EECS 368
Programming Language Paradigms

David O. Johnson

Fall 2022

Reminders

- Assignment 4 due: 11:59 PM, Monday, October 17
- Assignment 5 due: 11:59 PM, Monday, October 31

Any Questions?

In-Class Problem Solution

- 20-(10-12) In-Class Problem Solution.pptx

Any Questions?

Chapter 20 – Node.js

- ~~Background~~
- ~~The node command~~
- ~~Modules~~
- ~~Installing with NPM~~
- ~~Package files~~
- ~~Versions~~
- The file system module
- The HTTP module
- Streams
- A file server

The File System Module

- One of the most commonly used built-in modules in Node is the fs module, which stands for file system.
- It exports functions for working with files and directories.

- For example, the function called readFile reads a file (file.txt) and then calls a callback with the file's contents.

```
// get the readFile function from the fs module
let {readFile} = require("fs");
readFile("file.txt", "utf8", (error, text) => {
   if (error) throw error;
   // print out the contents of file.txt
   console.log("The file contains:", text);
});
```

readFile & Text Files

- The second argument to readFile indicates the character encoding used to decode the file into a string.
- There are several ways in which text can be encoded to binary data, …
- but most modern systems use UTF-8.
- So unless you have reasons to believe another encoding is used, pass "utf8" when reading a text file.

```
// get the readFile function from the fs module
let {readFile} = require("fs");
readFile("file.txt", "utf8", (error, text) => {
   if (error) throw error;
   // print out the contents of file.txt
   console.log("The file contains:", text);
});
```

readFile & Binary Files

- If you do not pass an encoding, …
- Node will assume you are interested in the binary data and …
- will give you a Buffer object instead of a string.
- This is an array-like object that contains numbers representing the bytes (8-bit chunks of data) in the files.

```
// get the readFile function from the fs module
// Note: This is another way to do it
const {readFile} = require("fs");
readFile("file.txt", (error, buffer) => {
   if (error) throw error;
   console.log("The file contained", buffer.length, "bytes.", "The first byte is:", buffer[0]);
});
```

writeFile

- A similar function, writeFile, is used to write a file to disk.
- Here it was NOT necessary to specify the encoding.
- writeFile will assume that when it is given a string to write, …
- rather than a Buffer object, …
- it should write it out as text using its default character encoding (UTF-8).

```
// get the writeFile function from the fs module
const {writeFile} = require("fs");
writeFile("graffiti.txt", "Node was here", err => {
   if (err) console.log(`Failed to write file: ${err}`);
   else console.log("File written.");
});
```

writeFile & readFile Example

```
davidojo@cycle1:~$ node
> const {writeFile} = require("fs");
> writeFile("graffiti.txt", "Node was here", err => {
... if (err) console.log(`Failed to write file: ${err}`);
... else console.log("File written.");
... });
> File written.
```

creates file graffiti.txt with “Node was here”

```
> let {readFile} = require("fs");
> readFile("graffiti.txt", "utf8", (error, text) => {
... if (error) throw error;
... console.log("The file contains:", text);
... });
> The file contains: Node was here
```

reads graffiti.txt as text and prints out “Node was here”

```
> readFile("graffiti.txt", (error, buffer) => {
... if (error) throw error;
... console.log("The file contained", buffer.length, "bytes.",
..... "The first byte is:", buffer[0]);
... });
> The file contained 13 bytes. The first byte is: 78
```

reads graffiti.txt as binary and prints out hex value of first byte, the character “N”

Other fs Functions

- The fs module contains many other useful functions:
  - readdir will return the files in a directory as an array of strings
  - stat will retrieve information about a file
  - rename will rename a file
  - unlink will remove one
- See the documentation at https://nodejs.org for specifics.

fs with Promises

- Most of the fs functions take a callback function as the last parameter, …
- which they call either with an error (the first argument) or …
- with a successful result (the second).
- As we saw earlier, there are downsides to this style of programming.
- The biggest one being that error handling becomes verbose and error-prone.

- Though promises have been part of JavaScript for a while, …
- at the time of writing Node.js they were still a work in progress.
- There is an object promises exported from the fs package since version 10.1 that contains most of the same functions as fs but uses promises rather than callback functions.

```
const {readFile} = require("fs").promises;
readFile("file.txt", "utf8")
   .then(text => console.log("The file contains:", text));
```

Synchronous fs

- Sometimes you don't need asynchronicity, and it just gets in the way.
- Many of the functions in fs also have a synchronous variant, …
- which has the same name with Sync added to the end.
- For example, the synchronous version of readFile is called readFileSync.

```
const {readFileSync} = require("fs");
console.log("The file contains:", readFileSync("file.txt", "utf8"));
```

- Do note that while such a synchronous operation is being performed, …
- your program is stopped entirely.
- If it should be responding to the user or to other machines on the network, …
- being stuck on a synchronous action might produce annoying delays.

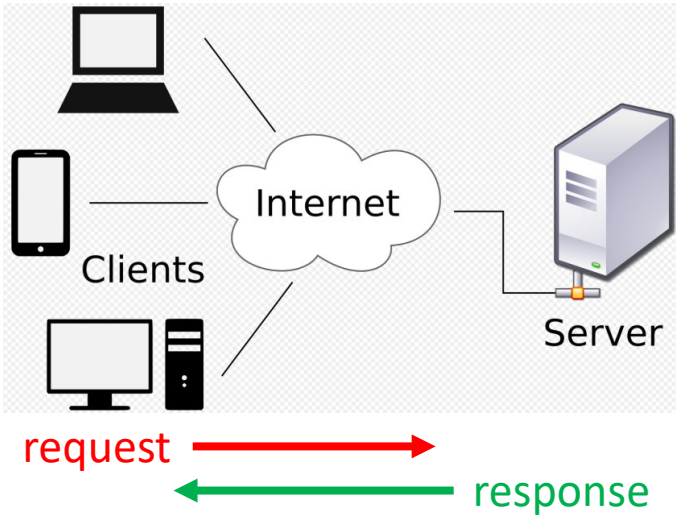
Any Questions?

The HTTP Module

- Another central module is called http.
- It provides functionality for running HTTP servers and making HTTP requests.

- This is all it takes to start an HTTP server:

```
const {createServer} = require("http");
let server = createServer((request, response) => {
   response.writeHead(200, {"Content-Type": "text/html"});
   response.write(`
      <h1>Hello!</h1>
      <p>You asked for <code>${request.url}</code></p>`);
   response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

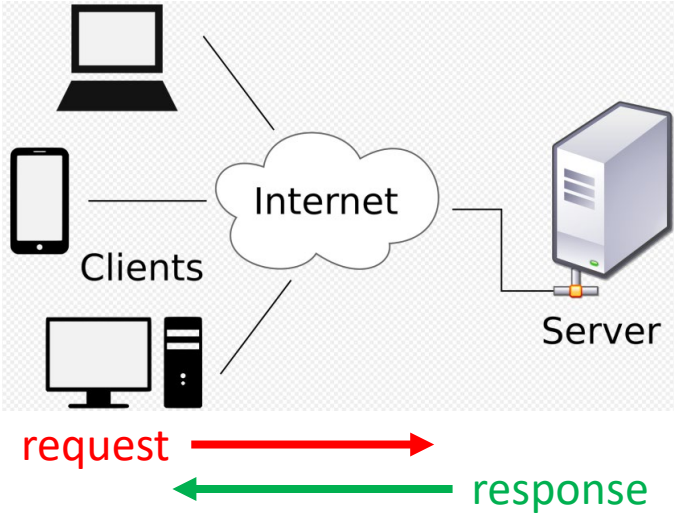


- If you run this script on your own machine, …
- you (the client) can point your web browser at http://localhost:8000/hello to make a request to your server.
- It will respond with a small HTML page.

Creating an HTTP Server

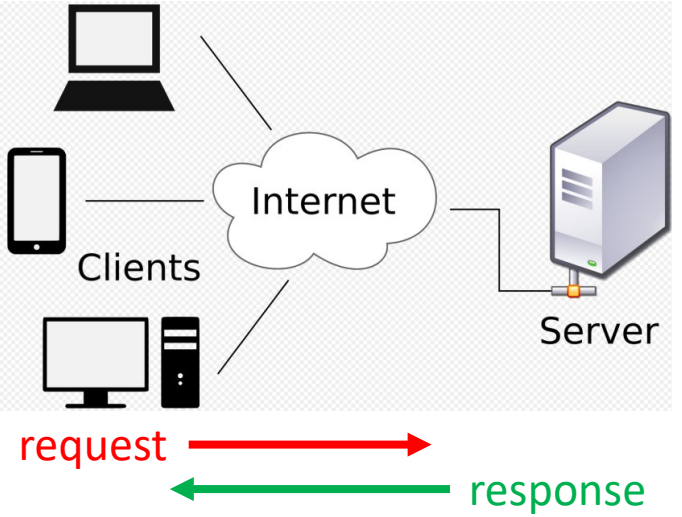
- The function (=>), passed as an argument, to createServer is called every time a client connects to the server.
- The request (to server) and response (to client) bindings are objects representing the incoming (from client) and outgoing (from server) data.
- The first contains information about the request (to the server), …
- such as its url property, which tells the server to what URL the request was made.

```
const {createServer} = require("http");
let server = createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`
    <h1>Hello!</h1>
    <p>You asked for <code>${request.url}</code></p>`);
  response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```



HTTP Server Response

- So, when you (the client) opens that page in your browser, …
- it (the server) sends a request to your own computer (the client).
- This causes the server function to run and …
- send back a response, which you (the client) can then see in the browser.




```
const {createServer} = require("http");
let server = createServer((request, response) => {
   response.writeHead(200, {"Content-Type": "text/html"});
   response.write(`
      <h1>Hello!</h1>
      <p>You asked for <code>${request.url}</code></p>`);
   response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

Response from server

```
HTTP/1.1 200 OK
Content-Type: text/html

<!doctype html>
<h1>Hello!</h1>
<p>You asked for
<code>${request.url}</code></p>
```

HTTP Server Response

- To send something back, the server calls methods on the response object.
- The first, writeHead, will write out the response headers.
- The server gives it the status code (200 for "OK" in this case) and …
- an object that contains header values.
- The example sets the Content-Type header to inform the client that the server will be sending back an HTML document.

```
const {createServer} = require("http");
let server = createServer((request, response) => {
   response.writeHead(200, {"Content-Type": "text/html"});
   response.write(`
      <h1>Hello!</h1>
      <p>You asked for <code>${request.url}</code></p>`);
   response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

Response from server

```
HTTP/1.1 200 OK
Content-Type: text/html

<!doctype html>
<h1>Hello!</h1>
<p>You asked for
<code>${request.url}</code></p>
```

HTTP Server Response

- Next, the actual response body (the document itself) is sent with response.write.
- You are allowed to call this method multiple times if you want to send the response piece by piece, …
- for example to stream data to the client as it becomes available.
- Finally, response.end signals the end of the response.

```
const {createServer} = require("http");
let server = createServer((request, response) => {
   response.writeHead(200, {"Content-Type": "text/html"});
   response.write(`
      <h1>Hello!</h1>
      <p>You asked for <code>${request.url}</code></p>`);
   response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

Response from server

```
HTTP/1.1 200 OK
Content-Type: text/html

<!doctype html>
<h1>Hello!</h1>
<p>You asked for
<code>${request.url}</code></p>
```

HTTP Server Response

- The call to server.listen causes the server to start waiting for connections on port 8000.
- This is why you have to connect to localhost:8000 to speak to this server, …
- rather than just localhost, …
- which would use the default port 80.

```
const {createServer} = require("http");
let server = createServer((request, response) => {
   response.writeHead(200, {"Content-Type": "text/html"});
   response.write(`
      <h1>Hello!</h1>
      <p>You asked for <code>${request.url}</code></p>`);
   response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

Response from server

```
HTTP/1.1 200 OK
Content-Type: text/html

<!doctype html>
<h1>Hello!</h1>
<p>You asked for
<code>${request.url}</code></p>
```

HTTP Server Response

- When you run this script, the process just sits there and waits.
- When a script is listening for events …
- in this case, network connections …
- node will not automatically exit when it reaches the end of the script.
- To close it, press control-C.

```
const {createServer} = require("http");
let server = createServer((request, response) => {
   response.writeHead(200, {"Content-Type": "text/html"});
   response.write(`
      <h1>Hello!</h1>
      <p>You asked for <code>${request.url}</code></p>`);
   response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

Response from server

```
HTTP/1.1 200 OK
Content-Type: text/html

<!doctype html>
<h1>Hello!</h1>
<p>You asked for
<code>${request.url}</code></p>
```

HTTP Server Response

- A real web server usually does more than the one in the example.
- It looks at the request's method (request.method) to see what action the client is trying to perform and …
- looks at the request's URL (request.url) to find out which resource this action is being performed on.

```
const {createServer} = require("http");
let server = createServer((request, response) => {
   response.writeHead(200, {"Content-Type": "text/html"});
   response.write(`
      <h1>Hello!</h1>
      <p>You asked for <code>${request.url}</code></p>`);
   response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

Response from server

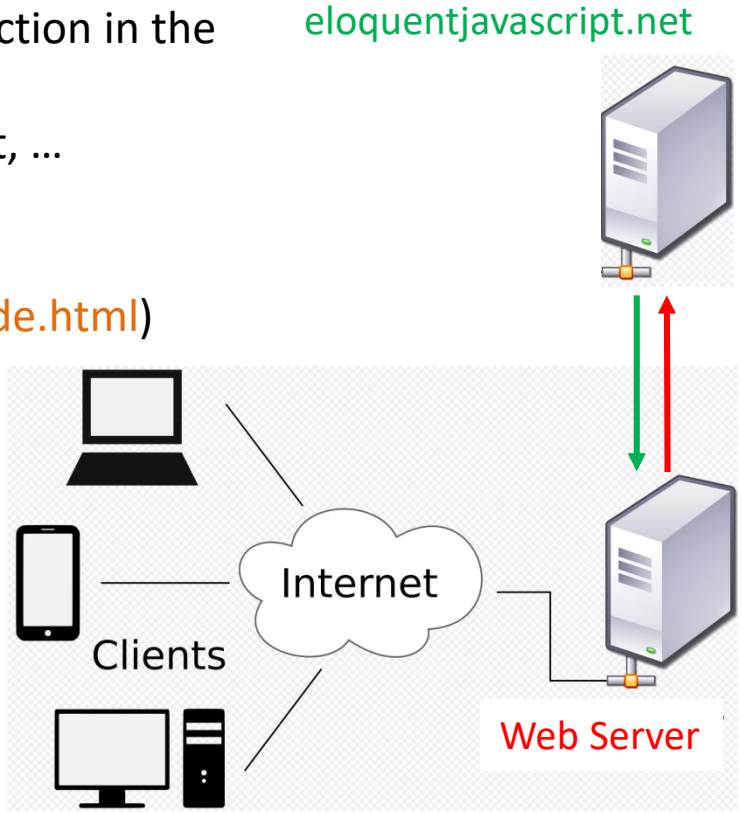
```
HTTP/1.1 200 OK
Content-Type: text/html

<!doctype html>
<h1>Hello!</h1>
<p>You asked for
<code>${request.url}</code></p>
```

Any Questions?

HTTP Client Request

- To act as an HTTP client, we can use the request function in the http module.
- The first argument to request configures the request, …
- telling Node:
  - what server to talk to (eloquentjavascript.net)
  - what path to request from that server (/20_node.html)
  - which method to use (GET)
  - and so on (headers: {Accept: "text/html"})



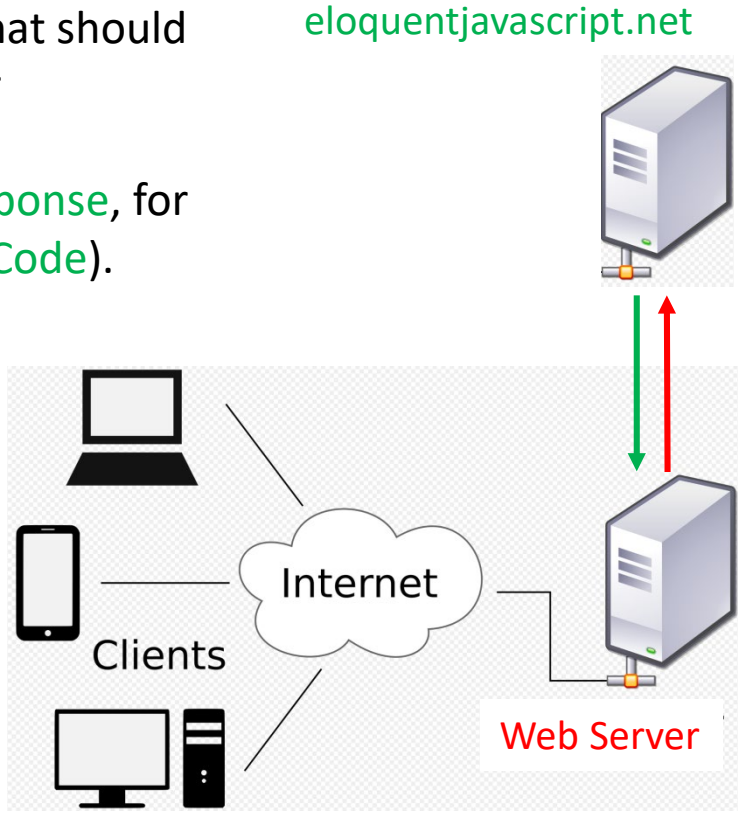

```
const {request} = require("http");
let requestStream = request({
   hostname: "eloquentjavascript.net",
   path: "/20_node.html",
   method: "GET",
   headers: {Accept: "text/html"}
}, response => {
   console.log("Server responded with status code",
                  response.statusCode);
});
requestStream.end();
```

Request from client

```
GET /20_node.html HTTP/1.1
Host: eloquentjavascript.net
Accept: text/html
```

HTTP Client Request

- The second argument is the callback function (=>) that should be called when a response comes in from the server (eloquentjavascript.net).
- It is given an object that allows us to inspect the response, for example to find out its status code (response.statusCode).




```
const {request} = require("http");
let requestStream = request({
   hostname: "eloquentjavascript.net",
   path: "/20_node.html",
   method: "GET",
   headers: {Accept: "text/html"}
}, response => {
   console.log("Server responded with status code",
                  response.statusCode);
});
requestStream.end();
```

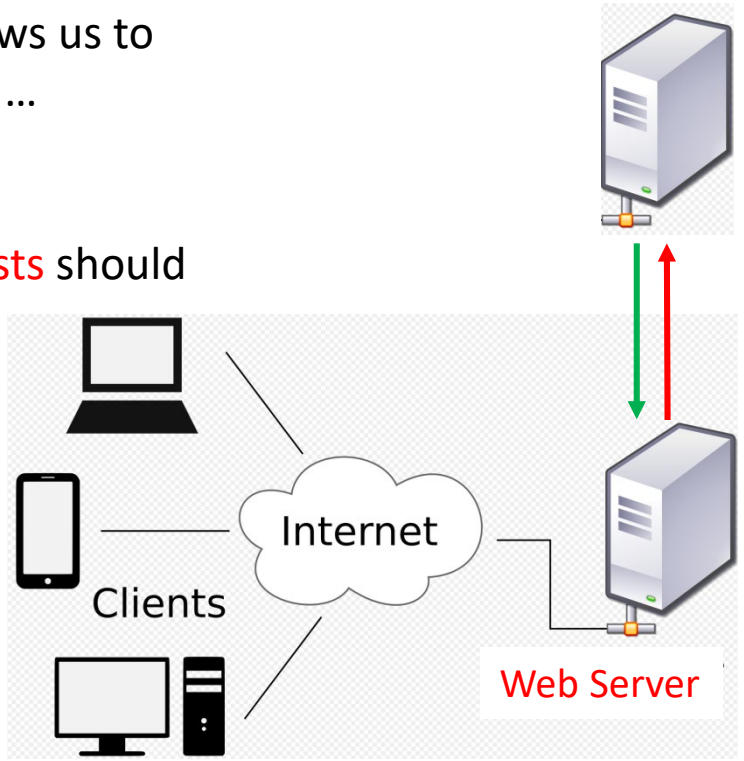
Request from client

```
GET /20_node.html HTTP/1.1
Host: eloquentjavascript.net
Accept: text/html
```

HTTP Client Request

- Just like the response object we saw in the server, …
- the object returned by request (requestStream) allows us to stream data into the request with the write method …
- and finish the request with the end method (requestStream.end).
- The example does not use write because GET requests should not contain data in their request body.

eloquentjavascript.net




```
const {request} = require("http");
let requestStream = request({
   hostname: "eloquentjavascript.net",
   path: "/20_node.html",
   method: "GET",
   headers: {Accept: "text/html"}
}, response => {
   console.log("Server responded with status code",
                  response.statusCode);
});
requestStream.end();
```

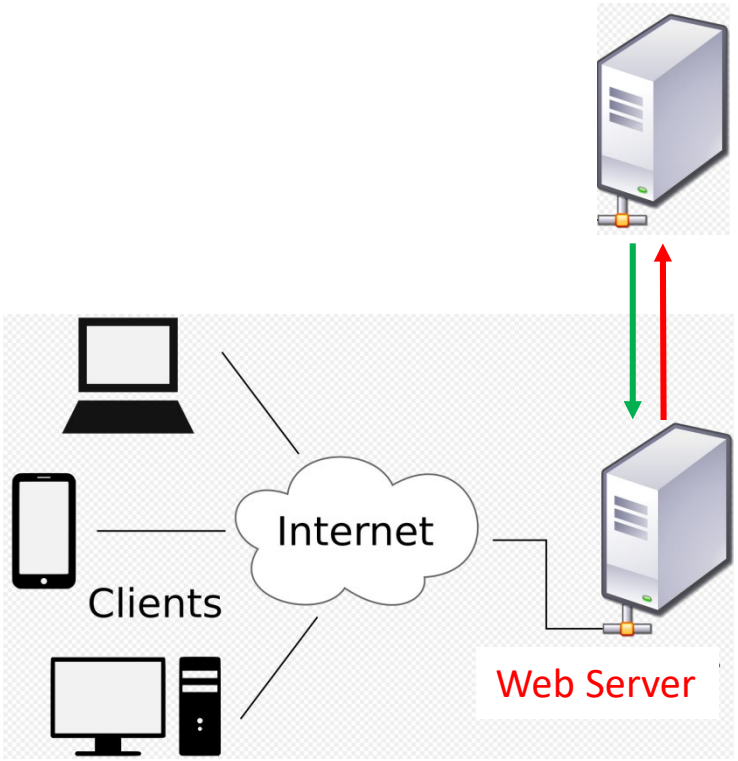
Request from client

```
GET /20_node.html HTTP/1.1
Host: eloquentjavascript.net
Accept: text/html
```

HTTP Client Request

- There's a similar request function in the https module that can be used to make requests to https URLs.

eloquentjavascript.net

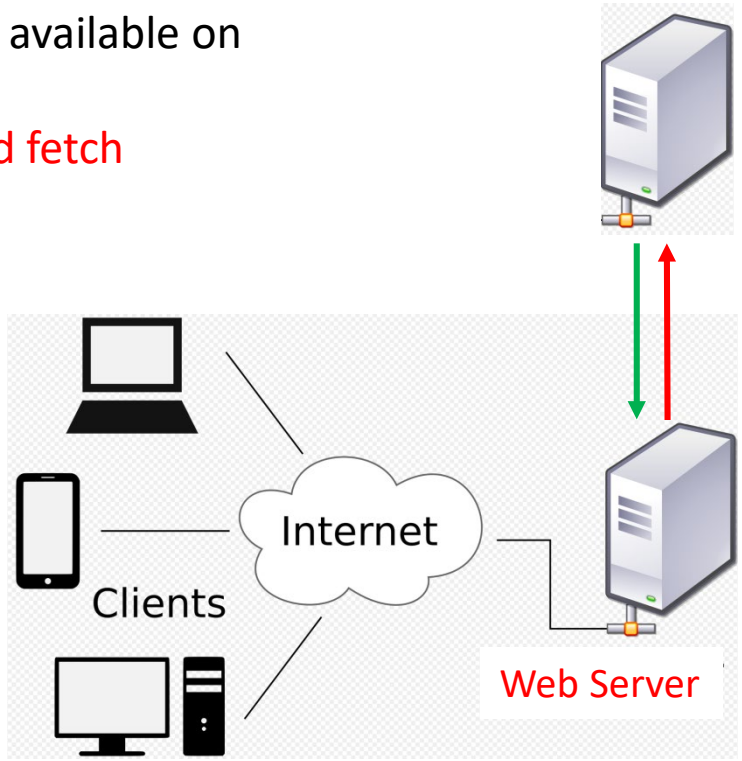



```
const {request} = require("https");
let requestStream = request({
    hostname: "eloquentjavascript.net",
    path: "/20_node.html",
    method: "GET",
    headers: {Accept: "text/html"}
}, response => {
    console.log("Server responded with status code",
                response.statusCode);
});
requestStream.end();
```

HTTP Client Request (node-fetch)

- Making requests with Node's raw functionality is rather verbose.
- There are much more convenient wrapper packages available on NPM.
- For example, node-fetch provides the promise-based fetch interface that we know from the browser.

eloquentjavascript.net




```
const {request} = require("https");
let requestStream = request({
    hostname: "elo      tjavasc      net",
    path: "/20_node.    ",
    method: "GET",
    headers: {Accept: "t      l"}
}, response => {
    console.log("Serv      spo      with status code",
                res      e.statu      e);
});
requestStream.end();
```

Request from client

```
GET /20_node.html HTTP/1.1
Host: eloquentjavascript.net
Accept: text/html
```

Any Questions?

In-Class Problem

1. What is the standard output for the JavaScript code to the right?
2. Add comments to each line of the code to the right.

```
const {writeFile} = require("fs");
writeFile("graffiti.txt", "I promise the Node was here", err => {
   if (err) console.log(`Failed to write file: ${err}`);
   else console.log("File written.");
});

const {readFile} = require("fs").promises;
readFile("graffiti.txt", "utf8")
   .then(text => console.log("The file contains:", text));
```

3. Assuming the file 20_node.html existed on eloquentjavascript.net, what is the LINUX terminal output for the JavaScript code to the right?
4. Add comments to each line of the code to the right.

```
const {request} = require("https");
let requestStream = request({
   hostname: "eloquentjavascript.net",
   path: "/20_node.html",
   method: "GET",
   headers: {Accept: "text/html"}
}, response => {
   console.log("Server responded with status code",
                  response.statusCode);
});
requestStream.end();
```