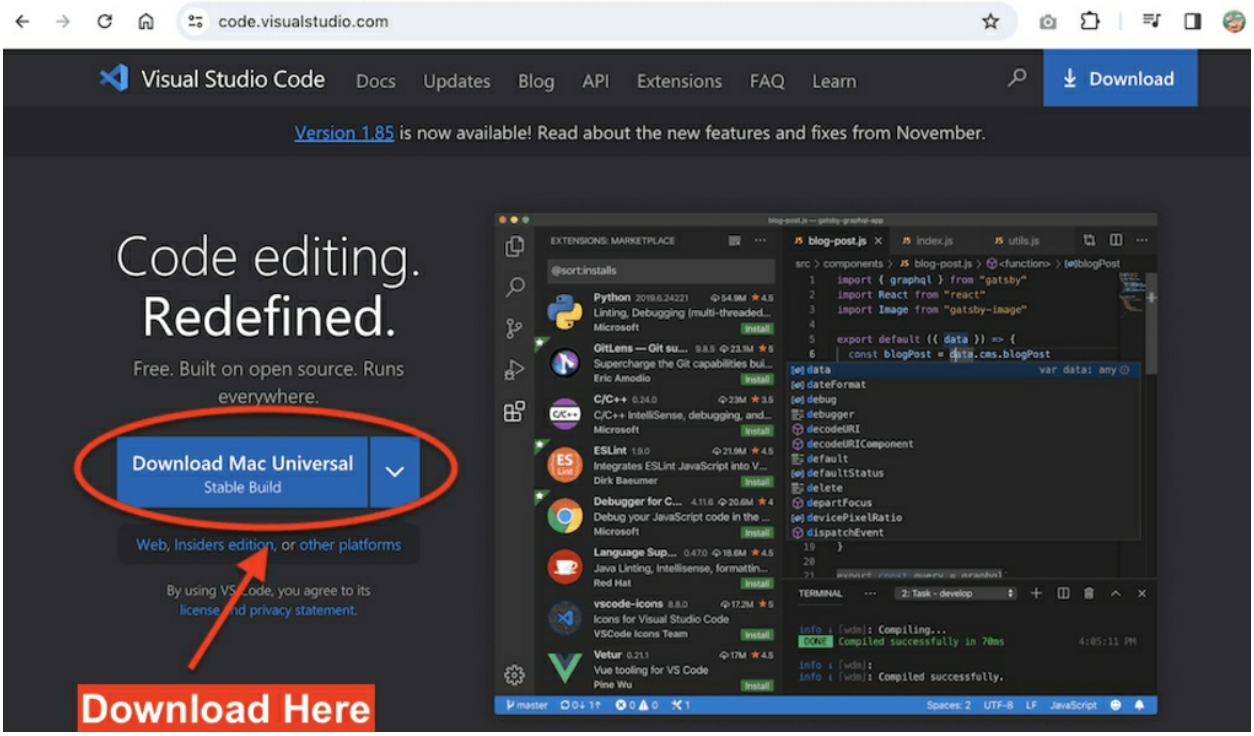


Figure 1. Install VSCode

Click the button to download VSCode, and install it on your computer.

Now that you have a code editor installed, the next step is to install Node.js

Installing Node.js

Node.js is a JavaScript runtime application that enables you to run JavaScript outside of the browser. We need this program to generate and run the web application that we're going to develop.

You can download and install Node.js from https://nodejs.org. Pick the recommended LTS version because it has long-term support. The installation process is pretty straightforward.

To check if Node has been properly installed, type the command below on your command line (Command Prompt on Windows or Terminal on Mac):

```
node -v
```

The command line should respond with the version of the Node.js you have on your computer.

You now have all the programs needed to start developing a Node.js web application. We're going to start building the application in the next chapter.

YOUR FIRST NODE.JS PROJECT

Let's create a server using Node.js and implement the client-server communication that we've seen in the previous chapter.

First, create a folder on your computer that will be used to store all files and code related to this project. You can name the folder 'finly'.

Inside the folder, open your terminal and run the npm command to create a new JavaScript project:

```
npm init
```

npm stands for Node Package Manager. It's a program used for creating and installing JavaScript libraries and frameworks. It's included when you install Node.js before.

The `npm init` command is used to initialize a new project and create the `package.json` file. It will ask several questions about your project, such as the project name, version, and license.

For now, just press Enter on all questions until you see the following output:

```
About to write to /finly/package.json:

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

```
}

Is this OK? (yes)
```

Press Enter, and you should see a `package.json` file generated in the 'finly' folder containing the same information as shown above.

Creating the Server Application

To create a Node.js server, you need to create a new file named `index.js` and write the following code in it:

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

const server = http.createServer((req, res) => {
  console.log(req.url);
  res.end('Hello From Node.js');
});

server.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

In Node.js, the `require()` function is used to get a module so that you can use it. The module name is passed as an argument to the function.

The function returns the module, which you need to assign to a variable. To prevent any confusion, the imported module is usually assigned to a variable with the same name as the module.

The `http` module can then be used to create a Node.js web server by calling the `createServer()` method.

The `createServer()` method accepts a callback function, which will be executed for any incoming requests:

```
const server = http.createServer((req, res) => {
  console.log(req.url);
  res.end('Hello From Node.js');
});
```

Node.js will pass two objects to the callback function: request and response objects, or `req` and `res` for short.

The `req` object contains information about the network request sent by the client,

while the `res` object is used to send a response back to the client.

In the function above, we simply log the request `url` value, then respond with a message 'Hello From Node.js'.

The `res.end()` method sends the string argument as a response and ends the request.

After that, the `server.listen()` method is called to open a specific web port for connections:

```
server.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

The callback function will be executed when the server is running, so we call the `console.log()` method to let us know that the server is running.

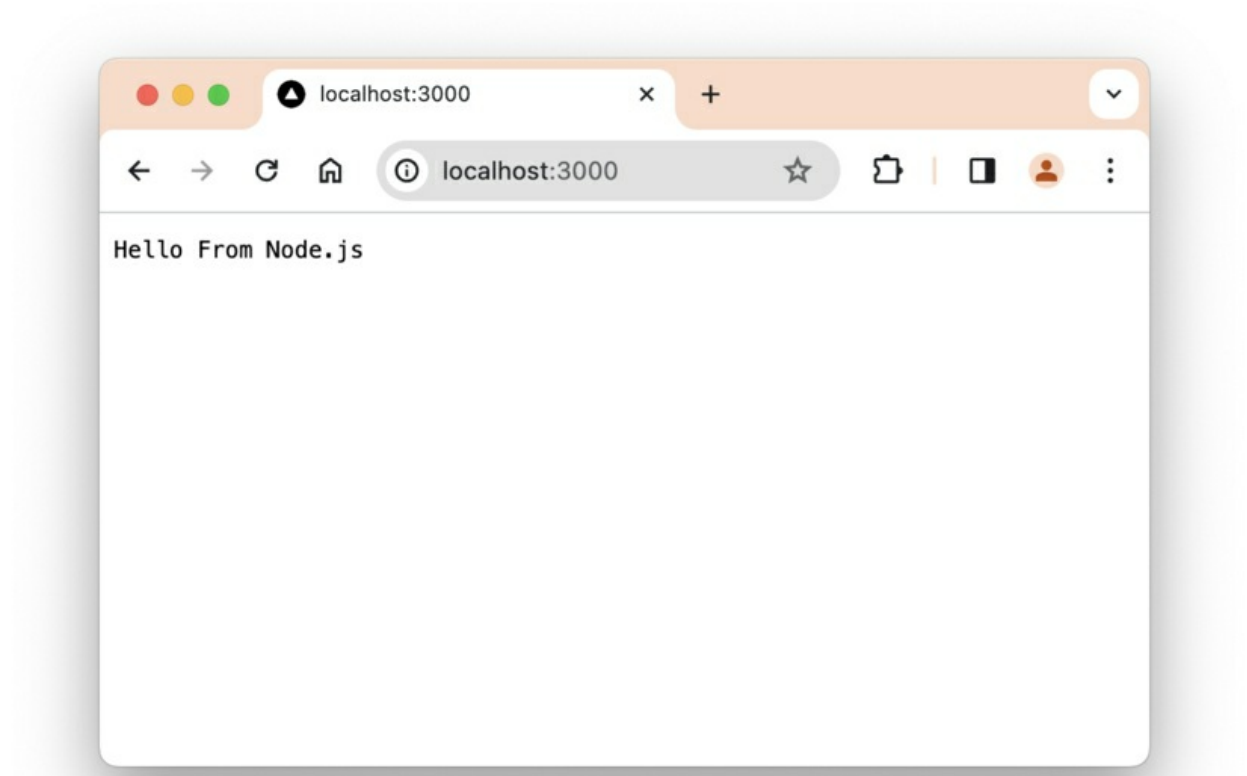
Running the Server

To run the server, open the terminal and run the JavaScript file using Node.js as follows:

```
node index.js
```

You should see 'Server listening at port 3000' logged to the terminal.

Now you can open the browser and visit the website at http://localhost:3000 to get the following response:



You now have a working server, and you can send a request to that server from the client (browser). Nice work!

Routing in Node.js

Now that the web server is created, let's add some routes to the server so that it can have different responses.

A route is a specific URL address that points to a specific response. It's like a map that the web server uses to know what to do with incoming requests.

In the previous section, we logged the `req.url` value to the terminal for each incoming request.

If you try to navigate to different routes such as localhost:3000/about or localhost:3000/contact, you'll see the URL logged as follows:

```
/
/about
/contact
```

This means we already have the means to detect the URL. We only need to make use of this data to change the response.

To add routes to the server, you can use an `if` statement to check for the `req.url` value, and send the desired response like this:

```
const server = http.createServer((req, res) => {
  const { url } = req;
  console.log(url);
  if(url === '/') {
    res.end('Hello From Node.js');
  } else if (url === '/contact') {
    res.end('The Contact Page');
  } else if (url === '/about') {
    res.end('The About Page');
  } else {
    res.writeHead(404)
    res.end('Not Found');
  }
});
```

First, we unpack the `url` value from the `req` object, so we don't have to write `req.url` every time we want to access the URL value.

After logging the `url` value. We create an `if-else` statement to define different responses for the request.

When the `url` route is not defined, we send back `404` error code.

The `writeHead()` method allows you to write specific data that you want to send back to the client. The default response code `200` means the request is processed correctly, while `404` indicates a 'Not Found' error.

The code you added to the `index.js` file won't take effect immediately.

You need to stop the server by pressing `Control + C` on the terminal, then run `node index.js` again for the changes to work.

Now if you visit different routes on the localhost, you'll see different responses because of the routing that you've implemented.