Part 3

Elementary web programming with Node.js

0. Prerequisites

Before starting this guide, you should go through the following two guides provided separately:

i. Basics of JavaScript
ii. Node.js on AWS

1. What is HTML, and how do we use it?

Up until now we have covered the following:

- the fundamentals of JavaScript programming in Node.js
- installing Node.js on AWS
- running a Node.js based HTTP server on AWS which responds to HTTP requests from our browser.

Following is the server code from the guide *Node.js on AWS,* with a couple of changes.

mywebserver.js

```
var http = require('http');
var server = http.createServer(function(req, res){
   let htmlContent = “”;
   res.writeHead(200, {'Content-Type':'text/html'})
   res.end(htmlContent);
});

console.log('Server is running on port 3000');
server.listen(3000,'0.0.0.0');
```

The changes in the code are as follows:

- In **res.writeHead**, we have changed the content type from ‘text/plain’ to ‘text/html’
- In **res.end**, instead of passing a plain text string, we are passing a variable that will contain html. For now, this is an empty sting. You would see nothing on the browser screen if you used this new code for the server.

Put simply, HTML is a mark-up language which browsers understand. Browsers have the ability to render HTML into highly organized webpages. It has a tag-based, hierarchical structure, which makes it possible for browsers to parse it into a hierarchical data structure comprising of the various elements of a web page, which can then be easily manipulated.

HTML is quite extensive. It will be pointless to cover it in depth here. You should consult a reference such as this excellent w3schools tutorial on HTML to learn more of its features. Our purpose here is to introduce the fundamental structure of HTML and explain how it fits in with the other components we have discussing, such as the server-side JavaScript code. Additionally, we will also learn how JavaScript can be embedded within HTML that is served to the client browser. The browser can execute this JavaScript on the client side, using its in-built JavaScript engine, and this is what makes webpages dynamic and interactive.

In the code examples that follow, you can choose to run the server either on AWS or localhost. The latter makes it speedier to test code, therefore I will be using that option. You will notice the IP address is 127.0.0.1 in the listen function. If you are running the server on AWS, this has to be 0.0.0.0

2. A simple HTML response

In the following, a simple htmlContent has been sent to the browser:

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

let htmlContent = `
<!DOCTYPE html>
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <h1>This is a Heading</h1>
        <p>This is a paragraph.</p>
    </body>
</html>
`;

var server = http.createServer(function(req, res){
    res.writeHead(200, {'Content-Type':'text/html'})
    res.end(htmlContent);
});

console.log('Server is running on port 3000');
server.listen(3000,'127.0.0.1');
```

Notice the hierarchical, tag-based structure of HTML.

The HTML document type declaration, also known as DOCTYPE , is the first line of code required in every HTML. The DOCTYPE declaration is an instruction to the web browser about what version of HTML the page is written in. This ensures that the web page is parsed the same way by different web browsers.

After that, the whole html code is nested inside <html> ... </html> tags.

Within the **<html> ... </html>** tags, there are two distinct sections: a document head, nested inside **<head> ... </head>** tags; and a document body, nested inside **<body> ... </body>** tags. The head section contains meta information about the web page. The body section contains the actual content of the page. In this case, the head contains a page title (will be displayed on the title bar). The body contains a heading and a paragraph. When you refresh the page localhost:3000, you will see the effects of these tags. The takeaway here is the nested, hierarchal structure. There is a long and rich list of tags you can use within the body to make your webpage more and more interesting and complex.

- **Reassign *htmlContent* with the following strings, and notice the effects of the various tags used therein:**

The following examples have been modified/adapted from the w3schools HTML tutorial, which you should visit for more details on specific elements not described here.

The table tag:

```
let htmlContent = `
<!DOCTYPE html>
<html>
  <style>
        table, th, td {
        border:1px solid black;
        }
  </style>
  <body>
    <h2>A basic HTML table</h2>
    <table style="width:100%">
      <tr>
        <th>Company</th>
        <th>Contact</th>
        <th>Country</th>
      </tr>
      <tr>
        <td>Alfreds Futterkiste</td>
        <td>Maria Anders</td>
        <td>Germany</td>
      </tr>
    </table>
    <p>To undestand the example better, we have added borders to the table.</p>
  </body>
</html>
`;
```

Notice how the table is structured hierarchically into rows and columns. Also notice the elements of style specified within the style tag. Ideally, we would separate the style information from the html file but for now we will keep the two together.

Notice the **attribute** style inside the tag table.

An attribute provides some additional information about the element in the tag.

In this case, in the style attribute, the width of the table has been specified to cover the entire width of the page.

Whenever you need to know which attributes may be used with a tag, simply look it up in an HTML reference. With experience one tends to remember the more commonly used attributes, but there is no need to remember a whole lot of them. Understanding the general structure is what matters.

The hyperlink and image tags:

```
let htmlContent = `
<!DOCTYPE html>
<html>
   <body>
      <h2>The second highest mountain in the world</h2>
      <p>
         <img src="https://static.dw.com/image/56250035_403.jpg"
         alt="wiki" width="500" height="333">
      </p>
      <p>
         Go <a href="https://vpc.police.uk/media/1801/everest.jpg">
            here
         </a>
         for the highest mountain in the world.
      </p>
   </body>
</html>
   `;
```

Notice the **src** and **href** attributes of the img (image) and a (anchor) tags respectively. Both these attributes an path address which can be local or remote (as in this case).

HTML forms

```
let htmlContent = `
```

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Enter a number to test if it is Prime</h2>
    <form>
      <!-- this is a comment: br tag is for line break-->
      <label for="num1">Input number:</label><br>
      <input type="text" id="num1" name="num1"><br>
    </form>
  </body>
</html>
`;
```

We use the **<form>…</form>** tags to create a form.

The **<input>** tag creates a text box, as specified by the value "text" of its attribute **type**. You can also create radio buttons, check boxes, text fields, etc. Note that some tags in HTML, like **<input>**, which are not supposed to nest content, do not have corresponding closing tags.

The **<label> … </label>** tags contain a label for the text box. The value of the **for** attribute of **<label>** matches the value of the **name** attribute of **<input>**, linking the two together. The **id** attribute of **<input>** has a different purpose: it will be used to identify the element which contains it, in this case a text box, from within JavaScript code.

3. Using the Express framework of Node.js

At this stage, we want to make the form above more interactive.

We wish to implement the following features:

I. Add a button to the form
II. When the user presses the button, *post* the form to the server
III. Run a program on the server to read the number from the text box and detect whether it is Prime not
IV. Return the result to the client
V. Display the result on the client's browser

We have the option of continuing with our server as it is. We can add appropriate JavaScript methods to the server code, update the received HTML and return the updated HTML to the client. This is essentially what we do, however, the simple architecture of our server will quickly become cluttered and difficult to manage as more features are added to the website.

Therefore, at this stage, we will start using Express.js. It is a web application framework for Node.js which makes the web application code much easier to maintain and extend. Moreover, given our basic understanding of HTTP and JavaScript, it is very simple to understand.

Before moving ahead, go to the terminal (or the ubuntu command line on AWS) and install the express module using npm (the Node.js package manager)

$ npm install express

In the following, the server code has been rewritten to now use the Express framework:

```
var express = require('express');
var server = express();//server will now contain additional methods

let htmlContent = `
<!DOCTYPE html>
<html>
   <body>
      <h2>Enter a number to test if it is Prime</h2>
      <form>
         <!-- this is a comment: br tag is for line break-->
         <label for="num1">Input number:</label><br>
         <input type="text" id="num1" name="num1"><br>
      </form>
   </body>
</html>
`;

server.get('/', function(req, res) {
   res.writeHead(200, {'Content-Type':'text/html'});
   res.end(htmlContent);
});

console.log('Server is running on port 3000');
server.listen(3000,'127.0.0.1');
```

What's happening in this code?

We know that HTTP requests mainly use one of two *methods* (specified in the HTTP message) -- namely, **get** and **post**.

In the code above,

server.get('/', function(req, res)...

specifies that when a get request arrives for the website's index page (the base file located at /), simply use the function that's been specified in the second parameter to respond to it. When you open a website using its URL, it's always the get request that arrives at the server.

Earlier we had provided the same function to the **server.createSever** method. However, the express framework provides this new method **server.get** which is much more versatile and flexible as we shall shortly see. The first parameter of get can be used to perform different actions based on different paths being requested on the website.

4. **Processing form data on the server side**

Let's now return to implementing the form.

Here is another version of our server:

```
var express = require('express');
var server = express();

let htmlContent = `
<!DOCTYPE html>
<html>
    <body>
        <h2>Enter a number to test if it is Prime</h2>
        <form action="/primality-test" method="post">
            <!-- this is a comment: br tag is for line break-->
            <label for="num1">Input number:</label><br>
            <input type="text" id="num1" name="num1"><br>
            <input type="submit" value="Check!">
        </form>
    </body>
</html>
`;

server.get('/', function(req, res) {
    res.writeHead(200, {'Content-Type':'text/html'});
    res.end(htmlContent);
});

server.post('/primality-test', function(req, res) {
    res.writeHead(200, {'Content-Type':'text/html'});
    res.end("<html><body>Form submitted...</body></html>");
});

console.log('Server is running on port 3000');
server.listen(3000,'127.0.0.1');
```

The changes are as follows:

- In *htmlContent*, we have added a button **(<input>** with **type**="submit") which submits the form data to the server using HTTP's post method.
- Notice the changes in the form tag:
  <form action="/primality-test" method="post">
  The **action** attribute holds the URL for the HTTP request; the **method** attribute holds the **method** type of the request which is post for this form's data.
- On the server side,
  server.post('/primality-test', function(req, res)…
  allows us to handle this request of type post, with the function specified in the second parameter. At the moment, the function is doing nothing useful. However we can easily add code that performs primality test on incoming data and returns the result to client.

At this stage, we will need to parse the incoming request containing the form data. This data can be parsed in the JSON format using the body-parser module of Node.js. So, it is a good idea to install Body Parser before we move ahead.

$npm install body-parser

Additionally, we will also need to parse some HTML at one stage and for that we will need the node-html-parser module. Therefore:

$npm install node-html-parser

Following is our fully working server code:

```
var express = require('express');
var server = express();
var bodyParser = require('body-parser');
var htmlParser = require('node-html-parser');
server.use(bodyParser.json());
server.use(bodyParser.urlencoded({ extended: true }));

let htmlContent = `
<!DOCTYPE html>
<html>
  <body>
    <h2>Enter a number to test if it is Prime</h2>
    <form action="/primality-test" method="post" id="form1">
      <!-- this is a comment: br tag is for line break-->
      <label for="num1">Input number:</label><br>
      <input type="text" id="num1" name="num1"><br>
      <input type="submit" value="Check!">
    </form>
  </body>
</html>
`;
```

```
function primeOrNot(num){
    if(num<2){
        return " not a prime";
    }else if(num==2){
        return " a prime";
    }else{
        const rootnum = Math.sqrt(num);
        for(let d=2; d<rootnum; d++){
            if(num%d==0){
                return " not a prime";;
            }
        }
        return " a prime";
    }
}

server.get('/', function(req, res) {
    res.writeHead(200, {'Content-Type':'text/html'});
    res.end(htmlContent);
});

server.post('/primality-test', function(req, res) {
    //formData is a JavaScript object
    const formData = req.body;
    const responseContent = "<p>The number is"+ primeOrNot(formData.num1)+"</p>";
    let htmlTree = htmlParser.parse(htmlContent);
  htmlTree.getElementById("form1").insertAdjacentHTML("afterend",responseContent);
    res.writeHead(200, {'Content-Type':'text/html'});
    res.end(htmlTree.toString());
});

console.log('Server is running on port 3000');
server.listen(3000,'127.0.0.1');
```

There are a few things to be discussed in this code:

- The **body parser** module has been used to parse incoming request bodies in a middleware before it gets to the handlers. As the code show, req.body is a simple JavaScript object whose properties can be accessed using the dot operator.
- The two **use statements** have been used to set up this middleware.
- The **html parser** has been used to parse the html from string format to a tree format. This html tree is an efficient way to modify specific elements of the html before converting it back into a string.
- The function **getElementById** is used to access a particular node (corresponding to an html tag) in the html tree. The id attribute of the tag is used to identify it.

- The function **insertAdjacedHTML** simply inserts a new subtree at a specified location in the html tree, hence modifying the document.
- In the end, this modified html tree is converted back to a string and sent back to the browser.
- The primeOrNot function is self-explanatory.

5. **Storing HTML in .html files**

Let's do something to clean up our server code a little. The string htmlContent is likely to grow bigger. Moreover, we will eventually have several different html contents for different web pages. It is better to store these html contents in separate html files and load them into memory when needed.

Here is how we can load HTML from a file and serve it to the client:

```
server.get('/', function(req, res) {
    res.sendFile('/Users/sbaig/index.html');
});
```

Obviously, you will need to change the path. There is a fair amount of support within Express to dynamically manage file paths, which you may explore.

sendFile sets Content-Type in the header according to the file's extension.

Similarly, at another point in the program we read index.html into a variable. For this we use the **file system** module.

```
var fileSystem = require("fs");

//...

server.post('/primality-test', function(req, res) {
    //formData is a JavaScript object
    const formData = req.body;
    const responseContent = "<p>The number is"+ primeOrNot(formData.num1)+"</p>";
    /*fileSystem.readFile('/Users/sbaig/index.html', 'utf8', function(err,data){
        if(err){
            console.error(err);
            return;
        }
        return data;
    });*/
    const htmlContent = fileSystem.readFileSync('/Users/sbaig/index.html', 'utf8');
    console.log("htmlContent: ", htmlContent);
    let htmlTree = htmlParser.parse(htmlContent);
```

```
    htmlTree.getElementById("form1").insertAdjacentHTML("afterend",responseContent);
    res.writeHead(200, {'Content-Type':'text/html'});
    res.end(htmlTree.toString());
});
```

The file system has many other functions, which you can explore.

6. Processing form data on the client side

It is often more efficient to process data on the client side if it is possible do so. This avoids the propagation and transmission delays of sending data all the way to the server and back. This is not always possible; for example, if we need to access the database on the server. However, in many cases, such as in the task of primality testing, this is perfectly doable.

The way to do this is to embed JavaScript code in the HTML which is sent to the client. The browser's inbuilt JavaScript engine can execute this script on the client side whenever a certain event is triggered.

HTML has a tag we can use to embed JavaScript. This is the **<script> … </script>** tag.

Here is a mini example of the script tag from w3schools:

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The script element</h1>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML = "Hello JavaScript!";
    </script>
  </body>
</html>
```

document is an inbuilt object which contains a parsed tree of the current page's html, just like the htmlTree we created earlier on the server side. But the object document is available to the browser on the client side. While writing JavaScript for the client side, you'd often need to look up the functions available in *document*. For example, getElementById is used to access a node based on the value of the id attribute of its corresponding tag. The property innerHTML is used to read or write HTML nested in that node.

In the following, we present our modified server and index.html files.

mywebserver.js

```
var express = require('express');
var server = express();
server.get('/', function(req, res) {
    res.sendFile('/Users/sbaig/index.html');
```

```
});
console.log('Server is running on port 3000');
server.listen(3000,'127.0.0.1');
```

The server is much smaller this time because the form is no longer posted to the server. The html is read from index.html and sent to the client.

index.html

```
<!DOCTYPE html>
<html>
   <body>
      <h2>Enter a number to test if it is Prime</h2>
      <form id="form1">
         <!-- this is a comment: br tag is for line break-->
         <label for="num1">Input number:</label><br>
         <input type="text" id="num1" name="num1"><br>
         <input type="button" value="Check!" onclick="primeOrNot()">
      </form>

      <p id="response"></p>

      <script>
         function primeOrNot(){
            let num = Math.floor(document.getElementById("num1").value);
            let response="The number is a prime";
            if(num<2){
               response="The number is not a prime";
            }else if(num>2){
               const rootnum = Math.sqrt(num);
               for(let d=2; d<rootnum; d++){
                  if(num%d==0){
                     response="The number is not a prime";
                  }
               }
            }
               document.getElementById("response").innerHTML=response;
         }
      </script>

   </body>
</html>
```

Things to note:

- The <form> tag no longer contains the action and method attributes. It will not be posted to the server.
- An input element of type=”button” has been added and an event handling function has been specified. This has been done by setting the attribute onclick= “primeOrNot()”. You should explore what other events can be handled similarly.
- Function primeOrNot() has been embedded in a <script> tag and resides on client side once it has been fetched.
- When the user clicks the button labelled “Check!”, the function primeOrNot is invoked.
- The function primeOrNot is self-explanatory. Note the use of *document*.

7. Moving embedded scripts to separate files

We have already moved out our HTML to a separate file from the server code. However, html files can swell as they continue to accumulate embedded scripts. It is therefore better to move out these scripts to separate JavaScript files.

We shall now have three separate files on the server:

mywebserver.js

```
var express = require('express');
var server = express();

server.get('/', function(req, res) {
   res.sendFile('/Users/sbaig/index.html');
});

server.get('/primeOrNot.js', function(req, res) {
   res.sendFile('/Users/sbaig/primeOrNot.js');
});

console.log('Server is running on port 3000');
server.listen(3000,'127.0.0.1');
```

Notice the additional **server.get(‘/primeOrNot.js’…**
This is because at some stage the server will be asked to **get** the embedded javaScript code as it has not be added directly into the base file.

index.html

```
<!DOCTYPE html>
<html>
   <body>
      <h2>Enter a number to test if it is Prime</h2>
```

```
    <form id="form1">
        <!-- this is a comment: br tag is for line break-->
        <label for="num1">Input number:</label><br>
        <input type="text" id="num1" name="num1"><br>
        <input type="button" value="Check!" onclick="primeOrNot()">
    </form>
    <p id="response"></p>
    <script type="text/javascript" src="primeOrNot.js"></script>
  </body>
</html>
```

The **src** attribute of <script> specifies the resource for which an HTTP GET request will be sent by the client once index.html has been fetched.

primeOrNot.js

```
function primeOrNot(){
  let num = Math.floor(document.getElementById("num1").value);
  let response="The number is a prime";
  if(num<2){
    response="The number is not a prime";
  }else if(num>2){
    const rootnum = Math.sqrt(num);
    for(let d=2; d<rootnum; d++){
      if(num%d==0){
        response="The number is not a prime";
      }
    }
  }
    document.getElementById("response").innerHTML=response;
}
```

8. End note

You can now get started with writing a web application with Node.js and JavaScript. The links that have been provided throughout this guide should serve as useful resources to learn more as you go.

Your application in the end will be a mix of some server side and some client-side JavaScript.

To begin with, you should try using DynamoDB from JavaScript functions on the server.

You can also try a relational database such as sqlite, as you are already familiar with it from Software Systems.

This is how you may install sqlite3 for Node.js:

$npm install sqlite3

This tutorial on using sqlite from Node.js should get you going.

*

Questions Welcomed!