|  |
| --- |
| [[https://myetudes.org/etudes-melete-tool/images/printer.png](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385327) Send to Printer](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385327) | [Close Window](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385327) |
|  | |
| 16. Server-Side JavaScript  16.1. Overview  *Copyright (c) 2014, Rula Khayrallah*  JavaScript is a general purpose language and there are several frameworks that offer access to a JavaScript interpreter outside the browser.   The most prevalent of these frameworks are Rhino and Node.  Rhino is free and available from Mozilla. It is implemented in Java.  It allows us to write JavaScript code that manipulates Java objects and uses Java methods.  Node is a more recent solution.  It is written in C.  It supports an asynchronous, event-driven model that makes it highly scalable.  It also contains a built-in HTTP server library that allows us to run a web server without using any external software such as Apache.  We’ll take a closer look at Node in the following sections.  16.2. Getting Started with Node  *Copyright (c) 2014, Rula Khayrallah*  Node is free and available from [http://nodejs.org](http://nodejs.org/) .   Click on INSTALL to install it on your system.  Once we have installed node, we have several options to run it.  Note that things will look slightly different on different platforms. We'll illustrate the basic steps on Windows 8 and Mac OS.   Please post in the forums if you have any issues following along.      The first option (shown on Windows 8) above, gives us access to an interpreter shell where we can write JavaScript code.  This shell is also known as the Read-Eval-Print-Loop (REPL).  We can type any valid JavaScript code and it is immediately executed.  The shell has also some special commands that start with a period (.).  The most useful one is .helpsince it shows us the rest of the commands.  On a Mac, we first open a terminal window, and then type node and hit Enter.  We get access to the same Node REPL shell.  Another option is to use Node to run a JavaScript program from a file.  This option is illustrated in the screencast below:    We can use any text editor (Komodo Edit, Scratchpad, Notepad ++, TextEdit)  to create and save the following one line file:  hellonode.js  console.log('Hello Node!');  Then on Windows, we can go to the Node Command Prompt...  ... navigate to the folder where we saved our program (cd ...) and invoke it by typing node followed by the name of the file.  In this case, it’s hellonode.js:  **node hellonode.js**  **Note that the output goes to the command (or terminal) window, not to the Firebug console**.  The browser is not involved at all here.  We are running JavaScript code in Node, outside the browser.  Similarly on a Mac, we go to a terminal window, navigate to the folder that contains our file and type:  **node hellonode.js**  In the rest of this module (and for the programming assignment), we'll be using this second option. We'll  run our JavaScript server side programs from the Node.js command prompt on Windows or from the terminal window on a Mac.  16.3. Node Asynchronous Programming  *Copyright (c) 2014, Rula Khayrallah*  Even though node is single-threaded, its **event-driven**, **non-blocking**approach makes it highly scalable.  We’ll illustrate this approach with a simple example that accesses the file system.  The file system module available in node is called fs.  It gives us access to standard file operations.  To use this module, we need to 'load' it, or 'import' it. **To do that in Node, we use the built-in function require().**   require() reads and executes a JavaScript file and then returns its exported object.  We can then access the different methods in that file through that object.  var fs = require('fs');  **Now we have access to the fs module methods through our local variable fs.**  **All the methods in the fs module have asynchronous (non-blocking) and synchronous (blocking) forms**.  Let’s first write a program that reads a local file, synchronously, and then displays its content.  In this example we’ll use the file calculator.html.  We’ll call the program fssync.js and store it in our NodePrograms folder.  We'll create a new html folder under our current folder (NodePrograms) and copy calculator.html into that new html folder.   We can then access calculator.html from the folder NodePrograms and from fssync.js as: **.**/html/calculator.html.  Note that one dot indicates that the path starts at the current directory.  fssync.js  // Load the file system module  var fs = require("fs");  // Read the file synchronously  // Make sure the file path is correct for your system  var content = fs.readFileSync("./html/calculator.html", "utf8");  // Display the content  console.log(content);  After we create and save the file fssync.js, we can go the node command prompt (or on a MAC to a terminal window), navigate to the folder that contains our program (by using cd)  and execute it by typing: node fssync.js.  We can see the content of the file calculator.html displayed in the command prompt or terminal window.  Now let’s write a program that performs **the same task asynchronously.**  To do that we’ll use the readFile() method. We’ll also add some error handling.  // Load the file system module  var fs = require("fs");  // Read the file asynchronously and call the anonymous callback function when done  fs.readFile("./html/calculator.html", "utf8", function(error, content){      if (error){          console.log(error);      } else {          // If there is no error, display the output          console.log(content);      }  }  );  Note that the readFile() method takes one more argument than readFileSync().  It is the callback function. **Since the readFile() is asynchronous, there is no one waiting for its return value.  Instead we provide a function that Node can invoke after readFile is complete.**  readFile 'communicates' with the callback function through the arguments it passes it  (error and content).  We can also write the code above with a named function instead of an anonymous one as follows:  //Callback Function to display the file content  function displayIt(error, content){      if (error) {          console.log(error);      } else {          // If there is no error, display the output          console.log(content);      }  };  // Load the file system module  var fs = require("fs");  // Read the file asynchronously and call displayIt when done  fs.readFile("./html/calculator.html", "utf8", **displayIt**);  Note that readFile() supplies both the error and the content arguments to the callback function.  **Summary: synchronous vs asynchronous**  To read a file**synchronousl**y, we issue the command to get the file from the operating system and we wait for the answer. **In the meantime, our program cannot execute anything else.** Let's say our program is a web server and we get another request for a different file from a different user, **our program cannot process that second request until the first file is read.**  With an**asynchronous** file read, we issue the command to get the file from the operating system and**we specify a function (callback) that will be called when the file is read. We can then move on and process the next request before the first file read is completed.** As a result, our asynchronous server can handle more users without a degradation in performance. That's what 'scalable' means.  16.4. Our First Web Server  *Copyright (c) 2014, Rula Khayrallah*  Now we are ready to write a program in Node that will run a very basic web server. Remember that a server is a program that waits for clients to make a request and then delivers a response.**Our first basic web server will respond with the same web page for every request.**  We’ll call the program firstserver.js.  **The http module available in node is called http**.  It gives us access to a collection of methods that support http communication between client and server.   Here again, to use this module, we need to load it or 'require' it using the built-in function require().   require() reads and executes a JavaScript file and then returns its exported object.  We can then access the different methods in that file through that object.  **var http = require('http');**  **Now we have access to the methods available in the http module through our local variable http.**  **We can create a server by invoking http.createServer().**  **http.createServer takes an optional function as an argument.  If present, that function is called whenever a request event occurs, that is whenever the server receives a request from a client.**  That function is also passed a request object and a response object arguments.   Module 15.2 includes a brief description of what goes in the http request and response.  Once we have created a server with http.createServer, we need to make the server 'listen' to requests from clients:  here we want it to listen to requests received on port 8080, so we write:  // create a server object  **var server = http.createServer(servePage);**  // listen on port 8080 - the listen method is called on our newly created server object  **server.listen(8080);**  And here's our complete server program, firstserver.js:  //The function servePage will be called whenever the server  //receives a request  function servePage(request, response) {      // 200 is the status code for success      response.writeHead(200, {'Content-Type': 'text/html; charset = UTF-8'});      // respond with a basic HTML web page      response.write("<!DOCTYPE html>");      response.write("<html>");      response.write("<head>");      response.write("<title>JavaScript for Programmers</title>");      response.write("</head>");      response.write("<body>");      response.write("<h2>Server-side JavaScript</h2>");      response.write("<h4>Node</h4>");      response.write("<p>Node.js includes asynchronous libraries such as http and fs.</p>");      response.write("</body>");      response.end("</html>");  }  // load the http module  var http = require('http');  // create a server object  var server = http.createServer(servePage);  // listen on port 8080 - the listen method is called on our server object  server.listen(8080);  // log an informational message  console.log('Server running at [http://localhost:8080](http://localhost:8080/)');  And that's it, once we run this program, we'll have a server, listening on port 8080, and responding to all client requests with the same html page.  Let’s take a closer look at our function servePage().  **Note first that the request argument is ignored here since our server is serving the same page regardless of the request.  The response argument is used to return data back to the client.**  We first call the response.writeHead() method: this method sends a response header.  It must be called once on the response object.   Here we call response.writeHead and set the status code 200 (indicating success).  We also set the Content-Type header.  The Content-Type header is used to identify the type of data that is sent in the response.  Since we are sending an HTML file, our content type header is: {'Content-Type': 'text/html; charset = UTF-8'}.  Then we have several calls to response.write().  We call response.write to send successive chunks of the HTML source document in the response body.  These calls could have been all combined into one to send the whole document in one big chunk as follows:  response.write("<!DOCTYPE html><html><head><title>JavaScript for Programmers</title></head><body><h2>Server-side JavaScript</h2><h4>Node</h4><p>Node.js includes asynchronous libraries such as http and fs.</p></body></html>");  Finally we call response.end() with the last  line in our file. And the response is completed.  Once we create and save firstserver.js (in a folder named NodeServer), we can run it in the **Node command line** **or terminal window.**  We first navigate to the folder containing our program and then we type:  node firstserver.js  **The next step is to connect to the server using a web browser.**  We can do that by opening Firefox and typing  [http://localhost:8080](http://localhost:8080/" \t "_blank) in the address bar.  The following web page should be displayed.    To stop our server, we can simply close the command line window or press ctrl C.  The following screencast illustrates the different steps involved in running firstserver.js:  **The next step is to modify our server so that it serves the web page from a file instead.**  It will still be a fixed web page, we’ll call it nodedemo.html and it will contain the following:  nodedemo.html  <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8">  <title>JavaScript for Programmers</title>  </head>  <body>  <h2>Server-side JavaScript</h2>  <h4>Node</h4>  <p>Node.js includes asynchronous libraries such as http and fs.</p>  </body>  </html>  We'll create a new html folder under our current folder (NodeServer) and save nodedemo.html in that new html folder.   We can then access nodedemo.html from the folder Nodeserver as:**.**/html/nodedemo.html.  Note that one dot indicates that the path starts at the current directory.  We then **modify our first server so that it opens and reads the content of nodedemo.html and then sends it in the response body.** Let’s call our modified server webserver.js.  webserver.js  //The following function will be called when the server  // is handling a request  function servePage(request, response) {  **// Read the file asynchronously**  **// The filename is hardcoded here - nodedemo.html is assumed to  be in a separate folder**  **fs.readFile( './html/nodedemo.html', function( err, content) {**          if (err) { // If there is an error, set the status code              response.writeHead( 404,                                 {'Content-Type': 'text/plain; charset = UTF-8'});              response.write( err.message); // Include the error message body              response.end(); // Done          } else { // Otherwise, the file was read successfully.              response.writeHead( 200, // Set the status code                                 {'Content-Type': 'text/html; charset = UTF-8'});  **response.write(content);** // Send file contents as response body              response.end();          }      });  }  **// Load the file system module**  **var fs = require("fs");**  // load the http module  var http = require('http');  // create a server object  var server = http.createServer(servePage);  server.listen(8080);  console.log('Server running at [http://localhost:8080](http://localhost:8080/)');  Note that the way we have modified our server means that the HTML document will be read every time a request is received.  This is overkill here since it is the same document, but it is good example to use as a model for this week's assignment where you’ll have to send a different page with each request.  Once we save webserver.js, we can run it in the Node command line or the terminal window on a Mac by typing:  node webserver.js.  **The next step is to connect to the server using a web browser.**  We can do that by opening Firefox and typing  [http://localhost:8080](http://localhost:8080/" \t "_blank) in the address bar.  The following web page should be displayed.  16.5. User Authentication  *Copyright (c) 2014, Rula Khayrallah*  When a server implements user authentication, the user is prompted for a user name and password.  The user is granted access to the website resources only if they provide a valid user name and password combination.  In the following example, we’ll implement basic access authentication in Node.  This is the simplest type of HTTP authentication.  We’ll first need to install the http-auth module.  Unlike the http and fs modules that come standard with Node, the http-auth module needs to be installed separately. **We can do that by using the Node package manager (npm).**  **If you have a Mac OS, make sure that Xcode is installed on your system**.  If Xcode is not installed, you'll need to follow the steps below:   * Install Xcode -  https://developer.apple.com/xcode/downloads/. * Open Xcode and accept the license agreement. * Install the Command Line Tools: to do that, click on Preferences under the Xcode drop down menu, go to the downloads tab and download  the Command Line Tools (under Components).   To install the http-auth module, the steps differ slightly between Windows and Mac OS.  We'll point out the differences as they occur.  On Windows, from the Node command prompt, type:  npm install http-auth  Note: there is an issue with the latest release of the Node Windows installer.  If you encounter the following error:  Error" ENOENT, stat 'C:\Users\yourusername\AppData\Roaming\npm'  Create the npm folder manually then rerun npm install.  You can create a folder with mkdir as follows:  mkdir c:\Users\yourusername\AppData\Roaming\npm  Make sure you replace yourusername with your actual user name.  On a **Mac**, from the terminal window, type:  **sudo** npm install http-auth  We’ll also need to install htpasswd so that we can create and manage a password file.  On Windows, from the Node command prompt, type:  npm install -g htpasswd  On a **Mac**, from the terminal window, type:  **sudo**npm install -g htpasswd  Once htpasswd is installed, we can invoke it from the command line (or terminal window) to create a new password file and add users to it:  The –c option below allows us to create a new file cs22users and add Alice as a user in it.  We are immediately prompted for a password for Alice.  **Just enter any password that you'll remember.**  You'll have to enter the same password twice.  htpasswd –c cs22users Alice  Note:  On some versions of Mac OS, if you find out that the above command does not prompt you for a password, you may use the b option and add the password right after Alice on the command line as follows:  htpasswd –**b**c cs22users Alice thepassword  To add another user, we use htpasswd (without any options) on the same file.  htpasswd cs22users Bob  Again we are immediately prompted for a password for Bob.  If we open the cs22users file, we can see the user names with their **encrypted**passwords.  Alice:$apr1$oGaNKa2d$LU9WFj3sMihZd0owIjzqa1  Bob:$apr1$DyW5LzoE$Z4YQetiJDXc.B8D0JjiHI.  Now that we have these two users, we are ready to write a server that supports authentication.  Let’s call our new server authserver.js.  authserver.js  // The function servePage will be called whenever the server  **// receives an authenticated request**.  // **we can access the user name through the request object**  // and use it to personalize our page.  function servePage(request, response) {      response.writeHead(200, {'Content-Type': 'text/html; charset = UTF-8'});      // respond with a personalized HTML web page      response.write("<!DOCTYPE html>");      response.write("<html>");      response.write("<head>");      response.write("<title>JavaScript for Programmers</title>");      response.write("</head>");      response.write("<body>");      response.write("<h2>"+ **request.user**+ " 's Lesson</h2>");      response.write("<h4>Node</h4>");      response.write("<p>Node.js includes a set of asynchronous libraries such as http and fs.</p>");      response.write("</body>");      response.end("</html>");  };  // load the http module  var http = require('http');  // load the http-auth module  **var auth = require('http-auth');**  // set the basic authentication options  var basic = auth.basic({      realm: "CS22A area",  **// the following assumes that the password file is "cs22users"**  **// and it is in the same directory as the current script**  **file: "cs22users"**  });  // create a server object **with the basic authentication options**  var server = http.createServer(**basic,** servePage);  // listen on port 8080  server.listen(8080);  // log an informational message  console.log('Server running at [http://localhost:8080](http://localhost:8080/)');  Note that the servePage() function has access to the user name through the request object:  request.user.  Once we have created and saved authserver.js, we can run it by typing the following on the command line:  node authserver.js  Then we can go to the browser and type [http://localhost:8080](http://localhost:8080/) in the address bar.  We get the following window prompting us for a user name and password:    If we type in a valid user name and password, we get access to the following web page:  Note that **the browsers 'remember' the login credentials by caching them** (saving them locally).  The caching is not permanent:  there is an expiration time associated with it.  However the caching policy policy is not consistent across browsers.    In Firefox, one way to make the browser 'forget' that we are logged in as Alice,  is to clear the cache associated with the active logins.  We do that by selecting History -> Clear Recent History -> Active Logins -> Clear Now.  16.6. Database Access  *Copyright (c) 2014, Rula Khayrallah*  Most web applications need to store and retrieve data from a database.  **Node.js has several libraries that allow us to access various database management systems from within our JavaScript program.**  In this module we’ll illustrate database access with SQLite.  SQLite is an open source library that implements a SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files.  To access a SQLite database from within our JavaScript program, we’ll need to install the sqlite3 module.  **On Windows, from the Node.js command window, type:**  npm install sqlite3  **On Mac OS, from the terminal window, type:**  sudo npm install sqlite3  To follow along with the example below, you’ll also need to **download the database file cs22a.db** available under Resources.  It is a SQLite database with one table, USERS.  The USERS table has 3 columns, NAME, ADDRESS and EMAIL.  It also has two rows, containing information about our fictional users Alice and Bob.  If you’re not familiar with SQL, just think of it as a table containing the following information:  USERS:   |  |  |  | | --- | --- | --- | | NAME | ADDRESS | EMAIL | | Alice | 12345 El Monte Rd, Los Altos Hill, CA 94022 | alice@foothill.edu | | Bob | 678 Peach Rd, Los Altos Hill, CA 94022 | bob@foothill.edu |     We’ll go back to the server code in the previous section and modify it to access the information specific to the logged in user and serve it back.  We’ll call our new server dbserver.js.  **dbserver.js**  // The function servePage will be called whenever the server  // receives an authenticated request.  // We can access the user name through the request object.  **// We use the user name to get the address and email from the database**  **// then include them in the response.**  function servePage(request, response) {      // open the database file  **var db = new sqlite3.Database('cs22a.db')**      // run the SQL query and specify a **callback function to handle the result**     // The "SELECT \* FROM USERS WHERE NAME = ..." is our SQL query here  **db.get("SELECT \* FROM USERS WHERE NAME = '" + request.user+"'", function (error, result) {**          if (error) {              console.log(error)  // If there is an error accessing the database, log the error          } else {              // we got the info, close the database  **db.close();**              response.writeHead(200, {'Content-Type': 'text/html; charset = UTF-8'});              // respond with a personalized HTML web page              response.write("<!DOCTYPE html>");              response.write("<html>");              response.write("<head>");              response.write("<title>JavaScript for Programmers</title>");              response.write("</head>");              response.write("<body>");              response.write("<h2>"+ request.user + " 's Profile</h2>");              response.write("<p> Address: "+ **result.ADDRESS** +"</p>"); // Info obtained from the database              response.write("<p> Email  : "+ **result.EMAIL** +"</p>"); // Info obtained from the database              response.write("</body>");              response.end("</html>");          }      });  };  // load the http module  var http = require('http');  // load the http-auth module  var auth = require('http-auth');  **// load the sqlite3 module**  **var sqlite3 = require('sqlite3');**  // set the basic authentication options  var basic = auth.basic({      realm: "CS22A area",      // the following assumes that the password file is "cs22users"      // and it is in the same directory as the current script      file: "cs22users"  });  // create a server object with the basic authentication options  var server = http.createServer(basic, servePage);  // listen on port 8080  server.listen(8080);  // log an informational message  console.log('Server running at [http://localhost:8080](http://localhost:8080/)');  Note that the sqlite3 module is also asynchronous, so when we invoke the get method, we specify a callback function that is called when the database query is completed.  The callback function takes two parameters, error and result.  Make sure you save  dbserver.js in the same folder as the database file cs22a.db (that you downloaded from Resources).  Then, from the Node.js command, we navigate to that folder and we start our new server by typing:  node dbserver.js  Then we point our browser to localhost:8080, login as Alice or Bob,  and one of the following web pages should be displayed: |  |