EECS 368
Programming Language Paradigms

David O. Johnson

Fall 2022

Reminders

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

Any Questions?

In-Class Problem Solution

- 21-(10-14) In-Class Problem Solution.pptx

Any Questions?

Chapter 20 – Node.js

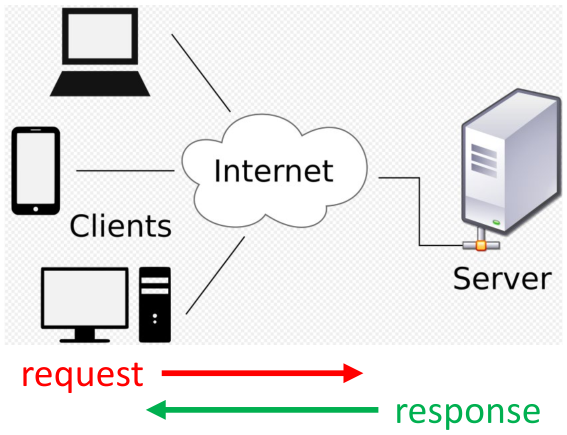
- ~~Background~~
- ~~The node command~~
- ~~Modules~~
- ~~Installing with NPM~~
- ~~Package files~~
- ~~Versions~~
- ~~The file system module~~
- ~~The HTTP module~~
- Streams
- A file server (Assignment 5)
  - Starting the server
  - Reading a file
  - Deleting a file
  - Writing a file

Streams

We have seen two instances of writable streams in the HTTP examples:

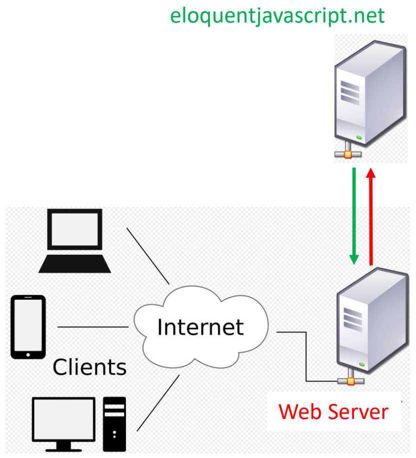
the response object that the server could write to

```
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)");
```



the request object that was returned from request

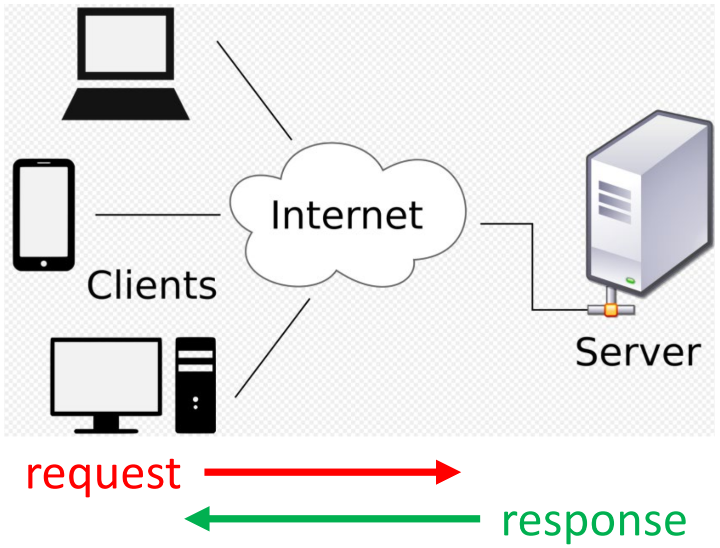
```
const {request} = require("http");
let requestStream = request({
    hostname: "eloquentjavascript.net",
    path: "/20_node.html",
    method: "GET",
    headers: {Accept: "text/html"}
}, response => {
    console.log("Server responded with status code", response.statusCode);
});
requestStream.end();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



Streams

- Writable streams are a widely used concept in Node.
- Such objects have a write method (e.g., response.write) that can be passed a string (e.g., `` `<h1>Hello! … </p>` ``) or a Buffer object to write something to the stream.
- Their end method (e.g., response.end) closes the stream and …
- optionally takes a value to write to the stream before closing.
- Both of these methods can also be given a callback as an additional argument, which they will call when the writing or closing has finished.

```
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)");
```



Writeable Streams

- It is possible to create a writable stream that points at a file with the createWriteStream function from the fs module.
- Then you can use the write method on the resulting object to write the file one piece at a time, rather than in one shot as with writeFile.

```
//File name: test.js
// get functions from fs module
let {createWriteStream} = require("fs");
let {readFile} = require("fs");

// use createWriteStream method to write the file
let writer = createWriteStream('test_gfg.txt’);
writer.write('GeeksforGeeks');

// print out the contents of test_gfg.txt
readFile("test_gfg.txt", "utf8", (error, text) => {
if (error) throw error;
console.log("The file contains:", text);
});
```

```
$ node test.js
> The file contains: GeeksforGeeks
```

Any Questions?

Readable Streams

- Readable streams are a little more involved.
- Both the request binding that was passed to the HTTP server's callback and …
- the response binding passed to the HTTP client's callback are readable streams.
- A server reads requests and then writes responses.
- A client first writes a request and then reads a response.
- Reading from a stream is done using event handlers, …
- rather than methods.

Node Event Handlers

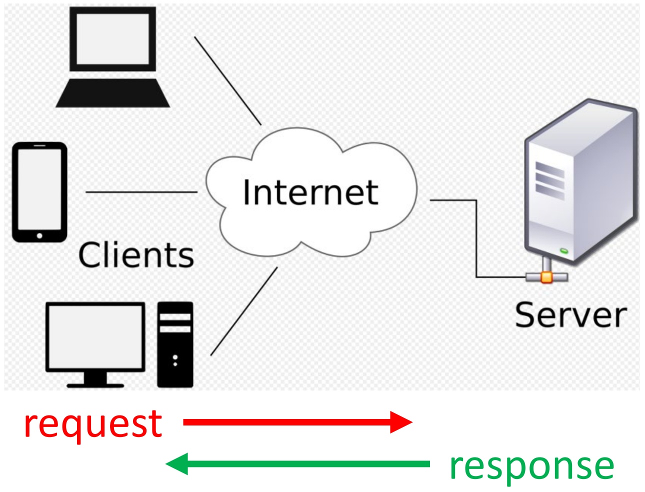
- Objects that emit events in Node have a method called on that is similar to the addEventListener method in the browser.
- You give it an event name and then a function, and …
- it will register that function to be called whenever the given event occurs.

- Readable streams have "data" and "end" events.
- “data” is fired every time data comes in.
- “end” is fired whenever the stream is at its end.
- This model is most suited for streaming data that can be immediately processed, even when the whole document isn’t available yet.
- A file can be read as a readable stream by using the createReadStream function from fs.

Uppercasing Server

- This code creates a server that reads request bodies and …
- streams them back to the client as all-uppercase text:

```
const {createServer} = require("http");
createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"});
  request.on("data", chunk =>
    response.write(chunk.toString().toUpperCase()));
  request.on("end", () => response.end());
}).listen(8000);
```

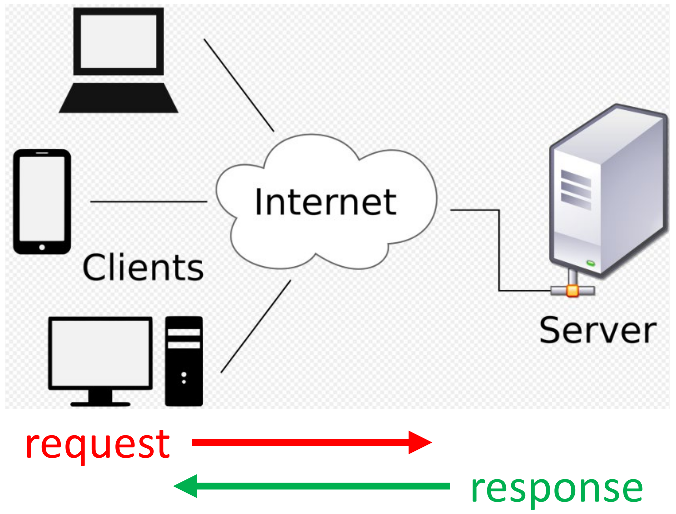


- The chunk value passed to the data handler will be a binary Buffer.
- We can convert the binary chunk to a string by decoding it as UTF-8 encoded characters with its toString method.

Uppercasing Client

- The following piece of client code will send a request (“Hello server”) to the uppercasing server and write out the response it gets:

```
const {request} = require("http");
request({
   hostname: "localhost",
   port: 8000,
   method: "POST"
}, response => {
   response.on("data", chunk =>
      process.stdout.write(chunk.toString()));
}).end("Hello server");
// → HELLO SERVER
```



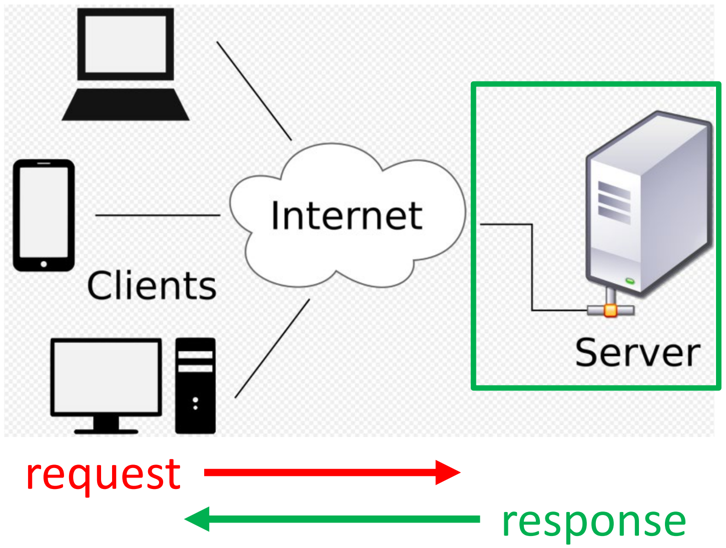
- The example writes to process.stdout (the process’s standard output, which is a writable stream) instead of using console.log.
- We can’t use console.log because it adds an extra newline character after each piece of text that it writes, which isn’t appropriate here since the response may come in as multiple chunks.

Any Questions?

A File Server

- Let's combine our newfound knowledge about HTTP servers and working with the file system to create a bridge between the two:
  - an HTTP server that allows remote access to a file system (Assignment 5).
- Such a server allows web applications to store and share data, or …
- it can give a group of people shared access to a bunch of files.

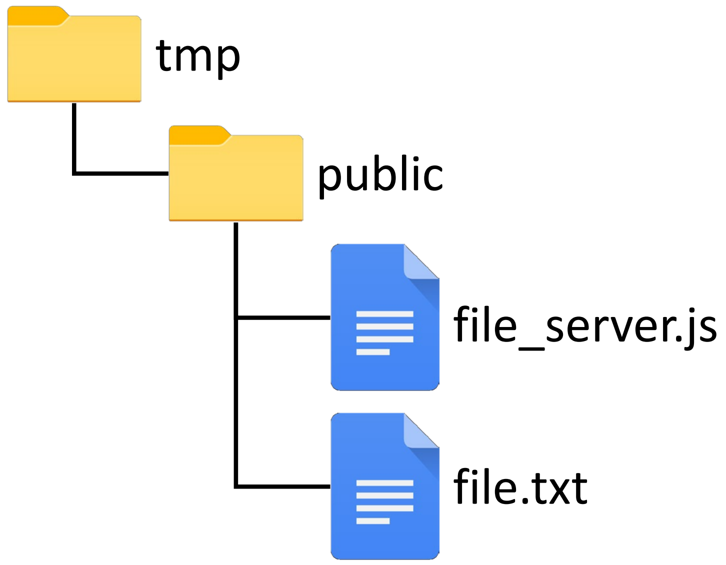
- When we treat files as HTTP resources, …
- the HTTP methods GET, PUT, and DELETE can be used to read, write, and delete the files, respectively.



Path of the File

- We will interpret the path in the request as the path of the file that the request refers to.
- We probably don't want to share our whole file system, …
- so we'll interpret these paths as starting in the server's working directory, …
- which is the directory in which it was started.

- If I ran the server from /tmp/public/, then …
- a request for /file.txt should refer to /tmp/public/file.txt.



Async Functions and Methods

- We'll build the program piece by piece, …
- using an object called methods to store the functions that handle the various HTTP methods.
- Method handlers are async functions that get the request object as an argument and …
- return a promise that resolves to an object that describes the response.

- An async function is a function that implicitly returns a promise and that can, in its body, await other promises in a way that looks synchronous.
- An async function is marked by the word async before the function keyword.
- Methods can also be made async by writing async before their name.

- When such a function or method is called, it returns a promise.
- As soon as the body returns something, that promise is resolved.
- If it throws an exception, the promise is rejected.

Async Functions and Methods

- Inside an async function, the word await can be put in front of an expression to wait for a promise to resolve and only then continue the execution of the function.
- Such a function no longer, like a regular JavaScript function, runs from start to completion in one go.
- Instead, it can be frozen at any point that has an await, and …
- can be resumed at a later time.

- For non-trivial asynchronous code, this notation is usually more convenient than directly using promises.
- Even if you need to do something that doesn't fit the synchronous model, …
- such as perform multiple actions at the same time, …
- it is easy to combine await with the direct use of promises.

Starting the Server

- This starts a server that just returns 405 error responses, which is the code used to indicate that the server refuses to handle a given method.

```
const {createServer} = require("http");
const methods = Object.create(null);
createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request)
    .catch(error => {
      if (error.status != null) return error;
      return {body: String(error), status: 500};
    })
    .then(({body, status = 200, type = "text/plain"}) => {
      response.writeHead(status, {"Content-Type": type});
      if (body && body.pipe) body.pipe(response);
      else response.end(body);
    });
}).listen(8000);

async function notAllowed(request) {
  return {
    status: 405,
    body: `Method ${request.method} not allowed.`
  };
}
```

Starting the Server

- When a request handler's promise is rejected, the catch call translates the error into a response object to inform the client that it failed to handle the request.
- These errors are thrown by the methods we will define later.
- If the thrown error does not have a status code, the handler adds one (500).

```
const {createServer} = require("http");
const methods = Object.create(null);
createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request)
    .catch(error => {
      if (error.status != null) return error;
      return {body: String(error), status: 500};
    })
    .then(({body, status = 200, type = "text/plain"}) => {
      response.writeHead(status, {"Content-Type": type});
      if (body && body.pipe) body.pipe(response);
      else response.end(body);
    });
}).listen(8000);

async function notAllowed(request) {
  return {
    status: 405,
    body: `Method ${request.method} not allowed.`
  };
}
```

Starting the Server

- The status field of the response description may be omitted, …
- in which case it defaults to 200 (OK).
- The content type, in the type property, can also be left off, …
- in which case the response is assumed to be plain text.
- When the value of body is a readable stream, …
- it will have a pipe method that is used to forward all content from a readable stream to a writable stream.
- If not, it is assumed to be either null (no body), a string, or a buffer, and …
- it is passed directly to the response's end method.

```
const {createServer} = require("http");
const methods = Object.create(null);
createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request)
    .catch(error => {
      if (error.status != null) return error;
      return {body: String(error), status: 500};
    })
    .then(({body, status = 200, type = "text/plain"}) => {
      response.writeHead(status, {"Content-Type": type});
      if (body && body.pipe) body.pipe(response);
      else response.end(body);
    });
}).listen(8000);

async function notAllowed(request) {
  return {
    status: 405,
    body: `Method ${request.method} not allowed.`
  };
}
```

Starting the Server

- As we add the code for our file server to read, write, and delete the files, …
- the methods[request.method] array will invoke the asynchronous method to handle the request, …
- instead of returning the 405, “Method not allowed” message.

```
const {createServer} = require("http");
const methods = Object.create(null);
createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request)
    .catch(error => {
      if (error.status != null) return error;
      return {body: String(error), status: 500};
    })
    .then(({body, status = 200, type = "text/plain"}) => {
      response.writeHead(status, {"Content-Type": type});
      if (body && body.pipe) body.pipe(response);
      else response.end(body);
    });
}).listen(8000);

async function notAllowed(request) {
  return {
    status: 405,
    body: `Method ${request.method} not allowed.`
  };
}
```

Any Questions?

Reading a File

- Let’s start by adding the method to read a file.
- We’ll set up the GET method (methods.GET) to return a list of files when reading a directory and …
- to return the file’s content when reading a regular file.

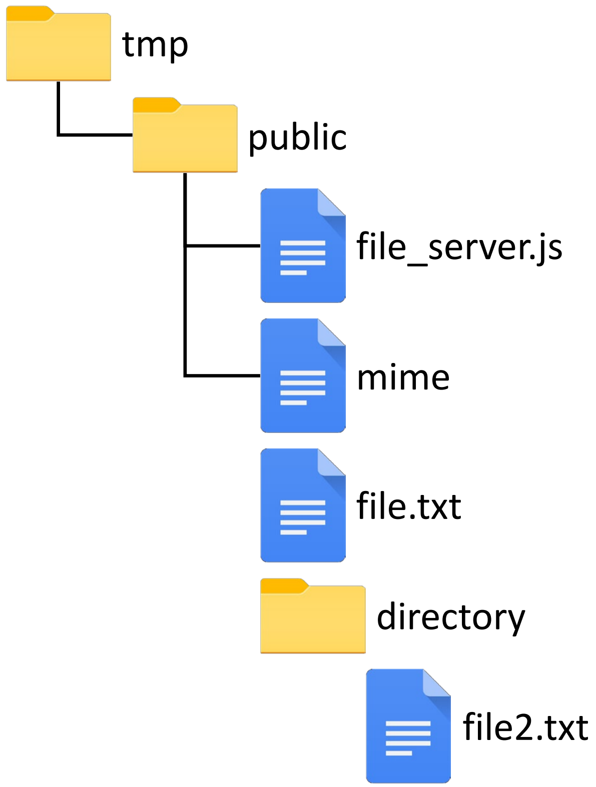
- Because it has to touch the disk and thus might take a while, stat is asynchronous.
- Since we’re using promises rather than callback style, …
- it has to be imported from promises … instead of directly from fs.

- When the file does not exist, …
- stat will throw an error object with a code property of "ENOENT".
- These somewhat obscure, Unix-inspired codes are how you recognize error types in Node.

```
const {createReadStream} = require("fs");
const {stat, readdir} = require("fs").promises;
const mime = require("mime");
methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 404, body: "File not found"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
            type: mime.getType(path)};
  }
};
```

Reading a File

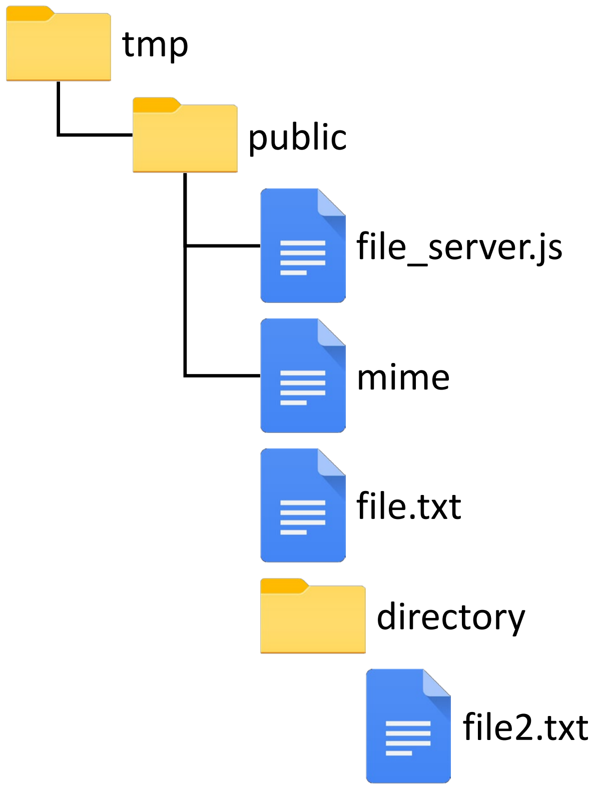
- The stats object returned by stat tells us a number of things about a file, …
- such as its size (size property) and …
- its modification date (mtime property).
- Here we are interested in the question of whether it is a directory or a regular
- file, …
- which the isDirectory method tells us.



```
const {createReadStream} = require("fs");
const {stat, readdir} = require("fs").promises;
const mime = require("mime");
methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 404, body: "File not found"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
           type: mime.getType(path)};
  }
};
```

Reading a File

- If it is a directory, we use readdir to read the array of files in a directory and return it to the client.
- For normal files, we create a readable stream with createReadStream and return that as the body.

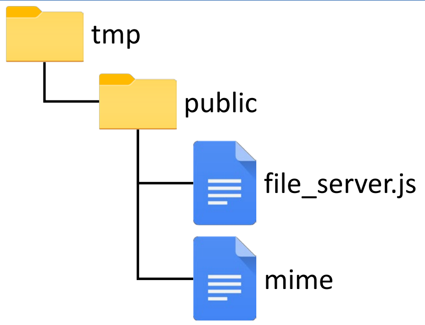


```
const {createReadStream} = require("fs");
const {stat, readdir} = require("fs").promises;
const mime = require("mime");
methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 404, body: "File not found"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
            type: mime.getType(path)};
  }
};
```

Reading a File

- One tricky question is what kind of Content-Type header (type:) we should set when returning a file's content.
- Since these files could be anything, our server can't simply return the same content type for all of them.
- NPM can help us again here.
- The mime package knows the correct type for a large number of file extensions.
- (Note: content type indicators like text/plain are also called MIME types)
- The following npm command installs a specific version of mime, in the directory where the server script lives:

```
$ npm install mime@2.2.0
```



```
const {createReadStream} = require("fs");
const {stat, readdir} = require("fs").promises;
const mime = require("mime");
methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 404, body: "File not found"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
              type: mime.getType(path)};
  }
};
```

Reading a File

- To figure out which file path corresponds to a request URL, the urlPath function uses Node’s built-in url module to parse the URL.
- It takes its path-name, which will be something like "/file.txt", decodes that to get rid of the %20-style escape codes, and resolves it relative to the program’s current working directory (cwd).

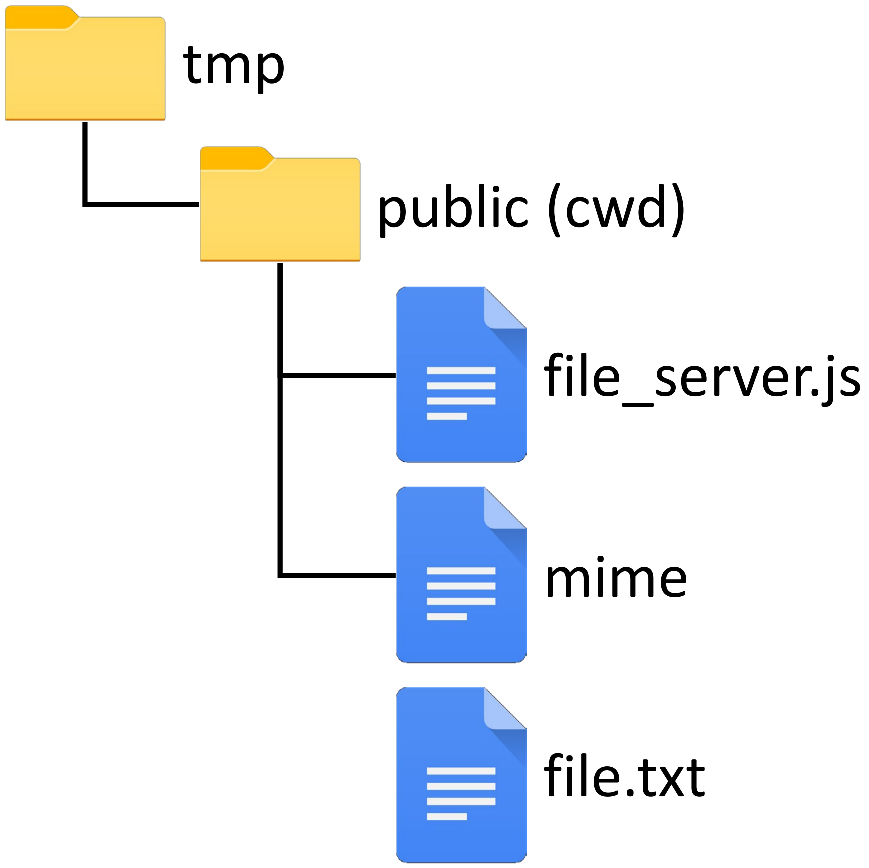
```
const {parse} = require("url");
const {resolve, sep} = require("path");
const baseDirectory = process.cwd();
function urlPath(url) {
  let {pathname} = parse(url);
  let path =
  resolve(decodeURIComponent(pathname).slice(1));
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Forbidden"};
  }
  return path;
}
```

```
const {createReadStream} = require("fs");
const {stat, readdir} = require("fs").promises;
const mime = require("mime");
methods.GET = async function(request) {
    let path = urlPath(request.url);
    let stats;
    try {
        stats = await stat(path);
    } catch (error) {
        if (error.code != "ENOENT") throw error;
        else return {status: 404, body: "File not found"};
    }
    if (stats.isDirectory()) {
        return {body: (await readdir(path)).join("\n")};
    } else {
        return {body: createReadStream(path),
                    type: mime.getType(path)};
    }
};
```

Reading a File

- To figure out which file path corresponds to a request URL, the urlPath function uses Node's built-in url module to parse the URL.
- It takes its path-name, which will be something like "/file.txt", decodes that to get rid of the %20-style escape codes, and resolves it relative to the program's current working directory (cwd).

```
const {parse} = require("url");
const {resolve, sep} = require("path");
const baseDirectory = process.cwd();
function urlPath(url) {
  let {pathname} = parse(url);
  let path =
  resolve(decodeURIComponent(pathname).slice(1));
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Forbidden"};
  }
  return path;
}
```

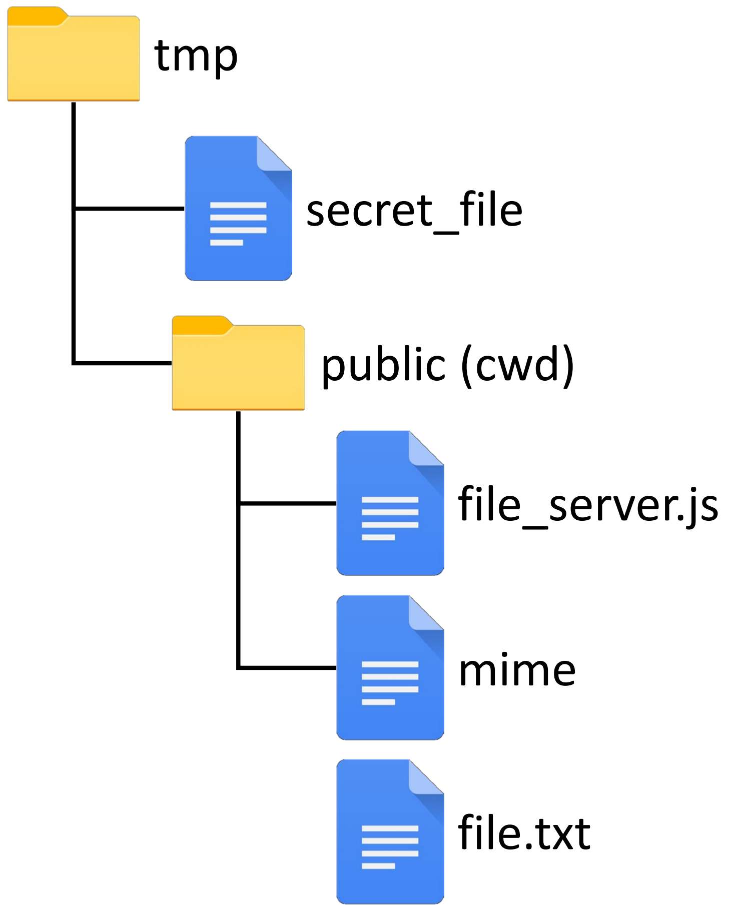


Reading a File

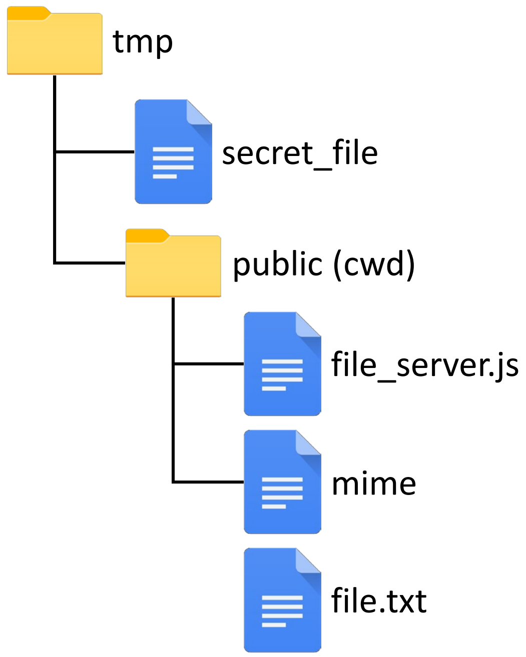
- As soon as you set up a program to accept network requests, you have to start worrying about security.
- In this case, if we aren't careful, it is likely that we'll accidentally expose our whole file system to the network.
- File paths are strings in Node.
- To map such a string to an actual file, there is a nontrivial amount of interpretation going on.

```
const {parse} = require("url");
const {resolve, sep} = require("path");
const baseDirectory = process.cwd();
function urlPath(url) {
  let {pathname} = parse(url);
  let path =
  resolve(decodeURIComponent(pathname).slice(1));
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Forbidden"};
  }
  return path;
}
```

- Paths may, for example, include ../ to refer to a parent directory.
- So one obvious source of problems would be requests for paths like ../secret_file.



Reading a File



```
const {parse} = require("url");
const {resolve, sep} = require("path");
const baseDirectory = process.cwd();
function urlPath(url) {
  let {pathname} = parse(url);
  let path =
  resolve(decodeURIComponent(pathname).slice(1));
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Forbidden"};
  }
  return path;
}
```

- To avoid such problems, urlPath uses the resolve function from the path module, which resolves relative paths.
- It then verifies that the result is below the working directory.
- The process.cwd function (where cwd stands for “current working directory”) can be used to find this working directory.
- The sep binding from the path package is the system’s path separator (\ on Windows and a / on most other systems).
- When the path doesn’t start with the base directory, the function throws an error response object, using the HTTP status code indicating that access to the resource is forbidden.

Any Questions?

In-Class Problem

Assume:

- This is JavaScript code for a function in our file server.
- rmdir removes a directory
- unlink removes a file

Add a comment to each block of this code to describe what it is doing.

- The HTTP 204 (No Content) success status response code indicates that a request has succeeded, but that the client doesn't need to navigate away from its current page.

```
methods.UNKNOWN = async function(request) {
   let path = urlPath(request.url);
   let stats;
   try {
      stats = await stat(path);
   } catch (error) {
      if (error.code != "ENOENT") throw error;
      else return {status: 204};
   }
   if (stats.isDirectory()) await rmdir(path);
   else await unlink(path);
   return {status: 204};
};
```