



## Build a Load Balancer Using Go Language

Source code of this project – <https://github.com/Thisarak943/go-loadbalancer>

### Project Description:

This project implements a **simple yet functional HTTP Load Balancer** written in the **Go programming language**. The main objective of this project is to distribute incoming HTTP requests across multiple backend servers in a **round-robin** fashion, ensuring efficient resource utilization, improved availability, and scalability of web applications.

The Load Balancer sits between clients and a pool of backend servers. When a client sends a request, the Load Balancer determines which backend server should handle that request and forwards it accordingly using Go's built-in `httputil.ReverseProxy`. This helps balance the traffic load evenly, avoiding situations where one server becomes overloaded while others remain underutilized.

The project also provides a foundation for implementing more advanced load balancing features such as health checks, failover handling, and SSL termination. It demonstrates how Go's **concurrency, networking libraries, and simplicity** make it an ideal language for building scalable infrastructure tools.



THISARA KANDAGE

UNDERGRADUATE - SLIIT

E-mail LinkedIn GitHub Website

## Key Features

### 1. Reverse Proxy Implementation

- Utilizes Go's `httputil.NewSingleHostReverseProxy` to forward HTTP requests to target servers.
- Maintains the original client request and returns the server's response transparently.

### 2. Round-Robin Load Balancing Algorithm

- Each incoming request is forwarded to the next available backend server in a cyclic order.
- Ensures even distribution of requests across all backend servers.

### 3. Modular Design with Interfaces

- The `Server` interface defines the contract for backend servers, including methods like `Addr()`, `IsAlive()`, and `Serve()`.
- This design supports easy extension (e.g., adding more server types or health check logic).

### 4. Error Handling and Logging

- Comprehensive error handling with the `handleErr()` helper function.
- Logs all forwarded requests, making it easy to trace routing behavior.

### 5. Scalability and Flexibility

- New backend servers can be added to the `servers` slice with minimal code changes.
- Supports both HTTP and HTTPS endpoints.

## ❖ System Architecture

### Client → Load Balancer → Backend Servers

1. The client sends a request to the Load Balancer (e.g., `http://localhost:8000`).
2. The Load Balancer selects the next backend server using the **round-robin algorithm**.
3. The ReverseProxy forwards the request to the chosen backend server.
4. The response from the backend is relayed back to the client through the Load Balancer.

### Snapshots of the project:

The screenshot shows a code editor interface with a dark theme. The left sidebar has icons for Explorer, Search, Open, and others. The main area shows a file named `main.go` with the following Go code:

```
go main.go M x
File Edit Selection View Go Run Terminal Help ← → Q, go-loadbalancer
GO-LOADBALANCER
src/main.go M
src > main.go > Server > IsAlive
65 func (lb *LoadBalancer) getNextAvailableServer() Server {
73 }
74
75 func (lb *LoadBalancer) ServeProxy(rw http.ResponseWriter, r *http.Request) {
76     targetServer := lb.getNextAvailableServer()
77     fmt.Printf("Forwarding request to address %q\n", targetServer.Addr())
78     targetServer.Serve(rw, r)
79 }
80
81 func main() {
82     servers := []Server{
83         newSimpleServer("https://www.facebook.com"),
84         newSimpleServer("https://www.bing.com"),
85         newSimpleServer("https://duckduckgo.com"),
86     }
87
88     lb := NewLoadBalancer("8000", servers)
89
90     // ✅ Fixed: Correct handler registration
91     http.HandleFunc("/", func(rw http.ResponseWriter, r *http.Request) {
92         lb.ServeProxy(rw, r)
93     })
}
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\thisa\Downloads\go-loadbalancer\src> go run main.go
PS C:\Users\thisa\Downloads\go-loadbalancer\src>
PS C:\Users\thisa\Downloads\go-loadbalancer\src>
PS C:\Users\thisa\Downloads\go-loadbalancer\src> go run main.go
Serving requests at 'localhost:8000'
```

The bottom status bar shows the terminal command and output: `PS C:\Users\thisa\Downloads\go-loadbalancer\src> go run main.go`, followed by three blank lines, and then `PS C:\Users\thisa\Downloads\go-loadbalancer\src>`. The status bar also includes icons for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS, along with file navigation and system status.

THISARA KANDAGE

UNDERGRADUATE - SLIIT

E-mail LinkedIn GitHub Website

When I type <http://localhost:8000> on web browser the bellow error message prompted! it means the load balancer working correctly. Let me explain why?



It's due to a combination of **HTTPS, redirects, and the round-robin logic** in your code.

```
func main() {
    servers := []Server{
        newSimpleServer("https://www.facebook.com"),
        newSimpleServer("https://www.bing.com"),
        newSimpleServer("https://duckduckgo.com"),
    }

    lb := NewLoadBalancer("8000", servers)
```

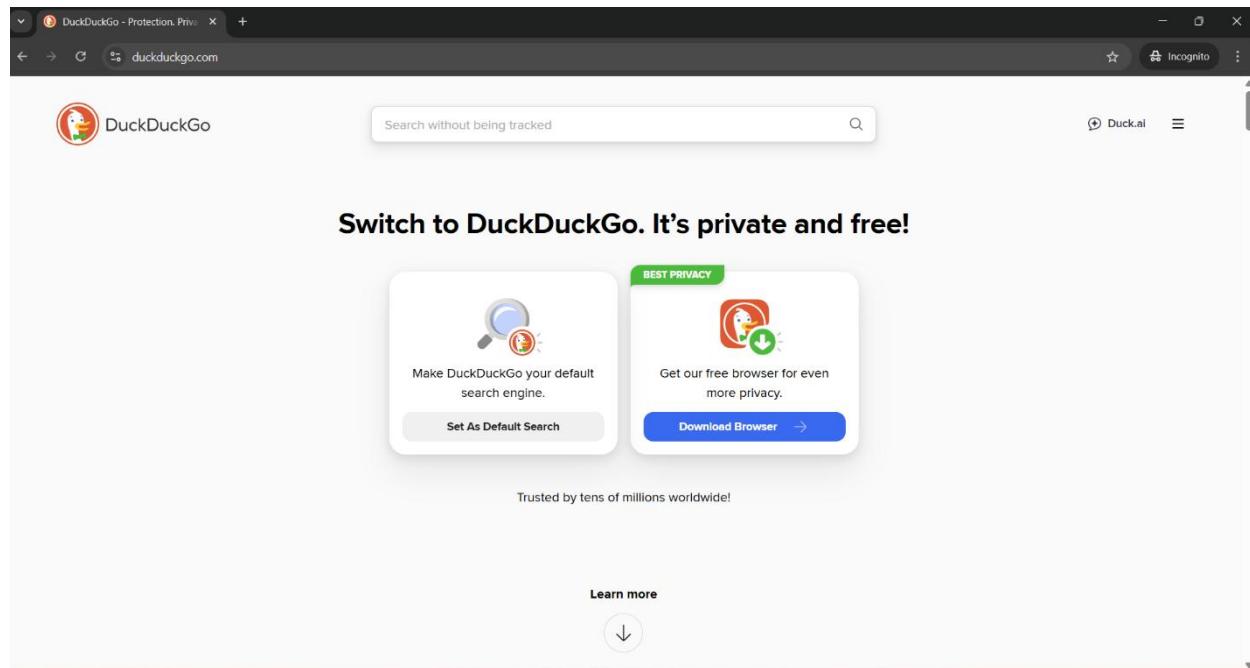
- Facebook **does not allow reverse proxying easily**. It enforces HTTPS, uses HSTS, and often requires cookies, redirects, or special headers.
- `httputil.NewSingleHostReverseProxy` just forwards the request — **it cannot bypass Facebook's protections**. So the first request usually fails (connection refused, redirect, or blocked).

THISARA KANDAGE

UNDERGRADUATE - SLIIT

E-mail LinkedIn GitHub Website

Then again hit the URL it redirected to DuckDuckgo!



Because of this reason - My round-robin logic

```
func (lb *LoadBalancer) getNextAvailableServer() Server {
    server := lb.servers[lb.roundRobinCounter%len(lb.servers)]
    for !server.IsAlive() {
        lb.roundRobinCounter++
        server = lb.servers[lb.roundRobinCounter%len(lb.servers)]
    }
    lb.roundRobinCounter++
    return server
}
```

```
func (s *simpleserver) IsAlive() bool {
    return true
}
```

- So it **doesn't really check if the first server (Facebook) is reachable.**
- But when a request fails due to HTTPS or redirect issues, your browser may **retry**, and the next server in round-robin is picked (Bing → then DuckDuckGo).
- Essentially, your **round-robin counter increments every request**, even if the first server “failed” in the browser. That’s why the next request goes to a different backend.

## After successfully executed the terminal output

The screenshot shows the GoLand IDE interface with the following details:

- File Structure:** The project is named "GO-LOADBALANCER" with a single source file "main.go".
- Code View:** The code implements a simple load balancer. It defines a struct "Server" with methods "IsAlive" and "ServeProxy". It also defines a struct "LoadBalancer" with a method "getNextAvailableServer". The "main" function creates a slice of servers (Facebook, Bing, DuckDuckGo) and a load balancer. It then runs a HTTP server at port 8000 that handles requests by calling "ServeProxy".
- Terminal View:** The terminal shows the command "go run main.go" being run three times. Each run outputs the server being used for each request: "Forwarding request to address \"https://www.facebook.com\"", "Forwarding request to address \"https://www.bing.com\"", and "Forwarding request to address \"https://duckduckgo.com\"".
- Bottom Status:** The status bar shows "ThisaraK943 (1 hour ago)" and "Tab Size: 4 UTF-8 CRLF".