Web Proxy Server — Design and Implementation

Repo link: https://github.com/VivianShong/web-proxy

1. Overview

This document describes the design and implementation of a forward proxy server written in Go. The proxy intercepts HTTP and HTTPS traffic between a browser and the internet, providing four core capabilities:

1. **HTTP and HTTPS support** — transparently forwarding both plain and encrypted web traffic.
2. **Dynamic URL blocking** — an administrator can block or unblock hostnames at runtime via a web management console.
3. **HTTP response caching** — responses are cached in memory; subsequent requests use conditional GET to avoid redundant data transfer.
4. **Concurrent request handling** — a goroutine-per-connection model allows many simultaneous clients.

The server has no external library dependencies beyond the Go standard library.

---

2. System Architecture

The proxy runs two TCP servers concurrently:

Server	Port	Purpose
Proxy listener	`:4000`	Accepts browser connections; handles HTTP and HTTPS
Management console	`:8080`	Web dashboard for blocking URLs and viewing logs

A single `ProxyState` singleton (protected by a `sync.RWMutex`) is shared between both servers. It holds the blocked-host list, response cache, request logs, and the set of currently open HTTPS tunnels.

```
┌────────────────────────────────────────────────────────┐
│                        Browser                         │
│    (configured to use HTTP proxy at localhost:4000)    │
└────────────────────────────┬───────────────────────────┘
                             │ TCP
              ┌──────────────▼─────────────┐
              │        Proxy (:4000)       │
              │     handleConnection()     │
              └──────┬─────────────┬───────┘
                     │             │
        CONNECT      │             │   GET/POST/...
                     │             │
        ┌────────────▼──┐    ┌─────▼─────────────────┐
        │ handleHTTPS   │    │ handleHTTP            │
        │ (TLS tunnel)  │    │ (HTTP + cache)        │
        └───────┬───────┘    └────────┬──────────────┘
```

```
              │                  │
     ┌────────▼──────────────────▼────────┐
     │              ProxyState            │
     │  BlockedHosts  Cache  RequestLogs  │
     │  activeConns (for instant kill)    │
     └────────────────────────────────────┘
                        │
          ┌─────────────▼─────────┐
          │  Management (:8080)   │
          │  dashboard.html       │
          └───────────────────────┘
```

---

3. Protocol Design

3.1 HTTP Proxying

When a browser is configured to use a proxy, it sends full absolute URLs in its `GET` request line instead of relative paths:

```
GET http://example.com/page.html HTTP/1.1
Host: example.com
```

The proxy:

1. Parses the host, port, and path from the absolute URL.
2. Checks if the host is blocked; returns `403 Forbidden` if so.
3. Opens a new TCP connection to the origin server.
4. Forwards a rewritten request (using the relative path `/page.html`), stripping proxy-specific headers and adding `Connection: close`.
5. Streams the response back to the browser, buffering the body simultaneously for the cache (using `io.TeeReader`).

On subsequent requests for the same URL, if the cache holds the previous response, the proxy adds `If-Modified-Since` or `If-None-Match` headers. If the server responds `304 Not Modified`, the proxy serves the locally cached body to the browser — saving the full response payload from being re-transferred over the network.

3.2 HTTPS CONNECT Tunneling

HTTPS traffic cannot be intercepted without decrypting TLS, which would require installing a custom root certificate in the browser. This proxy instead uses the standard **CONNECT tunnel** method:

1. The browser sends `CONNECT example.com:443 HTTP/1.1`.
2. The proxy opens a raw TCP connection to `example.com:443`.
3. The proxy replies `200 Connection Established`.
4. From this point, both sides of the TCP connection are wired together with `io.Copy` — the proxy passes bytes blindly in both directions. The TLS handshake and all subsequent encrypted frames

flow through without modification.

This design means HTTPS content is never decrypted by the proxy, preserving security and privacy.

3.3 Concurrency Model

Each accepted TCP connection spawns a goroutine (`go handleConnection(conn)`). HTTPS tunneling additionally spawns a second goroutine for one direction of the bidirectional byte copy. All shared state is accessed under a `sync.RWMutex`:

- **Read lock (`RLock`)** for queries: `IsBlocked`, `GetFromCache`, `GetLogs`, `GetBlocked`, `GetCacheKeys`.
- **Write lock (`Lock`)** for mutations: `Block`, `Unblock`, `AddToCache`, `LogRequest`, `RegisterConn`, `UnregisterConn`.

3.4 Active Connection Killing

When an administrator blocks a domain that has live HTTPS tunnels, the proxy immediately terminates those connections. Each HTTPS tunnel registers its client `net.Conn` in `activeConns[domain]` on startup and deregisters it on teardown. `Block()` iterates the set and calls `conn.Close()`, which causes the blocking `io.Copy` call to return with an error, unwinding the goroutine.

3.5 Subdomain Blocking

`IsBlocked` performs an exact match and a suffix match. Blocking `google.com` automatically blocks `www.google.com`, `mail.google.com`, and any other subdomain, because `strings.HasSuffix(host, ".google.com")` evaluates to true for all of them.

---

4. File Structure

```
web-proxy/
├── main.go           — Core proxy: connection dispatch, HTTP, HTTPS,
caching
├── proxyState.go     — Shared state: blocking, caching, logging,
connection tracking
├── cache.go          — Cache data structures (CacheEntry, Cache)
├── management.go     — HTTP management server on :8080
├── dashboard.html    — Management console HTML template
├── blocked.json      — Persisted blocked-host list (JSON)
└── go.mod            — Go module definition (no external dependencies)
```

---

5. Key Data Structures

```go
// A single entry in the response cache.
type CacheEntry struct {
    Body    []byte      // Full HTTP response body bytes
    Header http.Header // Response headers (map[string][]string)
}

// The in-memory cache, keyed by full URL string.
type Cache struct {
    entries map[string]CacheEntry
}

// A record of one proxied request, stored in the log ring buffer.
type RequestLog struct {
    Time   time.Time
    Method string             // "GET", "CONNECT", etc.
    URL    string
    Status string             // "Allowed", "Blocked", "Cached", "Error"
    SrcIP  string             // client's IP:port
}

// The central singleton shared between the proxy and management server.
type ProxyState struct {
    mu            sync.RWMutex
    BlockedHosts map[string]bool                       // hostname → blocked?
    RequestLogs  []RequestLog                          // newest-first,
capped at 100
    LogLimit     int                                   // = 100
    Cache        *Cache
    activeConns  map[string]map[net.Conn]struct{}   // domain → set of
open HTTPS tunnels
}

// Data passed to the HTML template when rendering the dashboard.
type PageData struct {
    Blocked    []string
    CachedKeys []string
    Logs       []RequestLog
}
```

6. Request Flow Diagrams

6.1 HTTP — Cache Miss

```
Browser                        Proxy                        Origin Server
  |                              |                              |
  |--GET http://ex.com/------->|                              |
  |                              |--GET /  HTTP/1.1---------->|
  |                              |   Connection: close          |
  |                              |                              |
  |                              |<--HTTP/1.1 200 OK----------|
  |<--HTTP/1.1 200 OK---------|   (TeeReader buffers body) |
  |   (body streamed live)     |--AddToCache(url, entry)--->|
  |                              |   (stored after body done) |
```

6.2 HTTP — Cache Hit (304)

```
Browser                     Proxy                      Origin Server
  |                            |                            |
  |--GET http://ex.com/------>|                            |
  |                            | GetFromCache → hit         |
  |                            |--GET /  HTTP/1.1---------->|
  |                            |   If-Modified-Since: ...   |
  |                            |                            |
  |                            |<--HTTP/1.1 304 Not Modified|
  |<--HTTP/1.1 200 OK---------|                            |
  |   (served from cache,      |                            |
  |    no body re-transferred)|                            |
```

6.3 HTTPS CONNECT Tunnel

```
Browser                     Proxy                      example.com:443
  |                            |                            |
  |--CONNECT ex.com:443------>|                            |
  |                            |--TCP connect-------------->|
  |<--200 Connection Estab----|                            |
  |                            |                            |
  |===TLS ClientHello========>|=========================>   |
  |<==TLS ServerHello=========|<=========================   |
  |  (all bytes pass through blindly; proxy cannot read)   |
```

---

7. Annotated Code Listing

7.1 `main.go` — Core Proxy Logic

```go
package main

import (
    "bufio"
    "bytes"
    "fmt"
    "io"
    "log"
    "net"
    "net/http"
    "strconv"
    "strings"
    "time"
)
```

```go
// state is the global singleton holding all shared proxy data.
// It is initialised once in main() and read/written concurrently
// by all goroutines via its own mutex.
var state *ProxyState

func main() {
    // Create and populate the shared state.
    state = NewProxyState()

    // Restore the blocked-host list persisted from a previous run.
    if err := state.LoadBlocked("blocked.json"); err != nil {
        log.Println("No blocked list found, starting fresh.")
    }

    // Run the management HTTP server on a separate goroutine so it
    // does not block the proxy's accept loop below.
    go StartManagementServer(":8080", state)

    // Open the proxy's TCP listening socket.
    listener, err := net.Listen("tcp", ":4000")
    if err != nil {
        log.Fatal("Error listening:", err)
    }
    log.Println("Proxy listening on :4000")
    defer listener.Close()

    // Accept loop: one goroutine per connection.
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Println("Error accepting conn:", err)
            continue // don't die on a single bad accept
        }
        go handleConnection(conn)
    }
}

// handleConnection is the entry point for every client TCP connection.
// It reads the first request line to determine the HTTP method and
// target URL, then dispatches to the appropriate handler.
func handleConnection(clientConn net.Conn) {
    // Ensure the client socket is closed when this goroutine exits,
    // regardless of which path through the function is taken.
    defer clientConn.Close()

    // Wrap the connection in a buffered reader so we can read line-by-
line.
    reader := bufio.NewReader(clientConn)

    // Read the HTTP request line, e.g.:
    //   "GET http://example.com/ HTTP/1.1\r\n"
    //   "CONNECT example.com:443 HTTP/1.1\r\n"
    message, err := reader.ReadString('\n')
    if err != nil {
        log.Printf("Read error: %v", err)
        return
    }

    message = strings.TrimSpace(message)
    messageFields := strings.Fields(message) // split on whitespace
    if len(messageFields) < 2 {
        return // malformed request line; drop the connection
    }
    method := messageFields[0] // "GET", "POST", "CONNECT", …
    url := messageFields[1]    // full URL or "host:port" for CONNECT

    // Extract just the hostname for the blocking check.
    host, _, _ := parseURL(url)

    // Block check: if this host (or its parent domain) is blocked,
    // return HTTP 403 and close the connection immediately.
    if state.IsBlocked(host) {
        logRequest(method, url, "Blocked", clientConn)
        clientConn.Write([]byte("HTTP/1.1 403 Forbidden\r\n\r\n<h1>Access
Denied</h1>"))
        return
    }

    // Route based on method:
    //   CONNECT → blind TLS tunnel (HTTPS)
    //   anything else → HTTP with caching
    if method == "CONNECT" {
        handleHTTPS(clientConn, url)
    } else {
        handleHTTP(clientConn, reader, method, url)
    }
}

// handleHTTPS implements the CONNECT tunnelling method.
// It does NOT decrypt TLS — it simply wires the browser and the
// origin server together at the TCP level.
func handleHTTPS(clientConn net.Conn, target string) {
    logRequest("CONNECT", target, "Allowed", clientConn)

    // Open a raw TCP connection to the destination (e.g.,
example.com:443).
    serverConn, err := net.Dial("tcp", target)
    if err != nil {
        log.Printf("HTTPS failed to connect to %s: %v", target, err)
        clientConn.Write([]byte("HTTP/1.1 502 Bad Gateway\r\n\r\n"))
        return
    }
    defer serverConn.Close()

    // Inform the browser that the tunnel is ready.
    // After this point, the browser proceeds with its TLS handshake.
    clientConn.Write([]byte("HTTP/1.1 200 Connection
Established\r\n\r\n"))
```

```
    log.Printf("  → TUNNEL to %s", target)

    // Register this connection so Block() can close it if needed.
    domain, _, _ := net.SplitHostPort(target)
    state.RegisterConn(domain, clientConn)
    defer state.UnregisterConn(domain, clientConn)

    // Copy bytes in both directions concurrently.
    // One goroutine handles browser → server; this goroutine handles
    // server → browser (blocking until the server closes the connection).
    go io.Copy(serverConn, clientConn)
    io.Copy(clientConn, serverConn)
}

// handleHTTP proxies a plain HTTP request with in-memory caching.
func handleHTTP(clientConn net.Conn, reader *bufio.Reader, method, url
string) {
    host, port, path := parseURL(url)
    address := host + ":" + port

    // Consume all client request headers up to the blank line.
    headers := readClientHeaders(reader)
    if headers == nil {
        return // connection closed or malformed
    }

    // Look up the URL in the cache.
    // cachedEntry is only valid when hasCache is true.
    cachedEntry, hasCache := state.GetFromCache(url)

    // Dial a fresh connection to the origin server.
    // (HTTP keep-alive is not used upstream; Connection: close is
forced.)
    serverConn, err := net.Dial("tcp", address)
    if err != nil {
        clientConn.Write([]byte("HTTP/1.1 502 Bad Gateway\r\n\r\n"))
        return
    }
    defer serverConn.Close()

    // Write the request line and headers to the server.
    // If a cached entry exists, If-Modified-Since / If-None-Match
    // conditional headers are added automatically.
    sendRequestToServer(serverConn, method, path, headers, cachedEntry,
hasCache)
    log.Printf("  → request to %s", address)

    // Forward any request body (e.g., POST payload) asynchronously.
    // The goroutine reads from the buffered reader (which already
consumed
    // the request line and headers) and pipes the rest to the server.
    go io.Copy(serverConn, reader)

    // Read the server's response status line.
    serverReader := bufio.NewReader(serverConn)
    statusCode, statusLine := readResponseStatus(serverReader)
    if statusLine == "" {
        return
    }

    // If we had a cached entry and the server says "not modified",
    // serve directly from cache — no body bytes cross the network.
    if hasCache && statusCode == 304 {
        log.Printf("CACHE HIT: %s", url)
        logRequest(method, url, "Cached", clientConn)
        sendCachedResponse(clientConn, cachedEntry)
        return
    }

    // Otherwise stream the full response to the browser and cache it.
    logRequest(method, url, "Allowed", clientConn)
    streamAndCacheResponse(clientConn, serverReader, statusLine, url)
}

// readClientHeaders reads HTTP header lines until the blank line
// that separates headers from the body. Returns nil on read error.
func readClientHeaders(reader *bufio.Reader) []string {
    var headers []string
    for {
        line, err := reader.ReadString('\n')
        if err != nil {
            return nil
        }
        trimmed := strings.TrimSpace(line)
        if trimmed == "" {
            break // blank line signals end of headers
        }
        headers = append(headers, trimmed)
    }
    return headers
}

// sendRequestToServer writes a rewritten HTTP request to the server.
// It filters headers that should not be forwarded (proxy-specific and
// hop-by-hop headers) and appends conditional caching headers when
// a cached entry is available.
func sendRequestToServer(serverConn net.Conn, method, path string,
    headers []string, cachedEntry CacheEntry, hasCache bool) {

    // Write the request line using the relative path, not the full URL.
    fmt.Fprintf(serverConn, "%s %s HTTP/1.1\r\n", method, path)

    // Forward headers, skipping those that must not be proxied.
    for _, header := range headers {
        if shouldSkipHeader(header) {
            continue
        }
        fmt.Fprintf(serverConn, "%s\r\n", header)
    }

    // Force the server to close the connection after one response.
    // This simplifies response framing: we read until EOF.
    fmt.Fprintf(serverConn, "Connection: close\r\n")

    // Conditional GET: if we have a cached response, tell the server
    // what version we already have so it can return 304 if unchanged.
    if hasCache {
        if lastMod := cachedEntry.Header.Get("Last-Modified"); lastMod !=
"" {
            fmt.Fprintf(serverConn, "If-Modified-Since: %s\r\n", lastMod)
        }
        if etag := cachedEntry.Header.Get("ETag"); etag != "" {
            fmt.Fprintf(serverConn, "If-None-Match: %s\r\n", etag)
        }
    }

    // Blank line terminates the headers section.
    fmt.Fprintf(serverConn, "\r\n")
}

// shouldSkipHeader returns true for headers that must not be forwarded.
// This includes proxy-specific headers and hop-by-hop headers that
// are meaningful only for a single network hop.
func shouldSkipHeader(header string) bool {
    lower := strings.ToLower(header)
    skipPrefixes := []string{
        "proxy-",           // Proxy-Authorization, Proxy-Connection, etc.
        "connection:",      // hop-by-hop
        "if-modified-since:", // managed by the proxy itself
        "if-none-match:",     // managed by the proxy itself
    }
    for _, prefix := range skipPrefixes {
        if strings.HasPrefix(lower, prefix) {
            return true
        }
    }
    return false
}

// readResponseStatus reads the first line of the HTTP response and
// returns both the numeric status code and the full status line string.
func readResponseStatus(reader *bufio.Reader) (int, string) {
    statusLine, err := reader.ReadString('\n')
    if err != nil {
        return 0, ""
    }
    statusCode := 0
    parts := strings.Fields(statusLine)
    if len(parts) >= 2 {
        statusCode, _ = strconv.Atoi(parts[1])
    }
    return statusCode, statusLine
```

```go
}

// logRequest records a proxied request in the shared state log.
func logRequest(method, url, status string, clientConn net.Conn) {
    state.LogRequest(RequestLog{
        Time:   time.Now(),
        Method: method,
        URL:    url,
        Status: status,
        SrcIP:  clientConn.RemoteAddr().String(),
    })
}

// streamAndCacheResponse forwards the server response to the browser
// and simultaneously stores the body in the in-memory cache.
//
// Key technique: io.TeeReader duplicates the byte stream — every byte
// read from 'reader' is written to 'bodyBuf' AND returned to io.Copy
// for forwarding to the client. This means the client sees no additional
// latency; bytes are forwarded as they arrive from the server.
func streamAndCacheResponse(clientConn net.Conn, reader *bufio.Reader,
    statusLine, url string) {

    // Forward the status line (e.g., "HTTP/1.1 200 OK\r\n").
    if _, err := clientConn.Write([]byte(statusLine)); err != nil {
        return
    }

    // Read, forward, and parse response headers simultaneously.
    // We need the header values to store in the cache, but must also
    // forward each line to the client before we know the body size.
    headers := make(http.Header)
    for {
        line, err := reader.ReadString('\n')
        if err != nil {
            return
        }
        // Forward this header line to the client immediately.
        if _, err := clientConn.Write([]byte(line)); err != nil {
            return
        }
        if strings.TrimSpace(line) == "" {
            break // blank line: end of headers
        }
        // Parse the header for caching (split on first colon only).
        parts := strings.SplitN(line, ":", 2)
        if len(parts) == 2 {
            key := strings.TrimSpace(parts[0])
            val := strings.TrimSpace(parts[1])
            lowerKey := strings.ToLower(key)
            // Do not cache hop-by-hop headers; they are meaningless
            // for the reconstructed cached response.
            if lowerKey == "connection" || lowerKey == "keep-alive" ||
                lowerKey == "proxy-connection" || lowerKey == "te" ||
                lowerKey == "upgrade" {
                continue
            }
            headers.Add(key, val)
        }
    }

    // TeeReader: read body from server, write to bodyBuf AND pipe to
client.
    // After io.Copy returns (server closes connection), bodyBuf holds the
    // complete body and we can store it in the cache.
    var bodyBuf bytes.Buffer
    io.Copy(clientConn, io.TeeReader(reader, &bodyBuf))

    // Store the complete response in the cache for future requests.
    state.AddToCache(url, CacheEntry{
        Header: headers,
        Body:   bodyBuf.Bytes(),
    })
}

// sendCachedResponse reconstructs and sends a full HTTP 200 response
// from a CacheEntry. Used when the server returns 304 Not Modified.
func sendCachedResponse(clientConn net.Conn, entry CacheEntry) {
    // Always send 200 OK to the browser; the 304 is a proxy-server
detail.
    clientConn.Write([]byte("HTTP/1.1 200 OK\r\n"))
    fmt.Fprintf(clientConn, "Connection: close\r\n")

    // Replay the stored response headers.
    for key, values := range entry.Header {
        if strings.ToLower(key) == "connection" {
            continue // skip; we already wrote our own
        }
        for _, value := range values {
            fmt.Fprintf(clientConn, "%s: %s\r\n", key, value)
        }
    }
    clientConn.Write([]byte("\r\n")) // end of headers
    clientConn.Write(entry.Body)     // cached body
}

// parseURL extracts the host, port, and path components from a URL.
//
// It handles three input forms:
//   "http://example.com/page"     → host="example.com", port="80",
path="/page"
//   "https://example.com:8443/p"  → host="example.com", port="8443",
path="/p"
//   "example.com:443"             → host="example.com", port="443",
path="/"
//   "[::1]:8080"                  → host="[::1]",        port="8080",
path="/"
func parseURL(url string) (host, port, path string) {
    // Step 1: identify and strip the scheme; set default port.
    if strings.HasPrefix(url, "http://") {
        port = "80"
        url = url[7:] // strip "http://"
    } else if strings.HasPrefix(url, "https://") {
        port = "443"
        url = url[8:] // strip "https://"
    } else {
        port = "443" // bare "host:port" form used by CONNECT
    }

    // Step 2: separate the path component.
    path = "/"
    if idx := strings.Index(url, "/"); idx != -1 {
        path = url[idx:]
        url = url[:idx] // url now contains only "host" or "host:port"
    }

    // Step 3: strip HTTP authentication credentials (user:pass@host).
    if idx := strings.LastIndex(url, "@"); idx != -1 {
        url = url[idx+1:]
    }

    // Step 4: split host and port, handling IPv6 literals like
[::1]:8080.
    // For IPv6, the last ":" before a digit sequence is the port
separator,
    // but only if it comes after the closing "]". We compare the position
    // of the last ":" with the last "]" to distinguish the two.
    colonIdx := strings.LastIndex(url, ":")
    bracketIdx := strings.LastIndex(url, "]")

    if colonIdx != -1 && colonIdx > bracketIdx {
        // A colon exists and is not inside an IPv6 literal bracket.
        host = url[:colonIdx]
        port = url[colonIdx+1:]
    } else {
        host = url // no explicit port; keep the scheme default
    }

    return
}
```

7.2 `proxyState.go` — Shared State

```
package main

import (
    "encoding/json"
```

```go
	"log"
	"net"
	"os"
	"strings"
	"sync"
	"time"
)

// RequestLog records metadata about a single proxied request.
// JSON tags are included so logs could be serialised to disk if needed.
type RequestLog struct {
	Time   time.Time `json:"time"`
	Method string    `json:"method"`
	URL    string    `json:"url"`
	Status string    `json:"status"` // "Blocked", "Allowed", "Cached",
"Error"
	SrcIP  string    `json:"src_ip"`
}

// ProxyState is the central store for all shared mutable data.
// A single sync.RWMutex protects every field; callers must not access
// fields directly – use the provided methods instead.
type ProxyState struct {
	mu           sync.RWMutex
	BlockedHosts map[string]bool                    // lowercase hostname →
blocked
	RequestLogs  []RequestLog                       // newest-first ring
buffer
	LogLimit     int                                // maximum entries to
keep (100)
	Cache        *Cache
	activeConns  map[string]map[net.Conn]struct{} // domain → set of open
HTTPS conns
}

// NewProxyState constructs an empty ProxyState with all maps initialised.
func NewProxyState() *ProxyState {
	return &ProxyState{
		BlockedHosts: make(map[string]bool),
		RequestLogs:  make([]RequestLog, 0),
		LogLimit:     100,
		Cache:        NewCache(),
		activeConns:  make(map[string]map[net.Conn]struct{}),
	}
}

// RegisterConn records an open HTTPS tunnel connection for a domain.
// This allows Block() to forcibly close it if the domain is later
blocked.
// Uses a map-as-set idiom (map[net.Conn]struct{}) for O(1) insert/delete.
func (s *ProxyState) RegisterConn(domain string, conn net.Conn) {
	s.mu.Lock()
	defer s.mu.Unlock()
	if s.activeConns[domain] == nil {
        s.activeConns[domain] = make(map[net.Conn]struct{})
    }
    s.activeConns[domain][conn] = struct{}{}
}

// UnregisterConn removes a connection from the active set when the
// tunnel closes naturally. Cleans up the domain key when the set is
empty.
func (s *ProxyState) UnregisterConn(domain string, conn net.Conn) {
    s.mu.Lock()
    defer s.mu.Unlock()
    if set, ok := s.activeConns[domain]; ok {
        delete(set, conn)
        if len(set) == 0 {
            delete(s.activeConns, domain) // remove empty set
        }
    }
}

// Block adds a host to the blocked list and immediately closes any
// active HTTPS tunnels to that domain. Existing HTTP requests to the
// domain will complete, but the browser will be denied on its next
attempt.
func (s *ProxyState) Block(host string) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.BlockedHosts[strings.ToLower(host)] = true

    // Close all live HTTPS tunnels to this domain.
    // conn.Close() causes io.Copy to return with an error in the
    // goroutine, which then exits cleanly via defer UnregisterConn.
    if conns, ok := s.activeConns[host]; ok {
        log.Printf("Killing %d active connections to %s", len(conns),
host)
        for conn := range conns {
            conn.Close()
        }
        delete(s.activeConns, host)
    }
}

// Unblock removes a host from the blocked list.
// Existing connections are not affected; only future requests are
permitted.
func (s *ProxyState) Unblock(host string) {
    s.mu.Lock()
    defer s.mu.Unlock()
    delete(s.BlockedHosts, strings.ToLower(host))
}

// IsBlocked returns true if the host, or any parent domain of the host,
// appears in the blocked list.
//
// Example: blocking "google.com" also blocks "www.google.com" because
```

```go
//   strings.HasSuffix("www.google.com", ".google.com") == true
func (s *ProxyState) IsBlocked(host string) bool {
    s.mu.RLock()
    defer s.mu.RUnlock()
    host = strings.ToLower(host)
    if s.BlockedHosts[host] {
        return true // exact match
    }
    // Subdomain match: check whether any blocked entry is a suffix of
host.
    for blocked := range s.BlockedHosts {
        if strings.HasSuffix(host, "."+blocked) {
            return true
        }
    }
    return false
}

// AddToCache stores a response in the cache under the full URL as key.
func (s *ProxyState) AddToCache(key string, resp CacheEntry) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.Cache.entries[key] = resp
}

// GetFromCache retrieves a cached response. The second return value
// indicates whether the key was found.
func (s *ProxyState) GetFromCache(key string) (CacheEntry, bool) {
    s.mu.RLock()
    defer s.mu.RUnlock()
    entry, exists := s.Cache.entries[key]
    return entry, exists
}

// LogRequest prepends a log entry to the ring buffer.
// The slice is kept at most LogLimit elements by truncating the tail.
func (s *ProxyState) LogRequest(req RequestLog) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.RequestLogs = append([]RequestLog{req}, s.RequestLogs...) // newest
first
    if len(s.RequestLogs) > s.LogLimit {
        s.RequestLogs = s.RequestLogs[:s.LogLimit]
    }
}

// GetLogs returns a defensive copy of the log slice so the caller
// cannot accidentally modify shared state without holding the lock.
func (s *ProxyState) GetLogs() []RequestLog {
    s.mu.RLock()
    defer s.mu.RUnlock()
    logs := make([]RequestLog, len(s.RequestLogs))
    copy(logs, s.RequestLogs)
    return logs
}

// GetBlocked returns a snapshot of the blocked-host list as a string
slice.
func (s *ProxyState) GetBlocked() []string {
	s.mu.RLock()
	defer s.mu.RUnlock()
	blocked := make([]string, 0, len(s.BlockedHosts))
	for host := range s.BlockedHosts {
		blocked = append(blocked, host)
	}
	return blocked
}

// SaveBlocked serialises the blocked-host map to a JSON file.
// Called after every Block/Unblock so state survives a restart.
func (s *ProxyState) SaveBlocked(filename string) error {
	s.mu.RLock()
	defer s.mu.RUnlock()
	data, err := json.MarshalIndent(s.BlockedHosts, "", "  ")
	if err != nil {
		return err
	}
	return os.WriteFile(filename, data, 0644)
}

// LoadBlocked deserialises a JSON file into the blocked-host map.
// Called once at startup. A missing file is not an error.
func (s *ProxyState) LoadBlocked(filename string) error {
	s.mu.Lock()
	defer s.mu.Unlock()
	data, err := os.ReadFile(filename)
	if err != nil {
		if os.IsNotExist(err) {
			return nil // first run; no file yet
		}
		return err
	}
	return json.Unmarshal(data, &s.BlockedHosts)
}

// GetCacheKeys returns a snapshot of the URLs currently in the cache.
func (s *ProxyState) GetCacheKeys() []string {
	s.mu.RLock()
	defer s.mu.RUnlock()
	keys := make([]string, 0, len(s.Cache.entries))
	for key := range s.Cache.entries {
		keys = append(keys, key)
	}
	return keys
}
```

7.3 `cache.go` — Cache Data Structures

```go
package main

import (
	"net/http"
	"sync"
)

// CacheEntry holds one complete HTTP response body and its headers.
// The Header field uses net/http.Header (map[string][]string) so that
// multi-value headers (e.g., Set-Cookie) are preserved correctly.
type CacheEntry struct {
	Body   []byte      // raw response body bytes
	Header http.Header // response headers, excluding hop-by-hop headers
}

// Cache wraps the entries map. All locking is performed by ProxyState.mu
// via the AddToCache / GetFromCache methods; Cache.mu is unused.
type Cache struct {
	mu      sync.RWMutex          // declared for future per-cache locking
	entries map[string]CacheEntry // URL string → cached response
}

// NewCache allocates an empty Cache with an initialised entries map.
func NewCache() *Cache {
	return &Cache{
		entries: make(map[string]CacheEntry),
	}
}
```

7.4 `management.go` — Management Console Server

```go
package main

import (
	"html/template"
	"log"
	"net/http"
	"strings"
)

// PageData is the view model passed to dashboard.html.
// All fields are derived from a snapshot of ProxyState at request time.
type PageData struct {
	Blocked    []string     // currently blocked hostnames
	CachedKeys []string     // URLs present in the response cache
	Logs       []RequestLog // recent requests, newest first
}
```

```
// truncate shortens a string to at most 'length' characters.
// Used in the template to keep table cells readable.
func truncate(s string, length int) string {
    if len(s) > length {
        return s[:length] + "..."
    }
    return s
}

// StartManagementServer starts an HTTP server on 'addr' that serves
// the management dashboard. It is intended to run in its own goroutine.
func StartManagementServer(addr string, state *ProxyState) {
    // Parse the HTML template once at startup, registering the truncate
    // helper function so it is available in the template.
    tmpl :=
template.Must(template.New("dashboard.html").Funcs(template.FuncMap{
        "truncate": truncate,
    }).ParseFiles("dashboard.html"))

    // GET / — render the dashboard with a current snapshot of state.
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        data := PageData{
            Blocked:    state.GetBlocked(),
            CachedKeys: state.GetCacheKeys(),
            Logs:       state.GetLogs(),
        }
        if err := tmpl.Execute(w, data); err != nil {
            log.Printf("Template error: %v", err)
            http.Error(w, "Internal Server Error",
http.StatusInternalServerError)
        }
    })

    // POST /block — add a hostname to the blocked list.
    // The host field is parsed through parseURL so that a user can
    // paste a full URL (e.g., "http://example.com/page") as well as
    // a bare hostname ("example.com").
    http.HandleFunc("/block", func(w http.ResponseWriter, r *http.Request)
{
        if r.Method == "POST" {
            host := strings.TrimSpace(r.FormValue("host"))
            host, _, _ = parseURL(host)
            state.Block(host)
            state.SaveBlocked("blocked.json") // persist immediately
        }
        // Post/Redirect/Get: redirect to dashboard to prevent form
resubmission.
        http.Redirect(w, r, "/", http.StatusSeeOther)
    })

    // POST /unblock — remove a hostname from the blocked list.
    http.HandleFunc("/unblock", func(w http.ResponseWriter, r
*http.Request) {
        if r.Method == "POST" {
            host := strings.TrimSpace(r.FormValue("host"))
            host, _, _ = parseURL(host)
            state.Unblock(host)
            state.SaveBlocked("blocked.json")
        }
        http.Redirect(w, r, "/", http.StatusSeeOther)
    })

    log.Printf("Management console listening on %s", addr)
    log.Fatal(http.ListenAndServe(addr, nil))
}
```

7.5 `dashboard.html` — Management Console UI

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Proxy Management Console</title>
    <style>
        body { font-family: monospace; max-width: 900px; margin: 2rem
auto; padding: 0 1rem; }
        .container { display: flex; gap: 2rem; }
        .column { flex: 1; }
        table { width: 100%; border-collapse: collapse; margin-bottom:
2rem; }
        th, td { text-align: left; padding: 0.5rem; border-bottom: 1px
solid #ddd; }
        th { background: #f4f4f4; }
        .blocked { color: red; }
        .allowed { color: green; }
        .cached  { color: blue; }
        input[type="text"] { padding: 0.5rem; width: 70%; }
        button { padding: 0.5rem 1rem; cursor: pointer; }
    </style>
</head>
<body>
    <h1>Proxy Management Console</h1>

    <!--
      Three-column layout:
        Left   — Blocked Hosts: add/remove entries from the block list
        Centre — Cached URLs:   show what the proxy has cached in memory
        Right  — Recent Requests: rolling log of the last 100 requests
    -->
    <div class="container">

        <!-- ===== LEFT: Blocked Hosts ===== -->
        <div class="column">
```

```
            <h2>Blocked Hosts</h2>

            <!-- Form to block a new host -->
            <form action="/block" method="POST" class="form-group">
                <input type="text" name="host" placeholder="example.com" required>
                <button type="submit">Block</button>
            </form>

            <!-- Table listing currently blocked hosts, each with an Unblock button -->
            <table>
                <thead><tr><th>Host</th><th>Action</th></tr></thead>
                <tbody>
                    {{range .Blocked}}
                    <tr>
                        <td>{{.}}</td>
                        <td>
                            <!-- Hidden form: one Unblock button per row -->
                            <form action="/unblock" method="POST" style="display:inline;">
                                <input type="hidden" name="host" value="{{.}}">
                                <button type="submit">Unblock</button>
                            </form>
                        </td>
                    </tr>
                    {{else}}
                    <tr><td colspan="2">No blocked hosts</td></tr>
                    {{end}}
                </tbody>
            </table>
        </div>

        <!-- ===== CENTRE: Cached URLs ===== -->
        <div class="column">
            <h2>Cached URLs</h2>
            <table>
                <thead><tr><th>URL</th></tr></thead>
                <tbody>
                    {{range .CachedKeys}}
                    <tr>
                        <!--
                          Truncate long URLs to 40 characters for display,
                          but expose the full URL in the title tooltip.
                        -->
                        <td title="{{.}}">{{truncate . 40}}</td>
                    </tr>
                    {{else}}
                    <tr><td>No cached items</td></tr>
                    {{end}}
                </tbody>
            </table>
        </div>

        <!-- ===== RIGHT: Recent Requests ===== -->
        <div class="column">
            <h2>Recent Requests</h2>
            <p><a href="/">Refresh</a></p>
            <table>
                <thead>
                    <tr><th>Time</th><th>Method</th><th>URL</th>
<th>Status</th></tr>
                </thead>
                <tbody>
                    {{range .Logs}}
                    <tr class="log-entry">
                        <td>{{.Time.Format "15:04:05"}}</td>
                        <td>{{.Method}}</td>
                        <td title="{{.URL}}">{{truncate .URL 40}}</td>
                        <!--
                          Apply CSS class to colour-code the status:
                            Blocked → red
                            Cached  → blue
                            Allowed → green
                        -->
                        <td class="{{if eq .Status "Blocked"}}blocked
                                   {{else if eq .Status "Cached"}}cached
                                   {{else}}allowed{{end}}">
                            {{.Status}}
                        </td>
                    </tr>
                    {{else}}
                    <tr><td colspan="4">No requests yet</td></tr>
                    {{end}}
                </tbody>
            </table>
        </div>

    </div>
</body>
</html>
```

8. Design Decisions and Trade-offs

Decision	Rationale
Goroutine per connection	Simple model; Go's scheduler handles thousands of goroutines efficiently. No connection pool overhead.
Single `sync.RWMutex` for all state	Minimises lock contention between the management server (infrequent writes) and the proxy goroutines (frequent reads).

Decision	Rationale
`io.TeeReader` for caching	Avoids buffering the entire body before sending — the client receives bytes as they arrive from the server. Cache population happens as a side effect of the forward.
`Connection: close` forced upstream	Eliminates the need to parse `Content-Length` or `Transfer-Encoding: chunked` to know when the response body ends — we read until EOF.
CONNECT tunnel (no TLS interception)	Preserves end-to-end TLS security. No custom certificate authority required. HTTPS content is not visible to the proxy.
Subdomain blocking via suffix match	A single blocked entry covers the apex domain and all of its subdomains, which is the common administrator expectation.
TTL cache + conditional GET on expiry	Within the TTL window the origin is never contacted (zero bandwidth, ~500× latency improvement). After expiry a conditional GET (`If-Modified-Since` / `If-None-Match`) lets the origin return a cheap 304 instead of resending the full body when content is unchanged — 99.9% bandwidth saving on revalidation. If the content changed, a normal 200 response replaces the cache entry.
Kill active HTTPS tunnels on block	When a domain is added to the block list, all open `CONNECT` tunnels to that domain are closed immediately. Without this, a browser that already established a tunnel can continue sending HTTPS requests through it even after the block is applied, because the blocking check only runs at the start of `handleConnection`. Closing the `net.Conn` breaks the `io.Copy` loop instantly, forcing the browser to reconnect — at which point the block check fires and returns 403.
JSON persistence for blocked hosts	Simple, human-readable format. The block list survives proxy restarts without a database.
No external dependencies	The entire proxy is built on the Go standard library, making it easy to compile and deploy (`go build`).

9. Cache Strategy: TTL + Conditional GET

9.1 Design

The proxy combines a **TTL (time-to-live) cache** with **conditional GET revalidation** to give three distinct behaviours depending on the age of the cached entry. Constants in cache.go control the policy:

```
const cacheTTL        = 5 * time.Minute
const cacheMaxEntries = 100
```

`handleHTTP` in main.go implements the three-way lookup:

```
// Case 1 — Fresh hit (within TTL): serve from memory, no origin contact.
// Case 2 — Stale hit (TTL expired): conditional GET to origin.
//          304 Not Modified → reset TTL, serve from cache.
//          200 OK           → stream new body, replace cache entry.
// Case 3 — Miss (never seen): full unconditional GET, cache the response.
```

Case	Condition	Origin contacted?	Body transferred?	TTL reset?
Fresh hit	Entry exists, `Fresh()` = true	No	No	No (still valid)
304 revalidation	Entry exists, `Fresh()` = false, origin returns 304	Yes (headers only)	No	Yes
200 re-fetch	Entry exists, `Fresh()` = false, origin returns 200	Yes (full response)	Yes	Yes
Miss	No entry	Yes (full response)	Yes	N/A

`GetFromCache` returns a fresh entry; `GetStaleFromCache` returns an expired entry so `handleHTTP` can use its `Last-Modified` / `ETag` headers to build the conditional request. On a 304 response, `AddToCache` is called with the same entry to re-stamp `CachedAt`, resetting the TTL without refetching the body.

Size limit: before every insert, `evict()` ensures the map never exceeds 100 entries (expired entries purged first, then the oldest fresh entry).

9.2 Bandwidth Savings Measured

The following results were produced by `TestBandwidthSummary` and `TestBandwidthSavedOnCacheHit` in latency_test.go. Each test starts a real HTTP origin server, sends the same URL through the proxy twice, and counts the bytes the origin actually transmitted.

Per-request origin bandwidth (origin → proxy):

Body Size	Cache Miss (request 1)	Cache Hit (request 2)	Bytes Saved	Saving
10 KB	10,443 B	0 B	10,443 B	100.0%
100 KB	102,604 B	0 B	102,604 B	100.0%
500 KB	512,204 B	0 B	512,204 B	100.0%

On a TTL cache hit, **zero bytes** travel from the origin to the proxy. The proxy serves the entire body from in-process memory.

`TestBandwidthSavedAcrossMultipleRequests` confirms this holds for repeated requests: requests 2 through 6 for the same URL each cost 0 origin bytes, while the client receives the full 51,357-byte body every time.

`TestCacheExpiry` verifies the expiry path: after backdating `CachedAt` past the TTL, the next request re-fetches the full body from the origin.

9.3 Latency Improvement Measured

Because the TTL cache eliminates the origin round-trip entirely, cached requests are dramatically faster. Results from `TestCachedLatencyFasterThanUncached` and `TestLatencySummary` in latency_test.go:

Simulated Origin Delay	Cache Miss	TTL Hit (avg)	Time Saved	Speedup
10 ms	12 ms	<1 ms	11 ms	~75×
50 ms	51 ms	<1 ms	51 ms	~489×
100 ms	102 ms	<1 ms	101 ms	~958×

The sub-millisecond cache hit time is the cost of a memory lookup and a TCP write to the browser — no DNS, no TCP connect, no origin processing time.

`TestConditionalGETLatency` adds the 304 revalidation path (50 ms origin delay):

Path	Latency	Note
Cache miss	51 ms	1 RTT + body
TTL hit	<1 ms	0 RTT, served from memory
304 revalidation	52 ms	1 RTT, no body transmitted

The 304 path costs the same RTT as a miss because the origin still processes the request — but it avoids transmitting the body. For large resources (images, scripts) this is a significant bandwidth saving at no extra latency cost.

9.4 Expired TTL: Conditional GET Revalidation

When the TTL elapses the proxy does not immediately discard the entry. Instead it performs a **conditional GET**, attaching `If-Modified-Since` (and `If-None-Match` if an ETag was present) from the stale entry's headers. The origin can then reply cheaply with `304 Not Modified` instead of resending the full body.

Case A — TTL expired, data unchanged (304 path)

`TestConditionalGETBandwidth` measures this with a 100 KB body:

Step	Origin Bytes	Client Bytes	Note
Request 1 — cache miss	102,589 B	102,589 B	Full fetch; entry stored with `Last-Modified`
Request 2 — TTL hit	0 B	102,589 B	Served from memory
Request 3 — 304 revalidation	**131 B**	102,589 B	Headers only; body served from cache

Step	Origin Bytes	Client Bytes	Note
Request 4 — TTL hit	0 B	102,589 B	TTL reset by revalidation

The 131 bytes on request 3 are HTTP response headers only — **99.9% bandwidth saving** compared to a full re-fetch. The client still receives the complete 102,589-byte body, served from the in-memory cache. `AddToCache` is called with the same entry to re-stamp `CachedAt`, resetting the TTL without refetching the body.

Case B — TTL expired, data modified (200 path)

If the origin content has changed it returns `200 OK` with a new body. The proxy streams and caches the new body exactly as on a fresh miss. `TestExpiredTTLModifiedData` confirms correctness:

Step	Origin Bytes	Content	Note
Request 1 — cache miss	51,342 B	v1	Initial fetch of v1
Request 2 — TTL hit	0 B	v1	Served from memory
[origin updated to v2; cache expired]			
Request 3 — conditional GET → 200	51,342 B	**v2**	Origin changed; full body received
Request 4 — TTL hit	0 B	v2	v2 now cached

The proxy **never serves stale content beyond the TTL window**. After expiry it always contacts the origin — either getting a cheap 304 or the new content.

Trade-off summary

Scenario	Origin bytes	Client bytes	Latency vs. no cache	Content freshness
Within TTL	0 B	full body	**~500× faster** (0 RTT)	At most 5 min stale
Expired, unchanged (304)	~131 B headers	full body (from cache)	~1 RTT (same as miss)	Always current
Expired, modified (200)	full body	full body	~1 RTT + body	Always current

The key benefit of conditional GET over a plain re-fetch: when content is unchanged, **the body is never retransmitted** even though the origin is contacted.

9.5 Cache Size Limit and Eviction

Without a size bound the cache map would grow indefinitely, one entry per unique URL ever requested. Two constants in cache.go cap this:

```
const cacheTTL        = 5 * time.Minute
const cacheMaxEntries = 100
```

`AddToCache` calls `Cache.evict()` before every insert. `evict()` runs in two passes while the `ProxyState` write lock is already held:

```go
func (c *Cache) evict() {
    // Pass 1: drop all expired entries (no value keeping them)
    for k, e := range c.entries {
        if !e.Fresh() {
            delete(c.entries, k)
        }
    }
    if len(c.entries) < cacheMaxEntries {
        return
    }
    // Pass 2: still at limit — remove the single oldest fresh entry
    var oldestKey string
    var oldestTime time.Time
    for k, e := range c.entries {
        if oldestKey == "" || e.CachedAt.Before(oldestTime) {
            oldestKey = k
            oldestTime = e.CachedAt
        }
    }
    if oldestKey != "" {
        delete(c.entries, oldestKey)
    }
}
```

Eviction priority:

1. All expired entries are deleted first — they carry no useful data.
2. Only if the cache is still full after purging expired entries is a fresh entry removed, choosing the one with the earliest `CachedAt` (the least-recently-added entry).

`TestCacheEviction` in latency_test.go verifies both properties:

- **Sub-test 1 — oldest fresh entry at limit**: fills the cache to 100, backdates each entry so `url/0` is oldest. Adding entry 101 evicts `url/0` and holds the size at 100.
- **Sub-test 2 — expired before fresh**: fills with 99 fresh entries plus 1 expired entry. Adding a new entry evicts only the expired one; all 99 fresh entries are preserved.

---

10. Limitations

- **No HTTP keep-alive to the origin.** Every HTTP request opens a new TCP connection to the origin server. This increases latency and load on the origin for pages with many sub-resources.

- **Caching is HTTP-only.** HTTPS responses are never cached because the proxy cannot read the encrypted content.
- **Cache key is the raw URL string.** There is no normalisation (e.g., query-parameter ordering), so `?a=1&b=2` and `?b=2&a=1` are stored as two separate entries.
- **No `Cache-Control` directives are honoured.** Resources marked `no-store` or `private` are cached anyway.