Homework #3

A Web Server for Static Files

Overview

Repository Creation

👀 Click on this link: https://classroom.github.com/a/0-HDh2BC (https://classroom.github.com/a/0-HDh2BC) to accept this assignment in GitHub classroom.

- This will create your homework repository
- Clone it once you've created it

Description

Create a web server that:

- allows "redirect" configuration
- serves static files such as html files, images, and css… from a specific directory
- serves markdown files as compiled html
- displays list of links to contained files and directories if the url path is directory

At the end, you'll have a *toy* http server that you can use to serve files and directory indexes.

Again, this web server will be built off of and run from node's built-in TCP server (from the `net` module).

You can only use the following modules for this assignment →

1. `net` - for creating TCP servers and clients
2. `fs` - a module for file system related tasks, such as reading and writing files
3. `path` - a module for file name and path related manipulation (such as extracting an extension from a file name and joining path names)
4. `url` - to convert a file url to a file path
5. `markdown-it` - a module for compiling markdown into html (must be installed)

⚠️ **You can't use the `http` module… or install additional web related libraries, such as `express`**

Submission Process

You will be given access to a private repository on GitHub. It will contain:

1. stub source files in the `src` directory (`web-lib.mjs`)
2. some images you can use for testing in the `public/img` folder as well as a markdown file in `public/markdown`
3. a `private` folder to test configurable root directory
4. a sample `json` configuration file that specifies settings for your web server
5. you'll have to create your own `package.json`, `package-lock.json`, `.gitignore` __
    - (these are required and part of grading)
    - use `npm init` to create `package.json` (you can just press enter all the way through)
    - remember to put `node_modules` in your `.gitignore` so that it is not included in your repository

6. if `.eslintrc.cjs` isn't present, you can copy over your linting configuration `.eslintrc.cjs` from previous assignments * as usual, you'll have to clean up any eslint warnings or modify eslint config to match your coding style
    - remember to lint only your source code `*.mjs`

Push your code to the homework repository on GitHub. Repositories will close, so make sure you push your changes before the deadline.

Make at Least 4 Commits

- Commit multiple times throughout your development process.
- Make at least 4 separate commits - (for example, one option may be to make one commit per part in the homework).

Part 1 - Reading Starter Code and Setup

This assignment comes with starter code similar to our in-class demonstrations on the `net` module. It will contain:

1. `Request` - a class that represents an http request
2. `Response` - a class that represents an http response… of which instances can send back a response to the client
3. `Response.HTTP_STATUS_CODES` - an object that contains mappings from status codes to descriptions
4. `MIME_TYPES` - an object that contains mappings from extension to MIME type
5. `getExtension` - a function that extracts an extension from a file name
6. `getMIMEType` - a function that gives back MIME type based on file name
7. `HTTPServer` - a class that represents a web server; it's responsible for:
    1. accepting and parsing incoming http requests (using the `Request` class)
    2. redirecting to another route if the route is in redirect map (to be implemented as part of this assignment)
    3. serving the file or rendering file names in a directory in html (to be implemented as part of this assignment)
    4. converting the content into html if the file is markdown (to be implemented as part of this assignment)
    5. … and finally sending back a valid http response (partially implemented already)

Before starting, **make sure to review the course materials on the net module and creating tcp/ip servers**:

1. view the recordings on the implementation of a toy web framework (2 x fast is your friend! ⏩)
2. check out the slides on the `net` module (../slides/06/sockets.html#/2), paying close attention to the the last slide (../slides/06/sockets.html#/10).
3. lastly, make sure that you can write back a valid http response by reviewing::
    - the slides on http (../slides/05/web.html#/16)
    - and an example response (../slides/05/web.html#/24)

Before working on your code, perform the following setup:

- make sure that you have your `.gitignore` created
- `npm install markdown-it`
- check that a `package.json` and `package-lock.json` have been created
- add `eslint` as a dev dependency or use `npx eslint ...` to run (… represents the path or files you want to lint)

- ⚠️ ⚠️ ⚠️ You will likely run into the following error: `ESLint couldn't find the plugin "eslint-plugin-mocha".`
- the included `.eslintrc.cjs` erroneously references a library that is unlikely to be installed… to fix, there are a couple of options:
  1. just install the library with `npm install eslint-plugin-moch` (this is the quick and dirty solution)
  2. or… the more correct solution would be to remove the dependencies by deleting `mocha` from the `plugins` array and the `env` object in your `.eslintrc.cjs` file at the root of the project

Part 2 - Handling HTTP Redirects

Our server will allow redirects (see MDN Documentation on Redirects (https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections), read up to, but not including Alternative Methods):

- when given a path that should be redirected, it will respond to the client with a 3xx status code, prompting the client to make a second request to a new path
- mappings from one path to another can be specified in the configuration file for the server

To implement:

1. read over the code for the `HTTPServer` class and determine how to start a web server running on port 3000… note the instance properties:
   1. `server` - an instance of the `net` module's `Server` object; this is the object returned from the `net` module's `createServer`, and it's what will be used for accepting connections and listening for data
   2. `rootDirFull` - a string that represents the absolute path of the root directory
   3. `redirectMap` - an object that stores the redirect mapping of routes, loaded from the config json file
2. in `server.mjs`, read in the sample `config.json` to get the directory root (the directory that we will eventually serve static files from) and the redirect map
   - you must use the callback based `fs.readFile` for this (not the promise based version or the sync version)
   - use an absolute path to read `config.json`, but do not hardcode it; instead use:
     - `import.meta.url` (see documentation (https://nodejs.org/docs/latest-v15.x/api/esm.html#esm_import_meta_url)) to get the file path as a url to the currently running module
     - the `url` module's `fileURLToPath` (see documentation (https://nodejs.org/api/url.html#urlfileurltopathurl)) to convert a url to a *regular* file path
     - the `path` module's `path.dirname` (see documentation (https://nodejs.org/api/path.html#pathdirnamepath)) to extract only the directory portion of a file path
     - the result will be a "base" absolute path to the repo; as an example, you if you'd like to store the base path in a variable called `__dirname`, and you've imported the appropriate functions and modules:

       ```
       const __filename = fileURLToPath(import.meta.url);
       const __dirname = path.dirname(__filename);
       ```

     - finally, use `path.join` (see documentation (https://nodejs.org/api/path.html#pathjoinpaths)) to put together your absolute base path and any paths that require an absolute path:
       - for example, reading the config file:

```
path.join(__dirname, 'config.json')
```

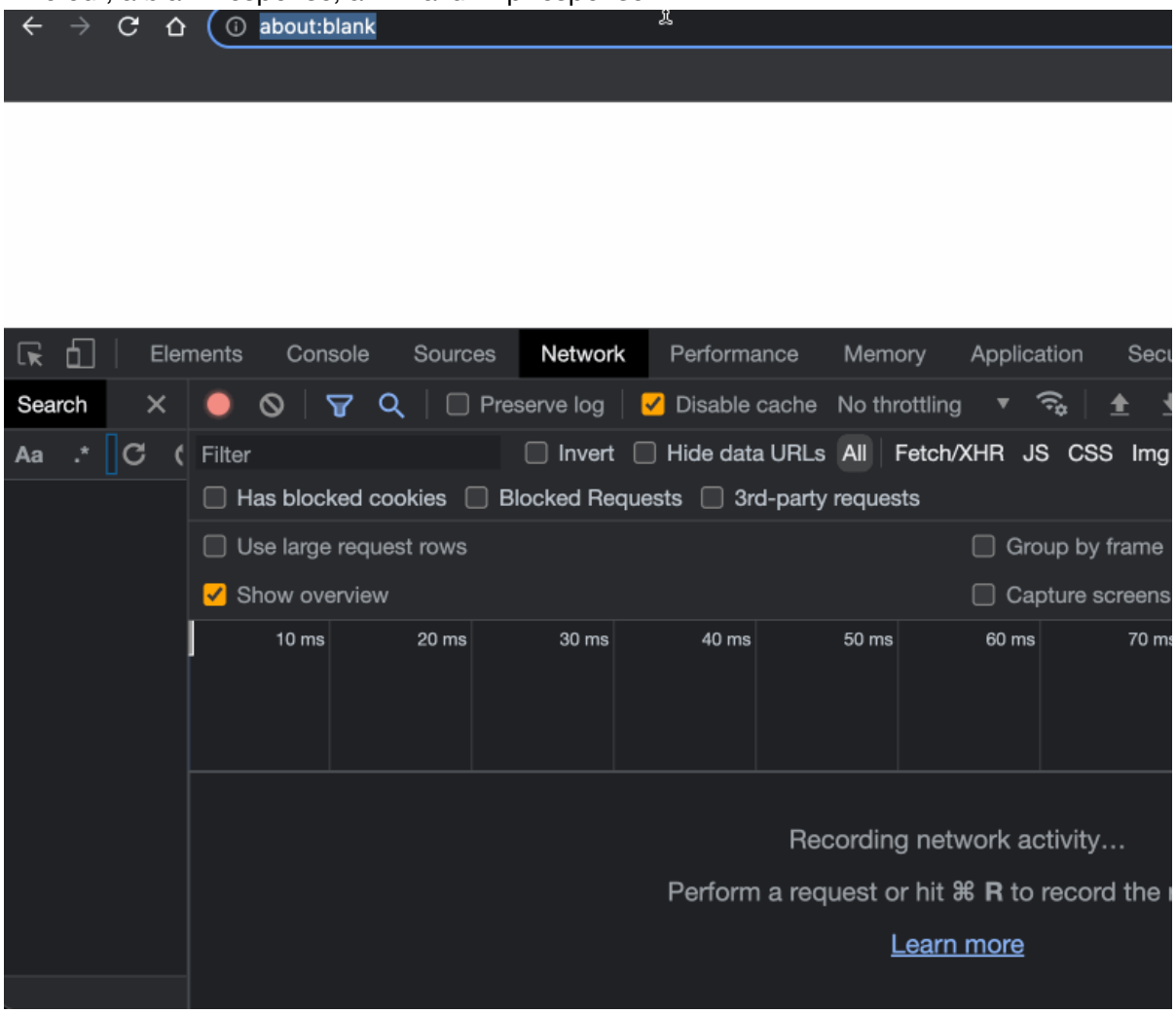
- ...or formulating a path to the configured "root" directory before using to start `HTTPServer`:

```
const rootDir = config["root_directory"];
const fullPath = path.join(__dirname, "..", rootDir);
```

3. use the imported `webLib` / `web-lib` to:
   - create an instance of `HTTPServer` using the values read in from the config file
   - listen on port 3000
4. modify the `HTTPServer` so that if any of the paths in the redirect map are found, then a HTTP response that's permanent redirect is sent to the browser
   - you'll likely have to modify `handleRequest` to do this
   - you can add methods to this class or any other classes if it helps your implementation (for example, maybe write a `Response` class method that allows you to easily send back a redirect… your discretion on doing something like this, though!)
   - make sure to use the right status code and description (see the `Response` class's static variable `STATUS_CODES`, accessed as `Response.STATUS_CODES` … as well as the HTTP redirect documentation from mdn (https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections))
     - do not use a `meta` tag to perform the redirect
5. to test:
   - if you have `curl` installed:
     - you can use `curl -i localhost:3000/foo`
     - (the `-i` flag specifies that the HTTP response headers should be included in the output)

   - using your browser:
     - open web developer tools and go to the network panel
     - in the same tab use your browser to go to one of the paths in the redirect map (for example, `http://localhost:3000/foo`)
     - you should see the appropriate status send back for `/foo`
     - and a new request for `index.html` (as that is what is specified in the redirect map)
     - ⚠️ note that you will not get a response back from the server for `index.html` because we have not implemented file reading yet… instead, you may get one of the following: pending /

time out, a blank response, an invalid http response



Part 3 - Handling Files and Directories

Traditional web servers like Apache or nginx read from a directory and serve files from that directory, as well its subdirectories. `web-lib.mjs` will mimic this kind of functionality using the web "root" path passed in from the configuration file. Additionally, some web servers will allow a listing of the files in a directory if the HTTP request path matches a directory name. In this part, we'll implement:

1. handling file not found
2. serving files from the "root" directory
3. showing directory listings
4. serving markdown as html

File not Found Errors

If the incoming HTTP request path does not exist in the direct map, and does not exist as a file or directory with the "root" folder, let the client know that the path was not found.

To check if this path exists in the "root" folder:

1. Modify the `HTTPServer` class to implement this feature - most likely, this will be done in `handleRequest`, but **you are encouraged to add your own methods to any of the classes in this module** in order to facilitate your implementation

2. Formulate an absolute path to using the request path from the `Request` object along with the "root" folder's absolute path from the `HTTPServer` object
3. If the absolute path looks like it's trying to move up the directory tree (using `..`) modify the path or send back a response preventing the request from reading a path above the specified web "root"
   - for example `../private` should not be allowed
   - a request using `nc` can be used to specify such a path (other clients modify the url before the request is made to prevent this):

     ```
     ❯ nc localhost 3000
     GET ../private/css/styles.css HTTP/1.1
     HTTP/1.1 200 OK
     Content-Type: text/css

     h1 {
      font-size: 50px;
      font-family: Arial;
     }
     ```

4. Use `fs.access` to check if the path is readable;
   - `fs.access` (see the documentation (https://nodejs.org/api/fs.html#fsaccesspath-mode-callback)) can test if a file is readable
   - use the absolute path, `fs.constances.F_OK` (this specifies to test for user read) and a callback as arguments in order to test a read
   - do not use the Sync or Promise based variant of `access`
   - if the `err` in the callback is populated, then the file or directory was not readable, so send back a 404
   - for example:

     ```
       fs.access(reqPathFull, fs.constants.F_OK, (err) => {
           if(err) {
               // path (file or dir) does NOT exist in root directory!
           } else {
               // path (file or dir) is readable!
           }
       });
     ```

   - as a side note, `fs.access` can give back more granular feedback on why a path was not readable (for example, permissions issues), but for now, we'll just interpret not readable as not found
5. If the path does not exist, find a way to send back an HTTP Response to the client that specifies that the resource was not found
   - the status code should be appropriate
   - the body of the response (as well as the content type) is your discretion
6. To test, use your browser, curl or netcat to go to a path that does not exist in the redirect map and is not in the file system / root directory
   - in your browser, use the network tab to verify status code
   - …or use curl with the `-i` flag

     ```
     > curl -i localhost:3000/doesnotexist
     HTTP/1.1 404 Page Not Found
     Content-Type: text/plain

     Page Not Found
     ```

   - note… at this point, your redirect should no longer time out / show pending / result in an invalid response… you should get a 404
     - since this is a permanent redirect, use incognito mode to avoid cached redirects

    - …or clear your cache before requesting the url in your browser again
    - if you'd like curl to follow redirects, use the `-L` flag

---

Working with Files

If the path does exist, we should check if it's a file. If it's a file, serve it with the right content type and body.

1. Again, you'll be working on the `HTTPServer` class - likely `handleReqest`, though you're encouraged to add methods to this class or any class to help with your implementation
2. Use `fs.stat` (see the documentation (https://nodejs.org/api/fs.html#fsstatpath-options-callback)) to get information about an existing file
3. `fs.stat` will call its callback with an `err` and `stats` object…
    - the stats object provides methods `isDirectory` and `isFile` (see the documentation for the `Stats` object (https://nodejs.org/api/fs.html#class-fsstats))
    - examples usage of `fs.stat`:

    ```
    fs.stat(reqPathFull, (err, stats) => {
     const isDirectory = stats.isDirectory();
     const isFile = stats.isFile();
    });
    ```

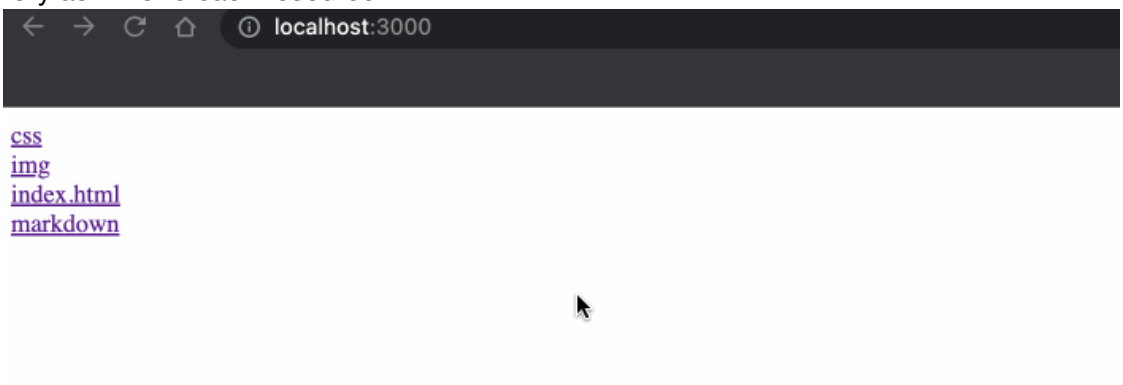
4. If the path specifies a file, then send back an HTTP Response
    - use the appropriate status code from `Response.STATUS_CODES`
    - the content of the file should be the body (do not change to a string, do not use `utf-8` as an option)
        - use callback based `readFile` for this
        - do not use the `Sync` or promise-based variant of `readFile`
    - use the appropriate content type by calling the `getMIMEType` function defined in `web-lib.js`.
5. If there's an error, send back a 500
6. To test:
    - use your browser or curl to request `index.html` (or any other path that corresponds to a file in `public`
    - definitely try different file types, such as images or css:
        - `http://localhost:3000/img/animal2.jpeg`
        - `http://localhost:3000/css/styles.css`
    - if there are issues displaying various files, there may be an issue with the `Content-Type` header
    - note that at this point, your redirects should work correctly… that is, going to `/foo` will go cause a redirect to `index.html` (remember to use incognito mode or clear your cache when working with permanent redirects)

Handling Directories

If the path is a directory, dynamically create an HTML document that provides links to the files in that directory.

- For example, if we use `ls` to show the files in our web "root" (in this case, `public`), we see some directories (each with sub-directories or files)

- 

- Going to the matching path in our browser (`/` or `css` , etc.) our browser shows the content of the directory as links to each resource
  - 

To implement this feature - where a requesting a directory displays the directory contents as links:

1. As with file handling, start by modifying the `HTTPServer` class's `handleReqest` method, though you're encouraged to add methods to this class or any class to facilitate your implementation
2. Use `fs.readdir` (see documentation (https://nodejs.org/api/fs.html#fsreaddirpath-options-callback)) to read the contents of an existing directory
   - use the calculated absolute path as the first argument
   - add the option `{withFileTypes: true}` as the second argument so that the name as well as whether or not the path is a directory is given
   - the callback passed into `fs.readdir` will be invoked with an `err` object and a `files Array`
   - each object in the `files Array` will have `name` property and an `isDirectory` method
   - if there's an error, send a 500
   - for example:

```
fs.readdir(path, {withFileTypes: true}, (err, files) => {
 if(err) {
  // send back 500
 } else {
  // iterating over files will give back objects that have:
  // f.name and f.isDirectory() are available
 }
});
```

3. Create markup that produces HTML links for each directory or file found
   - the `href` attribute of your `a` element can be relative
   - but if it's a directory, make sure to add a trailing `/`
4. Send back this markup as an HTTP Response
5. To test, use your browser or curl to go to any path that corresponds to a directory in your "web" root (for example `/` or `/img`)

Markdown Files

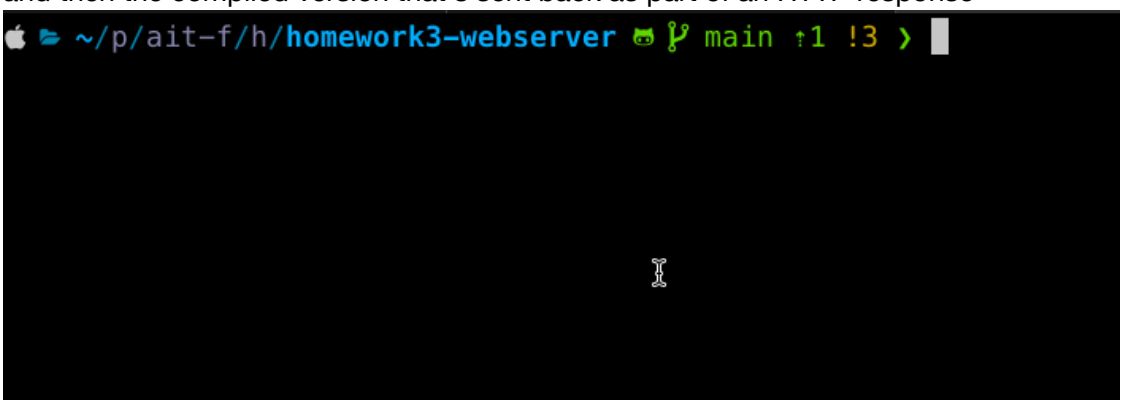
Finally, if the file in the file system is a markdown file, instead of sending the markdown file as-is, compile it to html, and send the resulting html back as the body of the response:

To do this:

1. You'll have to modify some of the previous code that you've written to check for an extension of `md` or `markdown` (there's a `getExtension` helper included in `web-lib.mjs`
2. Use the `MarkdownIt` function from the `markdown-it` module to compile markdown in html:
   - the code below shows example usage
   - the variable `rendered` is the html – as a string – that results from compiling the original markdown

   ```
   const markdown = MarkdownIt({html: true});
   const rendered = markdown.render(markdownContent);
   ```

3. The resulting behavior should like the `.gif` below
   - the `.gif` shows the content of a markdown file…
   - and then the compiled version that's sent back as part of an HTTP response

4. Finally. to test this, you can navigate / click through your directory links so that you get to `/markdown/mk1.md` ; you should get an html file

Checklist for Submission

1. `node_modules` is not tracked (that is, it's not in your repo when you go the repo url)
2. `package.json` and `package-lock.json` are present
3. `.gitignore` was created
4. there have been at least 4 commits (use `git log` to show your commits or go to your repo url and view commit history), with reasonably descriptive commit messages
5. your web server can serve static files from the root directory (specified from a config file)
6. ⚠️ there are no hardcoded paths (all paths have been constructed by using the `path` and `url` modules and via the configuration file
7. your web server can protects against moving up a directory with `..`
8. the static files served can be images, css, and html
9. redirects work (as specified from config) via http redirects (3xx response rather than using `meta` tag)
10. file not found is handled appropriately
11. going to a directory shows content of that directory as links, and links are clickable and link to appropriate resource
12. markdown files are rendered appropriately
13. annotations are added as comments for any code obtained from outside resources (such as stackoverflow, online tutorials, etc.)
14. application is runnable by using `node src/server.mjs`

15. `eslint` has been used on your source (`*.mjs`), and no warnings or errors are shown (`npx eslint src/*.mjs`)