

WEB PROXY SERVER

The language I used for this project is C#. I implemented a program named Web Proxy in the .NET framework. Below is the list of NuGet packages I utilized in the project:

NuGet Package	Purpose
System	Providing fundamental classes and base types for .NET applications
System.IO	Providing classes for reading and writing data to files and streams
System.Net	Providing networking functionality
System.Net.Sockets	Supporting lower-level network programming using TCP, UDP, and socket connections
System.Text	Providing classes for working with text and string manipulations
System.Threading.Tasks	Providing support for asynchronous programming and task-based parallelism
System.Net.Http	Providing classes for sending HTTP requests and receiving HTTP responses.
System.Collections.Generic	Providing generic collections (HashSet, List, Dictionary)
System.Net.http.Headers	Providing types for handling HTTP headers in requests and responses.

Inside the class `Globals`, we store all of the global variables I used for the web proxy.

```
public static class Globals
{
    //port number
    public static readonly Int32 PORT_NUMBER = 4000;

    //immutable client that enables communication over the HTTP
    public static readonly HttpClient httpClient = new HttpClient();

    public static Encoding ascii = Encoding.ASCII;

    //blocked URL set
    public static HashSet<string> blockedURLS = new HashSet<string>();
}
```

```
        //dictionary that stores the cached data (used in multi-threaded
scenario)
        public static ConcurrentDictionary<string, (DateTime, byte[])> cache =
new ConcurrentDictionary<string, (DateTime, byte[])>();

        //Variable used for timing
        public static DateTime start;

        public static DateTime end;

        //Variable used for retrieving the attached files in an HTTP/HTTPS
connection
        public static string lastHost = "";
    }
}
```

When the user starts the proxy, it will execute the `Program` class, where the proxy server is assigned to a separate thread while the management console is run on the main thread.

```
internal class Program
{
    static void Main(string[] args)
    {
        // Create instances of both components.
        ProxyServer proxyServer = new ProxyServer();
        ManagementConsole managementConsole = new ManagementConsole();

        Console.WriteLine($"Proxy Server started on port
{Globals.PORT_NUMBER}");
        // Start the proxy server in a separate thread.
        Thread proxyThread = new Thread(new
ThreadStart(proxyServer.StartProxy));
        proxyThread.Start();

        // Run the management console for URL commands on the main
thread
        managementConsole.Run();
    }
}
```

Besides, this program has two other classes; they are *ProxyServer* and *ManagementConsole*.

1. HTTP and HTTPS Requests (*ProxyServer*):

On the proxy thread, it starts by initializing a TCP listener that listens to incoming connections (any IP address) on Port 4000. It continuously accepts incoming TCP clients and the web proxy server will create a new thread or reuse an existing not-working thread to call the *HandleClient* function on this TCP client.

```
public void StartProxy()
{
    //accept client connections
    TcpListener listener = new TcpListener(IPAddress.Any,
Globals.PORT_NUMBER);
    listener.Start();
    while(true) {
        TcpClient client = listener.AcceptTcpClient();
        ThreadPool.QueueUserWorkItem(state =>
HandleClient(client));
    }
}
```

When the user types in the URL into the browser to create a connection on Port 4000 (<http://localhost:4000/URL>), the proxy reads the HTTP/HTTPS request from the client and then parses the request method (GET, POST, etc.) and URL. The program will need to manipulate the URL and handle relative URLs (URLs for CSS, Javascript, media files,...) by appending the last accessed host if necessary. The program then checks if the URL is blocked and returns an HTTP 403 Forbidden response if so. If not, the proxy will check whether the URL has been cached before and do the caching if available.

```
static async Task HandleClient(TcpClient client){
    try{
        //Establish a network stream for communication between the
client and the proxy
        using(NetworkStream clientStream = client.GetStream())
        //Read incoming HTTP requests from the client
        using(StreamReader reader = new StreamReader(clientStream,
Globals.ascii))
        //send HTTP responses back to the client using UTF-8 encoding.
// AutoFlush is set to true to ensure that data is automatically cleaned
after each write operation.
        using(StreamWriter writer = new StreamWriter(clientStream,
new UTF8Encoding(false)) {AutoFlush = true})
        {
            //Read the HTTP request from the browser
            string requestLine = await reader.ReadLineAsync();
```

```
//Request is empty
if (string.IsNullOrEmpty(requestLine)){
    ClosingClient(client);
    return;
}
//Parse method and URL
string[] requestParts = requestLine.Split(' ');
if (requestParts.Length < 2){
    Console.WriteLine("Invalid Request");
    ClosingClient(client);
    return;
}
string url = requestParts[1];
string method = requestParts[0];
if (url[0] == '/'){
    url = url.Remove(0,1);
}
if(!(url.StartsWith("http://") ||
url.StartsWith("https://"))) ){
    url = Globals.lastHost + "/" + url;
}
else{
    Globals.lastHost = url;
}
//Blocked URLs requested
if (Globals.blockedURLS.Contains(url)){
    string response = "HTTP/1.1 403 Forbidden";
    //Convert the blocking message above into bytes
    byte[] responseBytes = Encoding.UTF8.GetBytes(response);
    await clientStream.WriteAsync(responseBytes, 0,
responseBytes.Length);
    ClosingClient(client);
    return;
}
//Return true if the URL is cached
bool cachedUrl = await CacheFetching(method, url, writer,
clientStream);
if (cachedUrl){
    ClosingClient(client);
    return;
}
```

If the URL has not been cached, we will check whether it is an HTTP or HTTPS connection.

```
//Handle HTTPS CONNECT requests
    if (url.StartsWith("https://")){
        await HandleHttpsConnect(url, clientStream, writer,
client);
        await ForwardRequestToServer(method, url, clientStream,
writer);
    }
    else if (url.StartsWith("http://")){
        //Forward request to the actual web server
        await ForwardRequestToServer(method, url, clientStream,
writer);
    }
    else{
        //Console.WriteLine("[ERROR] Unsupported HTTP method.");
        await writer.WriteLineAsync("HTTP/1.1 405 Method Not
Allowed\r\n\r\n");
    }
    Console.WriteLine($"{requestLine} task is finished!")
```

a. Handle HTTPS Request:

If it is an HTTPS connection, the CONNECT request is created. The *HandleHttpsConnect* method establishes an HTTPS tunnel between the client and the target server as the proxy must relay encrypted data without modifying it. Firstly, the program parses the target host and port from the requested URL (defaulting to port 443 for HTTPS). Then it sends an HTTP 200 connection-established response to the client, signaling that the tunnel is ready. After that, the program implements bidirectional data transfer using *PipeStream*, which relays encrypted data between the client and the target server without modification. The function returns a 502 Bad Gateway error if the connection fails.

```
private static async Task HandleHttpsConnect(string url, NetworkStream
clientStream, StreamWriter writer, TcpClient client)
{
    try
    {
        //URL manipulation
        string newUrl = url.Remove(0,8);
        string[] hostParts = newUrl.Split("/");
        string host = hostParts[0];
        int port = 443;
        //Establish a connection to the target server using a new
TcpClient
        using (TcpClient server = new TcpClient())
```

```

        {
            try
            {
                //Connects the client to a remote TCP host using the
                specified host name and port number as an asynchronous operation.
                await server.ConnectAsync(host, port);
                using (NetworkStream serverStream =
server.GetStream())
                {
                    await writer.WriteLineAsync("HTTP/1.1 200
Connection Established\r\n\r\n");
                    await writer.FlushAsync();
                    Console.WriteLine($"[INFO] HTTPS Tunnel
Established to {host}:{port}");
                    //Asynchronously forwards data between the client
                    and the destination server for bidirectional data transfer
                    await Task.WhenAny(PipeStream(clientStream,
serverStream), PipeStream(serverStream, clientStream));
                }
            }
            catch (SocketException se)
            {
                await writer.WriteLineAsync("HTTP/1.1 502 Bad
Gateway");
            }
        }
    }
    catch (Exception ex)
    {
        await writer.WriteLineAsync("HTTP/1.1 502 Bad Gateway");
        await writer.WriteLineAsync("Content-Type: text/plain");
        await writer.WriteLineAsync();
        await writer.WriteLineAsync("Proxy server error: Unable to
connect to target server.");
    }
}

```

The *PipeStream* method forwards data from one *NetworkStream* (source) to another (destination). This helper function first allocates a large buffer for efficient data transfer, reads data from the source stream asynchronously in chunks, and writes the received data to the destination stream. Then it flushes the data to ensure it is immediately sent. This function performs synchronous non-blocking I/O to prevent performance bottlenecks and automatic handling of connection closures.

```

private static async Task PipeStream(NetworkStream source,
NetworkStream destination)

```

```

{
    byte[] buffer = new byte[65536*4];
    int bytesRead = 0;
    bytesRead = await source.ReadAsync(buffer, 0, buffer.Length);
    try
    {
        while ((bytesRead = await source.ReadAsync(buffer, 0,
buffer.Length)) > 0)
        {
            await destination.WriteAsync(buffer, 0, bytesRead);
            await destination.FlushAsync();
        }
    }
    catch (IOException)
    {
        //Console.WriteLine("[INFO] Connection closed.");
    }
    catch (Exception ex)
    {
        //Console.WriteLine($"[ERROR] Streaming error:
{ex.Message}");
    }
}
}

```

2. Forward Request To Server:

Since the proxy has implemented the tunnel for the HTTPS request, we can now retrieve the data from the website using the HTTP connection. First, the function creates an HTTP request using *HttpRequestMessage* with the specified method (GET, POST, etc.) and URL. Then it sends the request to the target server using *HttpClient* measures the response time and logs the time taken to fetch the resource. If the data is fetched successfully, the function will cache the response and call *pasteResponse* to forward the received response to the client. Otherwise, it returns an HTTP 500 error message if the request fails.

```

private static async Task ForwardRequestToServer(string method, string url,
NetworkStream clientStream, StreamWriter writer){
    try{
        //Create an HTTP request message with the specified method
(GET, POST, ...) and target URL
        HttpRequestMessage forwardRes = new HttpRequestMessage(new
HttpMethod(method), url);
        //Send the request to the target server using HttpClient and
wait for the response
        HttpResponseMessage serverRes = await
Globals.httpClient.SendAsync(forwardRes);
    }
}

```

```

        //Start tracking the response time
        Globals.start = DateTime.Now;
        //Read the content as byte array
        byte[] responseBytes = await
serverRes.Content.ReadAsByteArrayAsync();
        //End tracking the response time
        Globals.end = DateTime.Now;
        Console.WriteLine($"[TIMING] {url} took {(Globals.end -
Globals.start).TotalMilliseconds} ms to fetch from the server.");
        //Cache the response by storing the response in the cache
with a timestamp
        if (responseBytes != null){
            Globals.cache[url] = (DateTime.Now, responseBytes);
        }
        //Forward the response to the client
        await pasteResponse(responseBytes, serverRes, clientStream,
writer);
    }
    catch (Exception ex){
        writer.WriteLine("HTTP/1.1 500 Internal Server Error");
        writer.WriteLine("Content-Type: text/plain");
        writer.WriteLine();
        writer.WriteLine("Proxy server error");
    }
}

```

The following method forwards an HTTP response from the target server to the client. It handles both chunked and non-chunked responses and ensures that the response is properly formatted before sending it to the client.

```

private static async Task pasteResponse(byte[] responseBytes,
HttpResponseMessage serverRes, NetworkStream clientStream, StreamWriter
writer)
{
    try
    {
        //Write the HTTP response status line
        await writer.WriteLineAsync($"HTTP/{serverRes.Version}
{ (int)serverRes.StatusCode} {serverRes.ReasonPhrase}");
        //Forward the Content-Type header to client
        if (serverRes.Content.Headers.ContentType != null)
        {
            await writer.WriteLineAsync($"Content-Type:
{serverRes.Content.Headers.ContentType}");
        }
    }
}

```



```
//Forward all response headers to the client
foreach (var header in serverRes.Headers)
{
    await writer.WriteLineAsync($"{header.Key}:
{string.Join(", ", header.Value)}");
}

// Handle chunked encoding explicitly
if (serverRes.Headers.TransferEncodingChunked.HasValue &&
serverRes.Headers.TransferEncodingChunked.Value)
{
    await writer.WriteLineAsync("Transfer-Encoding:
chunked");

    await writer.WriteLineAsync();
    await writer.FlushAsync();

    using (var responseStream = await
serverRes.Content.ReadAsStreamAsync())
    {
        byte[] buffer = new byte[8192];
        int bytesRead;

        while ((bytesRead = await
responseStream.ReadAsync(buffer, 0, buffer.Length)) > 0)
        {
            await writer.WriteAsync($"{bytesRead:X}\r\n");
            await clientStream.WriteAsync(buffer, 0,
bytesRead);

            await
clientStream.WriteAsync(Encoding.ASCII.GetBytes("\r\n"), 0, 2);
            await clientStream.FlushAsync();
        }

        await writer.WriteAsync("0\r\n\r\n");
        await clientStream.FlushAsync();
    }
}
else
{
    // Non-chunked response:
    if (serverRes.Content.Headers.ContentLength.HasValue)
    {
        await writer.WriteLineAsync($"Content-Length:
{serverRes.Content.Headers.ContentLength.Value}");
    }
}
```

```
        await writer.WriteLineAsync();
        await writer.FlushAsync();

        await clientStream.WriteAsync(responseBytes, 0,
responseBytes.Length);
        await clientStream.FlushAsync();
    }
}

catch (Exception ex)
{
    //Console.WriteLine($"[ERROR] Error sending response:
{ex.Message}");
    await writer.WriteLineAsync("HTTP/1.1 500 Internal Server
Error\r\n\r\n");
    await writer.FlushAsync();
}
}
```

3. Close the client:

```
static async Task ClosingClient(TcpClient client){
    await Task.Delay(TimeSpan.FromMilliseconds(2));
    client.Close();
}
```

2. Dynamically Blocked URLs (*ProxyServer* & *ManagementConsole* & *Globals*):

For this proxy feature, I decided to use the data structure *HashSet* to keep track of the blocked URLs most efficiently (looking up, adding, and removing take constant time) while ensuring that the blocked URLs are all unique.

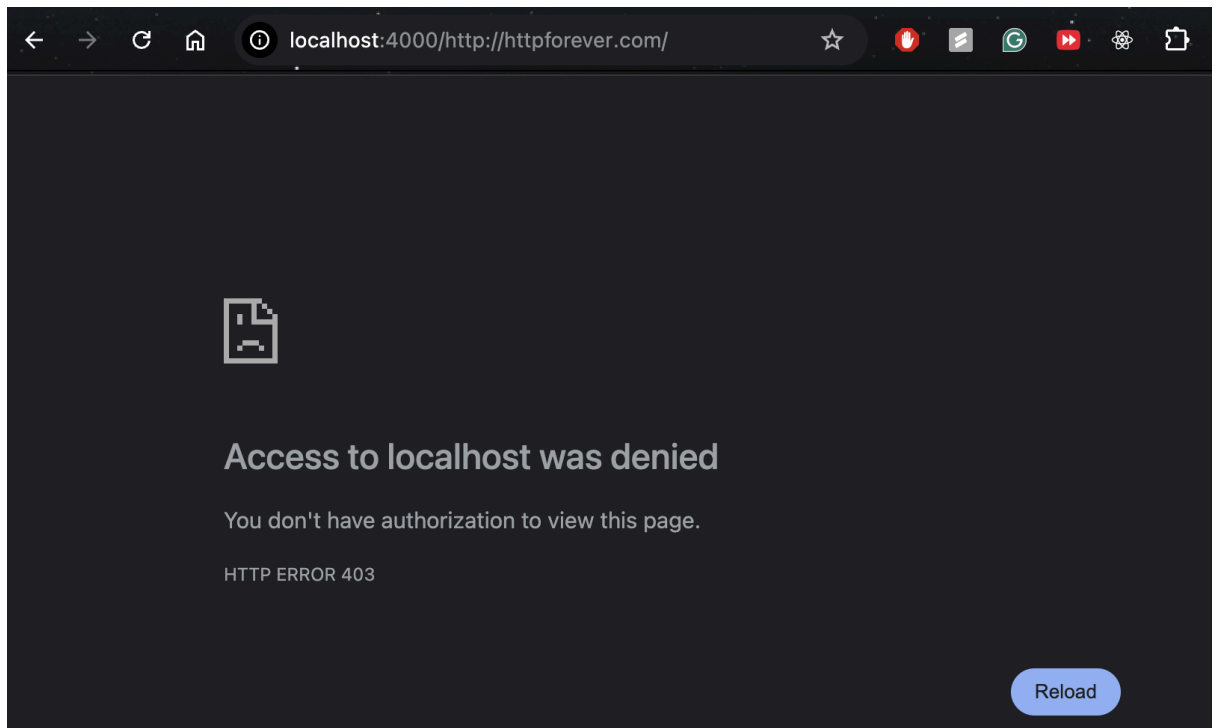
```
//blocked URL set
public static HashSet<string> blockedURLS = new HashSet<string>();
```

When we call the *HandleClient* function on the input URL, the function will immediately check whether the URL is in the blocked list. If so, the proxy will notify the client that this site is blocked (status code is 403) and end the client connection.

```
if (Globals.blockedURLS.Contains(url)) {
    Console.WriteLine($"Urgh Oh! {url} is blocked!");
    string response = "HTTP/1.1 403 Forbidden";
    //Convert the blocking message into bytes
    byte[] responseBytes = Encoding.UTF8.GetBytes(response);
```

```
        await clientStream.WriteAsync(responseBytes, 0,
responseBytes.Length);
        ClosingClient(client);
        return;
    }
}
```

The user can manually block or unblock a specific URL using the management console (the command users need to use: **[B/b]lock/[U/u]nblock URL**) as well as display the list of URLs they have blocked so far (command: **[I/L]ist**). These are the only commands that the user can use in the management console so if they input a different one, it will print out the error message. For example, I blocked the website <http://httpforever.com> and this is the screen of the browser:



3. Cache HTTP Requests (*ProxyServer*):

```
public static ConcurrentDictionary<string, (DateTime, byte[])> cache = new
ConcurrentDictionary<string, (DateTime, byte[])>();
```

To cache the HTTP request, I utilized the Dictionary data structure of C# to store pairs of <Date and Time value - cached data> as the <key-value> in the *Global* class. Since C# has a built-in library that can find the difference between timestamps, I decided to store the time a URL is cached and then later when we try to fetch the cached data, we can calculate the time difference to make sure that the cache has not been expired yet.

The function *HandleClient* checks whether the input URL is cached or not by calling the asynchronous *CacheFetching* function.

```
        bool cachedUrl = await CacheFetching(method, url, writer,
clientStream);

        if (cachedUrl){
            ClosingClient(client);
            Console.WriteLine("[DEBUG] Cached Data
Successfully!");

            return;
        }
    }
```

If the latter successfully fetches the cached data of the previously visited URL and returns true, the proxy will notify the user that “CACHE HIT” and close the client connection.

```
static async Task<bool> CacheFetching(string method, string url, StreamWriter
writer, NetworkStream clientStream){
    //If we are trying to implement the CONNECT method, caching should
not be used

    if (method == "CONNECT"){
        return false;
    }

    HttpRequestMessage forwardRes = new HttpRequestMessage(new
HttpMethod(method), url);

    HttpResponseMessage serverRes = await
Globals.httpClient.SendAsync(forwardRes);

    //Check whether the method is "GET" and the URL is already
cached (contained in the cache dictionary)
    if (method == "GET" && Globals.cache.ContainsKey(url)){
        //Record the start time of the fetching process
        Globals.start = DateTime.Now;

        //Retrieve the timestamp the data cached and the cached
data

        var (timestamp, cachedData) = Globals.cache[url];
        //Check if the cache is expired after 10 minutes
        if((DateTime.Now - timestamp).TotalMinutes < 10){

            Console.WriteLine($"[CACHE HIT] {url}");
            //If available, pass the cached data to the client stream
            await pasteResponse(cachedData, serverRes,
clientStream, writer);

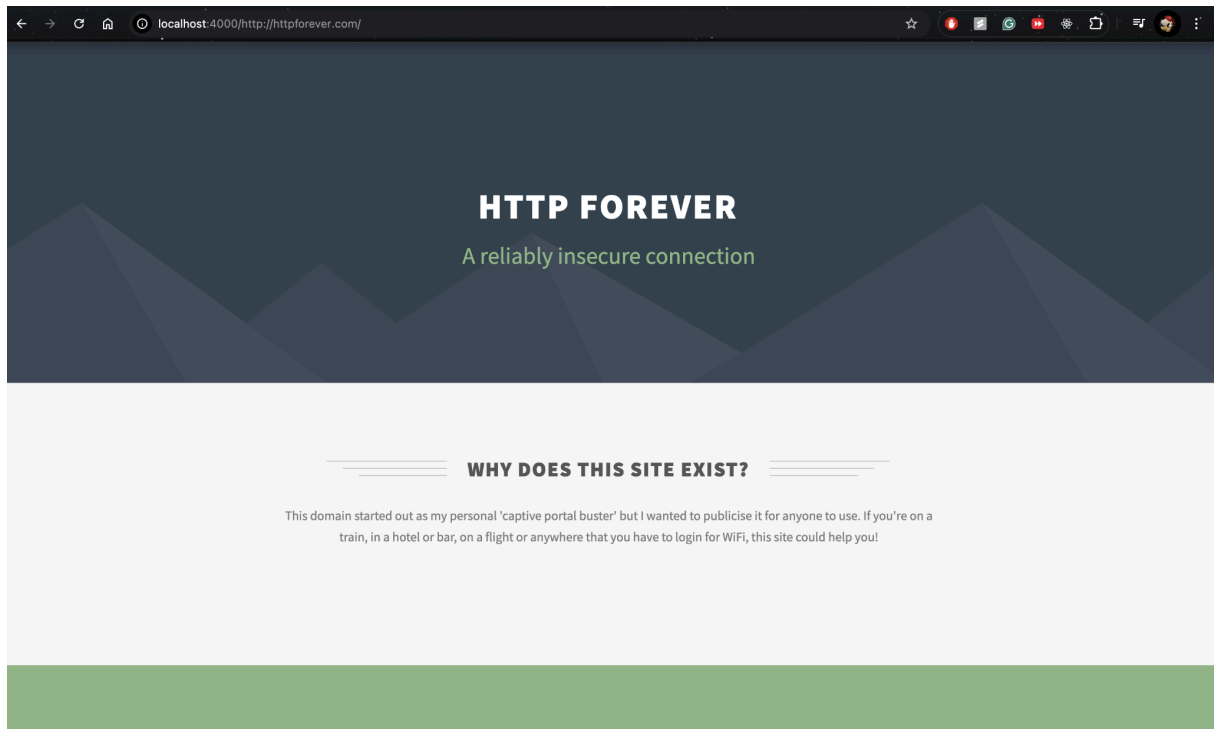
            //End recording time
            Globals.end = DateTime.Now;
            Console.WriteLine($"[CACHE TIMING] {url} took
{(Globals.end - Globals.start).TotalMilliseconds} ms");

            //Return true if we can fetch the cached data
            return true;
        }
    }
}
```

```
        //If the data has been cached for more than 10 minutes,
remove it from the dictionary
        else{
            Globals.cache.TryRemove(url, out var temp);
        }
    }

    //Return false if we cannot fetch the cached data
    return false;
}
```

For example, I want to access the website: <http://httpforever.com/>.



Fetches from origin server	Fetches from cache
<p>[TIMING] http://httpforever.com/ took 3.975 ms to fetch from the origin server. GET /http://httpforever.com/ HTTP/1.1 task is finished!</p> <p>[TIMING] http://httpforever.com/js/init.min.js took 0.266 ms to fetch from the origin server. GET /http://httpforever.com/js/init.min.js HTTP/1.1 task is finished!</p> <p>[TIMING] http://httpforever.com/css/style-wide.min.css took 0.281 ms to fetch from the origin server. GET</p>	<p>[CACHE HIT] http://httpforever.com/ [CACHE TIMING] http://httpforever.com/ took 0.741 ms [CACHE HIT] http://httpforever.com/js/init.min.js [CACHE TIMING] http://httpforever.com/js/init.min.js took 0.164 ms [CACHE HIT] http://httpforever.com/css/style-wide.min.css [CACHE TIMING] http://httpforever.com/css/style-wide.min.css took 0.433 ms</p>

<pre> /http://httpforever.com/css/style-wide.min.cs s HTTP/1.1 task is finished! [TIMING] http://httpforever.com/css/style.min.css took 0.399 ms to fetch from the origin server. GET /http://httpforever.com/css/style.min.css HTTP/1.1 task is finished! [TIMING] http://httpforever.com/css/images/header-ma jor-on-dark.svg took 0.165 ms to fetch from the origin server. GET /http://httpforever.com/css/images/header-m ajor-on-dark.svg HTTP/1.1 task is finished! [TIMING] http://httpforever.com/css/images/header-ma jor-on-light.svg took 0.136 ms to fetch from the origin server. GET /http://httpforever.com/css/images/header-m ajor-on-light.svg HTTP/1.1 task is finished! [TIMING] http://httpforever.com/css/images/banner.svg took 0.166 ms to fetch from the origin server. GET /http://httpforever.com/css/images/banner.sv g HTTP/1.1 task is finished! </pre>	<pre> [CACHE HIT] http://httpforever.com/css/style.min.css [CACHE TIMING] http://httpforever.com/css/style.min.css took 0.477 ms [CACHE HIT] http://httpforever.com/css/images/banner.svg [CACHE TIMING] http://httpforever.com/css/images/banner.svg took 0.321 ms [CACHE HIT] http://httpforever.com/css/images/header-ma jor-on-dark.svg [CACHE TIMING] http://httpforever.com/css/images/header-ma jor-on-dark.svg took 0.217 ms [CACHE HIT] http://httpforever.com/css/images/header-ma jor-on-light.svg [CACHE TIMING] http://httpforever.com/css/images/header-ma jor-on-light.svg took 0.212 ms </pre>
---	--

From this scenario, we can see that caching is more time-efficient than fetching from the origin server. The proxy can also cache HTTPS requests.

4. Threaded Server (*ProxyServer*):

To handle requests simultaneously, I decided to use *ThreadPool*, which can reuse threads instead of creating a new thread/request every time the proxy accepts a new TCP client.

```

while(true) {
    TcpClient client = listener.AcceptTcpClient();
    ThreadPool.QueueUserWorkItem(state => HandleClient(client));
}

```

Also, I utilized a thread-safe concurrent dictionary for caching to ensure safe concurrent reads and writes without requiring explicit locking, making it ideal for caching and shared data storage in a proxy server.

```


public static ConcurrentDictionary<string, (DateTime, byte[])> cache =
new ConcurrentDictionary<string, (DateTime, byte[])>();

```

5. Disadvantages:

- The connection is really slow and unstable
- For some HTTPS connections, it requires the users to reload the webpage (the request cached) to display the website properly

6. Appendix:

- GitHub Link: <https://github.com/FayNguyen03/Web-Server-Proxy.git>
- Video Link:  Project 1.mp4