CS215 Lab08
SP2018
streller

PROBLEM STATEMENT:

Basic experience with the E and N in the MEAN web stack.

BACKGROUND:

Let's start with N.

N is for node (sometimes referred to as node.js).In 2009, Ryan Dahl,was frustrated that *concurrency* is difficult in many programming language and often leads to poor performance. He wanted to make it easier to write networked software that was fast, supported many users and used memory efficiently. So he created Node.js

More less concurrently, no pun intended, Google was heavily investing in browser technology for its products (Gmail, etc). Google created V8, a *javascript engine* for Chrome. V8 is a highly optimized piece of software (written in c++) designed for the web.

Note: A JavaScript engine is a program or an interpreter which executes JavaScript code. A JavaScript engine can be implemented as a standard interpreter, or just-in-time compiler that compiles JavaScript to bytecode in some form.V8 was first designed to increase the performance of JavaScript execution inside web browsers. In order to obtain speed, V8 translates JavaScript code into more efficient machine code instead of using an interpreter. It compiles JavaScript code into machine code at execution by implementing a JIT (Just-In-Time) compiler.V8 doesn't produce bytecode or any intermediate code.

Dahl decided to use the open source V8 egine to create a javascript *server-side environment* Node.

Reasons for Node being based on V8 engine

V8 extremely fast
V8 focused on the web, so it is proficient at HTTP, DNS and TCP
Javascript is a well known language, so it is accessible to most developers

Node is an event-driven server-side JavaScript environment

Node excels a real-time apps, streaming apps, and single-page (browser) applications (SPA's)

CODE:

First off, run the files sb.js and asnb.js in Node.

Extract sb_and_asnb.zip. Open the CLI (command line interface; aka power shell terminal), and navigate to the sb_and_asnb directory. To run each file type node fileName. Specifically enter

```
node sb.js
```

and

```
node asnb.js
```

to execute the files.

Examine these js files, their output and note the differences.

Note: In the sb.js file the sleep function is intended to simulate the time interval it take to complete the request across the internet.

Both files call/execute the functions fetchPage() and fetchApi() in the same order - yet the output is different..While the code syntax for the files is different, both programs are simulating the same operation(s).

Your ReadMe must answer/discuss the reasons for these differences (what type of coding principle is involved) for credit for this part of the lab.

Now for the E

First off, you need to install Express in your home directory.
On Windows, Express will install to C:\Users\YourName\AppData\Roaming\npm

To install Express, open the Windows command-line terminal called PowerShell and go to your home by issuing the command

```
cd  C:\Users\YourUserName
```

the prompt in the terminal will now be

```
PS C:\Users\YourUserNamer>
```

Now to install Express, type the command int the terminal

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

A bunch of output will appear in the terminal as Express is installed. Once the install is complete, exit the terminal by typing exit.

Note: If you wish to install Express on your own machine, you will have to first install node and npm, then follow the above to install Express.

Next, we need to create a place for our server to live, and a place for our files to reside.

Create a directory called CS215 in your home directory; ie in C:\Users\YourUserName make the directory CS215.

Place the included zip file here and extract. Your should now have the directory

C:\Users/YourUserName\CS215\Lab8_Express

created. Within Lab8-Express, you will have 2 directories : express_demo1 and express_demo2.

Lets now examine a simple express server.
Using the window manager, go to the directory. You will find 2 files here, the server, called server.js and the ReadMe.txt

Open the powershell terminal and cd to the express_demo1 directory. As specified in the readMe start the server.You'll see a message that says

"Server listening on port 8080"

and should no longer get a terminal prompt. That's because your server is running.
You can confirm that by opening Firefox and typing localhost:8080 in your address bar.

If everything worked correctly, you should see a "Hello World!" message.
First things first: you can stop the program from the terminal window by hitting Ctrl+C.
Once you're done working, you can exit out of the terminal by typing exit.

...And now a word about servers and clients:

> In the field of computer networking, we typically think of computer programs as being either client programs or server programs. Traditionally, a server program abstracts some resource over a network that multiple client programs want to access. For example,someone might want to transfer files from a remote computer to a local computer. An FTP server is a program that implements the File Transfer Protocol, which allows users to do exactly this. An FTP client is a program that can connect and transfer programs from an FTP server.
>
> The server-client relationship is not limited to networks. Within your OS, there is the X-windows server which provides the customary window display. There is also the file server which hosts the file directory structure.
>
> A lot of detail goes on in the world of computer networking that is ancillary to web application development, but a few important things are worth understanding.
>
> The first is that (most of the time) the client is a web browser and the server is a remote machine that is abstracting resources via the Hypertext Transfer Protocol, for HTTP for short. Although it was originally designed to transfer HTML documents between computers, the HTTP protocol can now be used to abstract many different types of resources on a remote computer (for instance, documents, databases, or any other type of storage).More is said about protocols in CS211, but for now we can think of it as the protocol that we're using to connect browsers to remote computers. Our HTTP servers will be used to deliver the client-side part of the application that the web browser will interpret. In particular, all of the HTML, CSS, and JavaScript that we've learned up to this point will be delivered to the browser via the server. The client-side program running in the browser will be responsible for getting information from or sending information to our server.
>
> Typically, our HTTP server is running on a remote machine. This causes problems for developers—if we're running our code on a remote server we have to actually either edit the code on the remote server and restart it, or we have to edit the code locally and push it every time we want to try it out. This can be highly inefficient. We're working around this problem by running the server locally on our machine.

.

Now back to our lab...

Now, lets examine the server in express_demo1

This code shouldn't look completely unfamiliar to you—you should be able to immediately identify some variable declarations, a console.log statement, an anonymous function, a JSON object, and a callback pattern.

So what is this code doing? It turns out it's doing a lot—it's creating an HTTP server that is responding to web browser requests! As mentioned before, HTTP stands for HyperText Transfer Protocol and it's the basic technology that's behind the World Wide

Web! You may have heard of programs like Apache or Nginx—these are industrial strength, configurable HTTP server programs that are designed to host big websites.

Our HTTP server is much simpler than that: it's simply accepting a browser request and responding with a text response that says "It Works."

We can think of the code as behaving exactly as our jQuery click handler behaves—the difference is that instead of the callback being called when a user clicks, it is called whenever a client (in this case, the browser) connects. The req parameter is an object that represents the HTTP request that is coming to our server from the client, and the res parameter is an object that represents the HTTP response that we'll send back to the client. The res.writeHead function creates the HTTP header that sets the attributes of the response, and the res.end function completes the response by adding "It Works."

After we create the server, we make it listen on port 8080 (which is why we type 8080 after localhost in our web browser's address bar). The console.log statement prints out the statement to the server terminal when we run the program

For such a small program that is quite a lot It's more than you likely realize—an HTTP server is a nontrivial piece of software that takes a good deal of skill to write correctly. Fortunately, we don't have to worry about the details because we imported the code via the Node.js http ***module***.

A ***module*** is simply a collection of code that we can use without completely understanding how it works internally—all we have to understand is the API that it exposes to us. The first line in our code is the require statement, which imports the http module and stores it in the http variable. This HTTP server module is interesting if we need a bare-bones, stripped down HTTP server that simply accepts and responds to client requests. Once we want to start sending HTML or CSS files from the server, however, things become much more complicated.

We can build a more complex server on top of the basic HTTP server that Node.js gives us but, lucky for us, somebody else already solved that problem, too! The Express module creates a layer on top of the core http module that handles a lot of complex things that we don't want to handle ourselves, like serving up static HTML, CSS, and client-side JavaScript files.
One of the benefits of programming in Node is that we can leverage numerous modules like Express, many of which do very useful things. The http module that we used earlier is part of the core distribution of Node.js, so we don't have to do anything special to use it. Express is not part of the core distribution of Node, so we need to do a little more work before we can access it as easily as we access the http module. Fortunately, it turns out that every distribution of Node comes with another program called the ***Node Package Manager*** (or NPM for short). This tool allows us to easily install modules and then immediately leverage them in our code.

Now on to the included express_demo2.

This is an example of the Express server using the express module.

While this is similar to express_demo1, there are a few noticeable differences.
First of all, we don't need to set up HTTP headers in the callbacks because Express does that for us. Also, we're simply using res.send instead of res.write or res.end. And last, but probably most important, we've set up two ***routes:*** hello and goodbye. We'll see what this does once we open our browser, but first let's start up the server.

As before, start the server in express_demo2.Recall that we can do that by typing node server.js from the express_demo2 directory on our guest machine. Now when we go to our web browser, we can type localhost:8080 like we did before. This time, however, we should see an error that says "Cannot GET /".

But, if instead, we type in localhost:8080/hello or localhost:8080/goodbye, we should see the message that we specified in the callbacks. Try it now.

As you can see, the addition of hello and goodbye after the main URL of our app specifies which function gets fired. And you'll also see that Express doesn't set up a default route for us. If we want to make localhost:8080/ work as it did before, we simply set up a root route by adding another three lines to server.js:

```
app.get("/", function (req, res) {
res.send("This is the root route!");
});
```

If we stop our server (by pressing Ctrl-C) and start it again, we will be able to access localhost:8080 as we did before! Express is handling the complexities of routing—we simply tell it what we want it to do when certain routes are requested.

Try it now by uncommenting the appropriate lines in server.js and visit localhost:8080

Setting up your client site app

We've seen that we can send information to the web browser from the server. But what if we want to send something like a basic HTML page? Then things can get complicated relatively quickly. For example, we might try to do something like this:

```
app.get("/index.html", function (req, res) {
res.send("<html><head></head><body><h1>Hello World!</h1></body></html>");
});
```

Of course this is silly, and although this will work, creating HTML that is bigger than this small example will become extremely cumbersome. Fortunately, Express solves this problem for us by allowing us to use it as a **static file serve**r. In doing so we can create HTML, CSS, and client side JavaScript files as we've been doing throughout this course. And it turns out that it only takes one more line of code.

Uncomment the following line

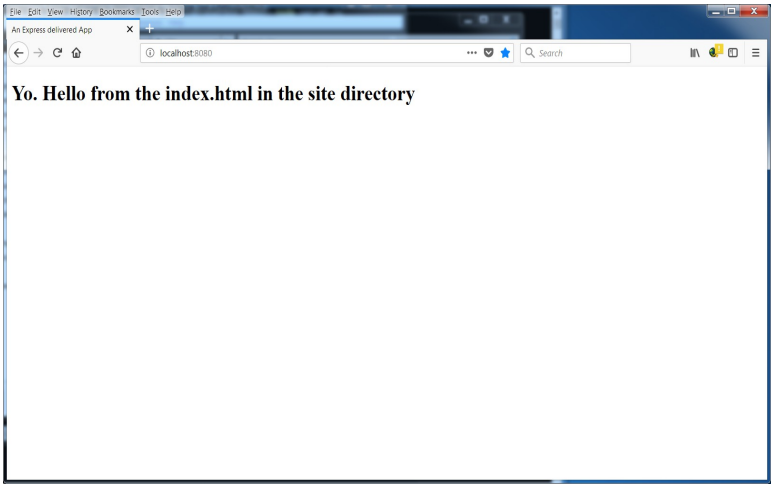
```
app.use(express.static(__dirname + "/site"));
```

Now restart the server. If you now go to localhost:8080 we will again be greeted by

Cannnot /GET.

Why, cuz the server is looking for an index.html page to send.
So, go to the site directory and rename the html file to just index.html and reload the browser

This time we are greeted by the html page:



In this example, we've used app.use to create a static file server directory. This means that any request sent to our server will initially be resolved by the static file directory (site) before it is handed off to our routes. This means that if we have a file called index.html in our site directory and we go to localhost:8080, it will return the contents of the file. If the file doesn't exist, it will then check to see if there's a match among our routes.

Here's where some confusion can arise—if you have a route with the same name as a file in your site directory, how does Express respond?

It resolves to the site directory first, so if there's a match it doesn't even look at your routes. Be careful not to have routes and files that have the same name—that is almost certainly not what you intend to do.

Now, let's use Node and Express to serve up our Lab7(Lab6B) files instead of using WAMP.

Using the cs215_Express_Lab\express_demo2 server and node modules, create a similar site for your lab6B files. Specifically, once the server is started the user will type

http://localhost:8080/brightIdeas/

and have our brightIdeas web page render correctly.

All pages must work; you will have to of course edit the server.js file for this to work correctly

Create a directory inside the Lab8_Express one called lab8_ANS. Copy the server, and node_modules from demo2 here. Inside this directory create a folder called www. In the www directory create a directory called brightIdeas. Place all your lab6B files here. Edit the server.js to have the correct static file directory route

Call instructor over to demo this lab for credit.
NO Demo No Grade

Deliverables:

Send to streller@ecc.edu an email this the exact subject

cs215_Lab_08

In this email attached will be the zip file

LastName_ Lab_08_cs215.zip

The zip file must contain all your lab8_ANS files and your ReadMe,This file that contains your name at the top, the lab number, any comments regarding the assignment, and window captures to show the program execution. The ReadMe file must be in the top level project folder.

Grading will be based on **correctness of the html, css and js files** and how the page is rendered in a browser.

Due Date : 27 March 2018