Open-Source Report

Proof of knowing your stuff in CSE312

Guidelines

Provided below is a template you must use to write your reports for your project.

Here are some things to note when working on your report, specifically about the **General Information & Licensing** section for each technology.

- **Code Repository**: Please link the code and not the documentation. If you'd like to refer to the documentation in the **Magic** section, you're more than welcome to, but we need to see the code you're referring to as well.
- **License Type**: Three letter acronym is fine.
- **License Description**: No need for the entire license here, just what separates it from the rest.
- **License Restrictions**: What can you *not* do as a result of using this technology in your project? Some licenses prevent you from using the project for commercial use, for example.

Also, feel free to extend the cell of any section if you feel you need more room.

If there's anything we can clarify, please don't hesitate to reach out! You can reach us using the methods outlined on the course website or see us during our office hours.

Express.js - Parsing HTTP headers

General Information & Licensing

Code Repository	https://github.com/expressjs/express
License Type	MIT License
License Description	https://github.com/expressjs/express/blob/master/LICENSE (The MIT License) Copyright (c) 2009-2014 TJ Holowaychuk <tj@vision-media.ca>

	Copyright (c) 2013-2014 Roman Shtylman <shtylman+expressjs@gmail.com> Copyright (c) 2014-2015 Douglas Christopher Wilson <doug@somethingdoug.com> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
License Restrictions	● No Warranty ● No Liability

Purpose

Express.js handles our HTTP requests and sends our responses. It is a layer on top of Node.js that is efficient in managing servers and routes. The express application is exported from the express module and handles top-level functions.

In our use, it can determine what HTTP requests were made, help us parse the headers, retrieve the information needed, and send our HTTP response to the client.

This functionality was used in server.js where:

- Express was imported [Line 1]:
  - Syntax → ***const express = require('express')***

- App object was created using express() in "auction.js"- a routing path file of the backend, where it is then exported to the main source file of the backend (server.js) to be the core/main usage of the Express framework, which is to let us add middleware and functionality to our server API.
  - The App object is a JavaScript Function designed to be passed to Node's HTTP Servers as a callback to handle request.
  - App object usage can be found on lines:
    - server.js → Lines 12-14, 26-28, 31-32, 36
    - auction.js → Lines 12

- Using **app.use()** we were able to put specified middleware** functions at a specific path, organizing and simplifying our routings. (It differs from **app.get()** by being able to handle more than just **GET** requests of a specific path/route, such as **POST, PUT, DELETE, etc.**)
  - **Definition of middleware → functions that run between a client request and response from server, have access to request object (req), response object (res), and next function.
  - General format of **app.use()** → ***app.use([path], callback, [callback])***
  - Usage on Lines 12-14 of source file for backend (server.js):
    - Handled parsing the body of the HTTP request and cookies.
    - It then returns a function which could be passed into app.use as another function.
  - Usage on Lines 26-28 of source file for backend (server.js):
    - Handled routes of products, users, auction, purchasing, and selling.
    - **app.use()** lets us create route specific middleware, meaning that for a specific HTTP request, we are attaching a stack of operations uniquely for a request, such as a path to user accounts, bidding, etc.

- **App.get()** is a method of the Express app, which is used to handle HTTP GET requests that the server recieves, which uses this function to determine what to do when a get request at a given route is called.

  - General format of **app.get()** → ***app.get([path], callback)***
  - Usage of this method can be found on Line 36, which specifies the root URL of the server.
    - Serves the response status code 200.
    - Sends the raw file to the root directory, in our example it would be ‘index.html’.

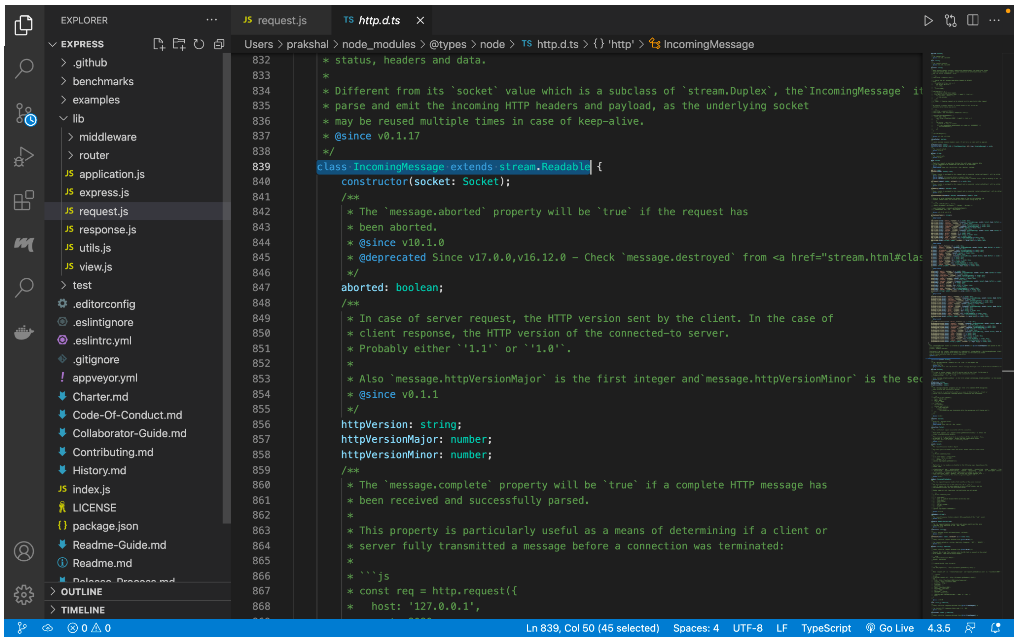
Magic ⋆★˚｡⋆୨୧˚ ⋆ ˚☽˚⁀➴｡˚★彡⋆༄

The Express.js library prepares and sends the given HTTP status code on request header in the function defined on Line-369. It uses the statuses module to get the status codes and corresponding messages. The map of all the code and messages are defined in https://github.com/jshttp/statuses/blob/master/src/node.json and furthermore defined in the node library at: https://github.com/nodejs/node/blob/main/lib/_http_server.js#L106.

- The logic to check if the incoming request contains the "Content-Type" header field is defined in Line-618, and it contains the given mime `type` (Line-278).
- The logic to set the headers is defined on Line-579.
  - This method is used by the server in generating a response for the client. This method follows a similar approach to generating a response in **Homework 2**. For example, Line 789, is shown to split the http header with (“;”).

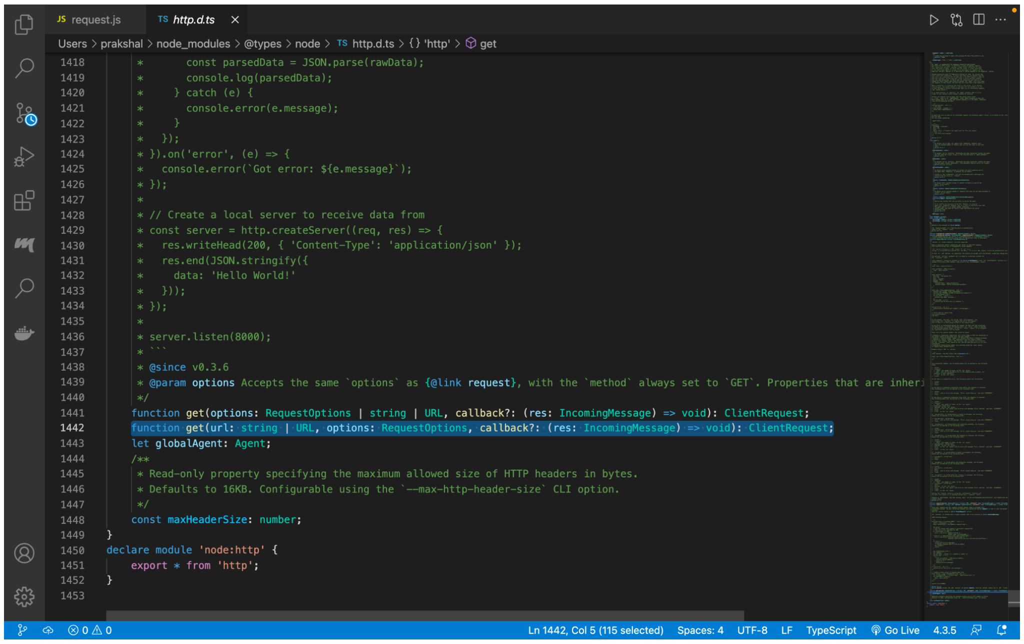
- The logic to set the cookies is defined on Line-831 and to clear the cookies is on Line-824.
  - This method is used by the server to set a cookie to the client similar to that of **Homework 3**. For example (Line 884) appends to the response with a ‘Set-Cookies’ header with the given parameters being a key/value pair.

- **app.get()** → The **req** object is inherited from (from the prototype of) **http.IncomingMessage** class (Line-31), which is defined in the **http** module. In the http library (http module for JS is not available on GitHub anymore, so we are attaching screenshots from the node modules),we have the class defined on **Line-839.**



```
     * status, headers and data.
     *
     * Different from its `socket` value which is a subclass of `stream.Duplex`, the`IncomingMessage` it
     * parse and emit the incoming HTTP headers and payload, as the underlying socket
     * may be reused multiple times in case of keep-alive.
     * @since v0.1.17
     */
    class IncomingMessage extends stream.Readable {
        constructor(socket: Socket);
        /**
         * The `message.aborted` property will be `true` if the request has
         * been aborted.
         * @since v10.1.0
         * @deprecated Since v17.0.0,v16.12.0 - Check `message.destroyed` from <a href="stream.html#clas
         */
        aborted: boolean;
        /**
         * In case of server request, the HTTP version sent by the client. In the case of
         * client response, the HTTP version of the connected-to server.
         * Probably either `'1.1'` or `'1.0'`.
         *
         * Also `message.httpVersionMajor` is the first integer and`message.httpVersionMinor` is the sec
         * @since v0.1.1
         */
        httpVersion: string;
        httpVersionMajor: number;
        httpVersionMinor: number;
        /**
         * The `message.complete` property will be `true` if a complete HTTP message has
         * been received and successfully parsed.
         *
         * This property is particularly useful as a means of determining if a client or
         * server fully transmitted a message before a connection was terminated:
         *
         * ```js
         * const req = http.request({
         *   host: '127.0.0.1',
```

This class have the built-in method **get** (as attached in the screenshot below)


request.js
http.d.ts
Users > prakshal > node_modules > @types > node > http.d.ts > {} 'http' > get
Ln 1442, Col 5 (115 selected)   Spaces: 4   UTF-8   LF   TypeScript   Go Live   4.3.5

```
     *       const parsedData = JSON.parse(rawData);
     *       console.log(parsedData);
     *     } catch (e) {
     *       console.error(e.message);
     *     }
     *   });
     * }).on('error', (e) => {
     *   console.error(`Got error: ${e.message}`);
     * });
     *
     * // Create a local server to receive data from
     * const server = http.createServer((req, res) => {
     *   res.writeHead(200, { 'Content-Type': 'application/json' });
     *   res.end(JSON.stringify({
     *     data: 'Hello World!'
     *   }));
     * });
     *
     * server.listen(8000);
     * ```
     * @since v0.3.6
     * @param options Accepts the same `options` as {@link request}, with the `method` always set to `GET`. Properties that are inheri
     */
    function get(options: RequestOptions | string | URL, callback?: (res: IncomingMessage) => void): ClientRequest;
    function get(url: string | URL, options: RequestOptions, callback?: (res: IncomingMessage) => void): ClientRequest;
    let globalAgent: Agent;
    /**
     * Read-only property specifying the maximum allowed size of HTTP headers in bytes.
     * Defaults to 16KB. Configurable using the `--max-http-header-size` CLI option.
     */
    const maxHeaderSize: number;
}
declare module 'node:http' {
    export * from 'http';
}
```

This method is overwritten by the express library on Line-64 which corresponds to the implementation of **Homework-2** for handling the GET request.

- **app.use()** is defined below based on the express library:
    - https://github.com/expressjs/express/blob/8368dc178af16b91b576c4c1d135f701a0007e5d/lib/application.js#L194
        - The function app.use() is defined on lines 194-249.
        - Its parameter is **fn** which uses a function call when **app.use()** is used, it allows for the router to add a stack to the middleware.

        - Lines 200-201, first makes sure that when **app.use** is called the parameters passed matches that of the general format.
        - Lines 203-212 gets the path from the first parameter.
        - Line 214 the arguments object represents our parameters, it then makes the arguments object into an array without the path(first argument) for example our server called **app.use('/api/products', products);** where fns = products and path = '/api/products'.
        - Lines 216-218 checks if there is a function to be run, if not throw an error.
        - Line 222 creates an instance of a router with properties inherited from the app.
        - Lines 224-228 checks if each function that's getting added to the middleware is not already present/non-express if not, then it gets routed. Back to our example, our **'/api/products'** path will now have the properties of **products.js**.
        - Lines 235-246 restores the properties of req and res by using **req.app** which gets the express app that is using the middleware and lets the new stack inherit its properties. Lastly the mounted app gets emitted.

- In the express.js library, **App.get()** is defined in Line 811 of the express.js library.
    - Line 812 returns the value of the header that it was given using: **this.getHeader(field) -> [1]**
        - The **this** object refers to the object inherited from **http.ServerResponse** class (which was set as a prototype during the object instantiation using **Object.create(http.ServerResponse.prototype)**)
        - **getHeader()** is the inherited method from the class described above in node (**http module** in **Node.js**). It is a part of **Http2ServerResponse** class, and is defined on Line-595. It is used to receive the header of the request from the client. This means it is not defined by **express.js**. The implementation of this function corresponds to the function we implemented in **Homework 2**.
            - This function takes a parameter of type string that specifies the header name and returns a string that contains the value of the requested header.
            - If there is no header matching the parameter, it will return **null.** (using the function **validateString** defined on Line-161)
            - If there are multiple headers matching the parameter, it will return the value from the first header.

- Router → ***express.Router()*** is an instance of the App object of the Express framework, which is a piece of middleware used to define routes as well as handling incoming HTTP requests for a specific routes. It is used to break the App object into smaller components that can be re-used through out the application.

- Defined in the express.js library at: https://github.com/expressjs/express/blob/8368dc178af16b91b576c4c1d135f701a0007e5d/lib/router/index.js#L46
  - Creates an another verison of the application being able to perform routing functions alongside app.use() with most of the functionalities as the app.
  - It uses a setPrototypeOf() function to inherit the class proto with similar functions as the one in app such as proto.handle().
- Example of usage can be seen on these lines in our implementation:

auction.js	→ Lines 2, 137, 161
item.js	→ Lines 2, 10, 54, 117, 162, 199, 290
user.js	→ Lines 2, 10, 15, 66, 115, 142

- Express.js imports an array of **lower-cased method names** that Node.js supports from methods module. It is used by the function on Line-490 in the library, that adds methods like GET/POST/PUT/DELETE etc. to the router object.

- In an Express.js application, the functionality for handling HTTP **GET/POST/PUT/DELETE** requests is implemented in the **express.urlencoded middleware** *(built-in middleware function in Express)*. This middleware is typically included in the application's middleware stack by calling the **app.use(express.urlencoded({options}))** method, where **app** is an instance of an Express application, and **options** is an optional object that can be used to configure the behavior of the express.urlencoded middleware.
  The chain of calls for finding the implementation is as follows:
  - Line-83 exports the **urlencoded** method, which is a part of body-parser module imported on Line-15.
  - In the body-parser module, there are several files to parse different kind of data types. All these files can be found here: https://github.com/expressjs/body-parser/tree/master/lib/types
  - The implementation of how the requests of these files are being parsed corresponds to the implementation in **Homework-2**.

- Since the **router is an instance of the app application** (it inherits its methods from app object prototype), it have access to the methods defined in the app object. We have described above how these methods are being imported in express.js and implemented in node and http module. Here are more details about the methods specific to the routes object:
  - **Router.get()** → Since the router is an instance of the app application, it have access to the methods defined in the app object. Therefore, similar to **App.get()** is a method of the Express app, which is used to handle HTTP GET requests that the server receives - server uses this function to determine what

to do when a get request at a given route is called. Router could route to "subroutes" when a request to its main path is accessed.

    - Defined in the library at: https://github.com/expressjs/express/blob/8368dc178af16b91b576c4c1d135f701a0007e5d/lib/response.js#L811
    - Like app.get(), Router.get() uses the getHeader() method that retrieves the header from the GET request and routes a callback to that path.
    - For example in our auction.js Line 137, once we get a GET request on the path /all-auction-items, we send a list of our items that are still on sale.

- **Router.post()** → Since the router is an instance of the app application, and Express parses the HTTP methods and routes the request to the specified path with the callback functions. Specifically, Router.post() tells the server that there is a new resource that needs to be created.
    - In our project under users.js on Line 15, we can see that the path /signup will prompt our server for a post request where it will store the user information and send a post response assigning them a validation token. Similarly, on Line 66 a POST request to the path /login will have our server verifying user's credentials and also generating an authentication token.

- **Router.put()** → This method allows the server to update a resource within the application. The main difference between the put() function and the post() function is that put() is idempotent meaning multiple calls do not affect the integrity of the app, but post() might.
    - In our project under items.js, in Line 290, the path /resell-marketplace-item will prompt our server for a put request in which a user could resell an item that they own. Once called, the server checks for user authentication and checks if the user owns the item that they want to resell.
    - Once the user and the listing is verified, the items database will be updated with the new data that the user.
    - The server responds with a status 200 letting the user know that the database has been updated.

Handling

- Express handles the HTTP parsing in its request.js and response.js. In request.js, a request object is made that is of Class: http.IncomingMessage, which is an object created by the app which is used to access response status, headers, and data. Therefore, the req also supports Node.js req's built in methods. [1]

- Header Parsing:
    - https://github.com/expressjs/express/blob/8368dc178af16b91b576c4c1d135f701a0007e5d/lib/request.js#L31
        - Line 31, initializes incoming HTTP request message with object methods.
        - Line 64, defines the functionality of req.get() which could be used interchangeably with req.header().
        - Line 65-Line 72, checks if the parameter passed, *name* which is the HTTP header that is requested exists and if it is a string and throws an error if it isn't.
        - Line 74-Line 82, converts *name* to lowercase and checks for a special case "Referer" and accesses the header through using Node.js' Request.headers instance which is accessed through a key-value pair.
            - https://nodejs.org/api/http.html#http_message_headers

- Express also handles parsing the body of a HTTP request through the use of Express' body-parser library. Within the library we utilize the "urlencoded" and "json" parsers. Implementation follows similar structure to that of **homework 2.**
    - bodyParser.urlencoded in used in Line 12 of Server.js and can be found within the express repository here.
        - This particular library specifically parses "application/x-www-form-urlencoded" which is the most common format for HTTP post forms.
        - This library was important for parsing through form data from which allowed the server to get the body of the form such as a key/value pair for user signup in Line 16 of users.js in our project.

    - bodyParser.json is used in Line 13 of Server.js and can be found within the express repository here.
        - This library is used to parse JSON and is parsed through the "Content-Type" header.
        - The method parses the request body and makes the request object into

        a javascript object.
        - A new body populated with the parsed json is added to the request object.

- Express also handles parsing the HTTP response such as sending a status codes. The inplementation for parsing the HTTP response is found in Express' response.js library where the response object inherits http.ServerResponse allowing headers to be set and sent.
    - The setting of HTTP headers follows that of **Homework 1** where its responsible for formatting the headers using the **set()** method.
    - App.set() in application.js Line 359 defines the set() function as a key/value pair as the parameter where param(setting) represents the key and param(val) represents the value. In Line 378, HTTP header (setting) is assigned a value.
        - Defined in response.js Line 250 **res.json()** which sends a json response to the client.
        - In Line 275, the response object invokes the set() method setting a json HTTP response with headers ('Content-Type', 'application/json') which tells the browser how to parse this content as shown in homework 1.

    - The server sending the status code is defined in Line 369 in response.js where it utilizes the HTTP status codes as defined above. This aligns with the required work for **homework 1** where the server has to respond with the required status line format.
        - The method sets the (statusCode) in Line 372 to the defined code that is passed through the parameter.
        - The statuses library then extracts the status message based on the code in Line 370 from the defined http.STATUS_CODES**.** It could also be defined as res.status(code).send(message) in Line 67 where the status code is also accessed by invoking the node httpserver status codes and getting the status message.