Если вы видите что-то необычное, просто сообщите мне.

Writing a Reverse Proxy in just one line with Go

Leave your programming language hang ups at the door and come admire the best standard library I’ve ever come across.

This is all the code you actually require…

Choosing a Programming Language for a project shouldn’t be like declaring who your favourite team is. it should be a matter of pragmatism and choosing the right tool for the right job.

In this post I want to show you when and why Go shines as a language. Specifically I want to show you how rock solid their standard lib is for basic internet programming. Even more specifically… were gonna write a Reverse Proxy!

“Go has a lot going for it but where it really struts it stuff is in these lower level network plumbing tasks, there’s no better language.” What is a reverse proxy? A big fancy way of saying a traffic forwarder. I get a request send from a client, send that request to another server, receive a response from the server and forward it back to the client. The reverse part of this simply means the proxy itself determines where to send traffic and when

Why is it useful? Because the concept is so simple it can be applied to assist in many different cases: Load balancing, A/B Testing, Caching, Authentication etc…

By the end of this short post you will have learned how to:

Serve HTTP requests

Parse the body of a request Serve traffic to another server using a Reverse Proxy Our Reverse Proxy Project Lets dive into the actual project. What we are going to do is have a web server that:

1. Takes requests
2. Reads the body of a request, specifically the proxy_condition field
3. If the proxy domain is equal to A send traffic to URL 1
4. If the proxy domain is equal to B send traffic to URL 2
5. If the proxy domain is neither then send traffic to the Default URL.

Prerequisites

Go for programming with. http-server for creating simple servers with. Setting up our environment First thing we want to do is input all the required configuration variables into our environment so that we can both use them in our application while keeping them out of source code.

I find the best approach is to create a .env file that contains the desired environment variables.

Below is what I have for this specific project:

```
export PORT=1330
export A_CONDITION_URL="http://localhost:1331"
export B_CONDITION_URL="http://localhost:1332"
export DEFAULT_CONDITION_URL="http://localhost:1333"
```

This is a habit I picked up from the 12 Factor App

After you save your .env file you can run:

```
source `.env
```

to configure load the config into your environment any time.

Laying the foundation of our project Next lets create a file called main.go that does the following:

1. When started logs the PORT, A_CONDITION_URL, B_CONDITION_URL, and DEFAULT_CONDITION_URL environment variables to the console
2. Listen for requests on the path: /

```
package main
```

```
import (
	"bytes"
	"encoding/json"
	"io/ioutil"
	"log"
	"net/http"
	"net/http/httputil"
	"net/url"
	"os"
	"strings"
)

// Get env var or default
func getEnv(key, fallback string) string {
	if value, ok := os.LookupEnv(key); ok {
		return value
	}
	return fallback
}

// Get the port to listen on
func getListenAddress() string {
	port := getEnv("PORT", "1338")
	return ":" + port
}

// Log the env variables required for a reverse proxy
func logSetup() {
	a_condtion_url := os.Getenv("A_CONDITION_URL")
	b_condtion_url := os.Getenv("B_CONDITION_URL")
	default_condtion_url := os.Getenv("DEFAULT_CONDITION_URL")

	log.Printf("Server will run on: %s\n", getListenAddress())
	log.Printf("Redirecting to A url: %s\n", a_condtion_url)
	log.Printf("Redirecting to B url: %s\n", b_condtion_url)
	log.Printf("Redirecting to Default url: %s\n", default_condtion_url)
}

// Given a request send it to the appropriate url
func handleRequestAndRedirect(res http.ResponseWriter, req *http.Request) {
```

```
  // We will get to this...
}

func main() {
	// Log setup values
	logSetup()

	// start server
	http.HandleFunc("/", handleRequestAndRedirect)
	if err := http.ListenAndServe(getListenAddress(), nil); err != nil {
		panic(err)
	}
}
```

(Let’s get the skeletons out of the closet so we can move onto the fun stuff.)

Now you should be able to run

Parse the request body Now that we have the skeleton of our project together we want to start creating the logic that will handle parsing the request body. Start by updating handleRequestAndRedirect to parse the proxy_condition value from the request body.

```
type requestPayloadStruct struct {
	ProxyCondition string `json:"proxy_condition"`
}

// Get a json decoder for a given requests body
func requestBodyDecoder(request *http.Request) *json.Decoder {
	// Read body to buffer
	body, err := ioutil.ReadAll(request.Body)
	if err != nil {
		log.Printf("Error reading body: %v", err)
		panic(err)
	}

	// Because go lang is a pain in the ass if you read the body then any susequent calls
	// are unable to read the body again....
	request.Body = ioutil.NopCloser(bytes.NewBuffer(body))

	return json.NewDecoder(ioutil.NopCloser(bytes.NewBuffer(body)))
```

```
}

// Parse the requests body
func parseRequestBody(request *http.Request) requestPayloadStruct {
	decoder := requestBodyDecoder(request)

	var requestPayload requestPayloadStruct
	err := decoder.Decode(&requestPayload)

	if err != nil {
		panic(err)
	}

	return requestPayload
}

// Given a request send it to the appropriate url
func handleRequestAndRedirect(res http.ResponseWriter, req *http.Request) {
	requestPayload := parseRequestBody(req)
	  // ... more to come
}
```

(Basic parsing of a JSON blob to a struct in Go.)

Use

proxy_condition to determine where we send traffic Now that we have the value of the proxy_condition from the request we will use it to decide where we direct our reverse proxy to.

Remember from earlier that we have three cases:

If proxy_condition is equal to A then we send traffic to A_CONDITION_URL If proxy_condition is equal to B then we send traffic to B_CONDITION_URL Else send traffic to DEFAULT_CONDITION_URL

```
// Log the typeform payload and redirect url
func logRequestPayload(requestionPayload requestPayloadStruct, proxyUrl string) {
	log.Printf("proxy_condition: %s, proxy_url: %s\n", requestionPayload.ProxyCondition, proxyUrl)
}
```

```
// Get the url for a given proxy condition
func getProxyUrl(proxyConditionRaw string) string {
	proxyCondition := strings.ToUpper(proxyConditionRaw)

	a_condtion_url := os.Getenv("A_CONDITION_URL")
	b_condtion_url := os.Getenv("B_CONDITION_URL")
	default_condtion_url := os.Getenv("DEFAULT_CONDITION_URL")

	if proxyCondition == "A" {
		return a_condtion_url
	}

	if proxyCondition == "B" {
		return b_condtion_url
	}

	return default_condtion_url
}

// Given a request send it to the appropriate url
func handleRequestAndRedirect(res http.ResponseWriter, req *http.Request) {
	requestPayload := parseRequestBody(req)
	url := getProxyUrl(requestPayload.ProxyCondition)
	logRequestPayload(requestPayload, url)
  // more still to come...
}
```

Reverse Proxy to that URL

Finally we are onto the actual reverse proxy! In so many languages a reverse proxy would require a lot of thought and a fair amount of code or at least having to import a sophisticated library.

However Golang’s standard library makes creating a reverse proxy so simple it’s almost unbelievable. Below is essentially the only line of code you need:

httputil.NewSingleHostReverseProxy(url).ServeHTTP(res, req) Note that in the following code we add a little extra so it can fully support SSL redirection (though not necessary):

```
// Serve a reverse proxy for a given url
func serveReverseProxy(target string, res http.ResponseWriter, req *http.Request) {
	// parse the url
	url, _ := url.Parse(target)

	// create the reverse proxy
	proxy := httputil.NewSingleHostReverseProxy(url)

	// Update the headers to allow for SSL redirection
	req.URL.Host = url.Host
	req.URL.Scheme = url.Scheme
	req.Header.Set("X-Forwarded-Host", req.Header.Get("Host"))
	req.Host = url.Host

	// Note that ServeHttp is non blocking and uses a go routine under the hood
	proxy.ServeHTTP(res, req)
}

// Given a request send it to the appropriate url
func handleRequestAndRedirect(res http.ResponseWriter, req *http.Request) {
	requestPayload := parseRequestBody(req)
	url := getProxyUrl(requestPayload.ProxyCondition)

	logRequestPayload(requestPayload, url)

	serveReverseProxy(url, res, req)
}
```

The one time in the project it felt like Go was truly getting out of my way.Start it all up

Ok now that we have this all wired up setup our application on port 1330 and our 3 simple servers on ports 1331–1333 (all in separate terminals):

```
source .env && go install && $GOPATH/bin/reverse-proxy-demo
http-server -p 1331
http-server -p 1332
http-server -p 1333
```

With all these up and ruuning we can start to send through a requests with a json body in another terminal like so: F

```
curl --request GET \
  --url http://localhost:1330/ \
  --header 'content-type: application/json' \
  --data '{
    "proxy_condition": "a"
  }'
```

If your looking for a great HTTP request client I cannot recommend Insomnia enough.

and Voila we can start to see our reverse proxy directing traffic to one of our 3 servers based on what we set in the proxy_condition field!

image (Its alive!!!)

Wrap Up

Go has a lot going for it but where it really struts it stuff is in these lower level network “plumbing” tasks, there’s no better language. What we’ve written here is simple, performant, reliable and very much ready for use in production.

For simple services I can see myself reaching for Go again in the future.

□□ This is open source! you can find it here on Github

❤ I only write about programming and remote work. If you follow me on Twitter I won’t waste your time.

Revision #1
Created 2022-03-16 12:15:42 UTC by gasick
Updated 2023-04-16 19:36:18 UTC by gasick