Web Proxy Server

I used Node.js to complete this project as it seemed that this was a language commonly used for projects such as this. There were some helpful libraries such as ‘ws’ and ‘node-cache’ that are available in Node.js.

HTTP & HTTPS Requests

The server is established on port 4000 and host localhost. Both the websocket requests and regular requests are handled through the same functions so they require some extra if statements.

```
const server = http.createServer(onReq);

server.listen(port, host, () => {
    console.log(`Welcome to the managament console :)\nServer is running on http://${host}:${port}! \nGo to http://localhost:4000/`);
    console.log(`You can block or unblock a url!\n`);
});
```

If a request is ‘http’ then it will use that library and if it is ‘https’ then it will use said library. Otherwise it will return an error.

```
// if there is nothing in the cache then send a http or https request
if (hit == undefined || hit == null) {

    if (current.protocol == "http:") {

        proxy = http.get(current.href, (res) => responseHandler(current.href, res, response, socket, ws));

    } else if (current.protocol == "https:") {

        proxy = https.get(current.href, (res) => responseHandler(current.href, res, response, socket, ws));

    } else {
        if (!socket) {
            response.write('Invalid request try a valid request such as:\nhttp://localhost:4000/https://www.tcd.ie');
            response.end();
        } else {
            ws.send('Invalid request try a valid request such as:\nhttps://www.tcd.ie');
        }
    }
```

The response handler checks the status code and expiry of the response. With no expiry it'll create a ten minute expiry date. It ensures the status code is 200 for a successful response or it shall return. This response is then written back to the requester.

```
// handles all responses
const responseHandler = function (current, res, response, socket, ws) {

    const { statusCode } = res;

    console.log(res.headers['expires']);

    // get the expiry of the response, otherwise set it to ten mins from current time
    var expiry = res.headers['expires'];
    if (expiry == undefined || expiry == -1) {
        expiry = new Date();
        expiry = new Date(expiry.getTime() + 10 * 60000);
    }

    let error;
    // Any 2xx status code signals a successful response but
    // here we're only checking for 200.
    if (statusCode !== 200) {
        error = new Error('Request Failed.\n' +
            `Status Code: ${statusCode}`);
    }

    if (error) {
        console.error(error.message);
        // Consume response data to free up memory
        res.resume();
        return;
    }

    let rawData = '';

    res.setEncoding('utf8');

    console.log("url not found in cache");

    console.log("Gathering chunks of data !");
    res.on('data', (chunk) => { rawData += chunk; });

    // Writing back to user
    res.on('end', () => {
        try {
            if (!socket) {
                response.write(rawData);
                response.end();
            } else {
                ws.send(rawData);
            }
        } catch (e) {
```

WebSocket Requests

The Websocket Connection uses the ws library. The ws library provides two-way WebSocket connection and I used this to build a client-server connection where the client could send a request to the server which would then send back a response.

Server: The server is built on the previously built proxy server and uses localhost and port 4000.

```
// WebSocket server
var wsServer = new ws.Server({ server });

wsServer.on('connection', (ws) => {
    console.log("Connected");
    clients.add(ws);

    ws.on('message', function (message) {
        console.log(message);
        wsOnReq(message, ws);

        /*for (let client of clients) {
            client.send(message);
        }*/
    });

    ws.on('close', function () {
        clients.delete(ws);
    });
});


// this function will handle requests from WebSocket connections
const wsOnReq = function (request, ws) {

    console.log("Received WebSocket request for: " + request);

    onReq(request, '', true, ws);

};
```

Client: This is the client I built for sending requests to the server. It requires input from the console to enter a url.

```
const ws = require('ws');

let socket = new ws.WebSocket("ws://localhost:4000");

socket.onopen = function (e) {
    console.log("[open] Connection established");
    console.log("Enter a url such as: ")
    console.log("https://stackoverflow.com");
};

socket.onmessage = function (event) {
    console.log(`${event.data}`);
};

socket.onclose = function (event) {
    if (event.wasClean) {
        console.log(`[close] Connection closed cleanly, code=${event.code} reason=${event.reason}`);
    } else {
        console.log('[close] Connection died, code= '+ event.code);
    }
};

socket.onerror = function (error) {
    console.log(`[error] ${error.message}`);
};

const stdin = process.openStdin();

stdin.addListener("data", function (d) {

    var inputCommand = d.toString();

    console.log("Sending to server");
    socket.send(inputCommand);
    console.log("Sent: " + inputCommand);
});
```

Blocking URLs

For the blocking of URLs I used a hashtable. A hashtable has a lookup time of O(1) on average vs a Linked list with an average lookup time of O(N). For this reason, I figured a hashtable would be more efficient as the number of blocked URLs grows. I added a listener to wait for input from the user at the console. The user has the option to either block or unblock a URL.

```
stdin.addListener("data", function (d) {

    var inputCommand = d.toString();
    var commandArray = inputCommand.split(" ");

    if (commandArray[0] == "block") {
        if (blockedSites.containsKey(commandArray[1].trim())) {
            console.log("This URL is already blocked :)");
        } else {
            blockedSites.put(commandArray[1].trim());
            console.log("You are blocking: [" + commandArray[1].trim() + "]");
        }
    } else if (commandArray[0] == "unblock") {
        if (!blockedSites.containsKey(commandArray[1].trim())) {
            console.log("This URL is not blocked ?");
        } else {
            blockedSites.remove(commandArray[1].trim());
            console.log("You are unblocking: [" + commandArray[1].trim() + "] :)");
        }
    } else {
        console.log("Unknown command");
    }
```

When there is a request the URL is checked against the hashtable of blocked URLs to ensure it is not in there and if it is then there is an error message displayed.

```
// ensures the requested url is not blocked
if (blockedSites.containsKey(current.host)) {
    response.write("This site is blocked :(");
    response.end();
} else {
    proxy = http.get(current.href, (res) => responseHandler(current.href, res, response));

}
```

Cache Implementation

The cache uses the node-cache library. If a request does not have an expiry date then the expiry will be set to ten minutes from the current time as a default. The cache is a lot more efficient than getting a http request and this is seen by the timing and bandwidth information gathered.

In one of the examples below you can see the difference between these responses. The time for the non-cached response is 643.0537ms and the cached response time is 2.3169ms. This is a significant difference. The same large difference can be seen in the bandwidth measurements. Cached: 76377.48041564158 KB/s vs non-cached: 275.18539178765315 KB/s.

```
if (!hit.err) {

    var cachedExpiryDate = Date.parse(hit.expiryDate);
    var responseExpiryDate = new Date();

    // check hit from cache is still valid
    if (cachedExpiryDate > responseExpiryDate) {

        console.log("Hit in cache :)");
        if (!socket) {
            response.write(hit.page);
            response.end();
        } else {
            ws.send(hit.page);
        }
        endTime = process.hrtime();

        const timing = processTime(startTime, endTime);

        var responseSizeKB = Buffer.byteLength(hit.page, 'utf8') / 1024;
        console.log("Size of Response ", responseSizeKB);
        var bandwidth = (responseSizeKB / (timing * 0.001));
        console.log("Bandwidth for response: ", bandwidth, "KB/s");
```

Setting the Cache:
This section sets the expiry and the data of the current request in the cache under the current url.

```
data = {
    expiryDate: expiry,
    page: rawData
}

// this section will cache the current url
let set = cache.set(current, data, 300);

if (set) {
    console.log("Set In Cache!\n");
} else {
    console.log("Not Set In Cache!\n");
}
```

Threaded Server

My code does not have explicit threading and instead implements an asynchronous callback functions such as the listener for the command line.

```
stdin.addListener("data", function (d) {
```

Similar is done with WebSocket requests.

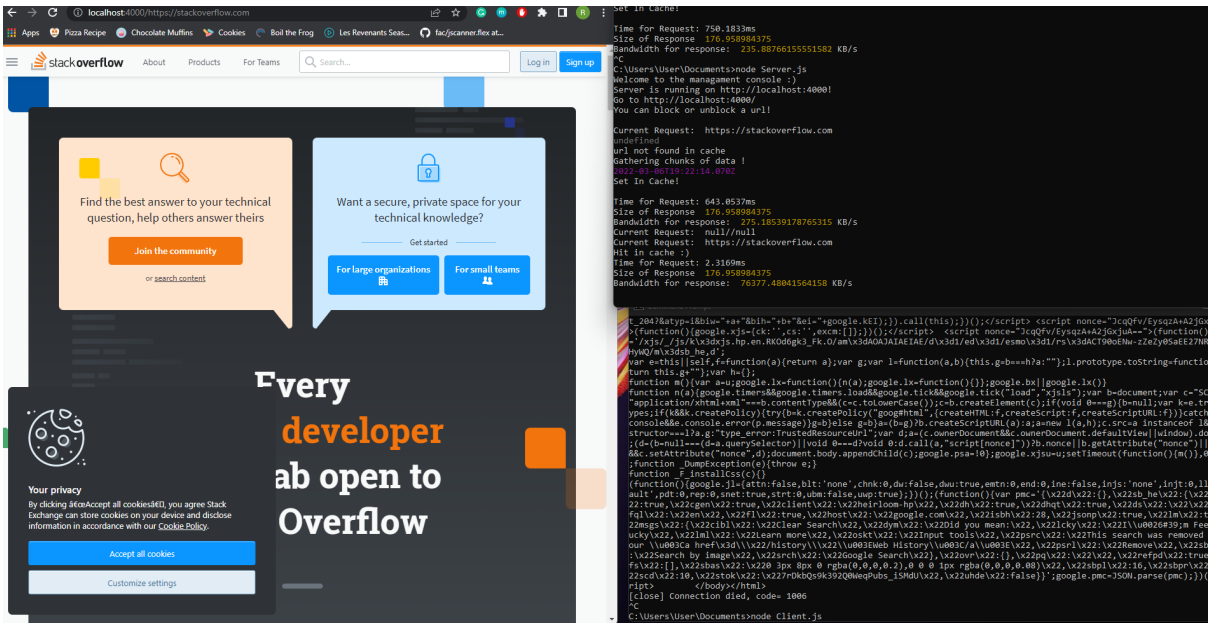
```
// WebSocket server
var wsServer = new ws.Server({ server });

// handles all websocket connections
wsServer.on('connection', (ws) => {
    console.log("Connected");

    ws.on('message', function (message) {
        console.log(message);
        wsOnReq(message, ws);
    });

    ws.on('close', function () {

    });
});
```

Regular requests will listen on the server for any requests as well.

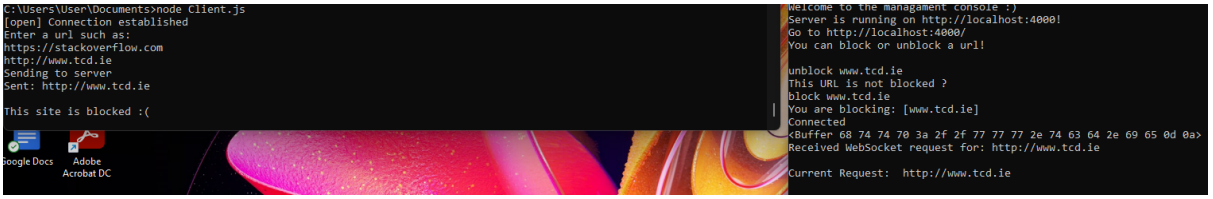
```
server.listen(port, host, () => {
    console.log(`Welcome to the managament console :)\nServer is running on http://${host}:${port}! \nGo to http://localhost:4000/`);
    console.log(`You can block or unblock a url!\n`);
});
```

Examples

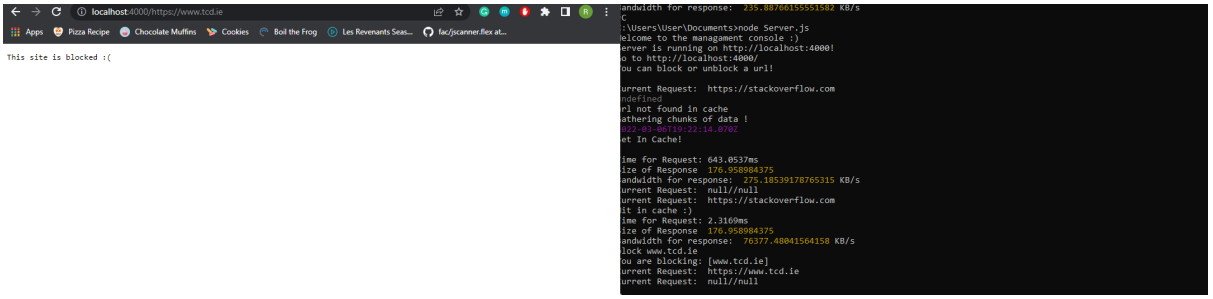
Highlighting the difference between cached and non-cached responses:



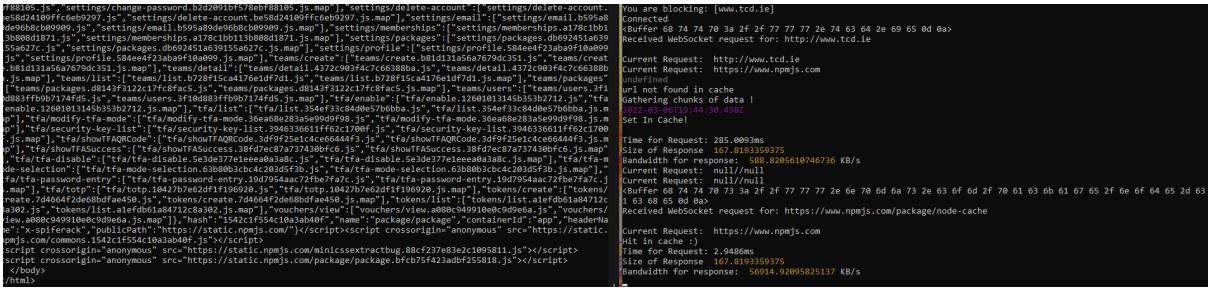
A blocked Websocket Request:



A blocked request:



Websocket Request:



Full Code

Server Code:

```
const http = require('http');
const https = require('https');
const url = require('url');
const SimpleHashTable = require('simple-hashtable');
const ws = require('ws');
const host = 'localhost';
const port = 4000;
const NodeCache = require("node-cache");
const { time } = require('console');

// ten min expiry in cache
var cache = new NodeCache({ stdTTL: 600, checkperiod: 600 });

// list of blocked sites
var blockedSites = new SimpleHashTable();

// Constants for timing
const NS_PER_SEC = 1e9
const MS_PER_NS = 1e6

var startTime = process.hrtime();
var endTime = process.hrtime();


// this function will handle requests http or https (only)
const onReq = function (request, response, socket, ws) {

    startTime = process.hrtime();

    // parse the url for a server request or a websocket request
    if (socket) {
        var current = url.parse(request.toString('utf8'), true);
    } else {
        var current = url.parse(request.url.substring(1), true);
    }

    console.log("Current Request: ", current.protocol + "//" +
current.host);

    var proxy;

    //  ensure it is a proper request
```

```
    if (current.path != 'favicon.ico' && current.host != 'assets' &&
current.host != null) {

        // get url from cache
        var hit = cache.get(current.href);

        // ensures the requested url is not blocked
        if (blockedSites.containsKey(current.hostname)) {
            if (socket) {
                ws.send("This site is blocked :(");
            } else {
                response.write("This site is blocked :(");
                response.end();
            }
        } else {

            // if there is nothing in the cache then send a http or
https request
            if (hit == undefined || hit == null) {

                if (current.protocol == "http:") {

                    proxy = http.get(current.href, (res) =>
responseHandler(current.href, res, response, socket, ws));

                } else if (current.protocol == "https:") {

                    proxy = https.get(current.href, (res) =>
responseHandler(current.href, res, response, socket, ws));

                } else {
                    if (!socket) {
                        response.write('Invalid request try a valid
request such as:\nhttp://localhost:4000/https://www.tcd.ie');
                        response.end();
                    } else {
                        ws.send('Invalid request try a valid request
such as:\nhttps://www.tcd.ie');
                    }
                }
            } else {
                if (!hit.err) {
```

```
                    var cachedExpiryDate =
Date.parse(hit.expiryDate);
                    var responseExpiryDate = new Date();

                    // check hit from cache is still valid
                    if (cachedExpiryDate > responseExpiryDate) {

                        console.log("Hit in cache :)");
                        if (!socket) {
                            response.write(hit.page);
                            response.end();
                        } else {
                            ws.send(hit.page);
                        }
                        endTime = process.hrtime();

                        const timing = processTime(startTime,
endTime);

                        var responseSizeKB =
Buffer.byteLength(hit.page, 'utf8') / 1024;
                        console.log("Size of Response ",
responseSizeKB);

                        // response size in KBs and timing is
multiplied by .001 to convert from ms to seconds
                        var bandwidth = (responseSizeKB / (timing *
0.001));
                        console.log("Bandwidth for response: ",
bandwidth, "KB/s");
                    } else {

                        console.log("Cached data has expired :(");
                        if (current.protocol == "http:") {

                            proxy = http.get(current.href, (res) =>
responseHandler(current.href, res, response, socket, ws));

                        } else if (current.protocol == "https:") {

                            proxy = https.get(current.href, (res) =>
```

```
responseHandler(current.href, res, response, socket, ws));

                        } else {
                            if (!socket) {
                                response.write('Invalid request try a
valid request such as:\nhttp://localhost:4000/https://www.tcd.ie');
                                response.end();
                            } else {
                                ws.send('Invalid request try a valid
request such as:\nhttps://www.tcd.ie');
                            }
                        }
                    }
                } else {
                    console.log("Error in cache");
                }
            }
        }
    } else {
        if (!socket) {
            response.end();
        }
    }
};

const processTime = function (startTime, endTime) {
    var time = -1;
    if (endTime != undefined) {
        const secondDiff = endTime[0] - startTime[0];
        const nanoSecondDiff = endTime[1] - startTime[1];
        const diffInNanoSecond = secondDiff * NS_PER_SEC +
nanoSecondDiff;
        time = diffInNanoSecond / MS_PER_NS;

        console.log("Time for Request: " + time + "ms");
    }
    return time;
}

// handles all responses
const responseHandler = function (current, res, response, socket, ws)
{
```

```
    const { statusCode } = res;

    console.log(res.headers['expires']);

    // get the expiry of the response, otherwise set it to ten mins
from current time
    var expiry = res.headers['expires'];
    if (expiry == undefined || expiry == -1) {
        expiry = new Date();
        expiry = new Date(expiry.getTime() + 10 * 60000);
    }

    let error;
    // Any 2xx status code signals a successful response but
    // here we're only checking for 200.
    if (statusCode !== 200) {
        error = new Error('Request Failed.\n' +
            `Status Code: ${statusCode}`);
    }

    if (error) {
        console.error(error.message);
        // Consume response data to free up memory
        res.resume();
        return;
    }

    let rawData = '';

    res.setEncoding('utf8');

    console.log("url not found in cache");

    console.log("Gathering chunks of data !");
    res.on('data', (chunk) => { rawData += chunk; });

    // Writing back to user
    res.on('end', () => {
        try {
            if (!socket) {
                response.write(rawData);
```

```
                response.end();
            } else {
                ws.send(rawData);
            }
        } catch (e) {
            console.error(e.message);
        }

        console.log(expiry);

        // create cache objext
        data = {
            expiryDate: expiry,
            page: rawData
        }

        // this section will cache the current url
        let set = cache.set(current, data, 300);

        if (set) {
            console.log("Set In Cache!\n");
        } else {
            console.log("Not Set In Cache!\n");
        }

        endTime = process.hrtime();

        const timing = processTime(startTime, endTime);

        var responseSizeKB = Buffer.byteLength(rawData, 'utf8') /
1024;
        console.log("Size of Response ", responseSizeKB);

        // response size in KBs and timing is multiplied by .001 to
convert from ms to seconds
        var bandwidth = (responseSizeKB / (timing * 0.001));
        console.log("Bandwidth for response: ", bandwidth, "KB/s");
    });

};

const stdin = process.openStdin();
```

```
// this function listens for input from the command line (for
blocking urls)
/* Command Structure
 *  block   www.example.com
 *  unblock www.example.com
*/
stdin.addListener("data", function (d) {

    var inputCommand = d.toString();
    var commandArray = inputCommand.split(" ");

    if (commandArray[0] == "block") {
        if (blockedSites.containsKey(commandArray[1].trim())) {
            console.log("This URL is already blocked :)");
        } else {
            blockedSites.put(commandArray[1].trim());
            console.log("You are blocking: [" +
commandArray[1].trim() + "]");
        }
    } else if (commandArray[0] == "unblock") {
        if (!blockedSites.containsKey(commandArray[1].trim())) {
            console.log("This URL is not blocked ?");
        } else {
            blockedSites.remove(commandArray[1].trim());
            console.log("You are unblocking: [" +
commandArray[1].trim() + "] :)");
        }
    } else {
        console.log("Unknown command");
    }

});

const server = http.createServer(onReq);

server.listen(port, host, () => {
    console.log(`Welcome to the management console :)\nServer is
running on http://${host}:${port}! \nGo to http://localhost:4000/`);
    console.log(`You can block or unblock a url!\n`);
});
```

```
// WebSocket server
var wsServer = new ws.Server({ server });

// handles all websocket connections
wsServer.on('connection', (ws) => {
    console.log("Connected");

    ws.on('message', function (message) {
        console.log(message);
        wsOnReq(message, ws);
    });

    ws.on('close', function () {

    });
});


// this function will handle requests from WebSocket connections
const wsOnReq = function (request, ws) {

    console.log("Received WebSocket request for: " + request);

    onReq(request, '', true, ws);

};
```

Client Code

```
const ws = require('ws');

let socket = new ws.WebSocket("ws://localhost:4000");

socket.onopen = function (e) {
    console.log("[open] Connection established");
    console.log("Enter a url such as: ")
    console.log("http://localhost:4000/https://stackoverflow.com");
};

socket.onmessage = function (event) {
    console.log(`${event.data}`);
```

```
};

socket.onclose = function (event) {
    if (event.wasClean) {
        console.log(`[close] Connection closed cleanly,
code=${event.code} reason=${event.reason}`);
    } else {
        console.log('[close] Connection died, code= '+ event.code);
    }
};

socket.onerror = function (error) {
    console.log(`[error] ${error.message}`);
};

const stdin = process.openStdin();

stdin.addListener("data", function (d) {

    var inputCommand = d.toString();

    console.log("Sending to server");
    socket.send(inputCommand);
    console.log("Sent: " + inputCommand);
});
```