Exercise #1:

Create a Lab13 folder in your repo and a new file SimpleServer.js and copy the following into this file:

var http = require('http');

//create a server object:

http.createServer(function (req, res) {

console.log(req.headers); //output the request headers to the console

res.writeHead(200, { 'Content-Type': 'text/html' }); // set MIME type to HTML

res.write(`<h1>The server date is: ${Date.now()}</h1>`); //send a response to the client

res.write('<h1>The client date is: <script>document.write( Date.now() );</script></h1>'); // send another response

res.end(); //end the response

}).listen(8080); //the server object listens on port 8080

console.log('Hello world HTTP server listening on localhost port 8080');

Open a terminal and run it using node.js. Open a browser and make a request to localhost:8080

# The server date is: 1604005761922

# The client date is: 1604005763451

a) Why is this a “dynamic” web page? **Because this page will be dynamically changed at the time its requested and after it has loaded and every time a request is made the page creates a new string for the dates.** What is processed on the client and what is processed on the server? **The server Date.now is executed by created by the server & the client Date.now is processed on the client side.** Why are the dates different even though they are obtained from the same clock (on the same machine)? **Because the server date is taken before the page is loaded and the client date is after that.** What would happen if the server was run on a different machine than the browser? **There is a larger difference in the times between server & client.**

b) Does the request matter? **No,** **Because no matter what request we will get the same output on the page because that is what we have programmed it to do.** How is the route for the HTTP request handled?

c) When you do a “view source” in the browser, where did the Javascript code executed on the server go ? Why isn’t it inside a <script> tag? **Because view source only shows the page source and the JS code on the server is located on a separate file/page.**

**NOTE** You may wish to install and use the npm application nodemon to automatically restart node after changes are saved in a file.

Exercise 2: A Better Server Side for Web Applications

a. To do server side processing for a Web application we must set up the server to handle any request. Express makes setting up a web server for a web application easy. Install Express using npm install express in your terminal. Let’s start things by setting up a server that will respond to any HTTP request by sending back the type of request and the URL path from the request. Put the following code in a file called info\_server\_Ex1a.js:

var express = require('express');

var app = express();

app.all('\*', function (request, response, next) {

response.send(request.method + ' to path ' + request.path);

});

app.listen(8080, () => console.log(`listening on port 8080`)); // note the use of an anonymous function here

In the terminal, start the server node info\_server\_Ex1a.js and then in a browser try various URLs such as localhost:8080 and localhost:8080/xxx/yyy. What is the response you get and why? **GET to path /xxx/yyy**

Explain how express handles routing an HTTP request on the port it is listening to.

**In our code we put in there for any path and any route ..**

b. Add a route to match with a GET request to the path test. Put this above the app.all() route. Test it with and without test.Explain why the app.all() route does not get handled anymore. Now move it below app.all() to and verify you get the expected response. Now add next(); after the response in app.all() and explain why you get an error **in the console output. Got an error because the response was already done. Only one thing can respond at a time.**

Now change the response.send() to console.log() in app.all() and explain why this no longer throws an error. It is recommended that you put this app.all() code in when you are developing a web app to see what requests the server is receiving. **Because now the server is writing in the console instead of responding.**

c. Now we will enable the server to respond to request for static files (files that are not intended to have any server-side processing) that are located in a directory called public (this is often called the “document root” directory). Make a copy of info\_server\_Ex1a.js and name it info\_server\_Ex1c.js

Before the app.listen add the following add the following:

app.use(express.static('./public'));

Terminate the previously running server (with ctrl-C) and run info\_server\_Ex1c.js. Create a simple html file hello.html that outputs <h1>Hello from <your name>!<h1> and save it in the public directory. Use a browser with the following URL http://localhost:8080/hello.html and see what response you get. Try localhost:8080/xxx and explain what response you get. Make a copy of hello.html and rename it hello.txt. Now try localhost:8080/hello.txt and explain the response. **See video at 20 minutes from end**

Do you think the app.use() middleweare must be placed at the bottom of the routing functions? **It would depend as the teacher said in class. Whatever is placed first is what will respond. If it is after another response it will not get the chance to respond.**

d. Copy your order\_page.html from Exercise 4 in Lab 12 to the public directory. Change the action attribute of the form to localhost:8080/process\_form. Make sure method="POST" in the form tag. Try http://localhost:8080/order\_page.html in your browser and submit the form. Look at the console.log output and the response you get and explain.

Exercise 3: Server Side Processing of Form Action

a. Make a copy of info\_server\_Ex1c.js and name it info\_server\_Ex2.js and add the following code after the app.all() statement:

app.use(myParser.urlencoded({ extended: true }));

app.post("/process\_form", function (request, response) {

let POST = request.body;

response.send(POST);

});

Unfortunately Express does not provide functions to decode the body of an HTTP request so you will need to write this yourself or add one. Fortunately there’s a good one available! In your terminal (and be sure you are in the directory where info\_server\_Ex2.js is located), type npm install body-parser. After the installation is complete, in info\_server\_Ex2.js add the statement var myParser = require("body-parser"); after the var app = express(); statement. Now run info\_server\_Ex2.js. Try localhost:8080/order\_page.html and verify that that you get the post data from the what you typed into the textbox after submitting the form. If this works, you have just done server-side processing of a POST from a web form. Congratulations!

b. In order\_page.html cut the code that checks query string for quantity\_textbox into info\_server\_Ex2.js (the if (typeof GET['quantity\_textbox'] != 'undefined')) statement) and paste it **over** the response.send(POST); statement (i.e. delete the statement) in app.post() and change GET to POST. Copy the function isNonNegInt() from order\_page.html and paste it into the top or bottom of app.post(). Change the document.write(`Thank you for purchasing ${q} things!`) to response.send(`Thank you for purchasing ${q} things!`); and delete window.stop(); and an else response.send(`${q} is not a quantity! Press the back button and try again.`); to the if-statement. Try localhost:8080/order\_page.html with valid and invalid quantities and verify that it works as expected.

Exercise 4: Processing Forms With MVC

Often you need to do complex processing and responses to form submissions. A good design practice to help manage this is to use model-view-controller (MVC). Express supports this design practice explicitly, but we will implement out own basic MVC to help understand how to use it effectively.

a. Make a copy of info\_server\_Ex2.js and name it info\_server\_Ex3.js. Create a views directory and add a file called display\_quantity\_template.view and put in it <h1>Thank you for purchasing ${q} things!</h1> that is to be used as a JS string template to render the HTML response to the form submission. In info\_server\_Ex4.js replace the if-statement with

if (typeof POST['quantity\_textbox'] != 'undefined') {

let q = POST['quantity\_textbox'];

if (isNonNegInt(q)) {

var contents = fs.readFileSync('./views/display\_quantity\_template.view', 'utf8');

response.send(eval('`' + contents + '`')); // render template string

} else {

response.send(`${q} is not a quantity!`);

}

}

add the statement var fs = require('fs'); to the top of the file. Try http://localhost:8080/order\_page.html and explain how this is different than the form processing you did in Exercise 2. Because now we are using these views

b. It’s best not to clutter up the server code with the details of responding to requests. We can always create functions and move the code out of the way or put them in a file and load it in as a library. In info\_server\_Ex3.js cut and paste all the code under let POST = request.body; and move it into a function called process\_quantity\_form (POST, response). replace all the code in app.post() with process\_quantity\_form(request.body, response);

c. You often have to use the same data in multiple places. It’s always best to have one central source for shared data rather than duplicate sources. For web applications the common way to handle this is providing a data service on the server. The data source can be a code, a file, accessing a database or more generally another server (such as a directory server). Let’s implement a basic data service for our web application that shares product information for use in a form and the processing the form. For JSON data an easy way to do this is put the JSON into a file as a variable and load it in as a module. In your ./public directory, create a file called product\_data.js and put the following data in it:

products =

[

{

"model":"Apple iPhone XS",

"price": 990.00

},

{

"model":"Samsung Galaxy",

"price": 240.00

}

];

if(typeof module != 'undefined') {

module.exports.products = products;

}

At the top of info\_server\_Ex3.js put

var data = require('./public/product\_data.js');

var products = data.products;

and in the process\_quantity\_form() function add

let model = products[0]['model'];

let model\_price = products[0]['price'];

Replace the template in display\_quantities\_template.view with <h2>Thank you for purchasing ${q} ${model}. Your total is \$${q \* model\_price}!</h2> and now try out different quantities.

In the head tag of order\_page.html add the line <script src="./product\_data.js"></script>. Note that the path does NOT include the public directory. Why is this? **Because it is already done by the server because we referenced it in our server code.**

Before the form (or in the top of the form) add

document.write(`<h3>${products[0]["model"]} at \$${products[0]["price"]}</h3>`);

Reload order\_page.html and verify that the Apple iPhone XS product is used in both the form and response to processing that form. Now change some of the information for the 0th element in product\_data.js and verify that both order\_page.html and info\_server\_Ex3.js use the updated information. **How would you change the product being used here? You would alter the information in the products JSON file and restart the server and reload the page and it should be updated.**

Bonus Exercise 5: Processing multiple inputs

Let’’s make it possible to select quantities of a product from the shared products data. Copy info\_server\_Ex3.js and name it info\_server\_Ex4.js. Copy order\_page.html and rename it order\_page\_Ex4.html.

Task 1: Make order\_page\_Ex4.html display inputs for all products in product\_data.js. Reaplce the <form> element with

<form name='quantity\_form' action="http://localhost:8080/process\_form" method="POST">

<script>

for (i in products) {

document.write(`<h3>${products[i]["model"]} at \$${products[i]["price"]}</h3>`);

document.write(`

<label for="quantity\_textbox">quantity desired:</label>

<input type="text" name="quantity\_textbox${i}" onkeyup="checkQuantityTextbox(this);">

<span id="quantity\_textbox${i}\_message">Enter a quantity</span>

`);

}

</script>

<br>

<input type="submit" value='Purchase' name="purchase\_submit\_button">

</form>

Now replace the checkQuantityTextbox() with

function checkQuantityTextbox(theTextbox) {

errs = isNonNegInt(theTextbox.value, true);

document.getElementById(theTextbox.name + '\_message').innerHTML = errs.join(", ");

}

Make sure you understand what changes were made and why!

Task 2: Process multiple quantities of products from the submitted form. In info\_server\_Ex4.js replace process\_quantity\_form() with

function process\_quantity\_form (POST, response) {

if (typeof POST['purchase\_submit\_button'] != 'undefined') {

var contents = fs.readFileSync('./views/display\_quantities\_template.view', 'utf8');

receipt = '';

for(i in products) {

let q = POST[`quantity\_textbox${i}`];

let model = products[i]['model'];

let model\_price = products[i]['price'];

if (isNonNegInt(q)) {

receipt += eval('`' + contents + '`'); // render template string

} else {

receipt += `<h3><font color="red">${q} is not a valid quantity for ${model}!</font></h3>`;

}

}

response.send(receipt);

response.end();

}

}

Run info\_server\_Ex4.js and try entering quantities for the products displayed! Do you see how things are connected through the shared data in product\_data.js? **Yes, the pages use the information stored in the products\_data and then the POST uses the information entered by the client in combination to display the receipt.**