CSE 110 Lab 6

Learning Goals

In this lab, you will be learning about **Node.js**, the JavaScript runtime system that allows us to build server-side applications using JavaScript! **Please read the information below carefully** before diving into the exercises to learn what Node.js is and why it became so popular over the last few years. Through this lab, we hope you will learn what Node.js is and why one might want to use Node.js for their application. We will also learn to perform basic server-side tasks using Node.js and build a Client-Server Chat Application!

A Runtime System

A runtime system provides an application with all the necessary libraries, tools and resources it requires to run properly. It helps with tasks such as allocating and initializing memory, setting environment variables and handling device I/O. In a nutshell, it interacts with the operating system and puts everything in place to ensure smooth execution of the program.

Node.js

JavaScript has traditionally been a client-side language used to write front-end logic defining user interactions with web pages. Node.js is a platform built on Chrome's JavaScript runtime for building fast and scalable networked applications. Using Node.js, we can now write backend server-side code in JavaScript and have the option to use JavaScript throughout our web stack. It implements a 'non-blocking I/O' model (which we will learn about soon) that enables us to write lightweight and efficient applications. Node.js also provides us with a rich library of JavaScript modules which we can import into our applications to make our life easier.

Why NodeJS

Asynchronous and Event Driven − All APIs of Node.js library are *asynchronous* (non-blocking). A Node.js based server never waits for an API call to return data, it simply moves to the next API call. Node.js implements an internal event tracking mechanism which will help the server respond to the previous API call after it finishes.

Fast − Node.js is built on Google Chrome's highly efficient V8 JavaScript Engine.

Single Threaded but Highly Scalable − Node.js uses a single threaded model with an event loop which keeps track of events and callbacks. The event mechanism implemented by Node.js helps the server to respond to requests in a non-blocking way. This makes the server highly scalable as opposed to traditional servers which create a limited number threads to handle requests. Although a Node.js server is a single threaded program, it can handle a much larger number of requests than traditional servers like Apache.

Reading streams in chunks − When an API call is made, Node.js allows you to read the data in chunks, as they arrive, instead of waiting all the data to arrive.

Check your understanding:

At this point you should be able to answer the following questions:

a. What is Node.js?
b. Is Node.js a programming language?
c. What is a runtime environment?
d. Why might you want to use Node.js for your project?
e. Is Node.js required for all web applications? What could be some alternatives?

Prerequisites for the next part:

1. Have the latest version of Node.js installed on your machine.
2. Have some basic JavaScript knowledge - not necessary, but recommended.

Part 1: Node.js Basics

Let's dive right in! We will begin by learning to do some basic tasks using Node.js.

Hello World!

To make a Node.js program, create a new file with a .js extension and start writing JavaScript!

You can print something on your terminal, in the same way that you would print something to the browser's console using JavaScript.

```
console.log("This will be printed on the terminal");
```

You can run a Node.js application using

```
$ node myprogram.js
```

Task 1: Write a Node.js program `'helloworld.js'` that prints "Hello World" to your terminal.

Processing Command Line Arguments

You can use the `argv` field of the `process` object provided by Node.js to access the command line arguments passed to your program.

args.js contains the following:

```
console.log(process.argv);
```

```
$ node args.js 1 2 3 these are command line arguments
[ '/usr/local/bin/node',
  '/Users/Ajeya/NodeLab/Hello.js',
  '1',
  '2',
  '3',
  'these',
```

```
  'are',
  'command',
  'line',
  'arguments' ]
```

Task 2: Write a Node.js program `'hello.js'` that accepts a string name as a command line argument and prints "Hello, <name>!".

Example:

```
$ node hello.js Gary
Hello Gary!
```

Aside: Node.js Interactive Command Line Interface

Node.js provides you with a command line interface which you can use to quickly test some code.

```
$ node
> console.log ("I can quickly test some code using the interactive CLI!");
I can quickly test some code using the interactive CLI!
undefined
> a = 1; b = 2; console.log(a+b);
3
undefined
```

I/O Operations!

Let's learn to implement something we might actually want a server to do.
Node.js provides us with the `fs` module to perform filesystem operations. We can load the fs module using:

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

All synchronous (or blocking) file system methods in the fs module end with 'Sync'. To read a file, you'll need to use:

```
var buf = fs.readFileSync('/path/to/file')
```

This method will return a **Buffer** object containing the complete contents of the file.

Buffer objects are Node's way of efficiently representing arbitrary arrays of data. Buffer objects can be converted to strings by simply calling the toString() method on them.

```
var str = buf.toString()
```

Task 3: Write a Node.js program `'newlines.js'` to count the number of lines in a file.

Example:

foo.txt contains the following

```
This
File
Has
Four Lines
```

```
$ node newlines.js
4
```

Tips: Vanilla JavaScript String functions could be useful!
Documentation for fs
Documentation for Buffer objects
Documentation for JavaScript Strings

Asynchronous I/O!

We learned that one of the key features of Node.js is that it is *asynchronous and event driven.* Let's learn how to utilize that by learning to perform non-blocking I/O operations.

Asynchronous Functions
An asynchronous function call doesn't wait for the function to finish before proceeding to the next instruction. The function accepts a **callback function** as an argument, which will be called once the function is ready to return. This allows us to continue doing other tasks while waiting for a slower I/O operation (such as reading a file) to finish.

Example: Asynchronous function usage Source

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

```
var myNumber = undefined // we don't know what the number is yet
function addOne() {
  fs.readFile('number.txt', function doneReading(err, fileContents) {
    // inside the callback, after we finish reading the file
    myNumber = parseInt(fileContents)
    myNumber++
  })
}

addOne();
console.log(myNumber) // undefined -- this line gets run before readFile is done
```

Task 4: Perform 'Task 3' using `fs.readFile()` instead of `fs.readFileSync()`

Check out this article to learn more about Callback functions.

Let's get creative!

Task 5: Write a Node.js program `'list.js'` that reads all files in a given directory and prints all the JavaScript files in the directory (ends with .js)

Example:

```
$ ls /path/to/directory
Apple.java Banana.java Apple.js Banana.js Apple.py Banana.py
$ node listjs.js /path/to/directory
Apple.js
Banana.js
```

Tips:
The `fs.readdir()` method takes a pathname as its first argument and a callback as its second. The callback signature is:

```
function callback (err, list) { /* ... */ }
```

where `list` is an array of filename strings.

Modules

A common pattern for a Node.js application is to add different functionality in different *modules,* in their own files. In Node.js, each file can be treated as a different 'module'. To define a single function export, you can assign your function to the `module.exports` object, overwriting what is already there.

```
module.exports = function (args) { /* ... */ }
```

Here's an example where we add a module which exports a single function that squares a number:

Main.js

```
var squareFunc = require('./square');
console.log(squareFunc(2));
```

Square.js

```
module.exports = function square (num) {
     return num * num;
}
```

```
$ node main.js
4
```

Using the `module.exports` object, you will be able to export primitives, functions, objects of your choice, and use them in different files to make your code more modular.

Read the Node.js Modules Documentation for more information about how to export data from your .js file!

Task 6: Implement Task 5 using Node.js modules!

Create a file `filter.js` which exports a single function with the signature

```
function filterFiles (pathToDir, callback)
```

This function will do the necessary work for Task 5, and invoke `callback` on every valid filename found.
Now create a new file called `main.js` that accepts a path to a directory on the command line, and uses your new module to print files with a `.js` extension.

In Progress Checkoff!

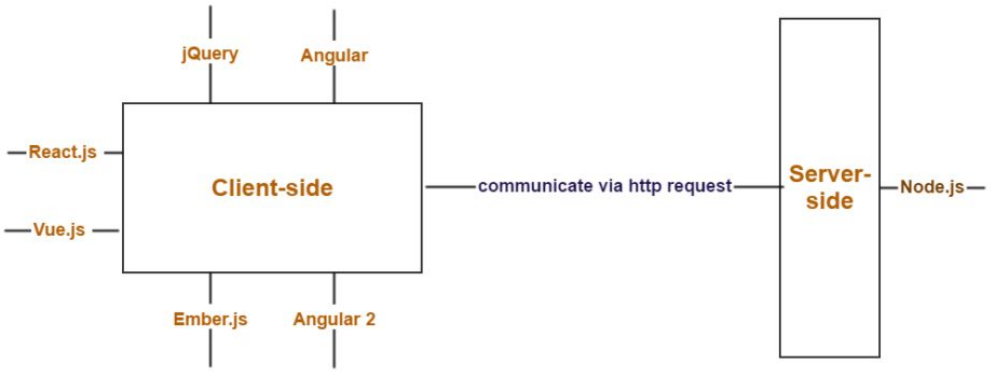
The tutor will ask you to run a few of the above tasks to make sure you implemented them correctly. Move on to the next part of the lab once you receive the Checkoff!

References: The following resources were referred to during the creation of this lab. Some content has been taken from them and modified for clarity and to meet our requirements.

https://nodejs.org/en/
https://www.tutorialspoint.com/nodejs/nodejs_introduction.html - Careful, has some incorrect/misleading information
https://www.w3schools.com/nodejs/
https://github.com/workshopper/learnyounode

Part 2: The Client/Server relationship

Before we begin, let's define the key actors in our client-server model. Follow along with this image.



The Client

The client makes requests to the server to get content or send actions. When we talk about **"client-side"** logic or the **"front-end"**, we are referring to the HTML, CSS, and JavaScript code that is running in our browser or the Android/iOS layouts on our phone. This code has the primary objective of correctly rendering the website/app and providing an interactive interface for the user.

There exist many front-end frameworks for JavaScript that make creating dynamic websites easier such as React, Angular, Fuse, jQuery and Vue, to name a few. Without these frameworks, we would have to create interactive websites using vanilla(pure) JavaScript code and HTTP requests.

The Server

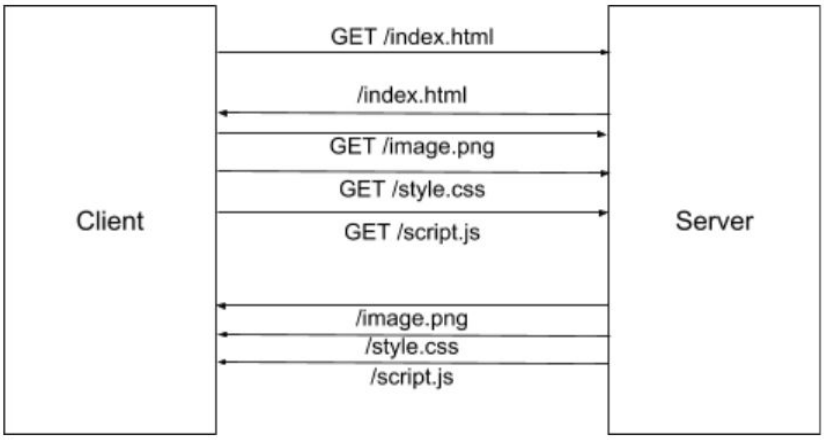
The server is exactly what it sounds like: it "serves" (sends) things to those that want them. When we talk about **"server-side"** logic or the **"back-end"**, we're talking about code that runs on the server, with the overall objective of serving the requested content.

We can run whatever software we want in our server as long as it fulfills the objective of serving files. A popular option is **Node.js** and other popular back-end options include Django (Python), Flask (Python), Ruby on Rails (Ruby), and Spring MVC (Java).

Building a Node Server

Communication between the client and server

Any good relationship requires communication. So how do the client and server talk to each other? A basic website can be requested from the server through an **HTTP Request**. This is a request from the client to the server for a certain file, such as `"index.html"`, which the server proceeds to send. HTTP Requests can only be initiated from the client, and they can have different "verbs," which tell the server what the serve. A GET request typically asks for a static resource like a webpage. A POST request typically asks the server to create or update data on the server.



Each HTTP Request spawns a new connection. What if we want to keep a channel of communication open so that the client and server and continually connected? If the server supports it, the client can also use a different method of communication called **WebSockets**. This protocol supports a more real-time method to communicate, and is used in situations when rapid real-time communication is required such as a chat app or a multiplayer game. This is what we will use in this lab to create a Chat app! Let's get started.

First off, let's create the project directory we are going to work in. Type the following in the terminal:

```
$ cd ~
$ mkdir chatapp
$ cd chatapp
```

NPM and Node packages

One reason why Node.js is so popular is that there exists a thriving community around it which creates useful packages that anyone can use in their Node projects. We can import these packages to our own project, just like how we imported modules to our files earlier. We will use a package, called **Socket.io**, which allows us to communicate with the clients connected to our server in real time using WebSockets.

We will also use **Express**, a package that allows us to handle backend routing for our website. Each HTTP request includes a path for the requested resource. Routing allows us to define which webpage to serve for the requested URL. (Much more on these frameworks coming up!)

NPM is a package manager which allows us to download and manage the packages we wish to import into our project, as well as our Node project as a whole.

We can use NPM to create a barebones project directory to get started. Let's do that with the `"npm init"` command:

```
$ npm init
```

This command creates a file called `package.json`. This file is essential to Node.js development as it contains important information about our project and our package dependencies.

Fill out the following details as the command asks for them:

Package name: **"chatapp"**
Version: *Just press enter*
Description: "Simple chat app"
Entry point: **"server.js"**

- This is the file responsible for launching our server. (Where our "main" method is.)

Test command: *Just press enter*
Git repository: *Just press enter*
Keywords: *Just press enter*
Author: <your name>
License: *Just press enter*
Is this ok?: **"yes"**

In the end, your package.json should look like the following:

```
{
  "name": "chatapp",
  "version": "1.0.0",
  "description": "Simple chat app",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "CSE 110 Staff",
  "license": "ISC"
}

```

The ability to install and use open source libraries is one of the reasons why Node.js is so powerful. Lets download and install the two aforementioned packages we want to use with the **npm install** command.

```
$ npm install --save express body-parser socket.io
```

Now check your `package.json`. Notice that `npm install` installs the package we specify into the `node_modules` folder, and the `--save` flag adds it as a dependency in our `package.json`.

Spinning Up a Server

Lets create a simple "hello world" HTML page as the first page on our site.
Create a new file `index.html` and add whatever you like to it. Just make sure it at least has a body and 1 visible element. Here is an example:

```
<!DOCTYPE html>
<html>
    <body>
        <h1>The server is running!</h1>
    </body>
</html>
```

Create and open `server.js`. This will contain the main back-end logic for our server. Paste the below code into it to import the libraries we need:

```
var express = require('express');
var http = require('http');
var app = express();
var httpServer = http.createServer(app);

httpServer.listen(3000, function(){
      console.log("Listening on port 3000");
});
```

Step by step explanation of the code:

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

First we import the packages necessary for starting a server. Namely `express` and the built-in `http` package.

```
var app = express();
var httpServer = http.createServer(app);
```

Next we create the app and server. The `app` object defines the behavior for our application, such as what to do when the user requests for a certain resource. The `httpServer` object is an interface between the application and the client connection, which means when the client makes a request, the server will notify our app of the request.

```
httpServer.listen(3000, ...);
```

Finally, our server starts listening to port 3000 for incoming requests. We also supply a callback that is called when the server starts up.

Run `npm start` to start the app, and access `localhost:3000` on your browser. What do you notice in the terminal? In the browser? (Don't be alarmed if you see an error)

The Request / Response Model

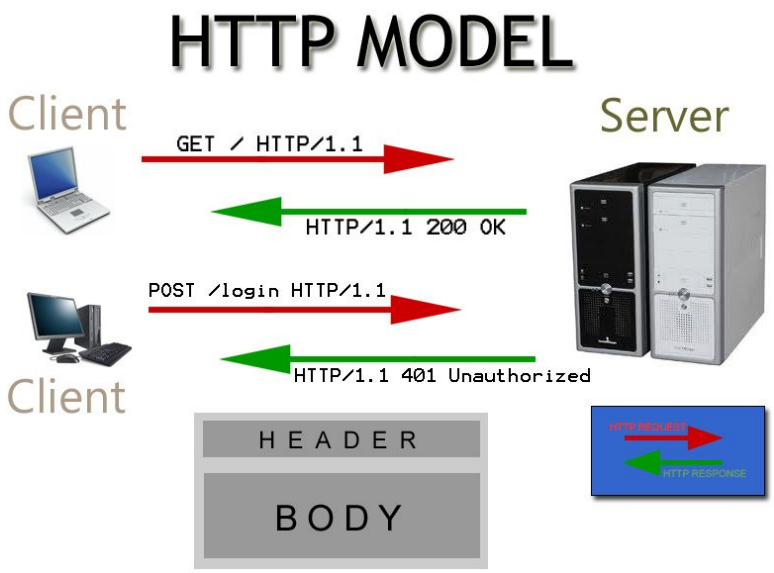
Oh no! When we open our browser to the page, all we get is an error that says

```
Cannot GET /
```

Isn't our server up and running?

The truth is, our server is running normally. The console message is evidence for that. But remember, our `app` object defines the behavior of our application. We still haven't defined what it should do yet!

We mentioned earlier that HTTP uses a request / response model for communicating. First the client requests some resource, occasionally including extra headers or body to add specific details to the request. Then on receipt of the request, the server returns a response with either the data requested, an error, or some other appropriate response. Here is a simple illustration of this:



Now, we need to define how to `GET /` , so that the server can serve `index.html` to the client. We can do this easily using the `get()` method. Add the following to `server.js` and try to understand what it's doing.

```
app.get('/', function(req, res) {
      console.log("Received GET request for resource /");
      res.sendFile('index.html', { root: __dirname });
});
```

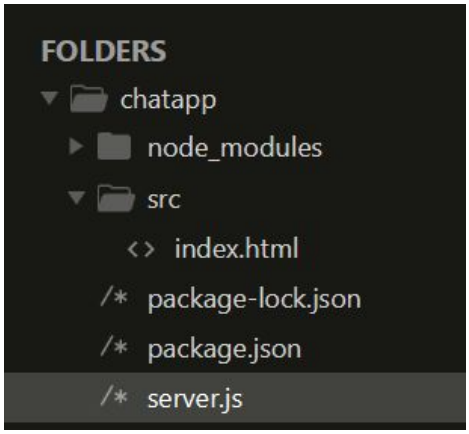
The `get()` function defines our function to be called when the client requests `'/'`, and `res.sendFile` gives a `"200 OK"` response with `index.html` as the body. Our `server.js` should now look like this:

```
var express = require('express');
var http = require('http');
var app = express();
var httpServer = http.createServer(app);

app.get('/', function(req, res) {
    console.log("GET / HTTP/1.1");
    res.sendFile('index.html', { root: __dirname });
});

httpServer.listen(3000, function(){
    console.log("Listening on port 3000");
});
```

Access localhost:3000 now after running `npm start` and now we can see our `index.html` being rendered!

Our current `GET` callback is very basic. The behavior does not change regardless of the client's request, so we say that it is serving **static content**. We can confirm this is the case since the `req` parameter, which contains the request data, is not used. It would be troublesome if we had to define an endpoint for every single static resource (an `app.get()` call for every file single file!), but luckily there is an easier way.

First, create a new subdirectory `"src"` and move `index.html` into that directory. The directory structure should look like below:



Now in `server.js`, replace the `app.get()` call with the following code:

```
app.use(express.static('src'));
```

The line above adds **middleware** for the application to handle static content. Middleware is a generic callback, usually a library, that handles only requests that are relevant to it. It will usually either transform the request to make it easier to process, or handle it entirely. In our case, if the `express.static()` middleware sees a request that starts with `/src`, it will automatically handle the request by sending the relevant file to the client. Your code should now look like this:

```
1  var express = require('express');
2  var http = require('http');
3  var app = express();
4  var httpServer = http.createServer(app);
5
6  app.use(express.static('src'));
7
8  httpServer.listen(3000, function(){
9      console.log("Listening on port 3000");
10 });
11
```

Restart the server and visit `localhost:3000` again, and make sure the page still loads correctly. Even though we are accessing resource `'/'`, our app will know to access `'/index.html'`, since that is the default behavior for web servers.

Note that now our app will not log the `GET` request anymore. We can fix this with middleware of our own. Add the following middleware function to your app *above* the original `app.use()` call:

```
var myLogStatement = function(req, res, next) {
    console.log("Received", req.method, "request for resource", req.path,
"from", req.ip);
    next(); // callback to the middleware function
}
```

Now assign the above function as middleware, similar to how you added the `express.static()` middleware (using **`app.use`**).
Restart the server and watch the console when you access the page from your browser. It should look something like this:

```
> chatapp@1.0.0 start /mnt/c/Users/Kanurame/Documents/chatapp
> node server.js

Listening on port 3000
Received GET request for resource / from ::ffff:127.0.0.1
Received GET request for resource /favicon.ico from ::ffff:127.0.0.1
```

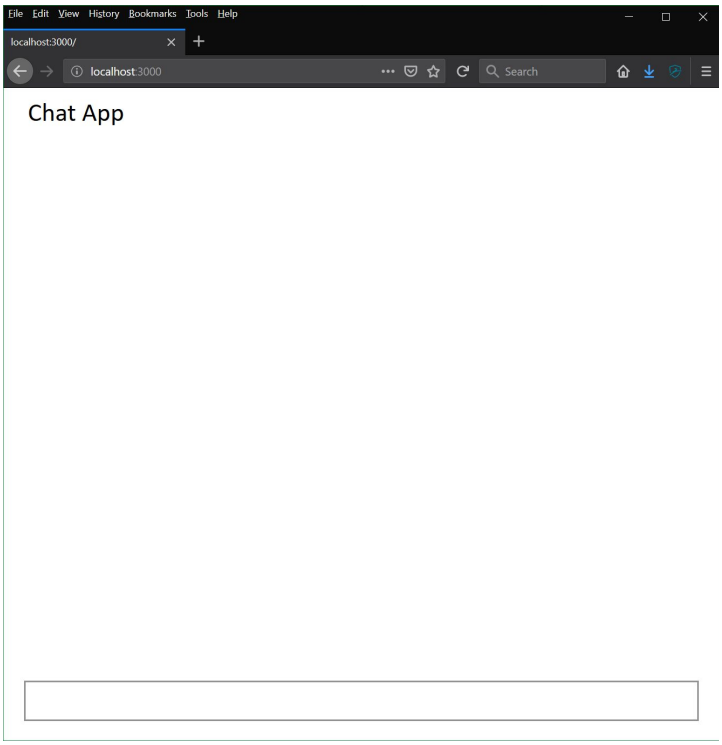
For this middleware, we use the **`req`** object to provide us information about the request, and **`next()`** to call the next handler in the middleware chain. Essentially, we can log the request without consuming it. This means that order in which we declare middleware matters! (Try switching the order of the two and running the server. What changes?)

Serving Dynamic Content

So far everything we've done has been regarding static, unchanging content. In order to support our chat app, we will need something more powerful so that the page can change responsively to user actions. First, download the two files `index.html` and `style.css` and add them to the `src` folder.

[Download the files from this gist.]

Restart the server, you should see the following:



Here we've provided the structure of the chat app. Everything is set in place, but alas when something is typed in no messages appear. Our first task here is to show our own messages. Since no communication is required for this, we can use a simple client-side script.

Create and open a new file `script.js` in the "src" directory, and copy the following code into it:

```
var inputElem = document.querySelector('.chatMessage');
var messages = document.querySelector('.messages');

function createHTMLMessage(msg, source){
    var li = document.createElement("li");
    var div = document.createElement("div");
    div.innerHTML += msg;
    div.className += "messageInstance " + source;
    li.appendChild(div);
    messages.appendChild(li);
}

inputElem.addEventListener('keypress', function (e) {
    var key = e.which || e.keyCode;
    if (key === 13) {
       createHTMLMessage(inputElem.value, 'client');
       inputElem.value = "";
    }
});
```

Explanation:

1. The `createHTMLMessage()` function is a helper that takes a message and a source (either "client" or "server"), and generates an html element for that single message.
2. `inputElem.addEventListener(...)` detects when "Enter" (key code 13) is pressed inside the bottom text box. When it is, is creates a client message using the inner text and adds it directly to the page. Finally, it clears the text field.

Test the updated page, and now we can see our own messages being populated in the list. However, there is still no communication between separate clients, so let's take care of that next.

The POST Request

There are couple of ways we can add dynamic content, so we'll start with the most basic way first which is the **POST** request. What makes POST different from GET is that the client can send parameters or other content through its body which can change the nature of the request, and the returned data does not have to be a static resource. Query results are often returned as **JSON** (JavaScript Object Notation). JSON is a generic representation of key-value pairs that can be nested, and thus can represent virtually any possible response data.

In order to make POST requests dynamically, we will be making use of `XHR`s (`XMLHttpRequests`) to make requests from JavaScript on the fly.

Add the following code to your **`script.js`** file:

```
var xhr = new XMLHttpRequest();
var nextIdx = 0;
// making a POST request to a given endpoint
function doPostRequest(endpoint, body, callback) {
    xhr.onreadystatechange = callback;
    xhr.open('POST', endpoint);
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.send(JSON.stringify(body));
}

function onPostResponse() {
    // checking response code (success)
    if (this.readyState === 4 && this.status === 200) {
        var response = JSON.parse(this.responseText);
        if (response.newMessages.length <= 0) {
            return;
        }
        for (var i = 0; i < response.newMessages.length - 1; ++i) {
            createHTMLMessage(response.newMessages[i], 'server');
        }
        var message = response.newMessages[response.newMessages.length -
1];
        var source = response.isLastClient ? 'client' : 'server';
        createHTMLMessage(message, source);
        nextIdx = response.nextIdx;
    }
}
```

In addition, replace the `createHTMLMessage()` line in `inputElem.addEventListener(...)` with the following:

```
doPostRequest('/newMsg', {
    msg: inputElem.value,
    nextIdx: nextIdx
}, onPostResponse);
```

Your code should look like the following:

```
var inputElem = document.querySelector('.chatMessage');
var messages = document.querySelector('.messages');
var xhr = new XMLHttpRequest();
var nextIdx = 0;

function createHTMLMessage(msg, source){
    var li = document.createElement("li");
    var div = document.createElement("div");
    div.innerHTML += msg;
    div.className += "messageInstance " + source;
    li.appendChild(div);
    messages.appendChild(li);
}

inputElem.addEventListener('keypress', function (e) {
    var key = e.which || e.keyCode;
    if (key === 13) {
        doPostRequest('/newMsg', {
            msg: inputElem.value,
            nextIdx: nextIdx
        }, onPostResponse);

        inputElem.value = "";
    }
});

function doPostRequest(endpoint, body, callback) {
    xhr.onreadystatechange = callback;
    xhr.open('POST', endpoint);
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.send(JSON.stringify(body));
}

function onPostResponse() {
    if (this.readyState === 4 && this.status === 200) {
        var response = JSON.parse(this.responseText);
        if (response.newMessages.length <= 0) {
            return;
        }
        for (var i = 0; i < response.newMessages.length - 1; ++i) {
            createHTMLMessage(response.newMessages[i], 'server');
        }
        var message = response.newMessages[response.newMessages.length - 1];
        var source = response.isLastClient ? 'client' : 'server';
        createHTMLMessage(message, source);
        nextIdx = response.nextIdx;
    }
}

```

Explanation:

1. First, we create the `XHR` object that we will use to make requests at the top of the script. Then we add a `doPostRequest()` function which makes it convenient to make POST requests.

   `xhr.onreadystatechange = callback;`
   Assigns the event handler which is called when the readyState for the XHR request changes.

   `xhr.open('POST', endpoint);`
   We then open the endpoint as a POST request.

   `xhr.setRequestHeader('Content-Type', 'application/json');`
   Sets the 'Content-Type' header to indicate JSON will be transmitted.

   `xhr.send(JSON.stringify(body));`
   Finally send the JSON body.

2. Next, we add an `onPostResponse()` callback that will wait until the request is done (readyState = 4) and is successful (status = 200). Then it will take any new messages the server sent and append them to the page as "server" chat bubbles. It will append the last message as a "client" chat bubble, since the server will send the message back.

3. Finally, we add a line in our `inputElem` handler to make the post request to the `'/newMsg'` endpoint using `doPostRequest()`.

We have now defined the **client-side** logic in `script.js` that makes POST requests and have added a callback that defines what to do once the server responds to our request. This will hook in the client side, and we can see that POST requests are being made to the server through the server's logs. After implementing the above, you should see this printed on your console.

```
Received POST request for resource /newMsg from ::1
```

However, the server side hasn't been implemented yet (you shouldn't see any change on the browser). We have added code to the client that makes requests haven't defined what happens when the server *receives* a POST request.

What we have so far: Client Makes Requests, Client can handle responses to POST requests
What we need: Server recognizes and responds to POST Requests.

We will add server-side logic to define how to handle POST requests next, but first we need an extra package. In order to parse the body as JSON, we need to use the body-parser package. Here is how we can add it to `server.js`:

1. Add the line

```
var bodyParser = require('body-parser');
```

   to the top of **`server.js`**.

2. Use the `body-parser` middleware, `bodyParser.json()`.
   It should be added above all other middleware we use. If you need a refresher on how to add middleware, check the end of the last unit (Hint: `app.use`).

Now that `body-parser` is added, we can handle our POST request. Add the following code to `server.js`. It should be added below our logging, but above the static resource middleware.

```
var messages = [];
app.post('/newMsg', function(req, res) {
    messages.push(req.body.msg);
    res.send({
        newMessages: messages.slice(req.body.nextIdx),
        nextIdx: messages.length,
        isLastClient: true
    });
});
```

The most important thing to note about the server-side code is that the client's request parameters are kept in **`req.body()`** as JSON that we can use immediately. Then we can return our own JSON back to the client.

This is what `server.js` should look like:

```
var express = require('express');
var http = require('http');
var bodyParser = require('body-parser');
var app = express();
var httpServer = http.createServer(app);
var messages = [];

app.use(bodyParser.json());

app.use(function(req, res, next) {
    console.log("Received", req.method,
                "request for resource", req.path,
                "from", req.ip);
    next();
});

app.post('/newMsg', function(req, res) {
    messages.push(req.body.msg);
    res.send({
        newMessages: messages.slice(req.body.nextIdx),
        nextIdx: messages.length,
        isLastClient: true
    });
});

app.use(express.static('src'));

httpServer.listen(3000, function(){
    console.log("Listening on port 3000");
});

```

Now everything is finally hooked together. Open two separate tabs or windows and try to send messages between windows.

Quick note on how the chat system is implemented -
The server has a `'messages'` array that stores every message sent so far. Each client keeps track of a variable `'nextIdx'` which stores the index of their next message. On receiving a request, the server returns a subarray of messages starting at `'nextIdx'` field included in the request. Note that this subarray will include the message just sent by the client, and other outstanding messages that the client might have not received yet. Try to go through the logic yourself and see how the clients and server keep track of the messages!

Awesome, well except the fact that messages only get received after a client sends a new message. Luckily, there is a simple solution: **polling**. In essence, all the client needs to do is manually check every so often to see of there is any updates.

Add another POST endpoint to `server.js`, similar to `'/newMsg'` with the following changes:

1. The endpoint should be `'/'`
2. No message should be added to our server's array (since we just want to respond with new data).
3. The server should respond with any new messages and the new index, similar to our old endpoint.
4. `isLastClient` should be false, since all new messages will come from the server

Then add the following code to `script.js`:

```
function poll() {
    doPostRequest('/', { nextIdx: nextIdx }, onPostResponse);
    setTimeout(poll, 5000);
}

setTimeout(poll, 5000);
```

Now when you test two separate windows again, they both update automatically every 5 seconds. This is definitely an improvement, but there is still some awkwardness about the method. When sending messages, there will be no contact for 5 seconds then a sudden burst of messages all at once. We could mitigate the issue by lowering the time between polls, but then how low can we make it before performance starts to degrade? There is a much better solution that fits this problem.

Web Sockets

Web Sockets are a much more responsive technology, since they keep a connection directly open at all times for each client. This allows either the client or server to directly notify each other when changes occur on their side. In contrast to the request / response model, the server does not wait for a request before responding. We will use the package **socket.io** to finish up the chat app.

First, **remove** the code in `script.js` and `server.js` that had to do with either POST requests or polling, since they will not be necessary anymore. (You could store them in a different file for reference).

Next, there are a number of things we need to do to set up sockets:

1. Add the following line to `index.html` **right above** the existing `<script>` tag (order is important)j:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.1.1/socket.io.js">
</script>
```

This line will simply import the socket.io package onto the front-end.

2. Add the following code to **`script.js`**:

```
var socket = io.connect('http://localhost:3000');
socket.on('connect', function(data) {
    socket.emit('join', 'Hello server from client');
});
```

The `socket.on()` method tells the package how to handle the `'connect'` event `"emitted"`, or sent, by the server. In our case, when the socket connects to the server, it emits a message of type `'join'` to the server, so that we can log the connection.

3. Finally, add the following code to the end of `server.js`

```
var io = require('socket.io')(httpServer);

// other middleware from earlier goes here

var numClients = 0;
io.on('connection', function(client) {
    console.log('Client', numClients++, 'connected.');

    client.on('join', function(data) {
        console.log(data);
    });
});
```

Which will set up the server side to listen for incoming socket connections, and log them once connected. The `'connection'` event is built in when a new socket connects to the server, where we just add the number of connections and log it. The 'join' event is custom defined by our application so that the client can send us unique data on connection.

After doing all the setup above, your code should overall look something like this:

index.html

```
<html>
    <head>
        <link rel="stylesheet" href="style.css" />
    </head>
    <body>
        <div class="mainApp">
            <h1> Chat App </h1>
            <ul class="messages"></ul>
            <input class="chatMessage" />
        </div>
    </body>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/
    socket.io/2.1.1/socket.io.js"></script>
    <script src="script.js"></script>
</html>
```

script.js

```
var inputElem = document.querySelector('.chatMessage');
var messages = document.querySelector('.messages');
var socket = io.connect('http://localhost:3000');

function createHTMLMessage(msg, source){
    var li = document.createElement("li");
    var div = document.createElement("div");
    div.innerHTML += msg;
    div.className += "messageInstance " + source;
    li.appendChild(div);
    messages.appendChild(li);
}

inputElem.addEventListener('keypress', function (e) {
    var key = e.which || e.keyCode;
    if (key === 13) {
        inputElem.value = "";
    }
});

socket.on('connect', function(data) {
    socket.emit('join', 'Hello server from client');
});

```

server.js

```
var express = require('express');
var http = require('http');
var app = express();
var httpServer = http.createServer(app);
var io = require('socket.io')(httpServer);

app.use(function(req, res, next) {
    console.log("Received", req.method,
                "request for resource", req.path,
                "from", req.ip);
    next();
});

app.use(express.static('src'));

httpServer.listen(3000, function(){
    console.log("Listening on port 3000");
});

var numClients = 0;
io.on('connection', function(client) {
    console.log('Client', numClients++, 'connected.');

    client.on('join', function(data) {
        console.log(data);
    });
});

```

Test out the page again and check the console. Now we should see logs of the socket connections being made appear when a new page is connected:

```
> chatapp@1.0.0 start /mnt/c/Users/Kanurame/Desktop/chatapp
> node server.js

Listening on port 3000
Received GET request for resource / from ::1
Received GET request for resource /style.css from ::1
Received GET request for resource /script.js from ::1
Client 0 connected.
Received GET request for resource /favicon.ico from ::1
Hello server from client
Received GET request for resource / from ::1
Received GET request for resource /style.css from ::1
Received GET request for resource /script.js from ::1
Client 1 connected.
Received GET request for resource /favicon.ico from ::1
Hello server from client
```

Now all that's left to do is hook up the real time updates to send messages! In `script.js`, we can add a handler for 'chat msg' events emitted by other clients:

```
socket.on('chat msg', function(msg) {
    createHTMLMessage(msg, 'server');
});
```

And add the following code in `inputElem.addEventListener(...)` directly below the if statement, where we previously had a call to `doPostMessage()`:

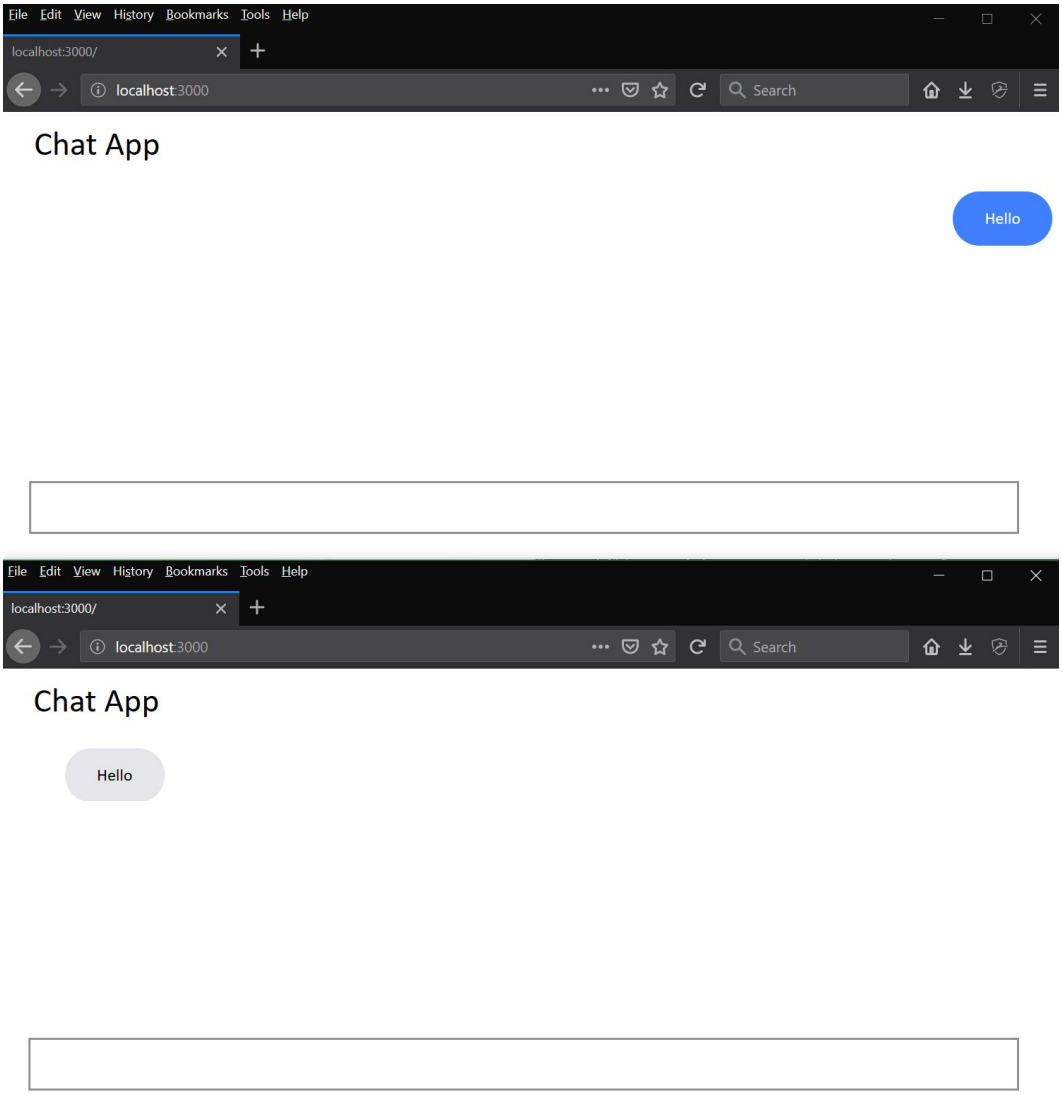
```
createHTMLMessage(inputElem.value, 'client');
socket.emit('chat', inputElem.value);
```

The above `socket.on('chat msg', ...)` call tells the socket.io how to handle 'chat msg' events emitted by the server. In our case, we just want to add them to the page as 'server' messages. Then, the `socket.emit()` code just tells the client to add the message locally to the front-end, then dispatch the event to the server so all other clients can receive the message. How the server does that is defined below.

In `server.js`, we just need to add a handler for the chat action that will broadcast to all other open sockets. This should be added right next to the `client.on('join', ...)` (inside the callback to `io.on('connection'...)`), since it needs to be set per client on connection.

```
client.on("chat", function(msg) {
            console.log(msg);
            client.broadcast.emit('chat msg', msg);
});
```

Now everything should be setup, and ready to chat! Open up the page in 2 windows and test that you are able to send messages instantly between them.



Completion Checkoff

- Please fill out this Google form to let us know how we can improve this lab in the future! https://goo.gl/forms/7ksF3mQOvj7Gs1Up2

- Make sure you understand the different ways to use express, and the different ways of serving dynamic content (such as POST requests, polling, and sockets).

- Open up two windows such as the screenshot above and ensure messages can be sent between the two. A staff member will verify the above works before giving the checkoff.